First-Class Phantom Types

James Cheney
Cornell University
Ithaca, NY 14850
jcheney@cs.cornell.edu

Ralf Hinze
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de

Abstract

Classical phantom types are datatypes in which type constraints are expressed using type variables that do not appear in the datatype cases themselves. They can be used to embed typed languages into Haskell or ML. However, while such encodings guarantee that only well-formed data can be constructed, they do not permit type-safe deconstruction without additional tagging and run-time checks. We introduce *first-class phantom types*, which make such constraints explicit via *type equations*. Examples of first-class phantom types include typed type representations and typed higher-order abstract syntax trees. These types can be used to support typed generic functions, dynamic typing, and staged compilation in higher-order, statically typed languages such as Haskell or Standard ML. In our system, type constraints can be equations between type constructors as well as type functions of higher-order kinds. We prove type soundness and decidability for a Haskell-like language extended by first-class phantom types.

1 Introduction

Generic functions, dynamic typing, and metacircular interpretation are powerful features found in dynamically typed programming languages but have proven difficult to incorporate into statically typed languages such as Haskell or Standard ML. Past approaches have included adding a first-class *Dynamic* type and **typecase** expressions [1, 2, 20], defining generic functions by translation from "polytypic" languages to existing languages [15, 13], and implementing staged computation with run-time typechecking [8, 25] or compile-time computation [24]. Recently, however, Cheney and Hinze [6] and Baars and Swierstra [4] found that many of these features can already be implemented via an encoding into Haskell based on *equality types* comprising executable *evidence* of type equality (e.g. embedding-projection pairs).

However, this encoding has several drawbacks:

- It imposes a high annotation burden on programmers, limiting its usability;
- it incurs unnecessary run-time overhead in the form of calls to embedding and projection functions;
- it is limited by the constraint-solving abilities of the underlying type-checker, so some interesting and safe programs do not typecheck.

The last limitation is particularly vexing because it places many interesting potential applications just out of reach. For example, some type-indexed types (e.g. generic generalized tries [12] and **Typerec** [10, 7]) can almost, but not quite, be implemented naturally. The obstacle is that the typechecker does not know certain valid properties of types that are not important for normal typechecking.

A similar difficulty arises in the context of so-called *phantom types*, that is, parameterized types that do not use their type argument(s). Leijen and Meijer [18]) first introduced and used phantom types to embed domain specific languages into Haskell type-safely. Operations on such types are typically restricted to a module which hides the internal (untyped) implementation, exposing only

the (typed) interface. Classical phantom type encodings can enforce type constraints on constructed values, but cannot make use of these constraints when decomposing a value. For example, it is possible to encode typed λ-terms in Haskell or ML with a module such as

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{module}\ \mathit{Lam}\ (\mathit{Lam}, \mathit{fun}, \mathit{app})\ \textbf{where}} \\
\textbf{data}\ \mathit{Lam0} & = & \mathit{Fun}\ (\mathit{Lam0} \to \mathit{Lam0}) \\
 & | & \mathit{App}\ \mathit{Lam0}\ \mathit{Lam0} \\
\textbf{type}\ \mathit{Lam}\ \alpha & = & \mathit{Lam0} \\
\mathit{fun} & :: & (\mathit{Lam}\ \alpha \to \mathit{Lam}\ \beta) \to \mathit{Lam}\ (\alpha \to \beta) \\
\mathit{fun}\ f & = & \mathit{Fun}\ f \\
\mathit{app} & :: & \mathit{Lam}\ (\alpha \to \beta) \to \mathit{Lam}\ \alpha \to \mathit{Lam}\ \beta \\
\mathit{app}\ m\ n & = & \mathit{App}\ m\ n
\end{array}
$$

such that only well-typed λ-terms can be formed using *fun* and *app*. But functions like

$$
\begin{array}{lcl}
\mathit{reduce} & :: & \mathit{Lam}\ \beta \to \mathit{Lam}\ \beta \\
\mathit{reduce}\ (\mathit{App}\ (\mathit{Fun}\ f)\ t) & = & f\ t
\end{array}
$$

typecheck within the module but not outside. Although it is true that $f :: \alpha \to \beta$ and $t :: \alpha$ for the same α, the standard type inference algorithm will conservatively infer only that $f :: \alpha' \to \beta$ for some fresh α', and typechecking will fail because α' is not "polymorphic enough". Thus, this phantom-type encoding is *second-class* in the sense that the phantom type constraints can be used to guarantee that only well-formed data is constructed, but cannot be used to help typecheck intuitively reasonable computations on such data.

We advocate an approach that fills this gap, and also encompasses encodings of typed type representations and other type-indexed types within Haskell. These encodings can be viewed as phantom types in which the abstract type variables *do* appear in the types of the constructors. Specifically, in our approach, the formerly unused type variables now appear within type equations that show how to refine the argument types. Conversely, this technique can be used to enrich more mainstream phantom types such as the λ-terms mentioned above, to enable typechecking computations over such data.

In this paper we present a programming construct for defining such *first-class phantom types*. In our formalism, datatype cases may be annotated by **with** clauses of the form **with** $\{\tau_1 = \tau_1'; \ldots; \tau_n = \tau_n'\}$. The equations listed in a **with** clause must hold whenever the corresponding constructor is applied and may be used in typechecking case bodies for the constructor. In combination with existential types, datatype definitions using type equations can be used to implement both introduction and elimination forms for many advanced applications of phantom types, including typed type representations and typed higher-order abstract syntax.

In the rest of this paper, we describe our proposal in more detail, discuss applications, present a type system for a core, explicitly typed Haskell-like language with first-class phantom types, and prove that its type system is sound and that typechecking is decidable. Finally, we discuss related and future work and conclude.

2 Enriching Phantom Types

2.1 Type representations and generic functions

As an introductory example, we show how to implement typed type representations using phantom types. The basic idea of type representations is to define for each type constructor a data constructor that represents the type. For instance, to represent types built from *Int* and *Char* using the list and the pair type constructor we introduce the following data constructors

$$
\begin{array}{lcl}
R_{\mathit{Int}} & :: & \mathit{Rep}\ \mathit{Int} \\
R_{\mathit{Char}} & :: & \mathit{Rep}\ \mathit{Char} \\
R_{[]} & :: & \forall\alpha\,.\,\mathit{Rep}\ \alpha \to \mathit{Rep}\ [\alpha] \\
R_{\times} & :: & \forall\alpha\ \beta\,.\,\mathit{Rep}\ \alpha \to \mathit{Rep}\ \beta \to \mathit{Rep}\ (\alpha, \beta).
\end{array}
$$

Thus, *Int* is represented by R_{Int}, the type $([\mathit{Char}], \mathit{Int})$ is represented by the value $R_\times\ (R_{[]}\ R_{Char})\ R_{Int}$.

In Haskell 98, the above signature cannot be translated into a **data** declaration. The reason is simply that all constructors of a data type must share the same result type, namely, the declared type on the left-hand side. Thus, we can assign R_{Int} the type $\mathit{Rep}\ \tau$ but not $\mathit{Rep}\ \mathit{Int}$. We can, of course, implement the above interface using classical phantom types. But the usefulness of such an implementation would be limited because we cannot pattern-match against functions defined in an interface, thus, outside of the abstraction barrier we cannot deconstruct a value of type $\mathit{Rep}\ a$. Consequently, every new interesting computation over $\mathit{Rep}\ a$ must be added to the module in which its implementation is available.

If only we had the means to constrain the type argument of *Rep* to a certain type. Now, this is exactly what a **with** clause allows us to do. Given this extension we declare the *Rep* datatype as follows:

$$\begin{array}{lcll} \textbf{data}\ \mathit{Rep}\ \tau & = & R_{Int} & \textbf{with}\ \tau = \mathit{Int} \\ & \mid & R_{Char} & \textbf{with}\ \tau = \mathit{Char} \\ & \mid\ \forall\alpha\,. & R_{[]}\ (\mathit{Rep}\ \alpha) & \textbf{with}\ \tau = [\alpha] \\ & \mid\ \forall\alpha\,\beta\,. & R_\times\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta) & \textbf{with}\ \tau = (\alpha, \beta). \end{array}$$

The **with** clause that is attached to each constructor records its type constraints. For instance, R_{Int} has type $\mathit{Rep}\ \tau$ with the additional constraint $\tau = \mathit{Int}$. The equations listed in the **with** clause are checked whenever the constructor is used to build a value. For example, the term $R_{[]}\ R_{Int}$ has type $\mathit{Rep}\ [\mathit{Int}]$ and is illegal in a context where an element of type $\mathit{Rep}\ \mathit{Char}$ is required (however, if the **with** clauses were left out it would be legal). A type equation may contain type variables that do not appear on the left-hand side of the data declaration.

Using these type representations we can now define *generic functions* that work for all representable types. Here is how we implement generic equality.

$$\begin{array}{lcl} \mathit{equal} & :: & \forall\tau\,.\,\mathit{Rep}\ \tau \to \tau \to \tau \to \mathit{Bool} \\ \mathit{equal}\ (R_{Int})\ i_1\ i_2 & = & i_1 == i_2 \\ \mathit{equal}\ (R_{Char})\ c_1\ c_2 & = & c_1 == c_2 \\ \mathit{equal}\ (R_{[]}\ r_\alpha)\ []\ [] & = & \mathit{True} \\ \mathit{equal}\ (R_{[]}\ r_\alpha)\ []\ (a_2 : x_2) & = & \mathit{False} \\ \mathit{equal}\ (R_{[]}\ r_\alpha)\ (a_1 : x_1)\ [] & = & \mathit{False} \\ \mathit{equal}\ (R_{[]}\ r_\alpha)\ (a_1 : x_1)\ (a_2 : x_2) & = & \mathit{equal}\ r_\alpha\ a_1\ a_2 \wedge \mathit{equal}\ (R_{[]}\ r_\alpha)\ x_1\ x_2 \\ \mathit{equal}\ (R_\times\ r_\alpha\ r_\beta)\ (a_1, b_1)\ (a_2, b_2) & = & \mathit{equal}\ r_\alpha\ a_1\ a_2 \wedge \mathit{equal}\ r_\beta\ b_1\ b_2 \end{array}$$

The equality function takes a type representation as a first argument, two values of the represented type and returns a Boolean. Even though *equal* is assigned the type $\forall\tau\,.\,\mathit{Rep}\ \tau \to \tau \to \tau \to \mathit{Bool}$, each equation has a more specific type as dictated by the type constraints. As an example, the first equation has type $\mathit{Rep}\ \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$ as R_{Int} constrains τ to *Int*. For further examples of generic functions and for an elegant way of combining generic functions with dynamic values the interested reader is referred to a recent paper by the authors [6].

In general, a **with** clause may comprise several type equations and each equation may relate any two type expressions of the same kind (including higher-order kinds). The main constraint on equations is that they be *satisfiable*: that is, there must be some substitution for the free type variables which makes the equations true. An example of a trivially unsatisfiable equation is $\mathit{Int} = \mathit{Bool}$; more subtle examples are $\alpha = \alpha \to \alpha$ and $\alpha = (\beta, \gamma), (\mathit{Int}, \gamma) = (\alpha, \delta)$. The reason for forbidding *prima facie* unsatisfiable constraints is that they do not help us typecheck any additional useful programs: if a constructor's constraints are unsatisfiable, then that constructor can never apepar during the execution of a program.

The reader may be curious whether allowing general type equations, or higher-kinded equations, is any more expressive than existing techniques, such as Guarded Recursive Datatypes [27]. The answers are *no* (as long as we forbid unsatisfiable guards) and *yes* respectively. We elaborate on this point in Section 6.6. We will encounter examples of equations at higher kinds in Section 2.4.

2.2 Typed λ-terms

We now show how to enrich the definition of typed higher-order λ-terms from the Introduction using **with** clauses to allow typechecking of functions like *reduce*. Working backward from the intended signature of the type constructors

$$\begin{array}{lcl} \mathit{Fun} & :: & (\mathit{Lam}\ \alpha \to \mathit{Lam}\ \beta) \to \mathit{Lam}\ (\alpha \to \beta) \\ \mathit{App} & :: & \mathit{Lam}\ (\alpha \to \beta) \to \mathit{Lam}\ \alpha \to \mathit{Lam}\ \beta \end{array}$$

we enrich the definition of *Lam* is as follows:

$$\begin{array}{lcll} \textbf{data}\ \mathit{Lam}\ \tau & = & \forall \beta\ \gamma\,. \quad \mathit{Fun}\ (\mathit{Lam}\ \beta \to \mathit{Lam}\ \gamma) & \textbf{with}\ \tau = \beta \to \gamma \\ & | & \forall \alpha\,. \quad \mathit{App}\ (\mathit{Lam}\ (\alpha \to \tau))\ (\mathit{Lam}\ \alpha) & \end{array}$$

Now consider

$$\begin{array}{lcl} \mathit{reduce} & :: & \mathit{Lam}\ \beta \to \mathit{Lam}\ \beta \\ \mathit{reduce}\ (\mathit{App}\ (\mathit{Fun}\ f)\ t) & = & f\ t \end{array}$$

The typechecker can now recover enough information to check that this function is type-safe. The typechecker will first observe that $\mathit{Fun}\ f :: \mathit{Lam}\ (\alpha \to \beta)$ and $t :: \mathit{Lam}\ \alpha$ for some α. Thus, $f :: \mathit{Lam}\ \gamma \to \mathit{Lam}\ \delta$ for some new γ, δ. But the typechecker also knows that the equation $\beta \to \alpha = \gamma \to \delta$ must hold; this implies that $\alpha = \gamma$ and $\beta = \delta$. Now, to typecheck $f\ t$, the typechecker needs to unify $\mathit{Lam}\ \gamma$ (the argument type of f) and $\mathit{Lam}\ \alpha$ (the type of t). This follows from $\alpha = \gamma$. Finally, to typecheck the whole function, it is necessary to unify $\mathit{Lam}\ \beta$ (the stated result type) and $\mathit{Lam}\ \delta$ (the return type of f). This too succeeds because $\beta = \delta$.

Here we are being cavalier about how the typechecker actually performs this reasoning. The actual mechanism uses unification to calculate the effect of the current set of type equations on typechecking, and also can discover that code is actually unreachable based on type equation information. For example, if we were to add integer constants and addition to *Lam*, attempting to pattern-match cases such as $\mathit{App}\ (\mathit{IntC}\ i)\ t$ and $\mathit{Plus}\ (\mathit{Fun}\ f)\ (\mathit{IntC}\ j)$ will result in collections of type equations that give rise to inconsistency; this indicates that such patterns can never be matched by well-formed data. The bodies of such cases are dead code. We make it an error for there to be dead code of this kind in order to avoid the problem of how to continue to typecheck code under inconsistent assumptions. Type inference also becomes more complicated in this setting (see Section 6.4).

2.3 Generic traversals

Building upon the *Rep* type introduced earlier in this section we proceed to implement a small combinator library that supports *typed tree traversals* (related approaches are strategic programming, visitor patterns, and adaptive programming). Broadly speaking, a generic traversal is a function that modifies data of *any* type.[1] The type of traversals is

$$\textbf{type}\ \mathit{Traversal} \quad = \quad \forall \tau\,.\, \mathit{Rep}\ \tau \to \tau \to \tau.$$

Thus, a generic traversal takes a type representation and transforms a value of the specified type. The universal quantifier makes explicit that the function works for *all* representable types.

As an example, here is an *ad hoc* traversal that converts strings to upper case and leaves integers, characters, pairs and so on unchanged.

$$\begin{array}{lcl} \mathit{capitalize} & :: & \forall \tau\,.\, \mathit{Rep}\ \tau \to \tau \to \tau \\ \mathit{capitalize}\ (R_{[]}\ R_{\mathit{Char}})\ s & = & \mathit{map}\ \mathit{toUpper}\ s \\ \mathit{capitalize}\ r_\tau\ t & = & t \end{array}$$

The simplest traversal is, of course, the identity idT, which does nothing.

$$\begin{array}{lcl} idT & :: & \mathit{Traversal} \\ idT\ r_\tau & = & \mathit{id} \end{array}$$

[1] Here, any *representable* type

Traversals can be composed using the operator '$\circ$', which has *copy* as its identity.

$$\begin{array}{lcl} (\circ) & :: & \mathit{Traversal} \to \mathit{Traversal} \to \mathit{Traversal} \\ (f \circ g)\ r_\tau & = & f\ r_\tau \cdot g\ r_\tau \end{array}$$

Note that '$\circ$' has a so-called *rank-2 type*: it takes polymorphic functions to polymorphic functions. While not legal Haskell 98 most implementations support rank-2 types or even rank-n types. With support for rank-n types, generic functions become true *first-class citizens*, a fact, which is put to good use in the combinator library.

