JFP **28**, e16, 50 pages, 2018.
doi:10.1017/S0956796818000114

*Ghostbuster: A tool for simplifying and converting GADTs**

TIMOTHY A. K. ZAKIAN
University of Oxford, Department of Computer Science, Oxford, UK
(*e-mail:* timothy.zakian@cs.ox.ac.uk)

TREVOR L. MCDONELL
University of New South Wales, School of Computer Science and Engineering, Sydney, AUS
(*e-mail:* tmcdonell@cse.unsw.edu.au)

MATTEO CIMINI
University of Massachusetts Lowell, Department of Computer Science, Lowell, MA, USA
(*e-mail:* matteo_cimini@uml.edu)

RYAN R. NEWTON
Indiana University, School of Informatics, Computing, and Engineering, Bloomington, IN, USA
(*e-mail:* rrnewton@indiana.edu)

Abstract

Generalized Algebraic Data Types, 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 toward *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.

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:

* This work was supported by NSF awards 1453508 and 1337242. Timothy Zakian was funded by the Clarendon Fund.

```
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 cannot 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 where
  Nil'  :: Vec' a
  Cons' :: a → Vec' 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 with type variables. A person tasked with working in an *existing* project is unlikely to be able to re-implement all of a project's datatypes and operations on them from scratch.

Inspired by the theory of *ornaments* (McBride, 2010; Dagand & McBride, 2012), 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) defines canonical simplified datatypes; and (2) creates 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—or that this class of datatypes is large enough to be meaningful—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 #-}
```

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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

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

Since `downVec` may fail at runtime if the actual size of the vector does not match the expected size as specified (in the type `n` and checked at runtime via `Typeable`) by the caller, we take the approach of returning the result wrapped in `Maybe`, but we could also choose to throw an `error` on failure or return a diagnostic message using `Either`. Furthermore, sometimes during the down-conversion process, we cannot determine the specific type index that must be *synthesized*, and we must therefore keep it *sealed* under an existential binding. We can then make use of this sealed type in contexts that operate over any instance of the sealed type:

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

downVecS :: 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 de-serialization 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, since Ghostbuster generates user-level code, we can leverage the `downVec` function that is created by the tool to 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 = downVec v' ]
```

Another option for de-serialization to GADTs is by using GHC's interpreter as a library via the `Hint` package.[2] Using this method, after we have de-serialized to a simpler—non-type-indexed—datatype a code generator then converts expressions in this simplified datatype into an equivalent Haskell expression using constructors of the original GADT. This is then passed to `Hint` *as a string and interpreted*, with the value returned to the running program. While this represents the most scalable

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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

method until now, it has many downsides, and we compare this method against Ghostbuster in Section 8.

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 up-conversion functions are total and up-then-down 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` typeclass 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 9,026 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. Further, while Haskell is used in this paper, care has been taken to ensure that the theory and tooling that we develop is applicable to other functional languages with GADTs.

The layout of this paper is as follows. In the next section, we describe the design constraints and prerequisites for Ghostbuster and give an intuition for our ambiguity criteria. In Section 3, we give some real-world examples and use cases. We then define and formalize the core language used by Ghostbuster in Section 4. After this, we present our ambiguity criteria in Section 5, and then detail our algorithm for down- and up-conversion functions and prove the round-trip property for our algorithm in Section 6. In Section 7, we discuss some of the Haskell-specific design decisions that we have made, possible extensions, and possible challenges we might face when extending to other languages. Section 8 then evaluates our prototype implementation of the algorithm against other methods. We finish with a discussion of related work and conclusions in Sections 9 and 10.

A preliminary version of this paper appeared in the *Proceedings of the 2016 International Conference on Functional Programming* (McDonell *et al.*, 2016). We

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

have added a discussion on how strongly typed GADTs may preclude certain common algorithms (Section 3.2), significantly expanded our formalization of the core language (Sections 4.3 and 4.4), expanded the exposition of our ambiguity criteria (Sections 5.2 and 5.3), and expanded Section 6 to include our algorithm for generating type representations and type equality operations.

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 an *up-conversion* function from the original datatype to the newly generated one, as well as a *down-conversion* function from the simplified type back to the original, re-establishing type-level invariants as necessary.

However, there are a number of different criteria that must be satisfied by the datatype and to-be-erased type variables in question before we can guarantee that we will be able to generate up- and down-conversion functions. This section motivates these criteria by highlighting some of the different problems that can arise when attempting to erase type variables from a datatype, along with detailing some of the language features that are needed in order to implement the Ghostbuster algorithm. In particular, Section 2.1 details the runtime type-testing facilities that we require for the Ghostbuster algorithm, Section 2.2 gives the intuition behind the different erasure settings that need to be considered for type variables, and Sections 2.3 and 2.4 informally introduce our type information flow criteria. 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 (up-conversion), which we then re-establish at some later point (down-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
  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
```

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

While this new `Typeable` type representation in GHC 8.2 could be used in Haskell, we have decided to instead generate these type-indexed `TypeRep` values ourselves for a couple of reasons: using embedded `TypeRep` *values* rather than embedded `Typeable` *class constraints* simplifies our core language since we do not have to handle typeclass constraints (Section 4); and other languages that have GADTs do not necessarily have such a way to connect runtime type tests to the type system (e.g., OCaml), thus using locally generated `TypeRep`s allows this work to be implemented in other languages that do not have typeclass constraints or built-in type-indexed type representations.

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 where
  Nil  :: List a
  Cons :: a → List 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 down-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:

```
data List' where
  Nil' :: List'
  Cons' :: ∀ a. TypeRep a → a → List' → List'

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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

In contrast to `Vec'`, where the erased type was synthesized during down-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 down-conversion, we check that each element of the list does indeed have the same type the user expects:

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

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

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

downVec :: Typeable n ⇒ Vec' a → Maybe (Vec a n)
downVec v' =
  case downVecS v' of
    SealedVec v → gcast v
```

This highlights the key difference between erasures in checked versus synthesized mode. In order to perform down-conversion on `List'`, we must examine the type of each element and compare it to the type that we expect; thus, we cannot 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, down-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.[3] In this sense, synthesized variables require *a posteriori* knowledge about what they should be, while checked variables require *a priori* knowledge of their type during the down-conversion process.

[3] Where `Data.Typeable.gcast` performs runtime type tests and type casts and returns a `Just` of the type cast value if test and cast are successful and `Nothing` otherwise.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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 yield the following simplified types for `ConsL`, respectively:

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

Thus, during down-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 cannot 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:

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

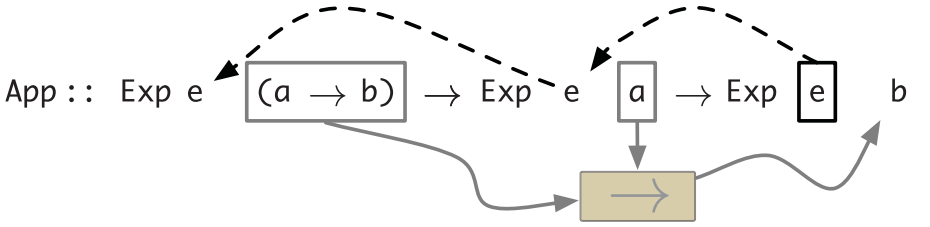

Fig. 1. Information flow within Ghostbuster for type variables in checked and synthesized contexts for the `App` constructor. Boxes are placed around those places where base type information is determined.

```
{-# 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 can be expressed as a series of information flow criteria that detail how the types of the fields in the data constructor constrain each other.[4]

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 `downExp` down-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 `downExp` 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 down-conversion process, since this type is synthesized by `downExp`, 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. Figure 1 shows how this type information is flowed through the `App` type constructor for type variables in checked and synthesized position, where to-be-checked type information during our recursive processing of the datatype is represented by dashed black arrows, synthesized type information being returned out from the recursive processing of the constructor is represented by grey arrows, and

[4] These information flow criteria are closely related to inherited and synthesized attributes in attribute grammars which are discussed in Section 9.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

the shaded grey arrow represents the determination of `b` based on the synthesized information about `a → b` and `a`.

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.[5] The same local reasoning holds for the requirements on checked types as well as synthesized. Together, we call these information flow checks our *ambiguity criteria* and formalize this in Section 5.

Erased types that escape. Ghostbuster performs one final check before declaring that 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? Moreover, even if a type constructor fits all these criteria, we cannot be assured that a valid `Functor` instance would give rise to valid erasures: if we took the default `Functor` instance for pairs in Haskell, the conversion would only be applied to the second element of the pair which is nothing like what we would like to get out of our conversion process. This is an incredibly tricky design space, and one in which it is not only difficult to determine the intended behavior that we would want for any particular `Functor` instance to give rise to valid erasures, but also one in which it is impossible to determine a priori whether or not a given `Functor` instance has those desired behaviors. We therefore do not handle these cases in Ghostbuster.

Thus, all the datatypes we consider—from `Vec` to `List` to `Exp` and the thousands of others we survey in Section 8—only have Ghostbusted types directly as fields, not as type arguments. Only when all of these constraints are met will Ghostbuster generate the requested datatypes and conversion functions, guaranteeing that they will type-check and successfully round-trip all (type-correct) values.

3 Life with Ghostbuster

In this section, we describe several concrete scenarios in which Ghostbuster can be used to make life easier for the programmer by allowing them to more easily implement standard algorithms over a GADT abstract syntax tree (AST) (Section 3.2), and derive standard typeclasses in Haskell for GADTs that would otherwise

[5] This is more local than other (tangentially related) features such as the "." notation in Idris (Brady *et al.*, 2004) and Agda, which signifies a type is runtime-irrelevant and should be erased during compilation. Irrelevance requires a whole-program check to verify whether the annotation can be fulfilled.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

require hand-written instances (Section 3.3). We take as a running example the simple expression language which we define below.

3.1 A type-safe expression language

Implementing type-safe ASTs is perhaps the most common application of GADTs. Consider the following language representation:[6]

```
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 (Altenkirch & Reus, 1999).

```
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 `(→)`.

```
data Typ a where
  Int :: Typ Int
  Arr :: Typ a → Typ b → Typ (a → b)
```

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 (Chakravarty *et al.*, 2011; McDonell *et al.*, 2013; McDonell *et al.*, 2015) 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.

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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

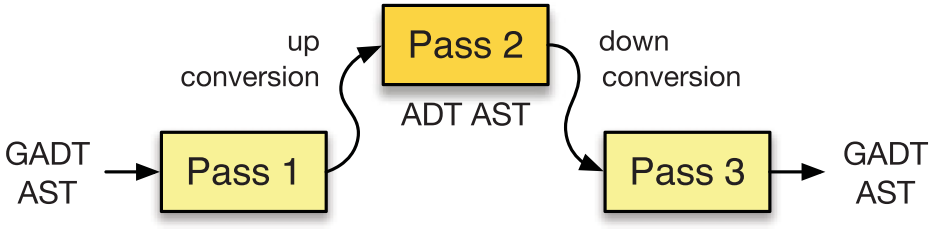

Fig. 2. In this scenario, we wish to add a prototype transformation into a compiler that uses sophisticated types, but against a simpler representation. For example, we may want to verify that an optimization does indeed improve performance, before tackling the type-preservation requirements of the GADT 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 2, 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,[7] a user could just as well build the same conversion functions to and from a less strictly typed AST representation themselves. However, this introduces a significant maintenance burden, since whenever the original (more strictly typed) AST is changed both the less strictly typed AST along with the conversion functions must be changed as well. Furthermore, these conversion functions are tricky to implement, and since the Haskell type checker cannot 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 down-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.

[7] Which is itself written in Haskell, with no modifications to GHC required to support it.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

```
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

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 (2005) provides exactly that, where renaming and substitution are instances of a single traversal, propagating operations on variables closed under shifting 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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

```
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

implementation shown in Listing 2; since no type-level environment manipulation needs to be taken into account when performing substitution (using `shift` and `rebuild`), this simply amounts to 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'
```

[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.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

and conversion functions:

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

downExp :: (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).

