Appendix A Runtime type representations

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

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

A.1 Data.Typeable

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

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

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

A.2 Runtime Types in Ghostbuster

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

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

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

A.3 Lowering Type Representation Primitives

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

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

```
data TypeRep a where
  Type$_{T_1}$ :: TypeRep $\overline{a}^{n_1}$ →TypeRep $T_1$
  Type$_{T_2}$ :: TypeRep $\overline{a}^{n_2}$ →TypeRep $T_2$
  ...
  ArrowType :: TypeRep a → TypeRep b
            → TypeRep (a → b)
  ExistentialType :: ∀ a . TypeRep a
```

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

```
⟦ typerep $T$ ⟧ ⇒TypeRep$_T$
```

```
⟦ typecase e$_1$ of
  ((typerep $T$) a$_1$ ... a$_n$) → e$_2$; _ → e$_3$ ⟧ ⇒
  case e$_1$ of
    Type$_{T_1}$ a$_1$ ... a$_n$ →⟦ e$_2$ ⟧
    Type$_{T_2}$ _ ...      → ⟦ e$_3$ ⟧
    ...
```

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

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

```
⟦ if e$_1$ $\simeq_\tau$ e$_2$ then e$_3$ else e$_4$ ⟧ ⇒
  case eqTT ⟦ e$_1$ ⟧ ⟦ e$_2$ ⟧ of
    Just Refl → ⟦ e$_3$ ⟧
    Nothing → ⟦ e$_4$ ⟧
```

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

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

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

Appendix B Formalism

B.1 Type System

The main judgement forms are the following:

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

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

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

B.2 GenUp/GenDown

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

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

3. Code that is *generated* is distinguished from code that is run in the generation process by the use of a grey background.

4. For a given type-constructor T, we denote the corresponding up-conversion and down-conversion functions by $\widehat{T}$ and $\widetilde{T}$ respectively.

5. To each type τ there is an associated type representation which we write as $\tau_{typerep}$. Thus in the code, when we see $a_{typerep}$ this is the type representation for the type-variable a that is in scope. Finding these type representations is not hard, but is tedious, and thus we elide the code for this and simply remark that we have a function $findPath(\psi, a)$ which, given a relational substitution ψ and a type variable a *generates code* that will access the correct substitution in ψ.

6. For a given data type definition d with type constructor T we denote the type-constructor for the sealed data type of d by $\mathcal{T}$ and overload this symbol to also represent the data constructor for the sealed data type definition.

7. We denote continuations by the Greek letter κ.

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

Appendix C Full proof for round-trip theorem

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

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

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

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

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

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

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

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

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

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

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

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

1. $\underline{\text{Case: } a_i \in \psi}$ Then we execute the following code:

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

   Then we know that the lookup will succeed by the assumption that $a_i \in \psi$. To see that the call to $buildTyRep$ will indeed return a type representation for τ' we proceed by induction on τ' and case-analysis on $buildTyRep$:

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

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

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

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

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

   We therefore have that $buildTyRep(\tau')$ will return the correct type representation for τ'. We then have that we simply create an alias $a_{i_{typerep}}$ for τ''s type representation and then return this.

2. $\underline{\text{Case: } a_i \notin \psi}$ This case happens when we may have a witness for a type representation $a_{typerep}$ buried inside some other type representation that is inside ψ e.g., $a_{typerep}$ could be found inside $((typerep\ ArrowTy)\ a_{typerep}\ b_{typerep})$ but we must "dig out" the witness for it from the larger type representation. This is what $findPath$ does. Now, moving on from this digression, in this case, we execute the following code:

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

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

We therefore have that

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

returns all needed type representations for K'.