The next combinator, called *everywhere*, allows us to apply a traversal 'everywhere' in a given value. It implements the generic part of traversals—sometimes called the *boilerplate code*. For instance, *everywhere capitalize* v converts all the strings contained in v to upper case. Note that v is an arbitrary value of an arbitrary (representable) type. The *everywhere* combinator is implemented in two steps. We first define a function that applies a traversal f to the *immediate* components of a value: the constructed value $C\ t_1\ \ldots\ t_n$ is mapped to $C\ (f\ r_{\tau_1}\ t_1)\ \ldots\ (f\ r_{\tau_n}\ t_n)$ where r_{τ_i} is the representation of t_i's type.

$$\begin{array}{lcl} \mathit{mapT} & :: & \mathit{Traversal} \to \mathit{Traversal} \\ \mathit{mapT}\ f\ (R_{Int})\ i & = & i \\ \mathit{mapT}\ f\ (R_{Char})\ c & = & c \\ \mathit{mapT}\ f\ (R_{[]}\ r_\alpha)\ [\,] & = & [\,] \\ \mathit{mapT}\ f\ (R_{[]}\ r_\alpha)\ (a : as) & = & f\ r_\alpha\ a : f\ (R_{[]}\ r_\alpha)\ as \\ \mathit{mapT}\ f\ (R_\times\ r_\alpha\ r_\beta)\ (a, b) & = & (f\ r_\alpha\ a, f\ r_\beta\ b) \end{array}$$

The function mapT can be seen as a 'traversal transformer'. It enjoys functor-like properties: $\mathit{mapT}\ \mathit{idT} = \mathit{idT}$ and $\mathit{mapT}\ (f \circ g) = \mathit{mapT}\ f \circ \mathit{mapT}\ g$.

$$\begin{array}{lcl} \mathit{everywhere}, \mathit{everywhere}' & :: & \mathit{Traversal} \to \mathit{Traversal} \\ \mathit{everywhere}\ f & = & f \circ \mathit{mapT}\ (\mathit{everywhere}\ f) \\ \mathit{everywhere}'\ f & = & \mathit{mapT}\ (\mathit{everywhere}'\ f) \circ f \end{array}$$

Actually, there are two flavors of the combinator: *everywhere* f applies f *after* the recursive calls (it traverses bottom-up, or post-order), whereas *everywhere*$'$ applies f *before* (it traverses top-down, or pre-order).

In a similar manner, we can also implement *generic queries* which transform trees into values of some fixed type. For further examples and applications we refer the interested reader to a recent paper by Lämmel and Peyton Jones [17].

2.4 Generalized tries

A trie is a search tree scheme that employs the structure of search keys to organize information [12]. In contrast to conventional search trees based on ordering relations tries support look-up and update in time proportional to the size of the search key. This means, however, that every key type gives rise to a different trie structure. In fact, tries serve as a nice example of so-called *type-indexed datatypes* [14], data types that are constructed in a generic way from an argument data type. Using phantom types we can implement tries and other type-indexed datatypes in at least two different ways.

A functional encoding A trie represents a finite map, a mapping from keys to values. Consequently, the trie type is parameterized by the type of keys and by the type of associated values and is defined via 'pattern matching' on the first argument.

$$\begin{array}{lcll} \textbf{data}\ \mathit{Trie}\ \kappa\ \nu & = & T_1\ (\mathit{Maybe}\ \nu) & \textbf{with}\ \kappa = () \\ & \mid & T_+\ (\mathit{Trie}\ \kappa_1\ \nu)\ (\mathit{Trie}\ \kappa_2\ \nu) & \textbf{with}\ \kappa = \mathit{Either}\ \kappa_1\ \kappa_2 \\ & \mid & T_\times\ (\mathit{Trie}\ \kappa_1\ (\mathit{Trie}\ \kappa_2\ \nu)) & \textbf{with}\ \kappa = (\kappa_1, \kappa_2) \end{array}$$

A trie for the unit type is a *Maybe* value (it is *Nothing* if the finite map is empty and of the form *Just* v otherwise), a trie for a sum is a product of tries and a trie for a product is a composition of tries. Note that *Trie* (as well as *Rep*) is not only a phantom type but also a so-called *nested* (or *non-regular*) *type* [5], as the definition involves 'recursive calls', e.g. $\mathit{Trie}\ \kappa_1\ (\mathit{Trie}\ \kappa_2\ \nu)$, that are substitution instances of the defined type.

The generic look-up function on tries is given by the following definition.

$$
\begin{array}{lcl}
\mathit{lookup} & :: & \forall \kappa\ \nu\ .\ \mathit{Trie}\ \kappa\ \nu \to \kappa \to \mathit{Maybe}\ \nu \\
\mathit{lookup}\ (T_1\ m)\ () & = & m \\
\mathit{lookup}\ (T_+\ ta\ tb)\ (\mathit{Left}\ a) & = & \mathit{lookup}\ a\ ta \\
\mathit{lookup}\ (T_+\ ta\ tb)\ (\mathit{Right}\ b) & = & \mathit{lookup}\ b\ tb \\
\mathit{lookup}\ (T_\times\ ta)\ (a, b) & = & \mathit{lookup}\ a\ ta \gg\!= \mathit{lookup}\ b
\end{array}
$$

where $(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$ is Haskell's built-in monad composition operator, here used with the *Maybe* monad. Note that *lookup* (as well as *equal*) requires a non-schematic form of recursion called *polymorphic recursion* [22]: the recursive calls are at types which are substitution instances of the declared type (see also Section 6.4).

Tries are also attractive because they support an efficient *merge* operation.

$$
\begin{array}{lcl}
\mathit{merge} & :: & \forall \kappa\ \nu\ .\ (\nu \to \nu \to \nu) \\
 & & \qquad \to (\mathit{Trie}\ \kappa\ \nu \to \mathit{Trie}\ \kappa\ \nu \to \mathit{Trie}\ \kappa\ \nu) \\
\mathit{merge}\ c\ (T_1\ \mathit{Nothing})\ (T_1\ \mathit{Nothing}) & = & T_1\ \mathit{Nothing} \\
\mathit{merge}\ c\ (T_1\ \mathit{Nothing})\ (T_1\ (\mathit{Just}\ v')) & = & T_1\ (\mathit{Just}\ v') \\
\mathit{merge}\ c\ (T_1\ (\mathit{Just}\ v))\ (T_1\ \mathit{Nothing}) & = & T_1\ (\mathit{Just}\ v) \\
\mathit{merge}\ c\ (T_1\ (\mathit{Just}\ v))\ (T_1\ (\mathit{Just}\ v')) & = & T_1\ (\mathit{Just}\ (c\ v\ v')) \\
\mathit{merge}\ c\ (T_+\ ta\ tb)\ (T_+\ ta'\ tb') & = & T_+\ (\mathit{merge}\ c\ ta\ ta')\ (\mathit{merge}\ c\ tb\ tb') \\
\mathit{merge}\ c\ (T_\times\ ta)\ (T_\times\ ta') & = & T_\times\ (\mathit{merge}\ (\mathit{merge}\ c)\ ta\ ta')
\end{array}
$$

The merge operation takes as a first argument a so-called combining function, which is applied whenever two bindings have the same key. It is instructive to reproduce why the definition typechecks. Consider the second but last equation: the patterns constrain κ to $\kappa = \mathit{Either}\ \kappa_1\ \kappa_2$ (first parameter) and $\kappa = \mathit{Either}\ \kappa_1'\ \kappa_2'$ (second parameter). The arguments of *Either* are different for each parameter because they are existentially quantified in the **data** declaration. However, we may conclude from $\mathit{Either}\ \kappa_1\ \kappa_2 = \kappa = \mathit{Either}\ \kappa_1'\ \kappa_2'$ that $\kappa_1 = \kappa_1'$ and $\kappa_2 = \kappa_2'$, which allows us to typecheck the two recursive calls. As innocent as this deduction may seem, this is exactly the point where the encoding of **with** types using an equality type fails: there is no way to prove this implication (see also Section 4).

Perhaps surprisingly, the equations defining *merge* are exhaustive though only six out of nine combinations of *Trie* constructors are covered. The reason is simply that, for instance, the case $\mathit{merge}\ c\ (T_+\ ta\ tb)\ (T_\times\ ta')$ need not be considered because the equations $\kappa = \mathit{Either}\ \kappa_1\ \kappa_2$ and $\kappa = (\kappa_1', \kappa_2')$ cannot be satisfied simultaneously.

A relational encoding A more intriguing alternative way of implementing tries is to encode the type-indexed type as a binary relation which relates each key type κ of kind $\star$ to the corresponding trie type *Trie* κ of kind $\star \to \star$.

$$
\begin{array}{lcl}
\mathbf{type}\ (\varphi_1 \times \varphi_2)\ v & = & (\varphi_1\ v, \varphi_2\ v) \\
\mathbf{type}\ (\varphi_1 \cdot \varphi_2)\ v & = & \varphi_1\ (\varphi_2\ v) \\
\mathbf{data}\ \mathit{Rep}\ \kappa\ \varphi & = & R_1 \\
 & & \quad \mathbf{with}\{\kappa = (); \varphi = \mathit{Maybe}\} \\
 & \mid & R_+\ (\mathit{Rep}\ \kappa_1\ \varphi_1)\ (\mathit{Rep}\ \kappa_2\ \varphi_2) \\
 & & \quad \mathbf{with}\{\kappa = \mathit{Either}\ \kappa_1\ \kappa_2; \varphi = \varphi_1 \times \varphi_2\} \\
 & \mid & R_\times\ (\mathit{Rep}\ \kappa_1\ \varphi_1)\ (\mathit{Rep}\ \kappa_2\ \varphi_2) \\
 & & \quad \mathbf{with}\{\kappa = (\kappa_1, \kappa_2); \varphi = \varphi_1 \cdot \varphi_2\}
\end{array}
$$

The type *Rep* can be seem as a type representation type that additionally incorporates the type-indexed datatype. Note that the second equation in each case relates two type constructors of kind $\star \to \star$. The operations on tries now take the type representation as a first argument and proceed as before (the definition of *merge* is left as an exercise to the reader).

$$\begin{array}{lcl}
\mathit{lookup} & :: & \forall\kappa\,\varphi\,.\,\mathit{Rep}\;\kappa\;\varphi \to \forall\nu\,.\,\kappa \to \varphi\;\nu \to \mathit{Maybe}\;\nu \\
\mathit{lookup}\;(R_1)\;()\;m & = & m \\
\mathit{lookup}\;(R_+\;r_\alpha\;r_\beta)\;(\mathit{Left}\;a)\;(ta, tb) & = & \mathit{lookup}\;r_\alpha\;a\;ta \\
\mathit{lookup}\;(R_+\;r_\alpha\;r_\beta)\;(\mathit{Right}\;b)\;(ta, tb) & = & \mathit{lookup}\;r_\beta\;b\;tb \\
\mathit{lookup}\;(R_\times\;r_\alpha\;r_\beta)\;(a, b)\;ta & = & \mathit{lookup}\;r_\alpha\;a\;ta \gg\!= \mathit{lookup}\;r_\beta\;b
\end{array}$$

In a sense, this implementation disentangles the type representation from the trie data structure.

3 Formal Presentation

We now give the static semantics of phantom types. We model the source language $\lambda_{\equiv}$ as a call-by-name, explicitly typed and kinded variant of F_ω with datatypes. Figure 1 shows the syntax modifications introduced in $\lambda_{\equiv}$; the full syntax, operational semantics, and static semantics can be found in Appendix A.

Phantom type expressions $C\;[\overline{\tau}]\;\overline{e}$ are constructed by applying a constructor C to existentially hidden type arguments $\overline{\tau}$ and expression arguments $\overline{e}$. Any constraints attached to the constructor C must be valid when $C\;[\overline{\tau}]\;\overline{e}$ is typechecked. Case expressions $\mathbf{case}[\sigma]\;e\;\mathbf{of}\;ms$ take a type argument σ describing the intended return type, as well as the discriminated expression e and pattern matches ms. Pattern matches have the form $C\;[\overline{\beta}]\;\overline{x} \to e_C$, where $\overline{\beta}$ are names for existentially hidden types, $\overline{x}$ are names for the constructor's arguments, and e_C is an expression which gives the result of the **case** when the $C\;[\overline{\beta}]\;\overline{x}$ pattern matches. The equations associated with C are assumed to hold within e_C. We consider only one level of pattern matching.

Our core calculus does not allow direct pattern matching within function definitions or **let**-bindings, but does include the basic building blocks for implementing them, including **fix** and **case**. Our system is presented *a la* Church—λ, **fix**, **case**, and constructor expressions are type-annotated so that typechecking is syntax-directed. We believe that it is (at least conceptually) straightforward to translate sufficiently annotated general Haskell programs to our calculus. We have not investigated type inference, although we outline some of the issues in Section 6.4.

Type equations are of the form $\tau \equiv \tau'$, often abbreviated ϵ when τ, τ' are not important. Phantom datatype declarations are given in a top-level, mutually recursive signature Σ. We write $\Sigma_{T.C\;\overline{\alpha}} = \exists\overline{\beta{:}\kappa}.C\;\overline{\sigma}\;\mathbf{with}\;\overline{\epsilon{:}\kappa'}$ to concisely indicate the components of the type constructor $T.C$ with type variables $\overline{\alpha}$. For brevity, we often omit irrelevant kind annotations.

As a running example, we consider simple type representations for right-associative integer n-tuples. The declaration of this type in our formalism is

$$\begin{array}{lcl}
\mathbf{data}\;\mathit{RTuple}\;a & = & R_{\mathit{Int}}\;\mathbf{with}\;a = \mathit{Int} \\
& \mid & \forall b\,.\,R_\times\;(\mathit{RTuple}\;b)\;\mathbf{with}\;a = (\mathit{Int}, b)
\end{array}$$

We can form a representation of integer triples as follows:

$$R_\times\;[(\mathit{Int}, \mathit{Int})]\;R_{\mathit{Int}}\;(R_\times\;[\mathit{Int}]\;R_{\mathit{Int}}\;R_{\mathit{Int}})$$

with type $\mathit{Rep}\;(\mathit{Int}, (\mathit{Int}, \mathit{Int}))$. Note that for large terms, we can expect considerable redundancy among the type annotations; however, we do not foresee any problems with inferring these annotations using standard techniques for existential types. The **case** expression is used to deconstruct representations. For example, we can define a *sum* function that adds up all the integers in an arbitrary n-tuple as follows:

$$\begin{array}{lcl}
\mathit{sum} & :: & \forall\alpha\,.\,\mathit{RTuple}\;\alpha \to \alpha \to \mathit{Int} \\
\mathit{sum}\;r\;e & = & \mathbf{case}\;[\mathit{Int}]\;r\;\mathbf{of} \\
& & \quad R_{\mathit{Int}} \to e \\
& & \mid R_\times\;[\beta]\;rb \to \mathbf{let}\;(x, y) = e\;\mathbf{in} \\
& & \qquad\qquad x + \mathit{sum}\;rb\;y
\end{array}$$

$$
\begin{array}{rcl}
\Sigma & ::= & \cdot \mid \Sigma; \mathbf{data}\ T\ \overline{\alpha{:}\kappa} = \Sigma_T \\
\Sigma_T & ::= & \cdot \mid \Sigma_T \mid \exists \overline{\beta{:}\kappa}.C\ \overline{\sigma}\ \mathbf{with}\ \overline{\epsilon{:}\kappa'} \\
\Psi & ::= & \cdot \mid \Psi, \epsilon{:}\kappa \\
\epsilon & ::= & \tau_1 \equiv \tau_2 \\
e & ::= & C\ [\overline{\tau}]\ \overline{e} \mid \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms \mid \mathbf{fail} \mid \cdots \\
ms & ::= & \cdot \mid C\ [\overline{\beta}]\ \overline{x} \to e \mid ms
\end{array}
$$

Figure 1: Syntax modifications for $\lambda_{\equiv}$

$$
\begin{array}{rcl}
v & ::= & C\ [\overline{\tau}]\ \overline{e} \mid \mathbf{fail} \mid \cdots \\
E & ::= & \mathbf{case}[\sigma]\ E\ \mathbf{of}\ ms \mid \cdots
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ C\ [\overline{\beta}]\ \overline{x} \mid ms & \mapsto & e[\overline{e}/\overline{x}, \overline{\tau}/\overline{\beta}] \\
\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ D\ [\overline{\beta}]\ \overline{x} \mid ms & \mapsto & \mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ ms \\
\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ \cdot & \mapsto & \mathbf{fail} \\
E[\mathbf{fail}] & \mapsto & \mathbf{fail}
\end{array}
$$

Figure 2: Operational semantics for **case**

Note that this function is polymorphically recursive, since we call *size* at *Rep b* and *Rep c* in the second case, where b and γ are unknown types such that $\alpha = (\beta, \gamma)$. Hence we must provide a type signature for the function.