While in this case, an algorithm for operating directly on the richly typed terms already existed, there is no guarantee that we will be so lucky for all of the operations we may wish to perform. An example of this arises in the common compiler optimization of shrinking (Appel, 2007) in which functions that are only used once are inlined, dead-code is eliminated, and constant folding is performed. In particular, while linear-time shrinking algorithms are known for normal ASTs (Appel & Jim, 1997; Benton *et al*., 2005), when using ASTs in which GADTs are used to maintain type-level invariants (such as in our richly typed expression language), we are no longer able to use these algorithms: the linear-time shrinking algorithm must be able to contract redexes in any order, however, doing this efficiently requires the ability to in-place update the AST of our program—which then requires us to prove (and re-prove) that the type-level invariants in our AST are maintained for each transformation. This represents at the very best a significant—if not insurmountable—barrier to implementing such an algorithm for richly typed ASTs.

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 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 derived instances for the simplified datatypes instead.

```
instance (...) ⇒ Show (Exp env t) where
  show = show . upExp
```

Similarly, some libraries include Template Haskell (Sheard & Peyton Jones, 2002) routines that can be used to automatically generate instances for the typeclasses of that library. Although these run into problems when applied to GADTs,[10] once more we can use Ghostbuster to circumvent this limitation. As an example, we can

[10] `https://www.reddit.com/r/haskell/comments/5acj3g/derive_fromjson_for_gadts/`

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

easily generate JSON (de-)serialization instances for the `aeson` package[11] applied to our richly typed terms:

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

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

instance (...) ⇒ FromJSON (Exp env t) where
  parseJSON v = do
    v' ← parseJSON v :: Parser Exp'
    return $ fromMaybe (error "...") (downExp 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, which may be otherwise precluded.

4 Core language definition

Before covering the ambiguity criteria and core algorithm for Ghostbuster in Sections 5 and 6, 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 4. 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 (Schrijvers *et al.*, 2009a), which supports decidable checking (Cheney & Hinze, 2003; Simonet & Pottier, 2007) (but 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 throughout the rest of the paper.

4.1 Syntax

The syntax of terms and types in Figure 4 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

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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

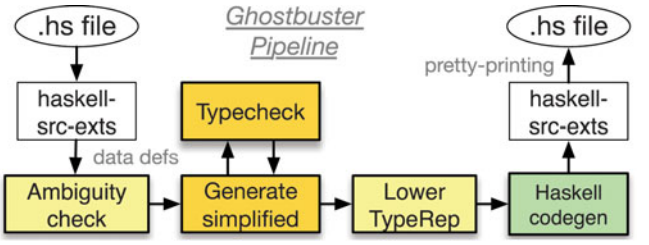

Fig. 3. The architecture of the Ghostbuster tool, which processes data definitions in several passes, resulting in pretty-printed Haskell source on disk. Note that only the ingestion and code generation phases are Haskell specific: the ambiguity check through lowering phases are implemented in terms of our core language.

Data constructors	K		
Type constructors	T, S		
Type variables	a, b, k, c, s		
Term variables	x, y, z		
Monotypes	τ	$::=$	$a \mid \tau \to \tau \mid T\ \overline{\tau} \mid \texttt{TypeRep}\ \tau$
Type Schemes	σ	$::=$	$\tau \mid \forall \overline{a}.\tau$
Typing Environments	Γ	$::=$	$\cdot \mid x : \sigma, \Gamma$
Constraints	C, D	$::=$	$\varepsilon \mid \tau \sim \tau \mid C \wedge C$
Substitutions	ϕ	$::=$	$\varnothing \mid \phi, \{a := \tau\}$
Programs	*prog*	$::=$	$dd_1 \ldots dd_n;\ vd_1 \ldots vd_m;\ e$
Data Definitions	*dd*	$::=$	$\texttt{data}\ T\ \overline{k}\ \overline{c}\ \overline{s}\ \texttt{where}$ $\texttt{K} :: \forall\ \overline{k}, \overline{c}, \overline{s}, \overline{b}.$ $\tau_1 \to \cdots \to \tau_p \to T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$
Value Definitions	*vd*	$::=$	$x :: \sigma;\ x = e$
Terms	e	$::=$	$x \mid \lambda x :: \tau.e \mid e\ e \mid e\ \tau_1 \rhd \tau_2$
		$\mid$	$\texttt{let}\ x :: \sigma = e\ \texttt{in}\ e$
		$\mid$	$\texttt{case}[\tau]\ e\ \texttt{of}\ [p_i \to e_i]_{i \in I}$
		$\mid$	$\texttt{typecase}[\tau]\ e\ \texttt{of}$ $(\texttt{typerep}\ \mathbb{T})\ x_1 \ldots x_n \to e \mid \_ \to e$
		$\mid$	$\texttt{if}\ e \simeq_\tau e\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid$	$K\ \{\overline{a}\} \mid \texttt{typerep}\ \mathbb{T}$
Values	v	$::=$	$K\ \{\overline{a}\}\ \overline{e} \mid \lambda x :: \tau.e$
Patterns	p	$::=$	$K\ x_1 \ldots x_n$
Type names	$\mathbb{T}$	$::=$	$T \mid \mathit{ArrowTy} \mid \mathit{Existential}$

Fig. 4. The core language manipulated by Ghostbuster.

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 *vd*s. 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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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 *de-constructing* type representations into their component parts—for example, splitting a function type into an input type and output type. An example of using `typecase` to perform this destructuring of `typereps` can be seen in the code on Page 33.

We specifically do not handle typeclasses. If we were to handle them, we would need to be able to discover and then prove the various typeclass constraints in the same way that we do for type constraints. However, verifying such constraints is impossible without either using `Constraints` from `GHC.Prim`, using `unsafeCoerce`, or dropping down into GHC's intermediate language. More generally, in order to allow typeclass constraints, we would need not only type-indexed, but *typeclass-indexed* type representations. And while GHC allows us to do this by hooking into the underlying intermediate language, this is not something that is a feature of Haskell or any other (non-intermediate) languages that we are aware of.

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 (C):

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

We avoid this normalization. Because we lack typeclass 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 C, which we will use during type-checking (Figure 7). 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 typeclass 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 2.2 and 2.1).

4.2 Type system

The typing rules for our language are syntax-directed and are given in Figures 5–8. The main judgment forms are the following:

$$\begin{array}{cc} \text{Well-typed expressions} & \text{Well-typed patterns} \\ C, \Gamma \vdash_e e : \tau & C, \Gamma \vdash_p p \rightarrow e : \tau_1 \rightarrow \tau_2 \end{array}$$

along with judgments for extending Γ with data definitions ($\Gamma \vdash_d dd : \Gamma'$), value definitions ($\Gamma \vdash_v vd : \Gamma'$), and for typing whole programs ($\Gamma \vdash_{prog} prog : \tau$). We also make use of some syntactic sugar in the typing rules and we will often write both $\tau_1, \ldots, \tau_n$ and $\tau_1 \ldots \tau_n$ as $\overline{\tau}^n$, and $\tau_1 \rightarrow \cdots \rightarrow \tau_n$ as $\overrightarrow{\tau}^n$.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

$$\textsc{TypeCase}\quad \frac{C,\Gamma \vdash_e \texttt{typerep}\ T : \overrightarrow{\texttt{TypeRep}\ \tau}^n \to \texttt{TypeRep}\ (T\ \overline{\tau}^n) \qquad C,\Gamma \vdash_e e : \texttt{TypeRep}\ \tau_0 \qquad C \wedge (\tau_0 \sim T\ \overline{\tau}^n),\Gamma \cup \{x_1 : \texttt{TypeRep}\ \tau_1,\ldots,x_n : \texttt{TypeRep}\ \tau_n\} \vdash_e e' : \tau \qquad C,\Gamma \vdash_e e'' : \tau}{C,\Gamma \vdash_e \texttt{typecase}[\tau]\ e\ \texttt{of}\ ((\texttt{typerep}\ T)\ x_1 \ldots x_n) \to e' \mid \_ \to e'' : \tau}$$

$$\textsc{TypeRep}\quad \frac{T : \overline{\star}^n \in \Gamma}{C,\Gamma \vdash_e \texttt{typerep}\ T : \overrightarrow{\texttt{TypeRep}\ \tau}^n \to \texttt{TypeRep}\ (T\ \overline{\tau}^n)}$$

$$\textsc{IfTyEq}\quad \frac{C,\Gamma \vdash_e e_1 : \texttt{TypeRep}\ \tau_1 \qquad C,\Gamma \vdash_e e_2 : \texttt{TypeRep}\ \tau_2 \qquad C \wedge (\tau_1 \sim \tau_2),\Gamma \vdash_e e' : \tau \qquad C,\Gamma \vdash_e e'' : \tau}{C,\Gamma \vdash_e \texttt{if}\ e_1 \simeq_\tau e_2\ \texttt{then}\ e'\ \texttt{else}\ e'' : \tau}$$

Fig. 5. Typing rules for type representations and operations on them.

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

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

$$\textsc{ArrStruct}\quad \frac{C \models \tau_i \sim \tau_i'}{C \models \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'} \qquad \textsc{ArrCon}\quad \frac{C \models \tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}{C \models \tau_i \sim \tau_i'} \qquad \textsc{TyRepStruct}\quad \frac{C \models \tau_1 \sim \tau_2}{C \models \texttt{typerep}\ \tau_1 \sim \texttt{typerep}\ \tau_2}$$

$$\textsc{TyRepCon}\quad \frac{C \models \texttt{typerep}\ \tau_1 \sim \texttt{typerep}\ \tau_2}{C \models \tau_1 \sim \tau_2}$$

Fig. 6. Equality theory for the Ghostbuster type system.

Many of the typing rules are standard, but a few—in particular, TypeRep, TypeCase, and IfTyEq—are unique to our language and separated into Figure 5. Here, the TypeRep and TypeCase rules only cover the type constructor T cases, the elided rules for the built-in type representations $\mathbb{T} = \mathit{Existential}$ and $\mathbb{T} = \mathit{ArrowTy}$ are nearly identical.