To see that this mapping is well-typed, simply note that the *only* things that bind can generate are those things of type $(TypeRep\ a)$, and therefore it generates the types expected by K'. What's more, due to the canonicity of ordering for $getTyReps$ (and since `map` doesn't permute its arguments), we have that the type representations expected by K' will be in the same order as the ones provided by $bind$. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 10. Up-conversion and helper functions

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

```
$openConstraints(\phi, [], \kappa)$ = $\kappa\ \phi$
$openConstraints(\phi, (\tau_{typerep}, \tau) : \phi'', \kappa)$ = let $\phi' = \{\tau := \tau_{typerep}, \phi\}$ in
    case $\tau$ of
        $\nu$ → if $\tau \in \phi$
                then if $\tau_{typerep} \simeq_{\tau} \beta_{typerep}$
                        $\{$where $\beta_{typerep} = lookup\ \tau\ \phi\}$
                     then $openConstraints(\phi', \phi'', \kappa)$
                     else $\bot$
                else let $\nu_{typerep} = a$
                     in $openConstraints(\phi', \phi'', \kappa)$
        $(T \tau_1 \ldots \tau_n)$ → typecase $\tau_{typerep}$ of
                (typerep $T$) $a_{1_{typerep}} \ldots a_{n_{typerep}}$ →
                    $(openConstraints($
                        $\{\tau_1 := a_{1_{typerep}}, \ldots, \tau_n := a_{n_{typerep}}, \phi'\}, \phi'', \kappa))$
```

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

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

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

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

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

$$openRecursion(\phi, (T, (\tau_1, \ldots, \tau_n)), v, \kappa) =$$
$$\kappa\ \phi\ (\ \check{T}\ \widetilde{buildTyRep}(\phi, \tau_1) \ldots \widetilde{buildTyRep}(\phi, \tau_n), v)$$

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

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

Figure 11. Down-conversion and helper functions

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

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

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

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

Figure 12. Down-conversion and unsealing functions, continued

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

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

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

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

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

```
if $\tau_{typerep} \simeq_\tau\ \beta_{typerep}$   {where $\beta_{typerep} = lookup\ \tau\ \phi$}
   then $openConstraints(\phi', D', \kappa)$
   else $\bot$
```

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

```
let $a_{typerep} = \tau_{typerep}$
in $openConstraints(\phi', D', \kappa)$
```

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

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

We then have by a similar argument as in the `if` case, that this `typecase` must succeed otherwise it would contradict our hypothesis on ϕ.
Now, by the definition of `typecase`, we have that this pattern match on (`typerep` T) and the various type representations that it's applied to will ensure (runtime) unification between τ'_i and $a_{i_{typerep}}$ – i.e. point wise type equality ($\simeq_\tau$) between $\tau_{typerep}$ and $a_{i_{typerep}}$ as well as unification of type constructors – and thus, we do not need to add these constraints into D', and instead can add them into our done-constraints ϕ'. We then continue under the additional substitutions.

$$\{\tau'_1 := a_{1_{typerep}}, \ldots, \tau'_n := a_{n_{typerep}}\}$$

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

□

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

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

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

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

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

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

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

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

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

then

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

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

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

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

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

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

```
K $x_1$ ... $x_n$ $\to$
   let $\phi$ = unify(T k c s, T $\tau_k$ $\tau_c$ $\tau_s$)
   K'$_{typereps}$ = map ($\lambda\ \tau \to$bind($(\!|\phi|\!)$, [$\tau$], buildTyRep($\tau$)))
                 getTyReps(K)
   in K' K'$_{typereps}$
         dispatch$_\uparrow$($\phi$, $x_1$, $\phi(\tau_1)$)
         ...
         dispatch$_\uparrow$($\phi$, $x_n$, $\phi(\tau_n)$)
```