The *sum* function is relatively straightforward to typecheck because the return type is *Int*. However, we do need the type equations in each case. In the first case, we need to use the fact that $\alpha = Int$ to determine that the use of e as an *Int* is correct. In the second case, similarly, we need to use the assumption $\alpha = (Int, \beta)$ to determine that it is safe to deconstruct e as a pair (x, y), use x as an *Int*, and recursively call *sum rb y*.

In these cases, the reasoning needed to verify type-safety seems clear, but how are we to automate this intuitive process? In general, our typechecking algorithm uses unification to distill the information implied by a given set of type equations to a substitution. Further typechecking is performed subject to the substitution. Thus, in the first case we unify α and *Int*, yielding substitution $\alpha \to Int$. Under this substitution, the typing problem $\alpha = Int$ reduces to $Int = Int$, which is obviously OK. However, this substitution is only active within the case to which the equation $\alpha = Int$ applies. In the other case, we typecheck modulo the substitution $\alpha \to (Int, \beta)$, which again makes the body typecheck.

So far this seems rather trivial. Some of the power of this technique becomes evident when we consider functions with more than one represented value:

$$
\begin{array}{lcl}
eq & :: & RTuple\ a \to RTuple\ a \to Bool \\
eq\ R_{Int}\ R_{Int} & = & True \\
eq\ (R_\times\ rb)\ (R_\times\ rb') & = & eq\ rb\ rb' \\
eq\ R_{Int}\ (R_\times\ \_) & = & False \\
eq\ (R_\times\ \_)\ R_{Int} & = & False
\end{array}
$$

In this example, in the third and fourth cases, we need to unify $\alpha = Int, \alpha = (Int, \beta)$. But there is no unifier for these equations. This means that these cases are statically dead code, which we treat as an error in source programs. Hence, the programmer should erase the latter two cases, resulting in:

$$
\begin{array}{l}
eq\ R_{Int}\ R_{Int} = True \\
eq\ (R_\times\ rb)\ (R_\times\ rb') = eq\ rb\ rb'
\end{array}
$$

This function is total, that is, there are no missed cases. (See Section 6.3 for more on exhaustiveness checking.)

The operational semantics of datatype and case expressions is completely standard; type equations have no effect on expression evaluation. As illustrated above, equations may make it possible

to deduce exhaustiveness or detect unreachable patterns in more cases. To avoid complicating the typing rules further, however, we do not include these checks in the base system. Section 6.3 shows how to do exhaustiveness checking in the presence of equations. If multiple patterns match, only the first match will be taken; if no match is found, the **case**$[\cdot]$ $\cdot$ **of** $\cdot$ expression steps to an exception value **fail**.

The static semantics requires a new form of context Ψ consisting of equational assumptions. We use it in the following judgments:

$\Delta \vdash \Psi$	equation context well-formedness
$\Delta \vdash \Psi \Downarrow \Theta$	equation context unification
$\Delta; \Psi \vdash \tau_1 \equiv \tau_2 : \kappa$	constructor equivalence
$\Delta; \Psi \vdash \sigma_1 \equiv \sigma_2$	type equivalence
$\Delta; \Psi; \Gamma \vdash e : \sigma$	expression well-formedness
$\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$	pattern-match well-formedness

Equation contexts are well-formed if the left and right sides of each equation are well-formed in the specified kinds. The well-formedness of all constructors, types, and contexts is assumed in type equivalence and expression well-formedness derivations. The unification judgment asserts that Θ syntactically unifies Ψ: that is, $\Theta\tau_1 = \Theta\tau_2$ for each $\tau_1 = \tau_2$ in Ψ. Unification is used explicitly during checking the well-formedness of datatype declarations to ensure that the equations associated with each case are satisfiable.[2]

The constructor equivalence judgment states that two type constructors (expressions describing monomorphic types or type functions) are equal; similarly, the type equivalence judgment asserts the equality of two possibly-quantified types. Expression well-formedness asserts that an expression has a given type, and pattern-match well-formedness asserts that a pattern match (a sequence of pairs of patterns and bodies) is well-formed; that is, each pattern is a well-formed pattern of type $T\ \overline{\tau}$, and under the resulting variable bindings the body is a well-formed expression of type σ. Other judgments are shown (with rules) in Appendix A.

The type equivalence rules are mostly standard, including reflexivity, transitivity, symmetry, and congruence rules. We treat the base type constructors $\times, \rightarrow, T$ as constants with the appropriate kinds, e.g. $(\times), (\rightarrow) : \star \rightarrow \star \rightarrow \star$. Thus, we need only one congruence rule, which asserts that if the left and right sides of corresponding applications are equal, then the result of the application is equal:

$$\frac{\Delta; \Psi \vdash \tau_1 \equiv \tau_1' : \kappa_1 \rightarrow \kappa \quad \Delta; \Psi \vdash \tau_2 \equiv \tau_2' : \kappa_1}{\Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa}$$

The other more typical congruence rules can then be derived.

The new rules are the *hypothesis* rule:

$$\frac{}{\Delta; \Psi, \epsilon : \kappa \vdash \epsilon : \kappa}\ ,$$

which allows us to use an assumed equation between two types to prove that the two types are equal, and the *decomposition rules* that invert the congruence rule of equational logic:

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa \quad \Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa}{\Delta; \Psi \vdash \tau_1 \equiv \tau_1' : \kappa_1 \rightarrow \kappa} \qquad \frac{\Delta \vdash \tau_2 : \kappa_1 \quad \Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa}{\Delta; \Psi \vdash \tau_2 \equiv \tau_2' : \kappa_1}$$

These rules are unusual and deserve further explanation.

In Haskell or ML, (mono)-types are drawn from a first-order language with "constants" such as *Int* and *Bool* and "function symbols" such as *List* a. Built-in type constructors such as function

[2] In Haskell, there are no *a priori* empty kinds, and types are syntactic objects, so a collection of equations among type constructors is satisfiable if and only if it has a unifier (i.e., there exists a type substitution Θ which makes all of the equations true). In the presence of an empty kind $\emptyset$, existence of a unifier does not imply validity, since $x = x : \emptyset$ is unifiable under the empty substitution but not satisfiable. In the presence of non-syntactic identities among ground types, such as *List* $(Int, Int) =$ *List* $(Bool, Int)$, satisfiability does not imply unifiability, since *List* $(\alpha, \alpha) =$ *List* $(Bool, Int)$ is satisfiable but does not syntactically unify. Fortunately Haskell's type system does not admit such identities, so unification is a *sound* and *complete* test for satisfiability of type equations.

space and pair constructions can also be thought of as (binary) function symbols in this language. Moreover, the intended interpretation of this language is syntactic. That is, no equations are assumed to hold among these types apart from syntactic equality. This fact is implicit in type inference: if we see a term of an unknown type used in integer addition, then we assume it must have type *Int*; we don't need to consider the possibility that there is some other type syntactically different from *Int* but semantically the same.

Pure equational logic (that is, deciding equality under no hypotheses using reflexivity, transitivity, symmetry, and congruence rules) is *complete* for syntactic theories. That is, all true single equations are derivable. However, this is not true when assumptions are added to the mix. For example, consider the assumption *List* α = *List* β. It is a fact that in the syntactic model, this equation only holds when $\alpha = \beta$ also holds. Thus, semantically $\alpha = \beta$ is a consequence of *List* α = *List* β. However, this cannot be derived using the pure equational rules, because the equational rules are sound in all models and there are non-syntactic models for which *List* α = *List* β does not imply $\alpha = \beta$.[3] Nevertheless, in the syntactic model the decomposition rules are sound, and they turn out to be both necessary for proving the adequacy of using unification during typechecking (see Section 5) and useful (see Section 4).

Most of the rules for expression typing are standard. The constructor typing rule is modified by adding the requirement that type constraints are checked:

$$\frac{\Delta; \Psi; \Gamma \vdash e_i : \sigma_i[\overline{\tau}'/\overline{\alpha}, \overline{\tau}/\overline{\beta}] \quad \Delta; \Psi \vdash \epsilon_i[\overline{\tau}'/\overline{\alpha}, \overline{\tau}/\overline{\beta}] : \kappa_i}{\Delta; \Psi; \Gamma \vdash C\ [\overline{\tau}]\ \overline{e} : T\ \overline{\tau}'}$$

where $\Sigma_{T.C\ \overline{\alpha}} = \exists\overline{\beta}.C\ \overline{\sigma}$ **with** $\overline{\epsilon}$. This rule states that a data constructor application typechecks if, in addition to the usual requirements, all of the type equations specified in the constructor's **with** clause are satisfied in the current context.

The case rule

$$\frac{\Delta; \Psi; \Gamma \vdash e : T\ \overline{\tau} \quad \Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma}{\Delta; \Psi; \Gamma \vdash \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms : \sigma}$$

simply checks that e is of a **data** type $T\ \overline{\tau}$ and passes the equation context Ψ on to pattern-match typechecking. The pattern-match typing rule introduces new type and variable bindings, and also looks up the equations associated with the current case, instantiates them with the arguments to T, and type checks the body under these added equational assumptions.

$$\frac{\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma' \quad \Delta, \overline{\beta}; \Psi, \overline{\epsilon}[\overline{\tau}/\overline{\alpha}, \overline{\gamma}/\overline{\beta}]; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}, \overline{\gamma}/\overline{\beta}] \vdash e : \sigma'}{\Delta; \Psi; \Gamma \vdash C\ [\overline{\gamma}]\ \overline{x} \rightarrow e\ |\ ms : T\ \overline{\tau} \Rightarrow \sigma'}$$

where $\Sigma_{T.C\ \overline{\alpha}} = \exists\overline{\beta}.C\ \overline{\sigma}$ **with** $\overline{\epsilon}$. Finally, we include a standard coercion rule:

$$\frac{\Delta; \Psi; \Gamma \vdash e : \sigma \quad \Delta; \Psi \vdash \sigma \equiv \sigma'}{\Delta; \Psi; \Gamma \vdash e : \sigma'}$$

4 Decomposition rules and applications

Type equivalence and unification are closely related. In Hindley-Milner-style type inference systems like Haskell, unification is used in a *bottom-up* fashion during type inference: type constraints are generated from a term, then unification is used to solve the constraints. For example, if we see a function application $f\ e$, then we generate a constraint of the form $\alpha = \beta \rightarrow \gamma$, where $f :: \alpha$, $e :: \beta$, and the result of the application has type γ. For an equation to typecheck, all such constraints must possess a single simultaneous solution, or *unifier*. Haskell 98 uses a slight generalization of first-order unification to solve equations like *Maybe Int* $= \phi\ \alpha$ for $\phi :: * \rightarrow *$ and $\alpha :: *$. In Haskell, the only solution is ϕ = *Maybe*, α = *Int*. Generally speaking, a term of the form $\tau_1\ \tau_2 = \sigma_1\ \sigma_2$ unifies if and only if both $\tau_1 = \sigma_1$ and $\tau_2 = \sigma_2$ do (with the same unifier), just as in the first-order case

[3] For example, consider the one-element model where every type is interpreted as that element.

with built-in functions. This is a form of sorted first-order unification, where the sorts are Haskell's kinds.

To take full advantage of the power of first-class phantom types, we need to employ *top-down* reasoning as well. That is, we need to be able to perform typechecking under temporary assumptions consisting of type equations. In Haskell, the language of types admits only syntactic equality (which is why syntactic unification suffices for type inference). However, some facts are true in the actual syntactic interpretation of types that are not true in general for equational theories. For example, in a general theory with a function symbol f, knowing $f\ x = f\ y$ does not imply $x = y$. (i.e., not every function is injective), so clearly this fact is not derivable from pure equational logic (that is, from the reflexive, symmetric, transitive, and congruence laws). But this is the case for types; for example, is we know $List\ a = List\ b$, then we can conclude that $a = b$. So pure equational logic is not complete for solving equations under hypotheses in Haskell.

The decomposition rules state that Haskell constructor application is injective. Put another way, these rules state that the usual congruence laws are *invertible* in Haskell: that is, from a proof of $\phi\ \alpha = \psi\ \beta$ we can always extract proofs of $\phi = \psi$ and $\alpha = \beta$. Using the decomposition rules, we can show that to typecheck code in the presence of hypothetical type equations, it suffices to find a unifier (most general solution) to the equations, apply the resulting substitution to the types in the environment, and proceed with typechecking as usual.

We have already seen more than one situation where the decomposition rule is useful: in *reduce* in the encoding of λ-terms in Section 2.2, and in *merge* in the functional encoding of generalized tries of Section 2.4. Generalized tries can be seen as an instance of **Typerec**, a type constructor found in intensionally polymorphic languages such as λ_i^{ML} and λ_R that defines a constructor by induction on the structure of another constructor. Thus, $\mathbf{Typerec}\ \tau\ \tau_{Int}\ \tau_\times$ is equivalent to τ_{Int} if $\tau = Int$, and $\tau_\times\ \tau_1\ \tau_2\ (\mathbf{Typerec}\ \tau_1\ \tau_{Int}\ \tau_\times)\ (\mathbf{Typerec}\ \tau_2\ \tau_{Int}\ \tau_\times)$ if $\tau = \tau_1 \times \tau_2$. Typical applications of **Typerec** include type-directed optimizations like automatic boxing and unboxing [19], array flattening, and automatic marshaling [10]. Using first-class phantom types, we can simulate **Typerec** as follows:

$$\begin{array}{lcll} \mathbf{data}\ Typerec\ \alpha\ \tau\ \phi\ \tau' & = & T_{Int}\ \tau & \mathbf{with}\ \alpha = Int \\ & | & T_\times\ (\phi\ \beta\ \gamma\ (Typerec\ \beta\ \tau\ \phi)\ (Typerec\ \gamma\ \tau\ \phi)) & \mathbf{with}\ \alpha = (\beta, \gamma) \end{array}$$

Without the decomposition rule, this form of *Typerec* would have limited usefulness. Consider as a simple example the following type function defined using *Typerec*:

$$\begin{array}{lcl} \mathbf{type}\ RevF\ \alpha\ \beta\ \gamma\ \delta & = & (\delta, \gamma) \\ \mathbf{type}\ Rev\ \alpha & = & Typerec\ \alpha\ Int\ RevF \end{array}$$

We'd like to define generic transformations from α to $Rev\ \alpha$ and back. As a first step, we show simple cases of constructing $Rev\ \alpha$ values from components:

$$\begin{array}{lcl} idInt & :: & Int \to Rev\ Int \\ idInt\ i & = & T_{Int}\ i \\ idPair & :: & (Rev\ \beta, Rev\ \gamma) \to Rev\ (\beta, \gamma) \\ idPair\ (b, c) & = & T_\times\ (c, b) \end{array}$$

Now we'd like to define reverse transformations as follows:

$$\begin{array}{lcl} unInt & :: & Rev\ Int \to Int \\ unInt\ (T_{Int}\ i) & = & i \\ unPair & :: & Rev\ (\beta, \gamma) \to (Rev\ \beta, Rev\ \gamma) \\ unPair\ (T_\times\ (c, b)) & = & (b, c) \end{array}$$

The *unInt* function typechecks even without decompositions, since we already know that $Int = Int$. But without the decomposition rules, the pair case does not typecheck, because the type of p is $(Rev\ \beta', Rev\ \gamma')$ for some existentially quantified β', γ' such that $(\beta, \gamma) = (\beta', \gamma')$, while the desired type is $(Rev\ \beta, Rev\ \gamma)$ with β, γ universally quantified. Without decompositions, we cannot conclude that $\beta = \beta'$, $\gamma = \gamma'$ or that $(Rev\ \beta, Rev\ \gamma) = (Rev\ \beta', Rev\ \gamma')$, as needed. Thus, phantom types are

not only a nice convenience, they are also strictly more powerful than the approach using equality types.

However, even with decomposition this encoding of **Typerec** is not ideal. Our encoding uses a datatype, which stores a tag along with the type-dependent data in each case. On one hand, when we know what type α is, the tag is redundant. On the other hand, when no other information about α is known, the tag can be used to disambiguate the two cases. The "standard" **Typerec** does not exhibit this behavior: in the absence of a representation for α, **Typerec** $\alpha\ \tau_i\ \tau_\times$ is an abstract type. We believe that a **union** construct may be a viable way to handle this behavior:

$$\begin{array}{lcl} \textbf{union}\ \mathit{Typerec}\ \alpha\ \tau\ \phi & = & T_{\mathit{Int}}\ \tau\ \textbf{with}\ \alpha = \mathit{Int} \\ & | & T_\times\ (\phi\ \beta\ \gamma\ (\mathit{Typerec}\ \beta\ \tau\ \phi)\ (\mathit{Typerec}\ \gamma\ \tau\ \phi)) \\ & & \textbf{with}\ \alpha = (\beta, \gamma) \end{array}$$

Unions are like standard phantom datatypes, except that the run-time datatype case tags are omitted. In a context where one of the equations is satisfied, a union is treated as a synonym for the matching body; otherwise, it is treated as an abstract type. Thus, for unions, the equations provide the *only* way of disambiguating the cases. As a result, to avoid confusion, the type equations in each case must be mutually exclusive. We have not studied this construct in detail, but plan to do so in the future.