The TypeRep and TypeCase rules together allow us to encapsulate the arguments to the type constructor T in such a way that we can later on—at runtime—de-structure and bind type information for later use, and is why the TypeRep rule requires that each of its arguments are TypeReps, and why each x_i is given the correct TypeRep type in the right-hand side (RHS) of the first branch in the TypeCase rule. Most importantly, both the TypeCase and IfTyEq rules are the way in which we reflect the runtime type-tests that are performed statically in the typing judgments for the different branches, notice in the TypeCase rule that not only is e' typed with each $x_i : \texttt{TypeRep}\ \tau_i$ but also with the added constraint that $\tau_0 \sim T\ \overline{\tau}^n$, and similarly for the IfTyEq rule the consequent is typed with the added constraint $\tau_0 \sim \tau_1$. This can be viewed as being similar to what is done in occurence-typing (Tobin-Hochstadt

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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

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

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

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

$$\textsc{App}\ \frac{C,\Gamma \vdash_e e_1 : \tau_1 \to \tau_2 \qquad C,\Gamma \vdash_e e_2 : \tau_1}{C,\Gamma \vdash_e e_1\ e_2 : \tau_2} \qquad \textsc{Eq}\ \frac{C,\Gamma \vdash e : \tau_1 \qquad C \models \tau_1 \sim \tau_2}{C,\Gamma \vdash e\ \tau_1 \rhd \tau_2 : \tau_2}$$

$$\textsc{Case}\ \frac{C,\Gamma \vdash_e e : \tau \qquad \forall i \in I.\ C,\Gamma \vdash_p p_i \to e_i : \tau \to \tau'}{C,\Gamma \vdash_e \texttt{case}\ [\tau']\ e\ \texttt{of}\ [p_i \to e_i]_{i \in I} : \tau'}$$

$$\textsc{Let}\ \frac{C,\Gamma \vdash_e e_1 : \tau_1 \qquad C,\Gamma \cup \{x : \forall \overline{a}.\tau_1\} \vdash_e e_2 : \tau_2}{C,\Gamma \vdash_e \texttt{let}\ x :: \forall \overline{a}.\tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2} \qquad \textsc{Con}\ \frac{(K : \forall \overline{a}.\tau_1 \to \ldots \to \tau_n \to T\ \overline{\tau}) \in \Gamma \qquad \phi = \{\overline{a := \tau}\} \qquad \tau'_1 \to \ldots \to \tau'_n \to T\ \overline{\tau'} = \phi(\tau_1 \to \ldots \to \tau_n \to T\ \overline{\tau}) \qquad C,\Gamma \vdash_e e_i : \tau'_i}{C,\Gamma \vdash_e K\ e_1 \ldots e_n : T\ \overline{\tau'}}$$

Fig. 7. Typing rules for the core language.

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

$$\textsc{VDef}\ \frac{\overline{\varepsilon,\Gamma' \vdash_e e : \tau} \qquad \Gamma' = \Gamma \cup \{\overline{x : \forall \overline{a}.\tau}\}}{\Gamma \vdash_v \overline{x :: \forall \overline{a}.\tau; x = e} : \Gamma'}$$

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

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

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

$$\textsc{Prog}\ \frac{\vdash_d \overline{dd} : \Gamma_d \qquad \Gamma_d \vdash_v \overline{vd} : \Gamma_v \qquad \varepsilon,\Gamma_v \vdash_e e : \tau}{\vdash_{prog} \overline{dd}; \overline{vd}; e : \tau}$$

Fig. 8. Environment and program typing rules.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

& Felleisen, 2010a) where type-level information is added to an expression based upon the knowledge of a given (runtime) predicate having passed.

We depart from previous approaches in the Eq rule in Figure 7 by making the coercion of an expression from one type (τ_1) to another (τ_2) explicit via the form $e\ \tau_1 \rhd \tau_2$ (cf. Schrijvers *et al.* (2009a)). Without these explicit coercions, when a reduction in our semantics pushes us under a true branch in a type equality check there would be no way of recovering the (possibly needed) equations in our constraint environment to show type equality within that subexpression. We could get around this need for explicit type coercions in our language by having our semantics return both an expression along with a constraint environment that is then used in the statement of type preservation, but this would significantly complicate our semantics.

We lack a full kind system, but we do track the arity of constructors, with $T : \star^n \in \Gamma$ 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.

It will prove useful later on to only deal with constraint contexts in which all of the types in our type constraints have been reduced to the smallest possible. We therefore define a reduction relation $\rightharpoonup$ on type constraints $\tau_1 \sim \tau_2$ and reduce the type constraints in C based upon these:

$$\begin{array}{lll} (\textsc{TConR}) & T\ \overline{\tau}^n \sim T\ \overline{\tau'}^n & \rightharpoonup \tau_1 \sim \tau'_1 \wedge \ldots \wedge \tau_n \sim \tau'_n \\ (\textsc{TyRepR}) & \texttt{typerep}\ \tau_1 \sim \texttt{typerep}\ \tau_2 & \rightharpoonup \tau_1 \sim \tau_2 \\ (\textsc{ArrowR}) & (\tau_1 \to \tau_2) \sim (\tau'_1 \to \tau'_2) & \rightharpoonup \tau_1 \sim \tau'_1 \wedge \tau_2 \sim \tau'_2 \end{array} \quad (1)$$

Since conjunction is commutative, the order in which we perform the reductions on the type constraints in C does not matter. If one of the reduction relations defined above applies to a type constraint $\tau_1 \sim \tau_2$, we will call that constraint *reducible*. The one-step reduction of a constraint $C \rightharpoonup C'$ is then defined to be the reduction of a single reducible type constraint $\tau_1 \sim \tau_2$ in C, where we say that a constraint $\tau_1 \sim \tau_2$ is in C (or $\tau_1 \sim \tau_2 \in C$) if there exist constraints C_L and/or C_R such that C can be written as a conjunction of $\tau_1 \sim \tau_2$ with one, or both, of C_L and C_R (i.e., C is equal to one of $\tau_1 \sim \tau_2 \wedge C_R$, $C_L \wedge \tau_1 \sim \tau_2$, or $C_L \wedge \tau_1 \sim \tau_2 \wedge C_R$).

We would like for our reduced constraints to have the same power as our original constraints. Luckily, this proves to be the case: since each constraint reduction relation has a corresponding expansion in our equality theory in Figure 6, any constraint reduction that we perform can always be "re-expanded" to get back to the original constraints.

Lemma 1 (*Constraint reduction preserves power*)
Let C be a constraint, and C' a one-step reduction of C. Then, if $C \models \tau_1 \sim \tau_2$, then $C' \models \tau_1 \sim \tau_2$.

Proof
It suffices to show that given any C, and one-step reduction C' of C, that we can transform C' back to C. We proceed by case analysis on the reduction relation used

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

from C to C'. If

$$C = C_L \wedge T\ \overline{\tau}^n \sim T\ \overline{\tau'}^n \wedge C_R,$$

then we will use the TConR reduction rule to get

$$C' = C_L \wedge \underbrace{\tau_1 \sim \tau_1' \wedge \ldots \wedge \tau_n \sim \tau_n'}_{(\star)} \wedge C_R$$

Applying rule TStruct in Figure 6 w.r.t. T and $(\star)$, we get

$$C_L \wedge T\ \overline{\tau}^n \sim T\ \overline{\tau'}^n \wedge C_R$$

which is equal to C.

We apply the exact same reasoning in the other cases, replacing the use of TStruct with TyRepStruct and ArrStruct, and then using TyRepR and ArrowR, respectively. □

Definition 1 (*Constraint normalization*)
We say that a constraint C is *normalized* if $\nexists \tau_1 \sim \tau_2 \in C$ such that $\tau_1 \sim \tau_2$ is reducible.

We can now easily show that normalized constraint contexts have the same expressive power as the original constraints using Figure 1.

Theorem 1 (*Normalization preserves power*)
Let C be a constraint, and C' its normalization. Then, if $C \models \tau_1 \sim \tau_2$, then $C' \models \tau_1 \sim \tau_2$.

Proof
By the definition of normalization, we have the following:

$$C = C_0 \rightharpoonup C_1 \rightharpoonup \ldots \rightharpoonup C_n = C'$$

where each C_i is the one-step reduction of C_{i-1} for $i \geqslant 1$. We then have by repeated application of Lemma 1 that for each C_i, $C_i \models \tau_1 \sim \tau_2$. From this, we get that $C_n \models \tau_1 \sim \tau_2$. Since $C = C_0$ and $C_n = C'$, we are done. □

Now that we have shown that we do not lose any expressive power through normalizing the constraints, from now on we assume that all constraints in C are normalized.

4.3 Semantics

The operational semantics for our language is largely straightforward, with the only intricacies arising from our embedded type representations and how we handle type coercions. As we mentioned in the previous section, since our operational semantics does not build up or keep a constraint environment, we need to be able to erase type coercions and substitute types in our semantics in order to reflect the runtime type information that we gather at the type-level. This substitution and type coercion erasure in our semantics can be seen as replacing the propositional equality between types that we have in our type system with syntactic equality.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

Since the type coercions in our language represent transformation-time *promises* of type equalities that must be proved later on at runtime, the erasure of a type coercion in the semantics represents a discharging of the proof obligation for that coercion to be valid. Likewise, a substitution of one type for another can be seen as discharging any possible future proof obligations of type equality between the two types that may be needed later—changing some propositional equalities (that would rely on some possibly no-longer-provable constraint) during type checking into syntactic equalities. To this end, once we have proved that two types are equal, we eagerly substitute one of them in for all occurrences of the other in the remaining term, and perform any coercion erasures that we can. This leads to the following definition of a type substitution and coercion erasure function $e[\![\tau_1/\tau_2]\!]$ that recurses naturally on terms, and performs a standard substitution on types and handles coercions as follows:

$$\begin{array}{lll} x[\![\tau_1/\tau_2]\!] & = x & \\ (\lambda x :: \tau.e)[\![\tau_1/\tau_2]\!] & = \lambda x :: \tau[\tau_1/\tau_2].e[\![\tau_1/\tau_2]\!] & \\ (e_1\ e_2)[\![\tau_1/\tau_2]\!] & = e_1[\![\tau_1/\tau_2]\!]\ e_2[\![\tau_1/\tau_2]\!] & \\ (e\ \tau_3 \rhd \tau_4)[\![\tau_1/\tau_2]\!] & = e[\![\tau_1/\tau_2]\!] & \\ & \quad \text{if } \tau_3[\tau_1/\tau_2] \equiv \tau_4[\tau_1/\tau_2] & \\ (e\ \tau_3 \rhd \tau_4)[\![\tau_1/\tau_2]\!] & = e[\![\tau_1/\tau_2]\!]\tau_3[\tau_1/\tau_2] \rhd \tau_4[\tau_1/\tau_2] & \\ & \quad \text{if } \tau_3[\tau_1/\tau_2] \not\equiv \tau_4[\tau_1/\tau_2] & \\ & \vdots & \end{array} \tag{2}$$

where the last rule handles the case where we are erasing a type that may occur within another type, a simple example of which can be seen in the following, where we are saying that we have shown τ_1 equal to τ_3, and τ_2 equal to τ_4:

$$\begin{array}{l} \quad (e\ (\tau_1 \to \tau_2) \rhd (\tau_3 \to \tau_4))\ [\![\tau_1/\tau_3]\!][\![\tau_2/\tau_4]\!] \\ = (e[\![\tau_1/\tau_3]\!]\ (\tau_1 \to \tau_2)[\tau_1/\tau_3] \rhd (\tau_3 \to \tau_4)[\tau_1/\tau_3])\ [\![\tau_2/\tau_4]\!] \\ = (e[\![\tau_1/\tau_3]\!]\ (\tau_1 \to \tau_2) \rhd (\tau_1 \to \tau_4))\ [\![\tau_2/\tau_4]\!] \\ = (e[\![\tau_1/\tau_3]\!][\![\tau_2/\tau_4]\!]\ (\tau_1 \to \tau_2)[\tau_2/\tau_4] \rhd (\tau_1 \to \tau_4)[\tau_2/\tau_4]) \\ = (e[\![\tau_1/\tau_3]\!][\![\tau_2/\tau_4]\!]\ (\tau_1 \to \tau_4) \rhd (\tau_1 \to \tau_4)) \\ = e[\![\tau_1/\tau_3]\!][\![\tau_2/\tau_4]\!] \end{array}$$