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

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

- <u>case $\tau_i = (S\ \tau_{k_i}\ \tau_{c_i}\ \tau_{s_i})\{S$ has erased variables$\}$</u>: Now recall that since S is a busted data definition, then $\texttt{amb-test}(prog, S)$ holds, and likewise by the proviso of the theorem that there exists a ϕ' such that $prog' \vdash e_i : \phi'(\tau_i)$ and since $\tau_i = S\ \tau_{k_i}\ \tau_{c_i}\ \tau_{s_i}$ we have that $prog' \vdash x_i : \phi'(S\ \tau_{k_i}\ \tau_{c_i}\ \tau_{s_i})$ which implies that $x_i = K_S\,y_1 \ldots y_m$ for some data constructor $K_S \in \texttt{def}(prog, S)$
  We then have by the definition of $dispatch_\uparrow$ that $dispatch_\uparrow(\phi, x_i, \phi(\tau_i))$ will result in the following code being executed:
  $$\begin{aligned}&\texttt{let}\ \overline{v} = \texttt{map}\ (\lambda\,\tau.bind((\!|\phi|\!), FV(\tau), buildTyRep(\tau)))\\ &\qquad\quad getTyReps(K_S)\\ &\ \texttt{in}\ \widehat{S}\ \overline{v}\ x_i,\end{aligned}$$
  Since we execute the code with the instantiation $x_i = e_i$, we now apply the inductive hypothesis, to get that
  $$prog' \vdash \widehat{S}\ \overline{v}\ e_i \equiv e_i'$$
  and that
  $$prog' \vdash x_i' :: S'\tau_{k_i} \text{ and } x_i' = K_S'\ K_{S'_{typereps}}\ y_1' \ldots\ y_n'$$
- <u>case $\tau = TypeRep\ \tau'$</u> By the definition of $dispatch_\uparrow$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.
- <u>case $(T\ \overline{\tau})\ \{T$ has no erased variables$\}$</u>: By the definition of $dispatch_\uparrow$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.
- <u>case $(ArrowTy\ \tau_1\tau_2)$</u>: By the definition of $dispatch_\uparrow$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.
- <u>case $\tau = a$ is a type variable</u>: By the definition of $dispatch_\uparrow$, we have that it will simply return x_i which has type τ_i. Therefore this case is satisfied.

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

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

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

```
K'$_{typereps}$ = map ($\lambda\ \tau\ \to$bind($(\!|\phi|\!)$, [$\tau$], buildTyRep($\tau$)))
              getTyReps(K)
```

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

We therefore have by the above, and by Lemma 1 that the constructor application of K' to $K'_{typereps}$ will apply K' to all needed type representation fields of K' and that each of these fields will be of the appropriate type and be the correct type representation for the given newly-existential variable(s) in K'. In the case of zero new existential variables, this map of *bind* will simply return nothing, since $getTyReps(K) = \{\}$ which agrees with what our LT constructor expects.

We also have by the above that each of the applications of K' to $dispatch_\uparrow(\phi, a_i)$ will be well typed, that each of the values will be preserved modulo ghostbusted data types, and that the application of K' to $K'_{typereps}$ and $x_1' \ldots x_n'$ will result in a total (non-partial) application of K'. From this, since K' is a data-constructor for $T'\ k$ we get that e' has type $T'\ k$. Furthermore, since the pattern matching on e is unique, e' must be of the form $K'\ K'_{typereps}\ e'_1 \ldots\ e'_n$. Where K' is the corresponding LT data constructor for K. Thus (1.) is proved.

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

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

```
K' K'$_{typereps}$ $x'_1$ ... $x'_n$ →
    ...
```

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

- We call $openConstraints(typeEnv, [(c_{typerep}, \tau_c)], \kappa_1)$, where

  $$typeEnv = \mathcal{D}_{check} \cup K'_{typereps}$$

  we then have by Lemma 2 that before $openConstraints$ relinquishes control to κ_1, that for each checked variable $\tau_c \in \overline{\tau_c}$ we have a checked substitution with its corresponding type representation $\tau_{c_{typerep}}$; $\{\tau_c := \tau_{c_{typerep}}\}$ in ϕ^1 before $(\kappa_1\ \phi^1)$ is called.
- In the above case, we have that

  $$\kappa_1 = \quad (\lambda\ \phi_1\ .\ openFields((x'_1, \ldots, x'_n), \phi_1, [], \\ (\tau'_1, \ldots, \tau'_n), \kappa_2))$$

  where each x'_i has type τ'_i. We now make the following claims:

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

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

  $$\breve{S}\ \{\widetilde{buildTyRep}(\phi, \tau_1), \ldots \widetilde{buildTyRep}(\phi, \tau_n)\} \\ (J'\ J'_{typereps}\ e'_1 \ldots\ e'_m)$$

  will succeed, and will return $(\mathcal{S}\ D_S\ (e_1 \ldots e_m))$ □

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

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

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

Case: $\tau' = (S\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s})$: {where S is a busted type constructor} In this case, we have by Claim 1 that before we pass control to

$$(\lambda\ \phi'\ rec.\ unseal(T\ \overline{\tau_k}\ \overline{\tau_c}\ \overline{\tau_s}, \phi', rec, \\ (\lambda\ \phi''\ \mathcal{D}_s\ a.\ openConstraints(\phi'', [(s_{typerep}, \tau_s)] \\ (\lambda\ \phi'''.\ (\kappa'\ \phi'''\ a))))))$$

that we will be able to open up any new constraints, and that rec will be of the form $(\mathcal{S}\ \mathcal{D}_s\ (e_1 \ldots e_m))$ and that

$$\phi' \supseteq \phi \supset \widetilde{\phi} \implies \phi' \supseteq \widetilde{\phi}$$

by the definition of $openRecursion$. We then have proceeding by case-analysis on $unseal$ that since S is a type constructor (i.e., datatype) that is busted, that we will call the following code

$$(\lambda\ \phi''\ \mathcal{D}_S\ a.\ openConstraints(\phi'', [(s_{typerep}, \tau_s)], \\ (\lambda\ \phi'''.\ (\kappa'\ \phi'''\ a))))$$

where

$$\phi'' = \phi' \supseteq \phi \supseteq \widetilde{\phi} \implies \phi'' \supseteq \widetilde{\phi}$$

We now have by Lemma 2 that the call to $openConstraints$ will create new substitutions $\{\tau_s := s_{typerep}\}$ (and witness them) for each $\tau_s \in \overline{\tau_s}$ and $s_{typerep} \in D_S$. What's more, we have by this same lemma that $openConstraints$ will then pass these new substitutions on to

$$(\lambda\ \phi'''.\ (\kappa'\ \phi'''\ a))$$

But this κ' is just

$$(\lambda\ \phi'\ v.\ openFields(fs, \phi', (vars, v), \tau_{rest}, \kappa))$$

and we can now apply the inductive hypothesis to the inner call of $openFields$ after β-reduction.

We have thus shown that $openField$ succeeds in down-converting the type field represented for f – renaming it a fresh name v – and also adds all newly-discovered (synthesized) type information from τ' to ϕ''' and therefore by the inductive hypothesis, that $openFields(fs, \phi''', (vars, v), \tau_{rest}, \kappa)$ succeeds within κ'.
Therefore $openFields$ will succeed in down-converting all fields in $varList$ and, for all newly-discovered synthesized type-variables in these fields, will witness and add a substitution

between the synthesized type-variable and it's corresponding type representations to ϕ before passing control to κ. □

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

- In the above case, we have that

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

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

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

```
let $\tau$ = lookup $x$ $\psi$ in
let $x_{typerep}$ = $buildTyRep(\tau)$
   in $bind(\psi, xs)$
```

    The τ that is returned from the lookup will be either a kept, checked, or synthesized type. If τ is kept or checked, then we have by the above parts (specifically, bullet one and two of this part of the proof) that we will have a type representation witness for τ. Likewise if τ is a synthesized type, then we have that the ambiguity criteria will ensure that we are able to construct a type representation witness for τ as well. We thus have that we can always construct the type representation needed, and that we have the appropriate substitutions and constraints within scope to ensure that these type representations are closed. We then have by the inductive hypothesis that we can construct type representations for xs.

  - $\overline{s' = []}$: In this case we have no more synthesized variables to find type representations for, we return

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

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

Thus, we see that when we construct

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

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

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

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

Thus we see that

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

This proves (2.) □