5 Type Soundness and Decidability

We now establish the type soundness and decidability of $\lambda_{\equiv}$. In the rest of this section, we give detailed proofs of these results, emphasizing the differences relative to proofs for similar systems.

5.1 Properties of Equivalence Derivations

We first prove some general substitution lemmas

Lemma 1 (Type Substitution). *If* $\Delta \vdash \tau : \kappa$ *and*

1. $\Delta, \alpha{:}\kappa \vdash \tau' : \kappa'$ *then* $\Delta[\tau/\alpha] \vdash \tau'[\tau/\alpha] : \kappa'$
2. $\Delta, \alpha{:}\kappa; \Psi \vdash \epsilon' : \kappa'$ *then* $\Delta[\tau/\alpha]; \Psi[\tau/\alpha] \vdash \epsilon'[\tau/\alpha] : \kappa'$
3. $\Delta, \alpha{:}\kappa; \Psi; \Gamma \vdash e : \sigma$ *then* $\Delta[\tau/\alpha]; \Psi[\tau/\alpha]; \Gamma[\tau/\alpha] \vdash e[\tau/\alpha] : \sigma[\tau/\alpha]$
4. $\Delta, \alpha{:}\kappa; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$ *then* $\Delta; \Psi[\tau/\alpha]; \Gamma[\tau/\alpha] \vdash ms[\tau/\alpha] : T\ \overline{\tau}[\tau/\alpha] \Rightarrow \sigma[\tau/\alpha]$

Proof. Standard. □

Lemma 2 (Weakening). *If* $\Delta; \Psi \vdash \epsilon : \kappa$ *and* $\Delta \vdash \tau_i : \kappa'$ *then* $\Delta; \Psi, \tau_1 \equiv \tau_2 \vdash \epsilon : \kappa'$. *If* $\Delta; \Psi \vdash \epsilon : \kappa$ *then* $\Delta, \alpha; \Psi \vdash \epsilon : \kappa$.

Proof. Both are simple inductions on the structure of the respective derivations. □

Lemma 3 (Syntactic Equality). *If* $\Delta \vdash \tau_i : \kappa$, *then* $\Delta; \cdot \vdash \tau_1 \equiv \tau_2 : \kappa$ *if and only if* $\tau_1 = \tau_2$. *If* $\Delta \vdash \sigma_i$, *then* $\Delta; \cdot \vdash \sigma_1 = \sigma_2$ *if and only if* $\sigma_1 =_\alpha \sigma_2$.

Proof. Easy induction on the structure of equivalence derivations. □

Definition 1. *A type constructor context is a constructor with a "hole"* $[\cdot]$ *given by the following grammar:*

$$C ::= [\cdot] \mid C\ \tau \mid \tau\ C$$

Lemma 4 (Constructor Congruence). *If* $\Delta; \Psi \vdash \tau \equiv \tau' : \kappa$ *and* $\Delta, \alpha{:}\kappa \vdash C[\alpha] : \kappa'$ *then* $\Delta; \Psi \vdash C[\tau] \equiv C[\tau'] : \kappa$.

Proof. Simple induction on the structure of C. □

Lemma 5. *If $\alpha \notin \Delta$ and $\Psi, \alpha = \tau$ is satisfiable and $\Delta \vdash \tau : \kappa'$ then $\Delta; \Psi[\tau/\alpha] \vdash \epsilon[\tau/\alpha] : \kappa$ if and only if $\Delta, \alpha{:}\kappa'; \Psi, \alpha \equiv \tau \vdash \epsilon : \kappa$.*

Proof. ($\Longrightarrow$): Proof is by induction on the number of occurrences of α in Ψ, ϵ. If there are none, then we have $\Delta; \Psi \vdash \epsilon : \kappa$, and by weakening we can conclude that $\Delta, \alpha; \Psi, \alpha \equiv \tau \vdash \epsilon : \kappa$. Suppose that the lemma holds for Ψ, ϵ with n occurrences of α and suppose we have Ψ, ϵ with $n+1$ occurrences. Pick one.

If it is in Ψ, without loss of generality assume $\Psi = \Psi', C[\alpha] \equiv \tau'$. By induction we can get $\Delta; \Psi, C[\tau] \equiv \tau', \alpha \equiv \tau \vdash \epsilon : \kappa$. We need to show that $\Delta; \Psi, C[\alpha] \equiv \tau', \alpha \equiv \tau \vdash \epsilon : \kappa$. But we can derive both $\Delta; \Psi, C[\alpha] \equiv \tau', \alpha \equiv \tau \vdash C[\tau] \equiv C[\alpha] : \kappa$ and $\Delta; \Psi, C[\alpha] \equiv \tau', \alpha \equiv \tau \vdash C[\alpha] \equiv \tau' : \kappa$ so by transitivity we have $\Delta; \Psi, C[\alpha] \equiv \tau', \alpha \equiv \tau \vdash C[\tau] \equiv \tau' : \kappa$. Using the Substitution Lemma we can conclude $\Delta; \Psi, C[\alpha] \equiv \tau', \alpha \equiv \tau \vdash \epsilon : \kappa$.

If the instance of α is in ϵ, without loss of generality assume $\epsilon = C[\alpha] \equiv \tau$. Then by induction we have $\Delta; \Psi, \alpha \equiv \tau \vdash C[\tau] \equiv \tau' : \kappa$. Then using context congruence we can derive $\Delta; \Psi, \alpha \equiv \tau \vdash C[\alpha] \equiv C[\tau] : \kappa$, so by transitivity $\Delta; \Psi, \alpha \equiv \tau \vdash C[\alpha] \equiv \tau' : \kappa$.

Note that since $\Psi, \alpha = \tau$ is satisfiable, we know that α does not occur in τ. This fact is crucial for the induction step, since it guarantees that there are fewer occurrences of α in $C[\tau]$ than $C[\alpha]$.

($\Longleftarrow$): Proof by induction on the structure of the derivation $\Delta, \alpha{:}\kappa'; \Psi, \alpha \equiv \tau \vdash \epsilon : \kappa$.

- Case (Hyp): If the hypothesis is $\alpha \equiv \tau$, then the desired judgment is $\Delta; \Psi[\tau/\alpha] \vdash \tau \equiv \tau : \kappa$, which follows by reflexivity.

  If the hypothesis used is some other hypothesis ϵ, then the hypothesis rule still applies to derive $\Delta; \Psi, \epsilon[\tau/\alpha] \vdash \epsilon[\tau/\alpha] : \kappa$

- Case (Refl): The desired judgment is also derivable by reflexivity.

- Case (Symm, Trans, Cong, Decomp): The desired judgment is derivable by applying the respective rules to the results of applying the induction hypothesis to their subderivations.

□

Corollary 1. *If $\Delta \vdash \Theta$ then $\Delta; \Theta \vdash \tau_1 = \tau_2 : \kappa$ if and only if $\Delta; \cdot \vdash \Theta\tau_1 = \Theta\tau_2 : \kappa$.*

Proof. Follows by induction on the size of Θ from Lemma 5. □

Lemma 6. *If $\Delta \vdash \Psi \Downarrow \Theta$ and $\Delta; \Theta \vdash \epsilon : \kappa$ then $\Delta; \Psi \vdash \epsilon : \kappa$.*

Proof. Proof by induction on the structure of the derivation of $\Delta \vdash \Psi \Downarrow \Theta$.

- If
$$\frac{}{\cdot \vdash \cdot \Downarrow},$$
then $\Psi = \cdot$, $\Theta = \cdot$, so $\Delta; \cdot \vdash \epsilon : \kappa$ holds by assumption.

- If
$$\frac{\Delta \vdash \Psi' \Downarrow \Theta}{\Delta \vdash \Psi', \alpha \equiv \alpha \Downarrow \Theta}$$
then Θ is a MGU for Ψ' also, so $\Delta; \Psi' \vdash \epsilon : \kappa$. By weakening we have $\Delta; \Psi', \alpha \equiv \alpha \vdash \epsilon : \kappa$.

- If
$$\frac{\Delta \vdash \Psi' \Downarrow \Theta}{\Delta \vdash \Psi', c \equiv c \Downarrow \Theta}$$
then the argument is similar to the last case.

- If
$$\frac{\Delta \vdash \Delta : \tau\kappa \quad \Delta \vdash \Psi'[\tau/\alpha] \Downarrow \Theta'}{\Delta, \alpha{:}\kappa \vdash \Psi', \alpha \equiv \tau \Downarrow \Theta', \alpha \equiv \tau}$$
then we have $\Delta; \Theta', \alpha \equiv \tau \vdash \epsilon : \kappa$ and need to show that $\Delta; \Psi', \alpha \equiv \tau \vdash \epsilon : \kappa$. By the reverse direction of Lemma 5, we have $\Delta; \Theta'[\tau/\alpha] \vdash \epsilon[\tau/\alpha] : \kappa$. But α does not occur in Θ, so $\Delta; \Theta'[\tau/\alpha] \vdash \epsilon[\tau/\alpha] : \kappa$. By induction since $\Delta \vdash \Psi'[\tau/\alpha] \Downarrow \Theta$ we can derive $\Delta; \Psi'[\tau/\alpha] \vdash \epsilon[\tau/\alpha] : \kappa$. By Lemma 5 in the forward direction, we can conclude $\Delta; \Psi', \alpha \equiv \tau \vdash \epsilon : \kappa$.

- If
$$\frac{\Delta \vdash \Psi', \alpha \equiv \tau \Downarrow \Theta \quad \tau \neq \beta}{\Delta \vdash \Psi', \tau \equiv \alpha \Downarrow \Theta}$$
then we proceed by induction.

- If
$$\frac{\Delta \vdash \Psi', \tau_1 \equiv \tau_1', \tau_2 \equiv \tau_2' \Downarrow \Theta}{\Delta \vdash \Psi', \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' \Downarrow \Theta}$$
then by induction we have $\Delta; \Psi', \tau_1 \equiv \tau_1', \tau_2 \equiv \tau_2' \vdash \epsilon : \kappa$. Moreover,
$$\frac{\Delta \vdash \tau_2 : \kappa_1 \quad \overline{\Delta; \Psi', \tau_1 \equiv \tau_1', \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa_2}}{\Delta; \Psi', \tau_1 \equiv \tau_1', \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' \vdash \tau_2 \equiv \tau_2' : \kappa_1}$$
so by the substitution lemma (proved in the next section) we have $\Delta; \Psi', \tau_1 \equiv \tau_1', \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' \vdash \epsilon : \kappa$. Similarly,
$$\frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad \overline{\Delta; \Psi', \tau_1 \equiv \tau_1', \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa_2}}{\Delta; \Psi', \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' \vdash \tau_1 \equiv \tau_1' : \kappa_1 \to \kappa_2}$$
so by substitution again we have $\Delta; \Psi', \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' \vdash \epsilon : \kappa$

□

Proposition 1. *Assume* $\Delta \vdash \tau_i : \kappa$ *and* $\Delta \vdash \Psi \Downarrow \Theta$ *(i.e.,* Θ *unifies* Ψ*). Then* $\Delta; \Psi \vdash \tau_1 \equiv \tau_2 : \kappa$ *if and only if* $\Theta\tau_1 = \Theta\tau_2$.

Proof. ($\Longrightarrow$): Proof is by induction on the derivation of $\Delta; \Psi \vdash \tau_1 \equiv \tau_2 : \kappa$.

- Case (Hyp). Then
$$\overline{\Delta; \Psi, \tau \equiv \tau' \vdash \tau \equiv \tau' : \kappa}$$
Then $\Theta\tau = \Theta\tau'$ since Θ unifies every equation in $\Psi, \tau \equiv \tau'$.

- Case (Refl). Then
$$\overline{\Delta; \Psi \vdash \tau \equiv \tau : \kappa}$$
Then $\Theta\tau = \Theta\tau$.

- Case (Symm). Then
$$\frac{\Delta; \Psi \vdash \tau' \equiv \tau : \kappa}{\Delta; \Psi \vdash \tau \equiv \tau' : \kappa}$$
Then by the induction hypothesis we have $\Theta\tau' = \Theta\tau$ so by symmetry of equality $\Theta\tau = \Theta\tau'$.

- Case (Trans). Then
$$\frac{\Delta; \Psi \vdash \tau \equiv \tau' : \kappa \quad \Delta; \Psi \vdash \tau' \equiv \tau'' : \kappa}{\Delta; \Psi \vdash \tau \equiv \tau'' : \kappa}$$
By induction hypothesis we have $\Theta\tau = \Theta\tau'$ and $\Theta\tau' = \Theta\tau''$, so $\Theta\tau = \Theta\tau''$.

- Case (Congr). Then
$$\frac{\Delta; \Psi \vdash \tau_1 \equiv \tau_1' : \kappa \quad \Delta; \Psi \vdash \tau_2 \equiv \tau_2' : \kappa}{\Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa}$$
By induction hypothesis we have $\Theta\tau_1 = \Theta\tau_1'$, $\Theta\tau_2 = \Theta\tau_2'$, so $\Theta(\tau_1\ \tau_2) = \Theta\tau_1\ \Theta\tau_2 = \Theta\tau_1'\ \Theta\tau_2' = \Theta(\tau_1'\ \tau_2')$.

- Case (Decomp1). Then
$$\frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad \Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa_2}{\Delta; \Psi \vdash \tau_1 \equiv \tau_1' : \kappa_1 \to \kappa_2}$$
By induction on the first hypothesis, we have $\Theta(\tau_1\ \tau_2) = \Theta(\tau_1'\ \tau_2')$, so $\Theta(\tau_1) = \Theta(\tau_1')$.

- Case (Decomp2). Then
$$\frac{\Delta \vdash \tau_2 : \kappa_1 \quad \Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa_2}{\Delta; \Psi \vdash \tau_2 \equiv \tau_2' : \kappa_1}$$
By induction on the first hypothesis, we have $\Theta(\tau_1\ \tau_2) = \Theta(\tau_1'\ \tau_2')$, so $\Theta(\tau_2) = \Theta(\tau_2')$.

($\Longleftarrow$): By Lemma 3, from $\Delta \vdash \Psi \Downarrow \Theta$ and $\Theta\tau = \Theta\tau'$ we can obtain $\Delta; \cdot \vdash \Theta\tau \equiv \Theta\tau' : \kappa$. By Corollary 1, we have $\Delta; \Theta \vdash \tau \equiv \tau' : \kappa$. By Lemma 6, we conclude $\Delta; \Psi \vdash \tau \equiv \tau' : \kappa$. □

The proofs of these properties are straightforward.

5.2 Type Soundness

The proof of type soundness is similar in structure to those for related systems λ_i^{ML} [21] and λ_R [7]. We already showed substitution lemmas for substituting types into various judgments. We prove additional equation and term substitution lemmas. We also show that typed values have canonical forms even with our stronger notion of type equivalence. Next we show that every typed term is either a value or can make progress, and finally we show that typedness is preserved by evaluation.

Lemma 7 (Equation Substitution). *If* $\Delta; \Psi \vdash \epsilon : \kappa$ *and*

1. $\Delta; \Psi, \epsilon \vdash \epsilon' : \kappa'$ *then* $\Delta; \Psi \vdash \epsilon' : \kappa'$
2. $\Delta; \Psi, \epsilon; \Gamma \vdash e : \sigma$ *then* $\Delta; \Psi; \Gamma \vdash e : \sigma$
3. $\Delta; \Psi, \epsilon; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$ *then* $\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$

Proof. The cases for typing derivations are straightforward. We show the interesting cases for equivalence derivations.

- (Hypothesis) If the hypothesis used is $\epsilon : \kappa$, then by assumption we already have a derivation of $\Delta; \Psi \vdash \epsilon : \kappa$. If it is some other hypothesis,
$$\frac{}{\Delta; \Psi, \epsilon', \epsilon \vdash \epsilon' : \kappa'}$$
then the hypothesis rule still applies without ϵ:
$$\frac{}{\Delta; \Psi, \epsilon' \vdash \epsilon' : \kappa'}$$
as desired.

- (Reflexivity) We have $\epsilon = (\tau \equiv \tau)$ and
$$\frac{}{\Delta; \Psi, \epsilon \vdash \tau \equiv \tau : \kappa'}$$
Then
$$\frac{}{\Delta; \Psi \vdash \tau \equiv \tau : \kappa'}$$

- (Symmetry) We have $\epsilon = (\tau \equiv \tau')$ and

$$\frac{\Delta; \Psi, \epsilon \vdash \tau' \equiv \tau : \kappa'}{\Delta; \Psi, \epsilon \vdash \tau \equiv \tau' : \kappa'}$$

  Then by induction we have $\Delta; \Psi \vdash \tau' \equiv \tau : \kappa$ so we conclude $\Delta; \Psi \vdash \tau \equiv \tau' : \kappa$ using the symmetry rule.