We are now in a position to present the (small-step) operational semantics for our language, which are given in Figures 9 and 10. The judgment takes the form

$$\Sigma; D \vdash e \hookrightarrow e'$$

where $\Sigma : TermVar \to Exp$ is an environment mapping term variables to expressions and is what we use to allow us to deal with function definitions and recursive functions in the language, and $D : TypeConstr \to DataConstr$ is an environment that maps type names and type constructors to their corresponding runtime type representation—a data constructor within the language. It is this latter environment that is of particular importance to us, since it is through this mapping that we are able to reify our static assumptions in the type system with dynamic (runtime) type checks.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

$$\textsc{ValDef}\ \frac{}{\Sigma \vdash x :: \sigma; x = e \rightsquigarrow \Sigma, x \mapsto e} \qquad \textsc{Prog}\ \frac{\begin{array}{c}\Sigma \vdash vd_1 \rightsquigarrow \Sigma_1 \\ \vdots \\ \Sigma_{n-1} \vdash vd_n \rightsquigarrow \Sigma_n \\ \Sigma_n; D \vdash e \hookrightarrow^{\star} v\end{array}}{\Sigma; D \vdash \overline{dd}; \overline{vd_i}; e \Longrightarrow v}$$

Fig. 9. Environment creation and evaluation judgments for programs.

How we build up the Σ environment is straightforward, and is given by the Prog and ValDef rules in Figure 9. However, in order to handle possibly mutually recursive functions, it is important that the ValDef rule does not inspect the body e of the value definition in the process of building up our bindings. Building the type representations D for our language on the other hand is quite a bit more nuanced in its realization and relies on a minor yet important property of the language: if we encounter a `typerep` during the evaluation of the program, that `typerep` must have occurred as a subterm of the larger program earlier on (i.e., we cannot synthesize `typereps`). This property is critical to the correctness of our algorithm when using a closed-world type representation since it ensures that we can *syntactically* determine after we have generated the program which type representations need to be created, and then insert these into the generated program. Thus, there is an important phase distinction between program generation, type representation creation, and actual evaluation of the program. This property about type representations in our language is formalized in Theorem 2, and we go into detail on precisely how we determine and generate the various type representations that need to be formed in Section 6.4. But for now, it suffices to know that D contains runtime type representations for all type names that we may encounter under a `typerep` form in the generated program.

Using a closed-world type representation in order to test runtime type equality means that each type representation will simply be a GADT data constructor in the source language. The operational semantics makes use of this fact and thus expresses both `typecase` and $\simeq_\tau$ in terms of `case` statements and equality checks on data constructors, respectively, in the TypeCase and IfTyEq rules. Note also, how in each of the TypeCase and IfTyEq rules, we perform type equality erasure with the new type-level information that is gained through the process of matching on the type representations, and where $D^{-1}(K)$ represents the pre-image of the constructor K in D. Moreover, in order to simplify data constructor application, we (implicitly) η-expand all data constructors in our language and thus all applications of data constructors are fully saturated, with substitution on the body of the enclosing lambda expressions substituting in the correct types for free type variables, and the correct expressions for the variables that have been bound in the η-expansion.

4.4 Metatheory

In this section, we detail some of the metatheoretic properties of our language and show our core language typesafe. Further, we show that we are able to syntactically

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

$$\textsc{Var}\ \frac{\Sigma(x) = e}{\Sigma; D \vdash x \hookrightarrow e} \qquad \textsc{App}\ \frac{\Sigma; D \vdash e_1 \hookrightarrow e_1'}{\Sigma; D \vdash e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{TypCoEq}\ \frac{\tau_1 \equiv \tau_2}{\Sigma; D \vdash e_1\ \tau_1 \rhd \tau_2 \hookrightarrow e_1}$$

$$\textsc{TypeRep}\ \frac{D(\mathbb{T}) = K\ \{\overline{a}\}}{\Sigma; D \vdash \texttt{typerep}\ \mathbb{T} \hookrightarrow \lambda x_1 :: \tau_1 \ldots . \rightarrow \lambda x_n :: \tau_n\ .\ K\ \{[\overline{\tau}/\overline{a}]\}\ \overline{x_i}}$$

$$\textsc{Beta}\ \frac{}{\Sigma; D \vdash (\lambda x :: \tau . e_b)\ e \hookrightarrow e_b[e/x]} \qquad \textsc{Let}\ \frac{}{\Sigma; D \vdash \texttt{let}\ x :: \sigma = e\ \texttt{in}\ e_b \hookrightarrow e_b[e/x]}$$

$$\textsc{CaseEval}\ \frac{\Sigma; D \vdash e \hookrightarrow e'}{\Sigma; D \vdash \texttt{case}\ e\ \texttt{of}\ \overline{p \rightarrow e} \hookrightarrow \texttt{case}\ e'\ \texttt{of}\ \overline{p \rightarrow e}}$$

$$\textsc{CaseMatch}\ \frac{K\{\overline{a}\}\ \overline{x_i} \rightarrow e_i \in \overline{p \rightarrow e}}{\Sigma; D \vdash \texttt{case}\ K\{\overline{\tau}\}\ \overline{e}\ \texttt{of}\ \overline{p \rightarrow e} \hookrightarrow e_i[\overline{e}/\overline{x_i}][\![\overline{\tau}/\overline{a}]\!]}$$

$$\textsc{TypeCaseEval}\ \frac{\Sigma; D \vdash e \hookrightarrow e'}{\Sigma; D \vdash \texttt{typecase}\ e\ \texttt{of}\ e_{tpat} \rightarrow e_1 \mid \_ \rightarrow e_2 \hookrightarrow \texttt{typecase}\ e'\ \texttt{of}\ e_{tpat} \rightarrow e_1 \mid \_ \rightarrow e_2}$$

$$\textsc{TypeCaseMatch}\ \frac{\Sigma; D \vdash \texttt{typerep}\ \mathbb{T} \hookrightarrow \lambda x_1 :: \tau_1 \ldots . \rightarrow \lambda x_n :: \tau_n\ .\ K\ \{[\overline{\tau_i}/\overline{a}]\}\ \overline{x_i}}{\Sigma; D \vdash \texttt{typecase}\ \ K\ \{\overline{\tau}\}\ \overline{e_\tau}\ \texttt{of}\ (\texttt{typerep}\ \mathbb{T})\ \overline{x} \rightarrow e_1 \mid \_ \rightarrow e_2 \hookrightarrow e_1[\overline{e_\tau}/\overline{x}][\![\overline{\tau}/\overline{a}]\!]}$$

$$\textsc{TypeCaseFail}\ \frac{\Sigma; D \vdash \texttt{typerep}\ \mathbb{T} \hookrightarrow \lambda x_1 :: \tau_1 \ldots . \rightarrow \lambda x_n :: \tau_n\ .\ K'\{[\overline{\tau_i}/\overline{b}]\}\ \overline{x_i} \qquad K' \neq K}{\Sigma; D \vdash \texttt{typecase}\ \ K\ \{\overline{\tau}\}\ \overline{e_\tau}\ \texttt{of}\ (\texttt{typerep}\ \mathbb{T})\ \overline{x} \rightarrow e_1 \mid \_ \rightarrow e_2 \hookrightarrow e_2}$$

$$\textsc{IfTyEqLeft}\ \frac{\Sigma; D \vdash e_1 \hookrightarrow e_1'}{\Sigma; D \vdash \texttt{if}\ e_1 \simeq_\tau e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \hookrightarrow \texttt{if}\ e_1' \simeq_\tau e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4}$$

$$\textsc{IfTyEqRight}\ \frac{\Sigma; D \vdash e_2 \hookrightarrow e_2'}{\Sigma; D \vdash \texttt{if}\ e_1 \simeq_\tau e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \hookrightarrow \texttt{if}\ e_1 \simeq_\tau e_2'\ \texttt{then}\ e_3\ \texttt{else}\ e_4}$$

$$\textsc{IfTyEqTrue}\ \frac{\texttt{TypeRep}\ \tau_1 = D^{-1}(K) \qquad \texttt{TypeRep}\ \tau_2 = D^{-1}(K') \qquad K \equiv K'}{\Sigma; D \vdash \texttt{if}\ K\ \{\overline{a}\} \simeq_\tau K'\ \{\overline{a}\}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_1[\![\tau_1/\tau_2]\!]}$$

$$\textsc{IfTyEqFalse}\ \frac{K \not\equiv K'}{\Sigma; D \vdash \texttt{if}\ K\ \{\overline{a}\} \simeq_\tau K'\ \{\overline{b}\}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow e_2}$$

Fig. 10. Operational semantics for expressions in Figure 4.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

determine the required type representations during the conversion process in order to use closed-world type representations for the generated program.

Syntactic determination of type representations. As we mentioned in the previous section, in order to use a closed-world type representation in the generated program, it is crucial that we are able to statically determine the various type representations that we need to generate. The following theorem formalizes this ability:

Theorem 2 (*Syntactic determination of type representations*)
Let $prog = \overline{dd}; \overline{vd}; e'$ and let $\vdash_d \overline{dd} : \Gamma_d$. Let e be an expression such that $\epsilon, \Gamma_d \vdash_e e : \tau$. Then, if $\Sigma; D \vdash e \hookrightarrow^\star \texttt{typerep}\ \mathbb{T}$ for some type name $\mathbb{T}$, then there exists a program context $\mathtt{C}$[12] such that $e = \mathtt{C}[\texttt{typerep}\ \mathbb{T}]$.

Proof
By induction on e and the operational semantics. □

Type coercion erasure. As was mentioned in Section 4.3, type coercions represent outstanding proof obligations of runtime type equality. Thus, encountering a closed term $\epsilon, \cdot \vdash_e e'\ \tau_1 \rhd \tau : \tau$ where $\tau_1 \not\equiv \tau$ during the reduction process represents the obligation to prove something from nothing: since we do not have any other parts of the program to build constraints that can prove the equality between τ_1 and τ, we will have reached a final unprovable expression if we encounter such a form. Thus, while the following theorem *could* be viewed as a corollary to progress for the language, in fact we cannot view it as such since we need this invariant in the proof to show that we do not encounter (non-identity) type coercions during our reduction process.

Theorem 3 (*Type coercion erasure*)
Suppose that $\epsilon, \cdot \vdash_e e : \tau$. Then there does not exist an e' such that $e \equiv e'\ \tau_1 \rhd \tau_2$ for $\tau_1 \not\equiv \tau_2$.

Proof
Assume for contradiction that such an e' did exist. Then by inversion on the Eq rule, we would have that $\epsilon \models \tau_1 \sim \tau_2$ and therefore $\tau_1 \equiv \tau_2$. But τ_1 was assumed to not be equal to τ_2 which is a contradiction. Therefore, no such e' can exist. □

Progress and preservation. Proving progress and preservation for our language is largely straightforward, however care must be taken with constraints, and in particular how we introduce them in the type system, and handle them in the semantics. The following lemma serves as a crucial link between the type-level constraint environment and the type-erasure that we perform in our semantics.

[12] Our program contexts are standard and elided for brevity.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

Lemma 2 (*Constraint substitution*)
Let C be a normalized constraint and $C \wedge \tau_1 \sim \tau_2, \cdot \vdash_e e : \tau$. Then, $C[\tau_1'/\tau_2'], \cdot \vdash_e e[\![\tau_1'/\tau_2']\!] : \tau[\tau_1'/\tau_2']$. Where we choose the τ_i' according to the following rules:

$$\begin{array}{ll} \tau_2 \in \tau, \tau_1 \notin \tau & \Rightarrow \tau_1' = \tau_2, \tau_2' = \tau_1 \\ \text{otherwise} & \Rightarrow \tau_1' = \tau_1, \tau_2' = \tau_2 \end{array}$$

Proof
The proof follows by induction on the structure of e and by inversion of the typing rules. The only interesting case arises from the explicit type coercions and the EQ rule—in particular in showing that

$$C \wedge \tau_1 \sim \tau_2 \models \tau' \sim \tau \implies C[\tau_1'/\tau_2'] \models \tau'[\tau_1'/\tau_2'] \sim \tau[\tau_1'/\tau_2']$$

This can be shown by proving a more general substitution lemma on constraints:

$$\frac{C \models \tau' \sim \tau}{C[\tau_1'/\tau_2'] \models \tau'[\tau_1'/\tau_2'] \sim \tau[\tau_1'/\tau_2']}$$

which follows by induction on the proof derivation. We then note that $(C \wedge \tau_1 \sim \tau_2)[\tau_1'/\tau_2'] = C[\tau_1'/\tau_2'] \wedge \tau_1' \sim \tau_1'$ and since $\tau_1' \sim \tau_1'$ holds by reflexivity, we can then eliminate this constraint from our environment. □

The fact that the constraint substitution in Lemma 2 does not necessarily lead to the same type (syntactically) presents a challenge to proving type safety, but it makes sense: a constraint $\tau_1 \sim \tau_2$ in C represents the *ability* to coerce a value of type τ_1 to a value of type τ_2 (and vice versa), however a substitution $[\tau_1/\tau_2]$ represents the *obligation* to change all types τ_2 to τ_1. Furthermore, Lemma 2 can be seen as a way to transform propositional into syntactic equality, so it makes sense that while it should preserve propositional equality between τ and $\tau[\tau_1/\tau_2]$ it *does not* necessarily preserve syntactic equality between these two types.

Since Lemma 2 is central to our proof of preservation, this lack of syntactic equality presents an issue to the normal formulation of preservation. We will therefore need to change the statement slightly to take into account that even though the types may change syntactically during reduction, they do not change semantically. This leads to the following definition of a propositional equality between types where we say that once we have proved two types to be equal at runtime then we can syntactically change our types to get rid of one of the (now known to be equal) types.

Definition 2
We say that $(e, \tau) \approx (e', \tau')$ if $e \hookrightarrow e'$ results in type-erasures $[\![\tau_1/\tau_1']\!], \ldots$, and $\tau_1 \sim \tau_1' \wedge \ldots \models \tau \sim \tau'$. If the reduction does not result in an erasure, then $\tau \equiv \tau'$.

Theorem 4 (*Preservation*)
Suppose that $\epsilon, \cdot \vdash_e e : \tau$ and that $\Sigma; D \vdash e \hookrightarrow e'$. Then, $\epsilon, \cdot \vdash_e e' : \tau'$ and $(e, \tau) \approx (e', \tau')$, where Σ is built-up as in the premises of the PROG rule.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

Proof
The majority of the proof is straightforward, and based on induction over the derivation of $\epsilon, \cdot \vdash_e e : \tau$ and a standard preservation proof. The only interesting part is in the handling of constraints. In particular, in showing that whenever type constraints are introduced in our typing rules, these correspond to type-erasures in the semantics. This correspondence is shown through a straightforward (and tedious) case analysis on our pattern matching and type equality testing rules in both the type system and semantics and in each case showing that the type-erasures that are performed in the semantics match with the constraints introduced in the typing rules, and then using Lemma 2 repeatedly on the normalized constraint (Definition 1, Theorem 1) to show that the resulting types are $\approx$ to each other after each type-erasure. □

We will need the following (standard) lemma for the proof of progress.

Lemma 3 (*Canonical forms*)

1. If v is a value of type $T\ \overline{\tau}$, then v is a data constructor for T (i.e., $K\ \overline{e}$).
2. If v is a value of type $\tau_1 \to \tau_2$, then v is a lambda expression.

Proof
By inspection of the definition of values in Figure 4, and the typing rules in Figure 7. □

Now that we have the Canonical Forms lemma, we have all the tools we will need in order to prove progress and preservation for our language.

Theorem 5 (*Progress*)
Suppose that $\epsilon, \cdot \vdash_e e : \tau$. Then, if e is not a value, then there exists an e' such that $\Sigma; D \vdash e \hookrightarrow e'$. Where Σ is built-up as in the premises for the PROG rule.

Proof
Straightforward induction on the derivation of $\epsilon, \cdot \vdash_e e : \tau$, using Theorem 3 to ensure that we do not encounter a type coercion when we perform the reduction step, and by use of Lemma 3. □

5 Pre-conditions and ambiguity checking

Before Ghostbuster can generate up- and down-conversion functions, it first performs a sanity check that the datatypes, together with requested parameter erasures, meet all pre-conditions 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 pre-conditions such that *if* these pre-conditions are met, the Ghostbuster tool is guaranteed to generate a pair of well-typed functions $(up, down)$, such that up-conversion followed by down-conversion is a total identity function. This section details these pre-conditions and ambiguity criteria.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

5.1 Ambiguity test

While it would be possible to issue errors at the point Ghostbuster is generating conversion functions (i.e., in a later pass of the "compiler"), our goal in the ambiguity criteria are a concise specification of the class of programs handled by Ghostbuster. These non-ambiguity pre-requisites apply per-data-constructor, $\mathtt{K}_i$, and for each datatype that requests a type-erasure (non-empty $\overline{c}$ or $\overline{s}$ variables). If all of the constructors for a datatype that is marked for erasure each individually pass the ambiguity check, then the datatype is marked as valid. And if all of the datatypes that have been marked for erasure individually pass the ambiguity check, then the program as a whole is valid. We will also need some terminology for our data constructors as we go forward in order to avoid ambiguity in our ambiguity criteria. Thus, given a data constructor:

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

we refer to the types τ_1 through τ_p as the *fields* $\overline{\tau}^p$ of the constructor, and the $T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}$ expression as the 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 primary pre-requisite-checks for Ghostbuster are for verifying computability of checked and synthesized type variables, and the ambiguity check is concerned with the *information flow* between the type variables in kept, checked, and synthesized contexts. That is, whether the type information erased from the simpler up-converted datatype can be recovered during down-conversion based on the properties and type information of the simpler datatype. If not, these type variables would not be recoverable upon down-conversion—and since we are only concerned with datatypes where we can generate both up- and down-conversion functions for datatypes Ghostbuster rejects the program.

5.2 Type variables synthesized on the RHS

In order to synthesize the types $\overline{\tau_s}$, we require that 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 the following:

1. 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 12.
2. $a \in Fv[\![\overline{\tau_k}]\!]$. That is, kept RHS types.
3. $a \in Fv[\![\overline{\tau_c}]\!]$. That is, a occurs in the *checked* (input) type.

Note that the occurrences of a in fields of the constructor can be in either kept or 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 down-conversion function as *inputs* to recursive

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

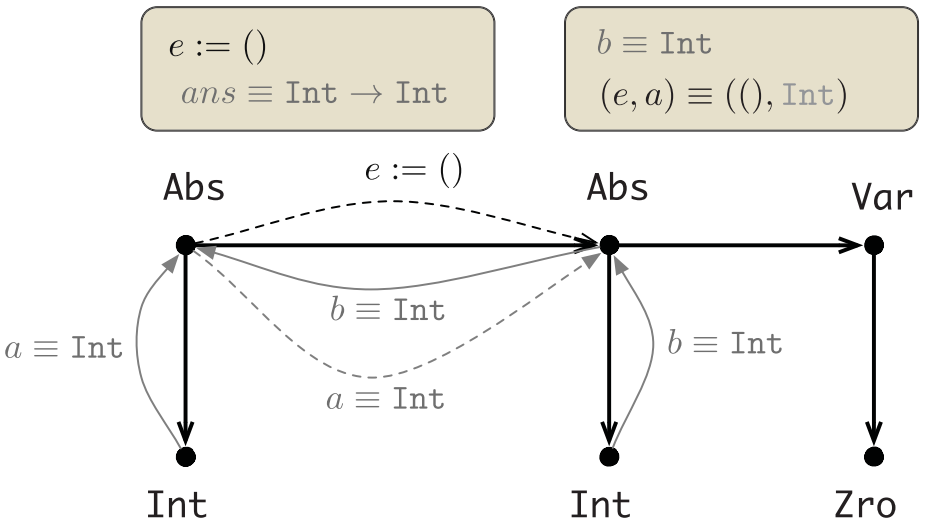

Fig. 11. Information flow for type variables in checked and synthesized contexts for the program `Abs Int (Abs Int (Var Zro))`. Vertical edges are processed before horizontal edges.

down-conversion calls on the value's fields. Thus, in (1), they cannot be a source of new information to determine synthesized outputs, and we have to use the $Fv_s[\![]\!]$ rather than the $Fv[\![]\!]$ metafunction above in order to determine the types available to us during the synthesization process. Conversely, notice that we do not worry about applying the above prerequisites to synthesized variables inside fields—these are the *outputs* of recursive down-conversion calls. Their computability is left to an inductive argument (bottoming out at "leaf" constructors such as `Exp`'s `Con`). An example of valid type-information flow for a small program in the language of Section 3.1 is given in Figure 11 where the grey arrows represent synthesized type-information being returned back out by the recursion, the black dashed arrows represent checked type-information being pushed down into the recursive calls on the datatype, and the grey dashed arrows represent synthesized type information that has been discovered at a previous step being pushed down into the recursive calls on the datatype.

5.3 Type variables in checked context

All types in checked context in $\tau_1 \ldots \tau_p$ are implicit arguments to the down-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

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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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 down-conversion function), it must be determinable from the other fields of the constructor, in this case `Typ a`. An example of what this information flow for inter-field type dependencies looks like is given by the grey dashed line in Figure 11.