- (Trans, Congr, Decomp) Straightforward induction steps

□

Lemma 8 (Term Substitution). *If $\Delta; \Psi; \Gamma \vdash e : \tau$ and $\Delta; \Psi; \Gamma, x : \tau \vdash e' : \sigma$ then $\Delta; \Psi; \Gamma \vdash e'[e/x] : \sigma$.*

Lemma 9 (Canonical Forms). *Suppose v is a value. If $v \neq$ **fail**, $\cdot;\cdot;\cdot \vdash v : \sigma$ and*

1. *$\cdot;\cdot;\cdot \vdash \sigma \equiv Int$, then $v = i$.*
2. *$\cdot;\cdot;\cdot \vdash \sigma \equiv \sigma_1 \to \sigma_2$, then $v = \lambda x : \sigma_1.e'$ for some x, e'.*
3. *$\cdot;\cdot;\cdot \vdash \sigma \equiv \sigma_1 \times \sigma_2$, then $v = \langle e_1, e_2 \rangle$ for some e_1, e_2.*
4. *$\cdot;\cdot;\cdot \vdash \sigma \equiv T\ \overline{\tau}$, then $v = C\ [\overline{\tau}']\ \overline{e}$ for some τ_i', e_i.*
5. *$\cdot;\cdot;\cdot \vdash \sigma \equiv \forall \alpha{:}\kappa.\sigma'$, then $v = \Lambda\alpha{:}\kappa.e$ for some e.*

Proof. Proof is by induction on the number of uses of the equality rule before we get to a non-equality rule at the head of the derivation. If there are no uses of the equality rule, then by Lemma 3, $\sigma = Int, \sigma_1 \to \sigma_2, \sigma_1 \times \sigma_2$, etc., respectively, so only the appropriate rule introducing i, $\lambda x{:}\sigma_1.e$, $\langle e_1, e_2 \rangle$, etc. is applicable. If there are $n+1$ uses of the equality rule, then we combine the last use with the assumption using transitivity and proceed by induction.

□

Definition 2. *We write $\gamma : \Gamma$ to indicate that γ is a function mapping variables bound in Γ to values such that $\cdot;\cdot;\cdot \vdash \gamma(x) : \Gamma(x)$, and similarly $\delta : \Delta; \Psi$ if δ maps type variables bound in Δ to type constructors of the appropriate kinds such that each equation in Ψ is satisfied; that is, $\cdot \vdash \delta(\alpha) : \Delta(\alpha)$ and $\cdot;\cdot \vdash \delta(\tau) \equiv \delta(\tau') :$ for each $\tau \equiv \tau' \in \Psi$.*

We write $\delta(\sigma)$ and $\gamma\delta(e)$ for the result of substituting all the type or term variables in a type or expression with their values in γ or δ.

Lemma 10 (Progress). *If $\cdot;\cdot;\cdot \vdash e : \tau$ then either e is a value or there exists e' such that $e \mapsto e'$.*

This proof is essentially the same as the standard proof, since equations have no run-time representation or effect.

Proof. We generalize the induction hypothesis to: If $\Delta; \Psi; \Gamma \vdash e : \tau$ and $\gamma : \Gamma$, $\delta : \Delta; \Psi$ then either $\gamma\delta(e)$ is a value or there exists e' such that $\gamma\delta(e) \mapsto \gamma\delta(e')$. First, note that if $\gamma\delta(e) = E[\textbf{fail}]$, then $\gamma\delta(E) \mapsto \textbf{fail} = \gamma\delta(\textbf{fail})$. So we may without loss assume that **fail** does not occur as part of the head redex in e.

Proof is by induction on the derivation of $\Delta; \Psi; \Gamma \vdash e : \sigma$. Most of the cases are standard. The only unusual case is for $\textbf{case}[\sigma]\ e\ \textbf{of}\ ms$ expressions.

- (Case) Then $e = \textbf{case}[\sigma]\ e'\ \textbf{of}\ ms$ and we have

$$\frac{\Delta; \Psi; \Gamma \vdash e' : T\ \overline{\tau} \quad \Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma}{\Delta; \Psi; \Gamma \vdash \textbf{case}[\sigma]\ e'\ \textbf{of}\ ms : \sigma}$$

  Case expressions can never be values, so we must find an evaluation step to take. If $\gamma\delta(e')$ is not a value, then by induction using the derivation $\Delta; \Psi; \Gamma \vdash e' : T\ \overline{\tau}$ we can find e'' such that

$\gamma\delta(e') \mapsto \gamma\delta(e'')$, and thus $\gamma\delta(\mathbf{case}[\sigma]\ e'\ \mathbf{of}\ ms) \mapsto \gamma\delta(\mathbf{case}[\sigma]\ e''\ \mathbf{of}\ ms)$. If $\gamma\delta(e')$ is a value, then by Lemma 9 it must be of the form $C\ [\overline{\tau}']\ \overline{e}$ since it is not **fail** and values of type $T\ \overline{\tau}$ consist only of constructors applied to values of the appropriate types. There are three cases. If $ms = \cdot$ then the expression steps to

$$\begin{array}{rcl} \gamma\delta(\mathbf{case}[\sigma]\ e'\ \mathbf{of}\ \cdot) & = & \mathbf{case}[\delta(\sigma)]\ \gamma\delta(e')\ \mathbf{of}\ \cdot \\ & \mapsto & \mathbf{fail} \\ & = & \gamma\delta(\mathbf{fail}) \end{array}$$

If $ms = D\ [\overline{\beta}]\ \overline{x} \to e_D \mid ms'$ where $C \neq D$ then we have

$$\begin{array}{rcl} \gamma\delta(\mathbf{case}[\sigma]\ e'\ \mathbf{of}\ ms) & = & \mathbf{case}[\delta(\sigma)]\ \gamma\delta(e')\ \mathbf{of}\ \gamma\delta(ms) \\ & \mapsto & \mathbf{case}[\delta(\sigma)]\ \gamma\delta(e')\ \mathbf{of}\ \gamma\delta(ms') \\ & = & \gamma\delta(\mathbf{case}[\sigma]\ v\ \mathbf{of}\ ms') \end{array}$$

Otherwise, $ms = C\ [\overline{\beta}]\ \overline{x} \to e_C \mid ms'$

$$\begin{array}{rcl} \gamma\delta(\mathbf{case}[\sigma]\ e'\ \mathbf{of}\ ms) & = & \mathbf{case}[\delta(\sigma)]\ \gamma\delta(e')\ \mathbf{of}\ \gamma\delta(ms) \\ & = & \mathbf{case}[\delta(\sigma)]\ C\ [\delta(\overline{\tau}')]\ \gamma\delta(\overline{e})\ \mathbf{of}\ \gamma\delta(ms) \\ & \mapsto & \gamma\delta(e_C)[\delta(\overline{\tau}')/\overline{\beta}, \gamma\delta(\overline{e})/\overline{x}] \\ & = & \gamma\delta(e_C[\overline{\tau}'/\overline{\beta}, \overline{e}/\overline{x}]) \end{array}$$

since $\overline{\beta}, \overline{x}$ are not in the domains of δ, γ.

□

Lemma 11 (Subject Reduction). *If* $\cdot;\cdot;\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot;\cdot;\cdot \vdash e' : \tau$.

The only unusual part of this proof is for $\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ C\ [\overline{\beta}]\ \overline{x} \to e \mid ms$ expressions, where we need to apply the equation substitution lemma as well as the type and term substitution lemmas in order to prove that $e[\overline{\tau}/\overline{\beta}, \overline{e}/\overline{v}]$ still typechecks.

Proof. We strengthen the induction hypothesis to: If $\Delta;\Psi;\Gamma \vdash e : \tau$ and $\delta : \Delta;\Psi$, and $\gamma : \Gamma$, and $\gamma\delta(e) \mapsto \gamma\delta(e')$ then $\Delta;\Psi;\Gamma \vdash e' : \tau$.

First note that if $\gamma\delta(e) \mapsto \mathbf{fail}$ then the conclusion follows immediately since $\mathbf{fail} = \gamma\delta(\mathbf{fail})$ inhabits any type. Otherwise, $\gamma\delta(e) = E[\gamma\delta(e')]) \mapsto E[\gamma\delta(e'')]$ for some $\gamma\delta(e') \mapsto \gamma\delta(e'')$. Since evaluation contexts preserve typing, it suffices to consider only the redex cases. The only interesting nonstandard case is that of $\mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms$ expressions.

- (Case) Then $e = \mathbf{case}[\sigma]\ C\ [\overline{\tau}']\ \overline{e'}\ \mathbf{of}\ ms$. We have

$$\dfrac{\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Delta;\Psi;\Gamma \vdash e_i : \sigma_i[\overline{\tau}/\overline{\alpha}, \overline{\tau}'/\overline{\beta}] \quad \Delta;\Psi \vdash \epsilon_i[\overline{\tau}'/\overline{\beta}] : \kappa_i} \quad \mathcal{D}_3}{\Delta;\Psi;\Gamma \vdash C\ [\overline{\tau}']\ \overline{e} : T\ \overline{\tau} \qquad \Delta;\Psi;\Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma}{\Delta;\Psi;\Gamma \vdash \mathbf{case}[\sigma]\ C\ [\overline{\tau}']\ \overline{e}\ \mathbf{of}\ ms : \sigma}$$

  If $ms = \cdot$, then

$$\gamma\delta(\mathbf{case}[\sigma]\ C\ [\overline{\tau}']\ \overline{e}\ \mathbf{of}\ \cdot) = \mathbf{case}[\delta(\sigma)]\ C\ [\delta(\overline{\tau}')]\ \gamma\delta(\overline{e})\ \mathbf{of}\ \cdot \mapsto \mathbf{fail}\,,$$

  and

$$\frac{}{\Delta;\Psi;\Gamma \vdash \mathbf{fail} : \sigma}\,.$$

  If $ms = D\ [\overline{\beta}']\ \overline{x}' \to e_D \mid ms'$ where $C \neq D$, then

$$\begin{array}{rcl} \gamma\delta(\mathbf{case}[\sigma]\ C\ [\overline{\tau}']\ \overline{e}\ \mathbf{of}\ ms) & = & \mathbf{case}[\delta(\sigma)]\ C\ [\delta(\overline{\tau}')]\ \gamma\delta(\overline{e})\ \mathbf{of}\ \gamma\delta(ms) \\ & \mapsto & \mathbf{case}[\delta(\sigma)]\ C\ [\delta(\overline{\tau}')]\ \gamma\delta(\overline{e})\ \mathbf{of}\ \gamma\delta(ms) \\ & = & \gamma\delta(\mathbf{case}[\sigma]\ C\ [\overline{\tau}']\ \overline{e}\ \mathbf{of}\ ms')\,. \end{array}$$

Also, $\mathcal{D}_3$ is of the form

$$\frac{\overset{\mathcal{D}_4}{\Delta;\Psi;\Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma} \quad \overset{\mathcal{D}_5}{\Delta, \overline{\beta}; \Psi, \overline{\epsilon}; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}] \vdash e_D : \sigma}}{\Delta;\Psi;\Gamma \vdash D\ [\overline{\beta}]\ \overline{x} \to e_D \mid ms' : T\ \overline{\tau} \Rightarrow \sigma}$$

so

$$\frac{\dfrac{\overset{\mathcal{D}_1}{\Delta;\Psi;\Gamma \vdash e_i : \sigma_i[\overline{\tau}/\overline{\alpha}, \overline{\tau}'/\overline{\beta}]} \quad \overset{\mathcal{D}_2}{\Delta;\Psi \vdash \epsilon_i[\overline{\tau}'/\overline{\beta}] : \kappa_i}}{\Delta;\Psi;\Gamma \vdash C\ [\overline{\tau}']\ \overline{e} : T\ \overline{\tau}} \quad \overset{\mathcal{D}_4}{\Delta;\Psi;\Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma}}{\Delta;\Psi;\Gamma \vdash \mathbf{case}[\sigma]\ C\ [\overline{\tau}']\ \overline{e}\ \mathbf{of}\ ms' : \sigma}$$

Otherwise, $ms = C\ [\overline{\beta}']\ \overline{x}' \to e_C \mid ms'$, and

$$\begin{array}{rcl} \gamma\delta(\mathbf{case}[\sigma]\ C\ [\overline{\tau}']\ \overline{e}\ \mathbf{of}\ ms) & = & \mathbf{case}[\delta(\sigma)]\ C\ [\delta(\overline{\tau}')]\ \gamma\delta(\overline{e})\ \mathbf{of}\ \gamma\delta(ms) \\ & \mapsto & \gamma\delta(e_C)[\delta(\overline{\tau}')/\overline{\beta}, \gamma\delta(\overline{e})/\overline{x}] \\ & = & \gamma\delta(e_C[\overline{\tau}'/\overline{\beta}, \overline{e}/\overline{x}]) \end{array}$$

We also have

$$\frac{\overset{\mathcal{D}_4}{\Delta;\Psi;\Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma} \quad \overset{\mathcal{D}_5}{\Delta, \overline{\beta}; \Psi, \overline{\epsilon}; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}] \vdash e_C : \sigma}}{\Delta;\Psi;\Gamma \vdash C\ [\overline{\beta}]\ \overline{x} \to e_C \mid ms' : T\ \overline{\tau} \Rightarrow \sigma}$$

where $\overline{\beta}$ are not free in Δ, Ψ, Γ, or σ. We first apply the type substitution lemma to substitute $\overline{\tau}'$ for $\overline{\beta}$ in the derivation $\mathcal{D}_5 :: \Delta, \overline{\beta}; \Psi, \overline{\epsilon}; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}] \vdash e_C : \sigma$. Since $\overline{\beta}$ do not occur in Δ, Ψ, Γ, or σ, the result of the type substitution lemma is

$$\Delta; \Psi, \overline{\epsilon}[\overline{\tau}'/\overline{\beta}]; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}, \overline{\tau}'/\overline{\beta}] \vdash e'[\overline{\tau}'/\overline{\beta}] : \sigma$$

Now we apply the equation substitution lemma to substitute the derivations of $\Delta; \Psi \vdash \epsilon_i[\overline{\tau}'/\overline{\beta}] : \kappa_i$ (obtained from the constructor typing derivation) in the above derivation, yielding

$$\Delta; \Psi; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}, \overline{\tau}'/\overline{\beta}] \vdash e'[\overline{\tau}'/\overline{\beta}] : \sigma$$

Finally, we use the term substitution lemma with to substitute the derivations of $\Delta; \Psi; \Gamma \vdash e_i : \sigma_i[\overline{\tau}/\overline{\alpha}, \overline{\tau}'/\overline{\beta}]$ into the above derivation, yielding

$$\Delta; \Psi; \Gamma \vdash e'[\overline{e}/\overline{x}, \overline{\tau}'/\overline{\beta}] : \sigma$$

as desired.

□

Theorem 1 (Type Safety). *If $\cdot;\cdot;\cdot \vdash e : \tau$ and $e \mapsto^* e'$ then e' is not stuck; that is, e' is a value or $e' \mapsto e''$ for some e''.*

Proof. Follows from the preservation and progress lemmas. By induction on the number of steps from e to e', we can use subject reduction to prove that $\cdot;\cdot;\cdot \vdash e' : \sigma$; by preservation, either e' is a value or there exists e'' such that $e' \mapsto e''$. □

5.3 Decidability

Now we turn to the question of the decidability of typechecking for $\lambda_{\equiv}$. Type equivalences are a special case of general equational implication, which is undecidable. However, we showed earlier that if Ψ is satisfiable, then $\Delta; \Psi \vdash \tau \equiv \tau' : \kappa$ if any only if the most general unifier Θ of Ψ satisfies $\Theta\tau = \Theta\tau'$. As a result, constructor and type equivalence are decidable provided the equational context unifies. We say a derivation is *satisfiable* if all contexts in it are satisfiable.

On the other hand, type equivalence problems in which Ψ is unsatisfiable are not decidable, because we can encode undecidable equational inference problems as equivalence problems. Even if general type equivalence were decidable, syntax-directed typechecking with general type equivalences would be hard, as there is no obvious choice of "natural" normal forms for types relative to an unsatisfiable context. We believe that it is best to avoid this problem by requiring all equation contexts that arise during typechecking to be satisfiable. This is a decidable property: we can check it by checking that the equational context that obtains at each program point unifies. Unsatisfiable contexts can only help to typecheck dead code, since the contexts can never be satisfied. So, if typechecking fails for this reason, then the user just needs to remove the dead code to get an equivalent program that does typecheck.