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 possible inter-field dependencies do not form a cycle. As an example, take the following piece of code:

```
{-# 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
```

Since the types `a` and `b` in the fields of the constructor *both* appear in checked mode, determining the type of `a` could depend on the determination of the type of `b` and vice versa. Thus, the inter-field dependency graph between `a` and `b` could form a cycle—so this constructor fails the ambiguity criteria and we cannot erase the type `t` from `Loop`.

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 $\tau_1 =$ `Typ a` occurs before $\tau_2 =$ `Exp (e,a) b`, therefore Ghostbuster accepts the definition.

Discussion: design choice. Finally, note that we could seek to loosen the inter-field dependency restriction to allow *intra*-field dependencies. For example, currently an uncurried version of the `Abs` constructor would be rejected by Ghostbuster:

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

Here, `a` in `(e,a)` must be determined by a synthesized portion of the *same* field's type, τ_1. In this particular case, we know that tuple values of type `(x,y)` can be broken into a value of type `x` and `y`, so we can recursively process one part of the tuple *before* the other. However, for arbitrary type constructors, this property does not hold: synthesization of type variables can be viewed as a type of effect and the order in which we synthesize type variables is important, so unless we know that a given type constructor has a `Traversable` instance (or some other canonical

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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

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

Fig. 12. Extracting free type variables in different contexts.

traversal ordering is imparted to it), we are unable to determine where we should start resolving intra-field dependencies. Furthermore, as discussed in Section 2.4, this would require us to also trust in the `Functor` instance provided to us for the type constructor, which in and of itself presents many challenging issues. For these reasons, we keep the allowed dependencies simple (inter-field), and types must be re-factored to meet this requirement.

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 6 (*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 3.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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 $T'\ \overline{\tau_k}$:

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

where *getTyReps* returns any newly existential variables for a constructor (Section 2.2):

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

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

6.2 Up-conversion generation

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

```
upT_i :: TypeRep c̄ → TypeRep s̄ → T_i k̄ c̄ s̄ → T_i' k̄
upT_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))
```

While the procedure is largely straightforward—pattern match on each $\mathtt{K}_j$ and apply the $\mathtt{K}_j'$ constructor—there is significant complexity in the type representation management of the *bind* and *dispatch*$_\uparrow$ 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 5–8. 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)
```

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

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 $dispatch_{\uparrow}$ function is charged with recursively processing each field `f` of K_j. Based on the type of `f`, this will take one of two actions:

- Opaque object: return it unmodified.
- Ghostbusted type T: call $\mathtt{up}T$.

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`.

Both of these behaviors can be seen in the snippet of actual Ghostbuster-generated code below:

```
upExp :: ∀ env ans . TypeRep env → TypeRep ans
        → Exp env ans → Exp'
upExp env_typerep ans_typerep orig
  = case orig of
      Abs a b → Abs'
      (upTyp
        (let r_typerep = typecase ans_typerep of
                           (typerep ArrowTy) left right → left
         in r_typerep)
        a)
      (upExp
        (let e_typerep = env_typerep in
         let r_typerep = typecase ans_typerep of
                           (typerep ArrowTy) left right → left
         in (typerep Tup) e_typerep r_typerep)
         ... )
```

Finally, when building type representations inside the $dispatch_{\uparrow}$ 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
```

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

In recursive calls to `upExp`, 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 $\forall$ `a. TypeRep a`.

Lemma 4 (*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 pre-existing existential variable, whereas only newly existential variables are returned by $getTyReps$. □

6.3 Down-conversion generation

Down-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 a down-converter takes type representation arguments only for checked type variables; synthesized types must be computed:

$$\texttt{down}T_i\ ::\ \overrightarrow{\texttt{TypeRep}\ c}\ \rightarrow\ T_i'\ \overline{k}\ \rightarrow\ \texttt{Sealed}T_i\ \overline{k}\ \overline{c}$$

where the `Sealed`T_i seals over any newly existential type variables for T_i that may be introduced during the down-conversion process just as `SealedVec` did for `Vec` in Section 1.

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 `up` function, `down` 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 up-conversion case above.

```
downT_i :: TypeRep c ... -> T_i' k ... -> SealedT_i k ... c ...
downT_i c1_typerep ... cm_typerep lower =
  case lower of
    K_j' ex_typerep ...  f_1 ... f_p ->
       let φ_0 = ... in
       openConstraints(φ_0, openFields(f_1...f_p))
 where
 openConstraints(∅, bod) = bod

 openConstraints(a := b : φ, bod) =
    if a_typerep ≃_τ b_typerep
    then openConstraints(φ, bod)
```

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

```
    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
```

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 $dispatch_\uparrow$, 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 so 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)
```

Fresh variables and naming conventions. Naming conventions are subtle when implementing the above code generation algorithm. By processing from left to

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

right, we ensure that earlier synthesized dictionaries are available for creating later checked dictionaries to pass to recursive calls. However, in the above presentation, we did not keep an explicit naming environment. This works because the structure of the types is static and available at all points in the code—it is possible to construct a unique name for the values and dictionaries returned by each recursive call, and to agree on this by convention. Alternatively, we could also pass a Γ as an argument to *openFields* which would keep track of all available term variables with dictionary type.

The result of code generation is that Ghostbuster has augmented the *prog* with up- and down-conversion functions in the language of Figure 4, 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 Runtime type representations

Before we can get rid of the `typecase` and $\simeq_\tau$ constructs in Ghostbuster's generated code, we must first choose an approach to dynamic type checks. Since we are generating Haskell code, one method is to use Haskell's type-indexed `Typeable` class introduced in GHC-8.2, 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 (Abadi *et al.*, 1989; Leroy & Mauny, 1991; Abadi *et al.*, 1995; Baars & Swierstra, 2002a).

6.4.1 Runtime Types in Ghostbuster

Since generating code against the new `Typeable` class in Haskell restricts the portability of the generated code,[13] we instead use the simple approach of generating a closed-world of type-indexed `TypeRep` values for all types mentioned in the datatypes passed to Ghostbuster: since the Ghostbuster tool can observe all the types mentioned in a set of data-types (Theorem 2) it creates an application-specific notion of a runtime type representation, which itself is a GADT. Since for a *closed* set of types, creating a GADT for runtime type representation is trivial (Peyton Jones *et al.*, 2016)—For example, the following is the `TypeRep` for representing Boolean, integer, and tuple types:

```
data TypeRep a where
  TypeInt  :: TypeRep Int
  TypeBool :: TypeRep Bool
  TypeTup2 :: TypeRep a → TypeRep b → TypeRep (a,b)
```

What is more, using an explicit dictionary type makes it trivial to construct a type equality check function of type `TypeRep a → TypeRep b → Either TypeError (a :∼: b)`

[13] Not just to other languages—but also to Haskell before the new `Data.Typeable` introduced in GHC-8.2.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

6.4.2 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 in our algorithm, and provides a simple solution to enabling an open union of dictionary types without using typeclasses or any more complex mechanisms in the formal language to achieve this. Now that we have chosen a representation for our types, we can describe how to de-sugar 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 type constructor T mentioned anywhere in a `typerep` or `typecase` form, plus the built-in types:

$$
\begin{aligned}
&\texttt{data TypeRep a where}\\
&\quad \texttt{Type}_{T_1} \texttt{ :: } \overrightarrow{\texttt{TypeRep } a}^{n_1} \texttt{ } \rightarrow \texttt{ TypeRep } (T_1\ \overline{a}^{n_1})\\
&\quad \texttt{Type}_{T_2} \texttt{ :: } \overrightarrow{\texttt{TypeRep } a}^{n_2} \texttt{ } \rightarrow \texttt{ TypeRep } (T_2\ \overline{a}^{n_2})\\
&\quad \ldots\\
&\quad \texttt{ArrowType :: TypeRep a} \rightarrow \texttt{TypeRep b} \rightarrow \texttt{TypeRep (a} \rightarrow \texttt{b)}\\
&\quad \texttt{ExistentialType :: } \forall \texttt{ a . TypeRep a}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathscr{D}[\![\texttt{typerep } T]\!] && \Longrightarrow \texttt{TypeRep}_T\\
&\mathscr{D}[\![\texttt{typecase e}_1 \texttt{ of } ((\texttt{typerep } T)\ \texttt{a}_1 \ldots \texttt{a}_n) \rightarrow \texttt{e}_2 ; \_ \rightarrow \texttt{e}_3]\!] && \Longrightarrow\\
&\quad \texttt{case } \mathscr{D}[\![\texttt{e}_1]\!] \texttt{ of}\\
&\qquad \texttt{Type}_T\ \texttt{a}_1 \quad \ldots \texttt{a}_n \rightarrow \mathscr{D}[\![\texttt{e}_2]\!]\\
&\qquad \texttt{Type}_{T_1}\ \_\_ \quad \ldots \quad \rightarrow \mathscr{D}[\![\texttt{e}_3]\!]\\
&\qquad\qquad \vdots
\end{aligned}
$$

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 non-trivial code, it is also important to `let`-bind it to avoid excessive code duplication, which slightly complicates the translation above.

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

$$
\begin{aligned}
&\mathscr{D}[\![\texttt{if e}_1 \simeq_\tau \texttt{e}_2 \texttt{ then e}_3 \texttt{ else e}_4]\!] \Longrightarrow\\
&\quad \texttt{case eqTT } \mathscr{D}[\![\texttt{e}_1]\!]\ \mathscr{D}[\![\texttt{e}_2]\!] \texttt{ of}\\
&\qquad \texttt{Just Refl} \rightarrow \mathscr{D}[\![\texttt{e}_3]\!]\\
&\qquad \texttt{Nothing} \quad \rightarrow \mathscr{D}[\![\texttt{e}_4]\!]
\end{aligned}
$$

This `eqTT` value definition 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:

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

```
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.

6.5 Validating Ghostbuster

We are now ready to state the main theorem about Ghostbuster: if all the datatypes in a program pass our ambiguity criteria, then up-conversion followed by down-conversion is the identity after unsealing synthesized type variables.

Theorem 7 (***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 $\texttt{prog}'$ with converted datatypes $\mathbf{T}' = \{(T'_1, k_1), \ldots, (T'_n, k_n)\}$, and functions $\texttt{up}T_i$ and $\texttt{down}T_i$ such that

$$\begin{aligned} &\forall e \in \texttt{prog}.\ \texttt{prog} \vdash e :: T_i\ k_i\ c_i\ s_i \wedge (T_i, k_i, c_i, s_i) \in \mathbf{T} \\ &\implies \texttt{prog}' \vdash (\texttt{up}T_i\ D_i\ e) :: T'_i\ k_i, \text{ where } (T'_i, k_i) \in \mathbf{T}' \end{aligned} \tag{3}$$

and

$$\begin{aligned} &\forall e \in \texttt{prog}.\ \texttt{prog} \vdash e :: T_i\ k_i\ c_i\ s_i \wedge (T_i, k_i, c_i, s_i) \in \mathbf{T} \\ &\implies \texttt{prog}' \vdash (\texttt{down}T_i\ D_i c\ (\texttt{up}T_i\ D_i\ e)) \\ &\equiv (\texttt{Sealed}T_i\ D_i s\ e :: \texttt{Sealed}T_i\ k_i\ c_i) \end{aligned} \tag{4}$$

The full proof, while being fairly lengthy and tedious—is not terribly interesting or enlightening. We thus provide a proof-sketch here.