Definition 3. *If $\Delta \vdash \Psi \Downarrow \Theta$, then we say that σ has normal form $\mathsf{NF}_{\Delta;\Psi}(\sigma) = \Theta\sigma$. When Δ and Ψ are obvious from context, we omit the subscript.*

A derivation is satisfiable if every context Ψ in it satisfiable. A judgment is satisfiably derivable if it has a satisfiable derivation.

A typing derivation is normal if it is of the form

$$\frac{\dfrac{\mathcal{D}_1 \ \cdots \ \mathcal{D}_n}{\Delta; \Psi; \Gamma \vdash e : \sigma}\ R \qquad \Delta; \Psi \vdash \sigma \equiv \sigma'}{\Delta; \Psi; \Gamma \vdash e : \sigma'}$$

where Ψ is satisfiable, σ' is in normal form, R is an axiom or inference rule other than coercion, and $\mathcal{D}_1 \ldots, \mathcal{D}_n$ are normal derivations.

A match typing derivation is normal if it is of the form

$$\frac{}{\Delta; \Psi; \Gamma \vdash \cdot : T\ \overline{\tau} \Rightarrow \sigma}$$

or

$$\frac{\dfrac{\mathcal{D}_1}{\Delta; \Psi; \Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma} \qquad \dfrac{\dfrac{\mathcal{D}_2}{\Delta'; \Psi'; \Gamma' \vdash e : \sigma' \quad \Delta; \Psi' \vdash \sigma' \equiv \sigma}}{\Delta'; \Psi'; \Gamma' \vdash e : \sigma}}{\Delta; \Psi; \Gamma \vdash C\ [\overline{\beta}]\ \overline{x} \to e \mid ms' : T\ \overline{\tau} \Rightarrow \sigma}$$

where $\Delta' = \Delta, \overline{\beta}$, $\Psi' = \Psi, \overline{\epsilon}[\overline{\tau}/\overline{\alpha}]$, $\Gamma' = \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}]$, Ψ is satisfiable, and $\mathcal{D}_1$, $\mathcal{D}_2$ are normal derivations. The type σ is not required to be in normal form, whereas σ' necessarily is.

Lemma 12 (Normal Forms Equivalent). *If Ψ is satisfiable and $\Delta \vdash \sigma$, we have $\Delta; \Psi \vdash \sigma \equiv \mathsf{NF}(\sigma)$. Conversely, if $\Delta; \Psi \vdash \sigma \equiv \sigma'$, then $\mathsf{NF}(\sigma) = \mathsf{NF}(\sigma')$.*

Proof. Both parts are immediate corollaries of Proposition 1. □

Lemma 13 (Normal Derivations). *If $\Delta; \Psi; \Gamma \vdash e : \sigma$ is satisfiably derivable them $\Delta; \Psi; \Gamma \vdash e : \mathsf{NF}(\sigma)$ is normally derivable.*

Proof. ($\Longleftarrow$): Suppose we have $\mathcal{D} :: \Delta; \Psi; \Gamma \vdash e : \mathsf{NF}(\sigma)$. Then by Lemma 12 we have

$$\frac{\dfrac{\mathcal{D}}{\Delta; \Psi; \Gamma \vdash e : \mathsf{NF}(\sigma)} \qquad \Delta; \Psi \vdash \mathsf{NF}(\sigma) \equiv \sigma}{\Delta; \Psi; \Gamma \vdash e : \sigma}$$

($\Longrightarrow$): We proceed by induction on the structure of $\mathcal{D} :: \Delta; \Psi; \Gamma \vdash e : \sigma$.

- Case (Var): Then

  $$\frac{}{\Delta; \Psi; \Gamma, x{:}\sigma \vdash x : \sigma}$$

  So we have

  $$\frac{\dfrac{}{\Delta; \Psi; \Gamma, x{:}\sigma \vdash x : \sigma} \qquad \Delta; \Psi \vdash \sigma \equiv \mathsf{NF}(\sigma)}{\Delta; \Psi; \Gamma, x{:}\sigma \vdash x : \mathsf{NF}(\sigma)}$$

- Case (Int): Then

$$\frac{}{\Delta; \Psi; \Gamma \vdash i : \mathit{Int}}$$

So we have

$$\frac{\dfrac{}{\Delta; \Psi; \Gamma \vdash i : \mathit{Int}} \quad \Delta; \Psi \vdash \mathit{Int} \equiv \mathsf{NF}(\mathit{Int})}{\Delta; \Psi; \Gamma \vdash i : \mathsf{NF}(\mathit{Int})}$$

- Case (App): Then

$$\frac{\overset{\mathcal{D}_1}{\Delta; \Psi; \Gamma \vdash e_1 : \sigma \to \sigma'} \quad \overset{\mathcal{D}_2}{\Delta; \Psi; \Gamma \vdash e_2 : \sigma}}{\Delta; \Psi; \Gamma \vdash e_1\ e_2 : \sigma'}$$

By induction we have normal derivations $\mathcal{D}_1' :: \Delta; \Psi; \Gamma \vdash e_1 : \mathsf{NF}(\sigma \to \sigma')$ and $\mathcal{D}_2' :: \Delta; \Psi; \Gamma \vdash e_2 : \mathsf{NF}(\sigma)$. But $\mathsf{NF}(\sigma \to \sigma') = \mathsf{NF}(\sigma) \to \mathsf{NF}(\sigma')$ so

$$\frac{\dfrac{\overset{\mathcal{D}_1'}{\Delta; \Psi; \Gamma \vdash e_1 : \mathsf{NF}(\sigma) \to \mathsf{NF}(\sigma')} \quad \overset{\mathcal{D}_2'}{\Delta; \Psi; \Gamma \vdash e_2 : \mathsf{NF}(\sigma)}}{\Delta; \Psi; \Gamma \vdash e_1\ e_2 : \mathsf{NF}(\sigma')} \qquad \Delta; \Psi \vdash \mathsf{NF}(\sigma') \equiv \mathsf{NF}(\sigma')}{\Delta; \Psi; \Gamma \vdash e_1\ e_2 : \mathsf{NF}(\sigma')}$$

- Case (Lam): Then

$$\frac{\overset{\mathcal{D}}{\Delta; \Psi; \Gamma, x{:}\sigma \vdash e : \sigma'}}{\Delta; \Psi; \Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \sigma'}$$

By induction we have $\mathcal{D}' :: \Delta; \Psi; \Gamma, x{:}\sigma \vdash e : \mathsf{NF}(\sigma')$. So since $\sigma \equiv \mathsf{NF}(\sigma)$ and $\mathsf{NF}(\sigma \to \sigma') = \mathsf{NF}(\sigma) \to \mathsf{NF}(\sigma')$, we can derive

$$\frac{\dfrac{\overset{\mathcal{D}'}{\Delta; \Psi; \Gamma, x{:}\sigma \vdash e : \mathsf{NF}(\sigma')}}{\Delta; \Psi; \Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \mathsf{NF}(\sigma')} \quad \Delta; \Psi \vdash \sigma \to \mathsf{NF}(\sigma') \equiv \mathsf{NF}(\sigma \to \sigma')}{\Delta; \Psi; \Gamma \vdash \lambda x{:}\sigma.e : \mathsf{NF}(\sigma \to \sigma')}$$

- Case (Coercion): Then we have

$$\frac{\overset{\mathcal{D}}{\Delta; \Psi; \Gamma \vdash e : \sigma} \quad \Delta; \Psi \vdash \sigma \equiv \sigma'}{\Delta; \Psi; \Gamma \vdash e : \sigma'}$$

By induction we have $\mathcal{D}' :: \Delta; \Psi; \Gamma \vdash e : \mathsf{NF}(\sigma)$, a normal derivation of the form

$$\frac{\dfrac{\mathcal{D}_1 \cdots \mathcal{D}_n}{\Delta; \Psi; \Gamma \vdash e : \sigma''}\ R \quad \Delta; \Psi \vdash \sigma'' \equiv \mathsf{NF}(\sigma'')}{\Delta; \Psi; \Gamma \vdash e : \mathsf{NF}(\sigma'')}\ .$$

for some σ'' such that $\mathsf{NF}(\sigma'') = \mathsf{NF}(\sigma)$. Observe that since $\sigma \equiv \sigma'$, $\mathsf{NF}(\sigma) = \mathsf{NF}(\sigma')$, so the above derivation is a normal derivation of $\Delta; \Psi; \Gamma \vdash e : \mathsf{NF}(\sigma')$ as well.

- (Pair, Proj, Fix, TLam, TApp) Straightforward.
- (Case, Con) straightforward.
- Case (Match). If

$$\frac{}{\Delta; \Psi; \Gamma \vdash \cdot : T\ \overline{\tau} \Rightarrow \sigma}$$

then the derivation is already normal. Otherwise, the derivation is of the form

$$\frac{\overset{\mathcal{D}_1}{\Delta; \Psi; \Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma} \quad \overset{\mathcal{D}_2}{\Delta'; \Psi'; \Gamma' \vdash e : \sigma}}{\Delta; \Psi; \Gamma \vdash C\ [\overline{\beta}]\ \overline{x} \to e \mid ms' : T\ \overline{\tau} \Rightarrow \sigma}$$

By induction we can construct normal derivations $\mathcal{D}_1' :: \Delta; \Psi; \Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma$ and $\mathcal{D}_2 :: \Delta'; \Psi'; \Gamma' \vdash e : NF_{\Delta';\Psi'}(\sigma)$. Since $\Delta'; \Psi' \vdash NF_{\Delta';\Psi'}(\sigma) \equiv \sigma :$, we have

$$\frac{\dfrac{\mathcal{D}_1'}{\Delta; \Psi; \Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma} \quad \dfrac{\dfrac{\mathcal{D}_2'}{\Delta'; \Psi'; \Gamma' \vdash e : NF_{\Delta';\Psi'}(\sigma)} \quad \Delta; \Psi' \vdash NF_{\Delta';\Psi'}(\sigma) \equiv \sigma}{\Delta'; \Psi'; \Gamma' \vdash e : \sigma}}{\Delta; \Psi; \Gamma \vdash C\ [\overline{\beta}]\ \overline{x} \to e \mid ms' : T\ \overline{\tau} \Rightarrow \sigma}$$

which is a normal match derivation.

□

Theorem 2 (Decidability of typechecking). *Assume $\Delta \vdash \Gamma$, $\Delta \vdash \Psi$, $\Delta \vdash \tau_i : \kappa$, $\Delta \vdash \sigma_i$, $\Delta \vdash \sigma$, and Ψ is satisfiable. It is decidable whether*

1. $\Delta; \Psi \vdash \tau_1 \equiv \tau_2 : \kappa$
2. $\Delta; \Psi \vdash \sigma_1 \equiv \sigma_2$
3. *there exists σ' such that $\Delta \vdash \sigma'$ and $\Delta; \Psi; \Gamma \vdash e : \sigma'$ has a satisfiable derivation*
4. $\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$ *has a satisfiable derivation*

Proof. The first two parts are immediate corollaries of Proposition 1.

The third and fourth parts are proved by induction on the structure of expressions and matches. We prove by induction that it is decidable whether there is a σ' in normal form such that $\Delta; \Psi; \Gamma \vdash e : \sigma'$ is normally derivable, and it is decidable whether $\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$ is normally derivable. Most of the cases are the same as in standard proofs that syntax-directed typechecking is sound and complete. The unusual cases are those for constructors, **case**s, and matches.

- Case (Con): Suppose $e = C\ [\overline{\beta}]\ \overline{e}$, where $\Sigma_{T.C\ \overline{\alpha:\kappa}} = \exists \overline{\beta : \kappa'}.C\ \overline{\sigma'}$ **with** $\overline{\epsilon : \kappa''}$. First, by induction, we synthesize types σ_i for the e_i. Then, using first-order matching, we can find $\overline{\tau}, \overline{\tau'}$ such that $\sigma_i'[\overline{\tau}/\overline{\alpha}, \overline{\tau'}/\overline{\beta}] = \sigma_i$ for each i. Next, we verify that the constructors are of the right kinds: $\Delta \vdash \overline{\tau} : \overline{\kappa}$ and $\Delta \vdash \overline{\tau'} : \overline{\kappa'}$. Next, we check that each $\Delta; \Psi \vdash \epsilon[\overline{\tau}/\overline{\alpha}, \overline{\tau'}/\overline{\beta}] : \kappa_i''$ holds. If all of these checks and constructions succeed, then we can use the constructor rule to type $e : T\ \overline{\tau}$, and then use the coercion rule to normalize $T\ \overline{\tau}$. Otherwise, no normal derivation is possible.
- Case (Case): Suppose $e = \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms$. By induction we can decide if there is a normal form σ' with $\Delta; \Psi; \Gamma \vdash e : \sigma'$. Since σ' is in normal form, we can tell by inspection whether it is $T\ \overline{\tau}$ for some T and constructors τ. If not, then typechecking fails. Otherwise, by part 4 we can decide if $\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$. If not, then no normal derivation can exist. Otherwise, we can use the case rule to immediately derive $\Delta; \Psi; \Gamma \vdash \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms : \sigma$, and then since $\sigma = \mathsf{NF}(\sigma)$, we can also derive $\Delta; \Psi; \Gamma \vdash \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms : \mathsf{NF}(\sigma)$.
- Case (Match): If $ms = \cdot$ then we succeed with the normal derivation $\Delta; \Psi; \Gamma \vdash \cdot : T\ \overline{\tau} \Rightarrow \sigma$. Otherwise, $ms = C\ [\overline{\beta}]\ \overline{x} \to e \mid ms'$. We need to show that it is decidable whether $\Delta; \Psi; \Gamma \vdash C\ [\overline{\beta}]\ \overline{x} \to e \mid ms' : T\ \overline{\tau} \Rightarrow \sigma$. First we construct a normal derivation of $\Delta; \Psi; \Gamma \vdash ms' : T\ \overline{\tau} \Rightarrow \sigma$. Next, we see if there is a normal form σ' and normal derivation of $\Delta, \overline{\beta}; \Psi, \overline{\epsilon}[\overline{\tau}/\overline{\alpha}]; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}] \vdash e : \sigma'$, and then we check that $\Delta, \overline{\beta}; \Psi, \overline{\epsilon}[\overline{\tau}/\overline{\alpha}] \vdash \sigma' \equiv \sigma :$. If these checks succeed then we can construct a normal derivation of $\Delta; \Psi; \Gamma \vdash C\ [\overline{\beta}]\ \overline{x} \to e \mid ms' : T\ \overline{\tau} \Rightarrow \sigma$; if one fails, then no such derivation is possible.

□

We believe that it is reasonable to restrict source programs to typecheck with satisfiable derivations. Programs that typecheck but do not have satisfiable derivations contain dead code that can

never be executed because its equation context can never be satisfied. This (contrived) typecheckable program

$$\begin{array}{lcl} \mathit{foo} & :: & \mathit{Rep}\ \mathit{Int} \to \mathit{Int} \\ \mathit{foo}\ (R_{\mathit{Int}}) & = & 1 \\ \mathit{foo}\ (R_\times\ \mathit{ra}\ \mathit{rb}) & = & 2 \end{array}$$

would be rejected by the satisfiability restriction because the $R_\times$ case involves an unsatisfiable context $\mathit{Int} = (\beta, \gamma)$. We treat the presence of statically dead code as an error; however, such errors can always be fixed by removing the dead code or making the type signature more general.

6 Discussion

6.1 Comparison with λ_R

One important difference between our system and systems like λ_i^{ML} and λ_R is that we are able to do without *trivialization* rules. These systems have **typecase** rules that distinguish between branching on types with known versus unknown top-level constructors. If the top-level constructor is not known (i.e., a variable), then appropriate refinements are substituted in each case. If the top-level constructor is known, then there is no type variable to refine, and so each case must typecheck without any additional knowledge of the top-level constructor. However, these rules alone do not suffice to prove the type substitution lemma needed for preservation. This is because there is a "gap" between what can be derived with the first rule and the second. When the first rule is used to typecheck a **typecase** on a type variable for which a known type is substituted, the refining rule no longer applies, and some of the cases may no longer typecheck using the non-refining rule. For example, in λ_R, the left hand side of the following reduction typechecks using the refining rule since $r : \mathit{Rep}\ \alpha$.

$$\begin{array}{l} (\Lambda\alpha.\lambda r{:}\mathit{Rep}\ \alpha.\lambda x{:}\alpha. \\ \quad \mathbf{case}[\delta.\delta]\ r\ \mathbf{of} \\ \qquad R_{\mathit{Int}} \to 1 \\ \quad \mid R_\times\ [\beta,\gamma]\ \mathit{rb}\ \mathit{rc} \to x)[\mathit{Int}]\ R_{\mathit{Int}}\ 42 \end{array} \quad \mapsto \quad \begin{array}{l} (\lambda r{:}\mathit{Rep}\ \mathit{Int}.\lambda x{:}\mathit{Int}. \\ \quad \mathbf{case}[\delta.\delta]\ r\ \mathbf{of} \\ \qquad R_{\mathit{Int}} \to 1 \\ \quad \mid R_\times\ [\beta,\gamma]\ \mathit{rb}\ \mathit{rc} \to x)\ R_{\mathit{Int}}\ 42 \end{array}$$

After the type substitution takes place, the refining rule no longer applies, since $r : \mathit{Rep}\ \mathit{Int}$. However, the non-refining rule no longer suffices to typecheck the right hand side, since the required result types are Int and $\beta \times \gamma$, and in the second case $x : \mathit{Int}$, not a pair type. As a result, in λ_R, it is necessary to add trivialization rules in order for the type substitution lemma to hold.

In our approach, information about type variables is put into the equation context where it may be used or ignored as needed. Consequently, it is straightforward to show that type substitution holds for $\lambda_\equiv$. In the above example, the first expression's typing would contain a derivation of $\alpha{:}\star; \alpha \equiv \beta \to \gamma \vdash \alpha \equiv \beta \to \gamma : \star$; substituting Int for α in this derivation results in $\cdot; \mathit{Int} \equiv \beta \to \gamma \vdash \mathit{Int} \equiv \beta \to \gamma : \star$, which is still valid.

Our approach is still not entirely satisfactory, since in order to prove type soundness we must permit intermediate program configurations to contain dead code that is forbidden at the source level. Typechecking (let alone type inference) for statically dead code under syntactically unsatisfiable hypotheses is both hard (undecidable) and pointless (since we know the code is dead). Although this approach seems cleaner than the λ_R solution, it leaves something to be desired. Yet other ways to address this problem (such as, for example, incorporating a "pruning" step into type substitution) seem even less desirable.

6.2 Comparison with type classes

Readers familiar with Haskell's powerful type class mechanism may be (justifiably) skeptical that first-class phantom types and representation-based generics solve any problems that type classes do not. We argue that type classes and first-class phantom types address different needs. Type classes are well-suited to many common cases in generic programming. Type classes are "open", that is,

users can define new instances for new user-defined types. Moreover, these instances can differ on types with different names but the same underlying structure. In contrast, representations the types to which generic functions apply to a universe built up of types like sums, products, function spaces, and basic types, and behavior on two types with the same underlying structure is the same. These are serious limitations, and we are not advocating the use of representations in their current form for situations for which type classes are already well suited.

Nevertheless, representations do have some advantages. First, representations are source-level data that can be manipulated more freely than type classes. Although type classes are translated to data (namely, dictionaries) in some implementations of Haskell, they are second-class entities within Haskell itself. Representations, on the other hand, have data structure that can be stored in data structures and analyzed at run time. For example, they can be used to implement a *Dynamic* type directly [6], as opposed to indirectly through type classes [26]. Second, representations facilitate the definition of type-dependent data structures such as generalized tries, which seem difficult to simulate using type classes. Finally, representations can in fact be used to implement type classes (see for example [27]): for a fixed, whole program, a type class dictionary can be factored into a representation type and a generic function each with one case for each class instance. Moreover, programming with first-class phantom types goes beyond representations to include embedding of typed languages within Haskell, as we have seen in Section 2.2.

On the other hand, type classes can be used to ease the burden of constructing and passing representations in the approaches to generic programming presented in the previous sections. By defining a class

$$\begin{array}{l}\textbf{class}\ \mathit{Representable}\ a\ \textbf{where}\\ \quad \mathit{rep} \quad :: \quad \mathit{Rep}\ a\end{array}$$

with instances such as

$$\begin{array}{l}\textbf{instance}\ \mathit{Representable}\ \mathit{Int}\ \textbf{where}\\ \quad \mathit{rep} \quad = \quad R_{\mathit{Int}}\\ \textbf{instance}\ (\mathit{Representable}\ a, \mathit{Representable}\ b) \Rightarrow \mathit{Representable}\ (a, b)\ \textbf{where}\\ \quad \mathit{rep} \quad = \quad R_{\times}\ \mathit{rep}\ \mathit{rep}\end{array}$$

for each case of *Rep a*, we can shift the burden of constructing and passing representations from the programmer to typechecker. This permits defining a more convenient external interface to generic functions such as *equal*:

$$\begin{array}{lcl}\mathit{equal}' & :: & \forall\tau\,.\,\mathit{Representable}\ \tau \Rightarrow \tau \to \tau \to \mathit{Bool}\\ \mathit{equal}'\ x\ y & = & \mathit{equal}\ \mathit{rep}\ x\ y\end{array}$$

6.3 Checking pattern-match exhaustiveness and redundancy

It is usually desirable to know whether a pattern-match covers all cases and whether there are any redundant patterns that can never be matched, for example because they are special cases of earlier patterns. We briefly sketch how this might be accomplished in this section, but the details remain to be worked out.

In Haskell or ML, redundancy and exhaustiveness checks can be carried out by building a decision tree with leaf nodes corresponding to **case** patterns and internal nodes corresponding to intermediate matching states. The nodes are labeled with the free variables of the corresponding pattern, and the edges are labeled with substitutions. We can recover the pattern corresponding to each node by concatenating the substitutions starting from the root.

Exhaustiveness and redundancy can be expressed in terms of *saturation*. A node is saturated if every term which matches it also matches a leaf below the it. Saturation can be checked by checking that the substitutions leading out of a node cover all of the possible data constructors of the free variables of the node. Exhaustiveness can be checked by checking whether the root of the tree is saturated. A pattern is redundant if when it is added to the tree, an ancestor is already saturated.

In the presence of equations, a pattern can be redundant in a new way: we may be able to statically detect that no typed terms exist which can match it. This happens precisely when the

set of equations that must hold as the result of the match are unsatisfiable, that is, when they fail to unify. Conversely, smaller collections of patterns should be considered exhaustive, because some patterns might be ruled out because of unsatisfiable equations. Saturation can be checked by checking that all the possible consistent combinations of data constructors leading out of a node are matched. The definitions of redundancy and exhaustiveness remain the same.

6.4 Type Inference

A full study of type inference in the presence of equational assumptions is beyond the scope of this paper. In this section we sketch the main issues and directions for future work.

Many uses of first-class phantom types require polymorphic recursion, for which type inference is undecidable [11]. In Haskell, polymorphically recursive functions must have explicit type signatures. This problem is not due to phantom types in and of themselves. But type equations do introduce type inference problems of their own. For example, it is not possible to infer principal types for unannotated expressions: the expression

$$\lambda x \to \textbf{case}\ x\ \textbf{of}\ R_{Int} \to 0$$

can have type $Rep\ \alpha \to \alpha$, but it can also have type $Rep\ \alpha \to Int$. These two types correspond to two distinct ways of solving $\alpha = Int \Rightarrow \beta = Int$ for β: the first sets $\beta = \alpha$ and uses the hypothesis, the second sets $\beta = Int$ and uses reflexivity. There is no principal type that generalizes these two types, so standard type inference does not work in the presence of type equations. The solution we (and others [27]) have adopted in this paper is to require result-type annotations for all phantom-type **case** expressions. Polymorphically recursive functions dealing with phantom types need to be annotated with their intended types anyway, so usually it will be possible to "kill two birds with one stone" by propagating these type annotations down to **case**s and other pattern matches.

In many reasonable cases type annotations seem unnecessary. The above example was contrived to point out the ambiguity arising from a single type equation, but the problem seems to get easier with more cases, not harder. Consider

$$\begin{array}{lcl} size\ R_{Int} & = & 1 \\ size\ (R_\times\ ra\ rb) & = & size\ ra + size\ rb \\ zero\ R_{Int} & = & 0 \\ zero\ R_\times\ ra\ rb & = & (zero\ ra, zero\ rb) \end{array}$$

The first function, *size* ought to have type $Rep\ \alpha \to Int$ and the second, *zero*, ought to have type $Rep\ \alpha \to \alpha$. This suggests the following algorithm for attempting to find a principal type: First, infer types for each case branch and normalize the inferred types to σ_i. Then, using first-order anti-unification, find a most-specific common generalization σ and substitutions Θ_i such that $\sigma_i = \Theta_i\sigma$ if possible. Finally, determine whether the substitutions Θ_i are compatible with the equation contexts for each case. This does not work for everything, but it would let us avoid the type annotations for case in the above two examples, because we can statically tell that *size* always returns an *Int* but that *zero* constructs an expression of the represented type. Of course, in Haskell we would have to supply type annotations anyway, in order to typecheck the polymorphically recursive calls.

There are *some* constraints on the types; they are just not expressible equationally. Since type equivalence involves implication in $\lambda_{\equiv}$, perhaps the type constraints generated by type inference need to take implication into account as well. To wit, in the first example, τ' must be *Int* only if $\tau = Int$; otherwise it may be anything. We can formalize this by writing $\tau = Int \Rightarrow \tau' = Int$. Indeed, this is exactly the type equivalence judgment that would be necessary to show that

$$\lambda(x :: Rep\ \tau) \to (\textbf{case}\ x\ \textbf{of}\ R_{Int} \to 0) :: \tau'$$

is well-typed. We conjecture that general type inference in the presence of type equations will require solving "implicational unification" problems in which we are given a set of equational Horn clauses $H_i = \epsilon_i^1, \ldots, \epsilon_i^n \Longrightarrow \epsilon_i$ and seek unifiers that make all of the implications (syntactically) true.

6.5 Parametricity

An important concern with introducing a substantial modification into Haskell's type system is whether good properties such as parametricity still hold. Since first-class phantom types permit polymorphic function definitions to vary based on type information (for example, we can define **typecase** operators), it may seem at first glance that the outlook for parametricity is bleak. However, type classes provide similar kinds of behavior, but do not violate parametricity, as can be seen from the fact that type classes can be implemented via a translation to F^ω (for example, this is how GHC implements them). This is also, we conjecture, essentially the case for first class phantom types as presented here. In many instances, **with**-equations can be translated to "evidence" expressed as type equations using embedding-projection pairs

$$\mathbf{data}\ a \equiv b = EP\{\, to :: a \to b, from :: b \to a \,\}$$

or Leibniz equality, that is,

$$\mathbf{data}\ a \equiv b = Eq\{\, unEq :: \forall f\,.\, f\ a \to f\ b \,\}$$

However, it appears that the decomposition rules step beyond what is expressible in pure F^ω, because for neither definition of equality is it possible to extract well-typed terms $t_1 :: a \equiv c$, $t_2 :: b \equiv d$ from a term $t :: (a, b) \equiv (c, d)$. (and similarly for other instances of the decomposition rules). This is because although type-level functions such as pair and function type formation are (semantically) injective, there is no way of making use of this fact within the type system. Hence, we cannot adequately encode first-class phantom types using embedding-projection pairs or Leibniz equality to reify the **with**-constraints. This does not preclude there being some more direct encoding, however.

6.6 Other Generalizations on Datatypes

Guarded recursive datatypes were developed by Xi, Chen, and Chen [27], prior to and independently of this work. A guarded recursive datatype is a ML datatype with local (i.e. existentially quantified) type variables associated with each data constructor. Each type argument to the datatype can be specialized within each datatype case. For example, GRDs can also be used to implement type representations:

```
datatype rep<*> =
           <int>    RInt
  | {'a,'b}<'a*'b>  RPair  of 'a rep * 'b rep
  | {'a,'b}<'a->'b> RArrow of 'a rep * 'b rep
```

as well as many other features, like subtyping and objects. They define an explicitly typed internal language and an implicitly typed, ML-style source language and define a constraint-based elaboration algorithm that infers type annotations for source language expressions; however, their form of **case**, like ours, requires type annotations. Our phantom types are slightly more general than guarded recursive datatypes, since the guards can be interpreted as equations of the form $\alpha \equiv \tau : \star$, whereas phantom type equations can be of any form and have any kind. However, to be fair, this is because this work is grounded in ML, which does not support constructor polymoirphism.

The Calculus of Inductive Constructions [23] permits *inductive datatypes* to be defined in the following way, by specifying the types of constructors:

$$\begin{array}{l}\mathbf{data}\ Rep :: * \to * = \{\\ \quad R_{Int} :: Rep\ Int\\ \mid R_\times :: \forall\alpha\ \beta\,.\, Rep\ \alpha \to Rep\ \beta \to Rep\ (\alpha, \beta)\\ \mid R_\to :: \forall\alpha\ \beta\,.\, Rep\ \alpha \to Rep\ \beta \to Rep\ (\alpha \to \beta)\\ \}\end{array}$$

The general form of such a datatype is a sequence of assignments of types to constructors such that the result type of each constructor is of the form $T\ \overline{\tau}$, where $T :: \kappa_1 \to \cdots \to \kappa_n \to *$ and $\tau_i : k_i$ for

each i. In the Calculus of Inductive Constructions, there is an additional positivity restriction on occurrences of T necessary to support inductive reasoning about these datatypes; for example, the type *Lam* α of section 2.2 would not be permitted.

The object of this section is to show that GRDs, inductive datatypes, and our first-class phantom types are all essentially the same (modulo inessential language differences such as the lack of higher kinds in ML, and curried vs. uncurried constructor forms).

To translate an inductive datatype with arguments of kind $*$ to a GRD, we translate the declaration **data** $T :: * \to ... \to *$ to `datatype T<*,...,*>`, and each constructor

$$C :: \forall\overline{\alpha}.\sigma_1 \to \cdots \to \sigma_n \to T\ \tau_1\ \ldots\ \tau_m$$

to a datatype case

```
{a1,...,an}<t1,...,tm> C of s1 * ... * sn
```

To translate a GRD into a first-class phantom type, we translate the declaration `datatype T<*,...,*>` to **data** $T\ \alpha_1\ ...\ \alpha_n$ (where $\alpha_1, \ldots, \alpha_m$ are fresh) and translate each constructor

```
{a1,...,an}<t1,...,tm> C of s1 * ... * sn
```

to a case

$$C\ \sigma_1 \cdots \sigma_n\ \textbf{with}\ \alpha_1 = \tau_1, \ldots, \alpha_m = \tau_m$$

The equation guards are clearly satisfiable because they are already in a solved form, and the α_i's are fresh variables so clearly do not appear in the τ_i's.

To translate a first-class phantom type (FCPT) with arguments of kind $*$ to an inductive datatype, we translate the declaration **data** $T\ \alpha_1\ ...\ \alpha_m$ to **data** $T :: * \to ... \to *$ and for each constructor

$$C\ \sigma_1 \cdots \sigma_n\ \textbf{with}\ \tau_1 = \tau_1', \ldots, \tau_m = \tau_m'$$

we unify the equations in its **with** clause, relative to the variables $\alpha_1, ..., \alpha_m$ bound in the datatype definition. Since well-formed FCPT cases always have satisfiable **with** clauses, a most general unifier Θ exists, and the corresponding inductive datatype constructor declaration is

$$C :: \forall\overline{\beta}.\Theta\sigma_1 \to \ldots \to \Theta\sigma_n \to T\ \Theta(\alpha_1) \ldots \Theta(\alpha_m)$$

where $\overline{\beta}$ consists of all the free type variables of the argument and result types.

Given that all three of the above generalizations of datatypes seems equivalent (ML vs. Haskell differences notwithstanding), how shall we choose among them? Our phantom types seem to require less drastic syntactic changes to Haskell than either GRDs or inductive datatypes. The equations of FCPTs make the connection between unification and typechecking explicit. On the other hand, FCPTs introduce a layer of type-variable indirection in order to refine the type of the data constructor, whereas the other two approaches express this refinement more directly (and perhaps more intuitively). Also, both inductve datatypes and GRDs manage to avoid requiring the extra satisfiability check that is required for FCPTs.

7 Related Work

Part of the motivation for this work comes from attempting to implement the type representations and **typecase** of Crary, Weirich, and Morrisett's λ_R [7]. First-class phantom types can be used to implement type representations, **typecase**, and a limited form of **Typerec**. The λ_R system has additional features like type-level λ-abstractions that are not present in Haskell or our $\lambda_{\equiv}$. Type λ-abstractions do not seem to be problematic (from a type-checking, rather than -inference, point of view) if they may not appear in **with** clauses. However, if λ-abstractions can occur in type equations, then determining whether a type equation context unifies may require higher-order unification, which is undecidable.

Yang [29] showed how to encode specific type-dependent values in ML's type system, and how to encode type representations using first-class polymorphism[4] and as embedding-projection pairs relating typed and untyped representations. The first and third encodings fit within the Hindley-Milner type system used by SML and Haskell 98, while the second can be implemented directly in some implementations: in SML/NJ using higher-order functors, and more directly in GHC or Hugs using first-class polymorphism. The first encoding amounts to simulating type classes "by hand" by dictionary passing. The third encoding is a generalization of the usual trick of defining type-dependent functions on a *Universal* datatype and coercing typed data to and from the *Universal*. Yang's second (and most interesting) encoding is, up to isomorphism, the same as implementing *Rep* in Haskell as in [6, 4].