Proof Sketch
We first show by the definition of up-conversion that given any data constructor K of the correct type, that the constructor will be matched. Proceeding by induction on the type of the data constructor and case analysis on *bind* and $dispatch_{\uparrow}$, 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. After showing that every individual type-field is up-converted successfully and that this up-conversion preserves values, we are then able to conclude that since we have managed to construct the correct type representations needed for the up-converted

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

data constructor K', and since we can successfully up-convert each field of K, that the application of K' to the typeReps for the newly existential types and the up-converted fields is well-typed and that the values that we wish to have preserved have been kept.

To show that down-conversion succeeds, we first show that given any data constructor K' of the correct type that the down-conversion function will match it. We then proceed by case analysis on the code-path executed on the RHS 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 down-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 down-converting the busted types that it encounters. We then use the fact that *openFields* has successfully down-converted the types it has encountered, coupled with the success of constructing suitable type representations to show that we are finally able to successfully construct the down-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 easily 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,[14] 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 typeclasses such as `Show`.

There is one important impedance mismatch between our core language's (more permissive) type system and Haskell's. In particular, we allow locally conflicting constraints in case statements, like Typed Racket (Tobin-Hochstadt & Felleisen, 2010b), but unlike GHC Haskell.[15] In these cases, GHC issues "inaccessible code" errors, which we would prefer could be turned into configurable warnings.

Of course, because these branches *are* inaccessible, they cannot cause a problem at runtime. Unless GHC makes a change, our recourse is to (1) predict which branches GHC will object to and omit those in code generation or (2) turn on deferred type errors locally for the generated conversion functions which have this problem. We currently do the former—avoiding the issue for Ghostbuster-generated conversion functions.

[14] `http://hackage.haskell.org/package/haskell-src-exts`
[15] See the *consistent* requirement in Section 3.2 of Schrijvers *et al.* (2009b).

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

7.1 Pre-processing options

There are several potential ways to connect the tool to a build environment, as well as several design decisions that we must address in constructing simplified types. As in the code snippets we have seen, the user of Ghostbuster writes the original type by hand, and uses a separate specification (pragma) to indicate which type variables should be erased. One option would be to generate the Ghostbusted code implicitly, e.g., by macro expansion,[16] but our intent is for the user to read the generated code and write functions consuming values of that type. Thus, we run Ghostbuster as a pre-processor that generates pretty-printed Haskell code in a stand-alone file.[17]

7.2 Current limitations and possible extensions

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-indexed `TypeRep` values, which just appeared in GHC-8.2. However, in order to make the theory and tool more easily generalizeable to other languages without this feature, we use our own (closed-world) representation of runtime types synthesized on demand by the Ghostbuster tool and described in Section 6.4.2.

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 (Chakravarty *et al*., 2005; Schrijvers *et al*., 2008) and typeclasses (Peterson & Jones, 1993; Hall *et al*., 1996).

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 behavior. Future work may allow a user to specify how Ghostbuster should traverse under type constructors to continue the erasure and conversion processes.

Typeclass constraints. As we saw in Section 4.1, we do not handle typeclass constraints in the datatypes that are passed to or generated by Ghostbuster. While we have decided against implementing this feature for the Haskell version of the

[16] For example, we could use Template Haskell (Sheard & Peyton Jones, 2002) with a top-level `$(ghostbuster ...)` splice, which inserts the generated code and conversion function declarations. This approach would be sufficient, but it suffers from a drawback. While it is possible to *dump* Template Haskell splices during compilation, this is not an ideal solution for examining the generated code.

[17] Both GHC and the build tool `cabal` have good support for invoking custom pre-processors.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

tool, there is nothing that prevents this. However, doing so in a formal and well-founded manner would complicate the formal language, type system, and operational semantics considerably, and would require updating the ambiguity criteria. Further, doing so would break the source-to-source nature of our tool since the generated code would need to access internal features of GHC (and Core) in order to prove typeclass constraints in the generated code.

Ghostbuster for other languages. As long as a given source/target language is parsable into our core language, updating the tool to handle other languages simply involves changing the parsing and code generation phases, and turning off the various Haskell-specific features that we have added (e.g., bang patterns). However, this leads to questions on how to handle other features that these languages have that can interact with GADTs, e.g., how should polymorphic variants be handled in OCaml when they appear as (or interact with) to-be-erased type indicies? Handling these language-specific features would present similar implementation challenges to those that would be faced in implementing typeclass constraints for Haskell, and could—depending on the feature—require non-trivial additions to both the ambiguity criteria, core language, and algorithm.

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 a 4-core Intel *i*7-4850HQ CPU (64-bit, 2.3 GHz, 16 GB RAM) running Mac OSX 10.12 and using GHC version 8.0.2 at `-O2` optimization level. Each data point is generated via linear regression using the `criterion` package.[18]

Figure 13 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 up- and down-conversion (`Abs`, `App`, etc.), and report the time to convert programs containing that number of terms.

Ghostbuster achieves comparable performance to a manually written up-conversion routine. The hand-written down-conversion routine, however, which uses embedded `Typeable` class constraints and is based on runtime type checks provided by the `Data.Typeable` library, 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 if we use the new open-world, type-indexed `Typeable` in GHC-8.2.

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

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

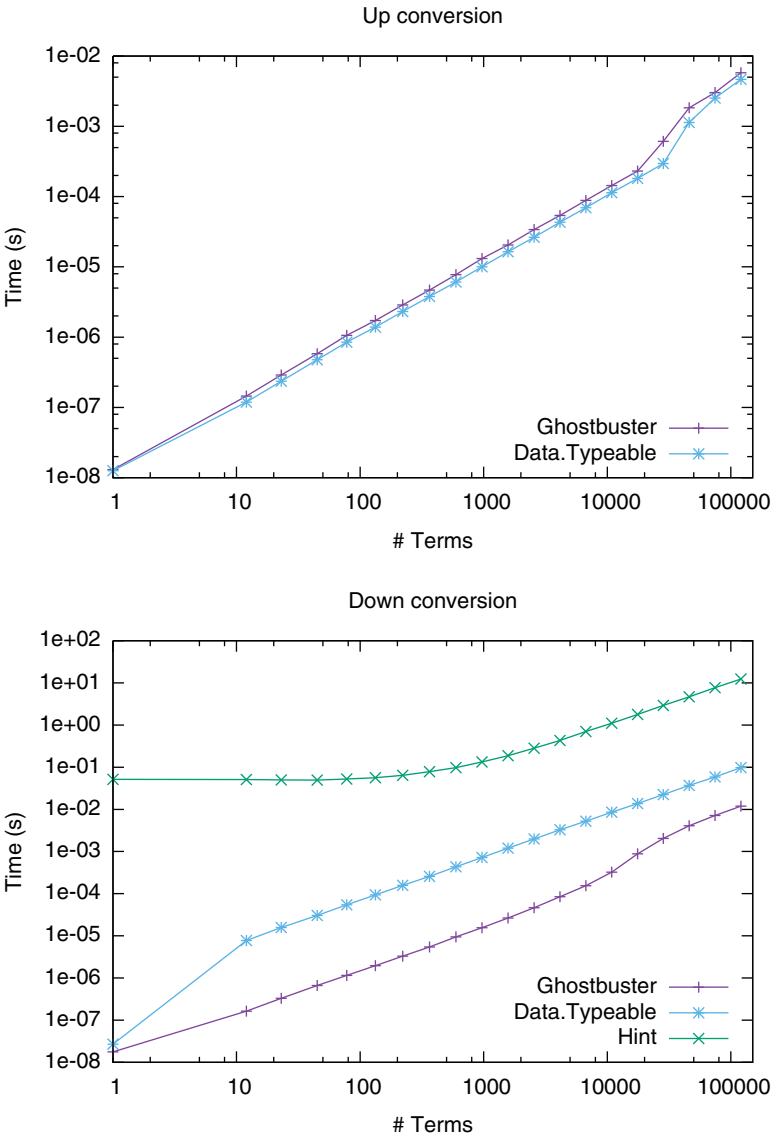

Fig. 13. Time to convert a program in our richly typed expression language (Section 3.1) with the given number of terms (i.e., nodes in the AST), from original GADT to simplified ADT (left) and vice-versa (right). Note the log-log scale.

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	1,426	1 MB
Data.Typeable	122	1,011	1 MB
Hint	78	451	45 MB

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

For the down-conversion process, we also compare against using GHC's interpreter as a library via the `Hint` package.[19] Due to the difficulty of writing the down-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 13, 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 took this approach is Hakaru.[20]

8.2 Package survey

We conclude our experimental evaluation by testing our prototype implementation against 9,026 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 datatypes and/or type variables, this can yield a huge search space, so we explore at most 10,000 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 8,773 "real" GADTs surveyed,[21] we were able to successfully ghostbust 5,525 (63%) of these down to regular ADTs.

9 Related work

Ornaments (McBride, 2010; Dagand & McBride, 2012; Ko & Gibbons, 2013), from the world of dependent type theory, provides an interesting theoretical substrate for

[19] `http://hackage.haskell.org/package/hint`
[20] `https://hackage.haskell.org/package/hakaru`
[21] Some types were written in GADT syntax that did not need to be.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

Table 1. *Summary of package survey*

Metric	
Total # packages	9,026
Total # source files	94,611
Total # SLOC	16,183,864
Total # datatypes using ADT syntax	9,261
Total # datatypes using GADT syntax	18,004
Total # connected components	15,409
ADTs with type variable(s)	1,341
GADTs with type variable(s)	11,213
GADTs with type indexed variable(s)	8,773
Actual search space	185,056,322,576,712
Explored search space	9,589,356
Ghostbuster succeeded	2,582,572
GADTs turned into ADTs	5,525
Ambiguity check failure	5,374,628
Unimplemented feature in Ghostbuster	1,632,156

moving between inductive data structures that share the same recursive structure, 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 (Williams *et al*., 2014). 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` in Abadi *et al.* (1989; 1995). 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* (Shields *et al*., 1998), 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 down-conversion. Likewise, Haskell's *deferred type errors* (Vytiniotis *et al*., 2012) 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.[22] 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).

[22] 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/`.

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

F# type providers (Syme *et al.*, 2013) 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 modes 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 (Somogyi *et al.*, 1995) has modes: input, output, deterministic. Our ambiguity checking process is similar.

The rules in our ambiguity criteria for types in checked and synthesized mode are very similar to those for synthesized and inherited attributes in attribute grammars (Knuth, 1968): information for synthesized types must be determinable from the children of that type just as synthesized attributes for a production are determined from the attributes of its children; and checked types receive information from their parents just as inherited attributes for a production are determined from the attributes of its parents. Moreover, the ambiguity criteria—and in particular, the restriction that checked types can only gather synthesized type information from their left-hand-side is similar to what one would see in an *L*-attribute grammar. However, while our ambiguity criteria are very similar to that in an *L*-attribute grammar, we differ slightly due to the presence of kept type variables. In particular, we allow non-erased type variables *anywhere* to be used in the determination of checked typed variables—and in this sense our checked types do not fully align with inherited attributes.

Ou *et al.* (2004) 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 (Casinghino *et al.*, 2014) 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.

Recent work by Dagand *et al.* (2016) defines a system based on partial type equivalences and runtime checks that provides interoperability between simply and dependently typed regions of code in a similar manner to us. However, while they are interested in partial type equivalences in general (and user-specified equivalences in particular) and how this can be used to allow cross-world usage—by lifting and lowering functions over the more- and less-specified datatypes—we are interested in a very specific partial equivalence between the more and less specified datatypes that permits us to round-trip them.

Gradual typing is an approach to integrating static and dynamic typing within a single language (Siek & Taha, 2006). The gradual typing approach is characterized by implicit conversions in the source language, while our work makes use of explicit conversions. Our work is therefore more closely related to calculi with explicit conversions (Abadi *et al.*, 1989; Henglein, 1994), including blame calculi (Findler & Felleisen, 2002; Tobin-Hochstadt & Felleisen, 2006; Wadler & Findler, 2009).

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

However, our work involves inhabiting expressions at a more or less detailed version of the same datatype, rather than integrating dynamically typed code. Blame calculi allocate blame to the origin of the conversion error in the source language, which is essential when running code in which implicit conversions have been compiled into explicit conversions. Due to the coarse-grained nature of our usage scenario implicit casting is not needed, and while blame tracking would be a nice feature we see this as only complicating the theory with little real-world benefit.

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., down-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 Conclusions and future work

We have 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.

Future work. While the theory presented here can handle a number of different GADT features, the class of GADTs that we can handle is still restricted. An interesting avenue of future work is to not treat GADTs as a primitive in the language, and instead explore re-phrasing the theory and implementation presented here in terms of existential types, types witnesses, and Guarded Recursive Datatypes (Weirich, 2000; Baars & Swierstra, 2002b; Cheney & Hinze, 2002; Sulzmann & Wang, 2004), and then see if, and how, this might change the class of datatypes that can be handled.

Another interesting aspect to be explored is how the conversion process interacts with GADTs (and ADTs) that use linear types (Bernardy *et al.*, 2018). In particular, we feel as though it should be possible to show that while the down-conversion process might use type-level information in a non-linear way, the actual converted datatypes retain the original linearity properties.

At the moment, we do not support multiple erasure variants and furthermore, we do not support transforming one erasure variant into the other. It would be interesting to see whether the round-trip property could be shown for a *set* of erasure variants, ensuring that given an up-converted datatype of any variance, we could transform from one erasure datatype variant to another while still allowing down-conversion to happen at any point. This would be interesting, however we feel that it may be quite tricky and difficult to show the round-trip property for a set of erasures; since any up-converted datatype for a given erasure variant would have to

https://doi.org/10.1017/S0956796818000114 Published online by Cambridge University Press

retain enough information for all the other erasure variants and their corresponding down-conversion functions to succeed.[23]

Acknowledgments

This work has benefited greatly from several conversations with Chung-chieh Shan and Jeremy Siek. We would also like to thank the anonymous reviewers of ICFP 2016 and this journal for their helpful feedback on this paper.

References

Abadi, M., Cardelli, L., Pierce, B., & Rémy, D. (1995) Dynamic typing in polymorphic languages. *J. Funct. Program.* **5**, 111–130.

Abadi, M., Cardelli, L., Pierce, B. & Plotkin, G. (1989) Dynamic typing in a statically-typed language. *ACM Trans. Program. Lang. Syst.* **13**(2), 237–268.

Altenkirch, T. & Reus, B. (1999) Monadic presentation of lambda terms using generalised inductive types. In *Computer Science Logic,* Flum, J. & Rodriguez-Artalejo, M. (eds). Berlin, Heidelberg: Springer, pp. 453–468.

Appel, A. W. (2007) *Compiling with Continuations.* New York, NY, USA: Cambridge University Press.

Appel, A. W. & Jim, T. (1997) Shrinking lambda expressions in linear time. *J. Funct. Program.* **7**(5), 515–540.

Baars, A. I. & Swierstra, S. D. (2002a) Typing dynamic typing. In ICFP02: International Conference on Functional Programming, pp. 157–166.

Baars, A. I. & Swierstra, S. D. (2002b) Typing dynamic typing. In Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ICFP '02. New York, NY, USA: ACM, pp. 157–166.

Benton, N., Kennedy, A., Lindley, S. & Russo, C. (2005) *Shrinking Reductions in sml.net*. Berlin, Heidelberg: Springer, pp. 142–159.

Bernardy, J.-P., Boespflug, M., Newton, R. R., Peyton Jones, S. & Spiwack, A. (2018) Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* **2** (POPL), 5:1–5:29

Brady, E., McBride, C. & McKinna, J. (2004) Inductive families need not store their indices. In *Types for Proofs and Programs*, Stefano, B., Mario, C. & Damiani, F. (eds). Berlin, Heidelberg: Springer, pp. 115–129.

Casinghino, C., Sjöberg, V. & Weirich, S. (2014) Combining proofs and programs in a dependently typed language. In POPL'14: Principles of Programming Languages, pp. 33–45.

Chakravarty, M. M. T., Keller, G., Lee, S., McDonell, T. L. & Grover, V. (2011) Accelerating Haskell array codes with multicore GPUs. In DAMP'11: Declarative Aspects of Multicore Programming, pp. 3–14.

Chakravarty, M. M. T., Keller, G. & Peyton Jones, S. (2005) Associated type synonyms. In POPL'05: Principles of Programming Languages, pp. 241–253.

Cheney, J. & Hinze, R. (2002) A lightweight implementation of generics and dynamics. In Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell'02. New York, NY, USA: ACM, pp. 90–104.

[23] Although one could possibly prove this by first down-converting and then up-converting using the specific up-conversion function.

Cheney, J. & Hinze, R. (2003) *First-Class Phantom Types*. Technical Report, Cornell University.

Dagand, P.-E. & McBride, C. (2012) Transporting functions across ornaments. In ICFP'12: International Conference on Functional Programming, pp. 103–114.

Dagand, P.-E., Tabareau, N. & Tanter, É. (2016) Partial type equivalences for verified dependent interoperability. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP'16. New York, NY, USA: ACM, pp. 298–310.

Findler, R. B. & Felleisen, M. (2002 October) Contracts for higher-order functions. In Proceedings of the International Conference on Functional Programming, ICFP, pp. 48–59.

Hall, C. V., Hammond, K., Peyton Jones, S. & Wadler, P. L. (1996) Type classes in Haskell. *ACM Trans. Program. Lang. Syst.* **18**(2), 109–138.

Henglein, F. (1994) Dynamic typing: Syntax and proof theory. *Sci. Comput. Program.* **22**(3), 197–230.

Knuth, D. E. (1968) Semantics of context-free languages. *Math. Syst. Theory* **2**(2), 127–145.

Ko, H.-S. & Gibbons, J. (2013) Relational algebraic ornaments. In DTP'13: Dependently-Typed Programming, pp. 37–48.

Leroy, X. & Mauny, M. (1991) Dynamics in ML. In Functional Programming Languages and Computer Architecture, pp. 406–426.

McBride, C. (2005) Type-preserving renaming and substitution [online]. Accessed March 30, 2018. Available at: http://strictlypositive.org/ren-sub.pdf

McBride, C. (2010) Ornamental algebras, algebraic ornaments [online]. Accessed March 30, 2018. Available at: https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/Ornament.pdf

McDonell, T. L., Chakravarty, M. M. T., Keller, G. & Lippmeier, B. (2013) Optimising purely functional GPU programs. In ICFP'13: International Conference on Functional Programming, pp. 49–60.

McDonell, T. L., Chakravarty, M. M. T., Grover, V. & Newton, R. R. (2015) Type-safe runtime code generation: Accelerate to LLVM. In Proceedings of the Haskell Symposium, pp. 201–212.

McDonell, T. L., Zakian, T. A. K., Cimini, M. & Newton, R. R. (2016) Ghostbuster: A tool for simplifying and converting GADTs. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP'16. New York, NY, USA: ACM, pp. 338–350.

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

Peterson, J. & Jones, M. (1993 June) Implementing type classes. In PLDI'93: Programming Language Design and Implementation, pp. 227–236.

Peyton Jones, S., Weirich, S., Eisenberg, R. A. & Vytiniotis, D. (2016) *A Reflection on Types*. Cham: Springer International Publishing, pp. 292–317.

Schrijvers, T., Peyton Jones, S., Chakravarty, M. M. T. & Sulzmann, M. (2008) Type checking with open type functions. In ICFP'08: International Conference on Functional Programming, pp. 51–62.

Schrijvers, T., Peyton Jones, S., Sulzmann, M. & Vytiniotis, D. (2009a) Complete and decidable type inference for GADTs. In ICFP'09: International Conference on Functional Programming, pp. 341–352.

Schrijvers, T., Peyton Jones, S., Sulzmann, M. & Vytiniotis, D. (2009b) Complete and decidable type inference for GADTs. In Proceedings of the 14th ACM SIGPLAN International

Conference on Functional Programming, ICFP'09. New York, NY, USA: ACM, pp. 341–352.

Sheard, T. & Peyton Jones, S. (2002) Template meta-programming for Haskell. In Proceedings of the Haskell Workshop, pp. 1–16.

Shields, M., Sheard, T. & Peyton Jones, S. (1998) Dynamic typing as staged type inference. In POPL'98: Principles of Programming Languages, pp. 289–302.

Siek, J. G. & Taha, W. (2006) Gradual typing for functional languages. In Proceedings of the Scheme and Functional Programming Workshop, vol. 6, pp. 81–92.

Simonet, V. & Pottier, F. (2007) A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.* **29**(1), 1.

Somogyi, Z., Henderson, F. J. & Conway, T. C. (1995) Mercury, an efficient purely declarative logic programming language. *Aust. Comput. Sci. Commun.* **17**, 499–512.

Sulzmann, M. & Wang, M. (2004) *A Systematic Translation of Guarded Recursive Data Types to Existential Types.* Technical Report, National University of Singapore.

Syme, D., Battocchi, K., Takeda, K., Malayeri, D. & Petricek, T. (2013) Themes in information-rich functional programming for internet-scale data sources. In DDFP'13: Data Driven Functional Programming, pp. 1–4.

Tobin-Hochstadt, S. & Felleisen, M. (2006) Interlanguage migration: From scripts to programs. In Proceedings of the Dynamic Languages Symposium.

Tobin-Hochstadt, S. & Felleisen, M. (2010a) Logical types for untyped languages. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP'10. New York, NY, USA: ACM, pp. 117–128.

Tobin-Hochstadt, S. & Felleisen, M. (2010b) Logical types for untyped languages. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP'10. New York, NY, USA: ACM, pp. 117–128.

Vytiniotis, D., Peyton Jones, S. & Magalhães, J. P. (2012) Equality proofs and deferred type errors: A compiler pearl. In ICFP'12: International Conference on Functional Programming, pp. 341–352.

Wadler, P. & Findler, R. B. (2009 March) Well-typed programs can't be blamed. In Proceedings of the European Symposium on Programming, ESOP, pp. 1–16.

Weirich, S. (2000) Type-safe cast: (functional pearl). In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ICFP'00. New York, NY, USA: ACM, pp. 58–67.

Williams, T., Dagand, P.-É. & Rémy, D. (2014) Ornaments in practice. In WGP'14: Workshop on Generic Programming, pp. 15–24.