Zenger's *indexed types* [30] are very similar in form to our first-class phantom types. Indexed types are datatypes which may be parameterized by numeric (specifically complex) variables, and guarded by systems of polynomial equations. Although indexed types do not permit any new expressions to be typed, they make it possible to give more accurate types to existing functions like matrix multiply and quicksort, which can be used to detect more errors statically (for example, multiplying matrices of incompatible dimensions, or dropping elements during sorting). Zenger's system can be seen as an instance of Jones' *qualified types* framework [16] where the qualifiers are polynomial equations.

Xi, Chen, and Chen [27] have introduced *guarded recursive datatypes*, or datatypes whose type arguments may be constrained in each case. We have compared their approach with ours in detail in Section 6.6. Guarded recursive datatypes can be seen as a special case of Xi and Pfenning's Dependent ML framework [28] where the constraint domain is just syntactic equality among type expressions. Another related line of research is work on *singleton kinds*, originating from efforts to characterize type sharing constraints in module systems like that of Standard ML [9]. Singleton kinds are kinds $S(\tau)$ containing only the type τ; thus, the kind judgment $\Delta \vdash \tau : S(\tau')$ implies the equivalence judgment $\Delta \vdash \tau \equiv \tau'$. Singleton kinds can encode hypotheses of the form $\alpha = \tau$ as $\alpha : S(\tau)$.

8 Future Work

Our system has several limitations. The most serious is that type representations and generic functions definable using first-class phantom types are limited to a fixed universe of types, in contrast to type classes which are extensible to user-defined datatypes in a modular way. Another limitation is that we only allow *predicative* type constraints (between type constructors), as opposed to *impredicative* constraints (between arbitrarily quantified types). We have focused on this simpler system because it is useful as it stands and because impredicative constraints seem likely to cause problems similar to those encountered in polymorphic **typecase** pattern matching [2, 20]. A third direction for future work is broadening the scope of phantom type constraints beyond type equations. Some related systems, like indexed types [30], Cayenne's dependent type system [3] and Dependent ML [28], can handle richer constraint domains such as integer inequalities in order to capture more interesting data-structure invariants.

Type representations implemented using phantom types are fairly limited: in contrast to some of the encodings presented in [6] they cannot deal gracefully with user-defined datatypes, because type representations fix a "closed world" of representable types. Furthermore, both approaches encounter difficulties with higher-order type representations and representations of type-indexed types, although the relational encoding explored in Section 2.4 can often help. It is a open problem whether representation-based techniques can be generalized to deal with these problems.

[4] we mean polymorphic recursion and rank-n polymorphism

9 Conclusions

We have presented a language extension called first-class phantom types that generalizes phantom type encoding tricks found in the literature in that it allows both type-safe construction and deconstruction of constrained values, whereas existing approaches require additional run-time tagging and dynamic checks for safe destruction. Our phantom types can also be used to define type representations, a statically type-safe *Dynamic* type, and generic functions. They are more usable, more efficient, and more expressive than previous encodings of these features in Haskell via equation types.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Remy. Dynamic typing in polymorphic languages. Technical Report 120, DEC SRC, January 1994.

[3] Lennart Augustsson. Cayenne – a language with dependent types. *SIGPLAN Notices*, 34(1):239–250, January 1999.

[4] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In Simon Peyton Jones, editor, *Proceedings of the 2002 International Conference on Functional Programming, Pittsburgh, PA, USA, October 4-6, 2002*, pages 157–166. ACM Press, October 2002.

[5] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, June 1998.

[6] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel M. T. Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop (Haskell '02)*, pages 90–104, Pittsburgh, PA, October 2002. ACM Press.

[7] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, MD*, volume (**34**)1 of *ACM SIGPLAN Notices*, pages 301–312. ACM Press, June 1999.

[8] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[9] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94), Portland, Oregon*, pages 123–137, New York, NY, January 1994. ACM.

[10] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), San Francisco, California*, pages 130–141. ACM Press, 1995.

[11] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[12] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, July 2000.

[13] Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 00), Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

[14] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In Eerke A. Boiten and Bernhard Möller, editors, *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002), Dagstuhl, Germany, July 8-10, 2002*, volume 2386 of *Lecture Notes in Computer Science*, pages 148–174. Springer-Verlag, July 2002.

[15] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 470–482. ACM Press, January 1997.

[16] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[17] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37, March 2003. *SIGPLAN Notices,* 38(3), March 2003.

[18] Daan Leijen and Erik Meijer. Domain-specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, October 1999. USENIX Association.

[19] Xaver Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[20] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[21] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995. Published as CMU Technical Report CMU-CS-95-226.

[22] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228, 1984.

[23] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664, pages 328–345. Springer-Verlag, Berlin, 1993.

[24] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In Manuel M. T. Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop (Haskell '02)*, pages 1–16, Pittsburgh, PA, October 2002. ACM Press.

[25] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 289–302, New York, January 1998. Association for Computing Machinery.

[26] Stephanie Weirich. Type-safe cast: functional pearl. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, volume (**35**)9 of *ACM SIGPLAN Notices*, pages 58–67, N.Y., September 2000. ACM Press.

[27] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 2003 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, pages 224–235, January 2002. ACM SIGPLAN Notices, 38(1), January 2003.

[28] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227. ACM, January 1999.

[29] Zhe Yang. Encoding types in ML-like languages. In *International Conference on Functional Programming*, pages 289–300, 1998.

[30] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1–2):147–165, November 1997.

A $\lambda_{\equiv}$ details

A.1 Syntax and Operational Semantics

$$
\begin{array}{rcll}
(\textit{type contexts}) & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \\
(\textit{equation contexts}) & \Psi & ::= & \cdot \mid \Psi, \epsilon{:}\kappa \\
(\textit{substitutions}) & \Theta & ::= & \cdot \mid \Theta, \alpha \equiv \tau \\
(\textit{datatype contexts}) & \Sigma & ::= & \cdot \mid \Sigma; \mathbf{data}\ T\ \overline{\alpha{:}\kappa} = \Sigma_T \\
(\textit{datatype signatures}) & \Sigma_T & ::= & \cdot \mid \Sigma_T \mid \exists \overline{\beta{:}\kappa}.C\ \overline{\sigma}\ \mathbf{with}\ \overline{\epsilon{:}\kappa'} \\
(\textit{term contexts}) & \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \\
 & & & \\
(\textit{kinds}) & \kappa & ::= & \star \mid \kappa \to \kappa \\
(\textit{constructors}) & \tau & ::= & \alpha \mid \tau_1\ \tau_2 \mid \mathit{Int} \mid \to \mid \times \mid T \\
(\textit{equations}) & \epsilon & ::= & \tau_1 \equiv \tau_2 \\
(\textit{types}) & \sigma & ::= & \tau \mid \mathit{Int} \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall \alpha{:}\kappa.\sigma \\
(\textit{expressions}) & e & ::= & i \mid x \mid \lambda x : \sigma.e \mid \mathbf{fix}\ f{:}\sigma.e \mid e_1\ e_2 \mid \langle e_1, e_2 \rangle \\
 & & \mid & \pi_1 e \mid \pi_2 e \mid \Lambda \alpha{:}\kappa.e \mid e[\tau] \\
 & & \mid & C\ [\overline{\tau}]\ \overline{e} \mid \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms \mid \mathbf{fail} \\
(\textit{matches}) & ms & ::= & \cdot \mid C\ [\overline{\beta}]\ \overline{x} \to e \mid ms \\
 & & & \\
(\textit{values}) & v & ::= & i \mid \lambda x{:}\sigma.e \mid \langle e_1, e_2 \rangle \mid \Lambda \alpha{:}\kappa.e \\
 & & \mid & C\ [\overline{\tau}]\ \overline{e} \mid \mathbf{fail} \\
(\textit{evaluators}) & E & ::= & [\cdot] \mid E\ e \mid \pi_1\ E \mid \pi_2\ E \mid E[c] \mid \mathbf{case}[\sigma]\ E\ \mathbf{of}\ ms
\end{array}
$$

$$
\begin{array}{cc}
(\lambda x : \sigma.e)e' \mapsto e[e'/x] & \mathbf{fix}\ f{:}\sigma.e \mapsto e[\mathbf{fix}\ f{:}\sigma.e/f] \\
\pi_1\langle e_1, e_2 \rangle \mapsto e_1 & \pi_2\langle e_1, e_2 \rangle \mapsto e_2 \\
(\Lambda \alpha{:}\kappa.e)[c] \mapsto e[c/\alpha] & \dfrac{e \mapsto e'}{E[e] \mapsto E[e']}
\end{array}
$$

$$
\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ C\ [\overline{\beta}]\ \overline{x} \mid P \mapsto e[\overline{e}/\overline{x}, \overline{\tau}/\overline{\beta}]
$$

$$
\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ D\ [\overline{\beta}]\ \overline{x} \mid P \mapsto \mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ ms
$$

$$
\begin{array}{cc}
\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ \cdot \mapsto \mathbf{fail} & E[\mathbf{fail}] \mapsto \mathbf{fail}
\end{array}
$$

A.2 Well-Formedness

$\boxed{\Delta \vdash \tau : \kappa}$

$$
\frac{}{\Delta \vdash \mathit{Int} : \star} \qquad \frac{}{\Delta \vdash \to : \star \to \star \to \star} \qquad \frac{}{\Delta \vdash \times : \star \to \star \to \star}
$$

$$
\frac{}{\Delta, \alpha{:}\kappa \vdash \alpha : \kappa} \qquad \frac{\Delta \vdash \tau_1 : \kappa \to \kappa' \quad \Delta \vdash \tau_2 : \kappa}{\Delta \vdash \tau_1\ \tau_2 : \kappa'} \qquad \frac{\Sigma \vdash \mathbf{data}\ T\ \overline{\alpha{:}\kappa} = \Sigma_T}{\Delta \vdash T : \overline{\kappa} \to \star}
$$

$\boxed{\Delta \vdash \sigma}$

$$
\frac{\Delta \vdash \tau : \star}{\Delta \vdash \tau} \qquad \frac{}{\Delta \vdash \mathit{Int}} \qquad \frac{\Delta \vdash \sigma_i}{\Delta \vdash \sigma_1 \to \sigma_2} \qquad \frac{\Delta \vdash \sigma_i}{\Delta \vdash \sigma_1 \times \sigma_2} \qquad \frac{\Delta, \alpha{:}\kappa \vdash \sigma}{\Delta \vdash \forall \alpha{:}\kappa.\sigma}
$$

$\boxed{\Delta \vdash \Psi}$

$$
\frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Psi \quad \Delta \vdash \tau_i : \kappa}{\Delta \vdash \Psi, \tau_1 \equiv \tau_2{:}\kappa}
$$

$\boxed{\Delta \vdash \Sigma}$

$$\frac{}{\Delta \vdash \cdot} \quad \frac{\Delta \vdash \Sigma \quad \Delta, \overline{\alpha{:}\kappa} \vdash \Sigma_T}{\Delta \vdash \Sigma, \mathbf{data}\ T\ \overline{\alpha{:}\kappa} = \Sigma_T}$$

$\boxed{\Delta \vdash \Sigma_T}$

$$\frac{}{\Delta \vdash \cdot} \quad \frac{\Delta \vdash \Sigma_T \quad \Delta, \overline{\beta{:}\kappa} \vdash \tau_i^0 : \star \quad \Delta, \overline{\beta{:}\kappa} \vdash \tau_i : \kappa_i' \quad \Delta, \overline{\beta{:}\kappa} \vdash \tau_i' : \kappa_i' \quad \overline{\beta} \vdash \overline{\tau \equiv \tau' : \kappa'} \Downarrow \Theta}{\Delta \vdash \Sigma_T \mid \exists \overline{\beta{:}\kappa}.C\ \overline{\sigma}\ \mathbf{with}\ \overline{\tau \equiv \tau' : \kappa'}}$$

$\boxed{\Delta \vdash \Gamma}$

$$\frac{}{\Delta \vdash \cdot} \quad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \sigma}{\Delta \vdash \Gamma, x : \sigma}$$

A.3 Unification

$\boxed{\Delta \vdash \Psi \Downarrow \Theta}$

$$\frac{}{\Delta \vdash \cdot \Downarrow \cdot} \qquad \frac{\Delta \vdash \Psi \Downarrow \Theta}{\Delta \vdash \Psi, c = c \Downarrow \Theta}$$

$$\frac{\Delta \vdash \Psi \Downarrow \Theta}{\Delta \vdash \Psi, \alpha = \alpha \Downarrow \Theta} \qquad \frac{\Delta \vdash \Psi, \alpha = \tau \Downarrow \Theta \quad \tau \neq \beta}{\Delta \vdash \Psi, \tau = \alpha \Downarrow \Theta}$$

$$\frac{\Delta \vdash \tau : \kappa \quad \Delta \vdash \Psi[\tau/\alpha] \Downarrow \Theta}{\Delta, \alpha{:}\kappa \vdash \Psi, \alpha = \tau \Downarrow \Theta, \alpha = \tau} \qquad \frac{\Delta \vdash \Psi, \tau_1 = \tau_1', \tau_2 = \tau_2' \Downarrow \Theta}{\Delta \vdash \Psi, \tau_1\ \tau_2 = \tau_1'\ \tau_2' \Downarrow \Theta}$$

A.4 Type Equivalence

$\boxed{\Delta; \Psi \vdash \tau \equiv \tau' : \kappa}$

$$\frac{}{\Delta; \Psi, \tau_1 \equiv \tau_2{:}\kappa \vdash \tau_1 \equiv \tau_2 : \kappa} \qquad \frac{\Delta \vdash \tau : \kappa}{\Delta; \Psi \vdash \tau \equiv \tau : \kappa}$$

$$\frac{\Delta; \Psi \vdash \tau' \equiv \tau : \kappa}{\Delta; \Psi \vdash \tau \equiv \tau' : \kappa} \qquad \frac{\Delta; \Psi \vdash \tau \equiv \tau' : \kappa \quad \Delta; \Psi \vdash \tau' \equiv \tau'' : \kappa}{\Delta; \Psi \vdash \tau \equiv \tau'' : \kappa}$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa \quad \Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa}{\Delta; \Psi \vdash \tau_1 \equiv \tau_1' : \kappa_1 \to \kappa} \qquad \frac{\Delta \vdash \tau_2 : \kappa_1 \quad \Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa}{\Delta; \Psi \vdash \tau_2 \equiv \tau_2' : \kappa_1}$$

$$\frac{\Delta; \Psi \vdash \tau_1 \equiv \tau_1' : \kappa_1 \to \kappa \quad \Delta; \Psi \vdash \tau_2 \equiv \tau_2' : \kappa_1}{\Delta; \Psi \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2' : \kappa}$$

$\boxed{\Delta; \Psi \vdash \sigma = \sigma'}$

$$\frac{\Delta \vdash \sigma}{\Delta; \Psi \vdash \sigma = \sigma} \qquad \frac{\Delta; \Psi \vdash \sigma' = \sigma}{\Delta; \Psi \vdash \sigma = \sigma'}$$

$$\frac{\Delta; \Psi \vdash \sigma = \sigma' \quad \Delta; \Psi \vdash \sigma' = \sigma''}{\Delta; \Psi \vdash \sigma = \sigma''} \qquad \frac{\Delta; \Psi \vdash \sigma_1 = \sigma_1' \quad \Delta; \Psi \vdash \sigma_2 = \sigma_2'}{\Delta; \Psi \vdash \sigma_1 \times \sigma_2 = \sigma_1' \times \sigma_2'}$$

$$\frac{\Delta; \Psi \vdash \sigma_1 = \sigma_1' \quad \Delta; \Psi \vdash \sigma_2 = \sigma_2'}{\Delta; \Psi \vdash \sigma_1 \to \sigma_2 = \sigma_1' \to \sigma_2'} \qquad \frac{\Delta, \alpha{:}\kappa; \Psi \vdash \sigma_1 = \sigma_2}{\Delta; \Psi \vdash \forall \alpha{:}\kappa.\sigma_1 = \forall \alpha{:}\kappa.\sigma_2}$$

$$\frac{\Delta; \Psi \vdash \tau = \tau' : \star}{\Delta; \Psi \vdash \tau = \tau'}$$

A.5 Typing

$\boxed{\Delta; \Psi; \Gamma \vdash e : \sigma}$

$$\frac{}{\Delta; \Psi; \Gamma \vdash i : \mathit{Int}} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Delta; \Psi; \Gamma, x{:}\sigma \vdash e : \sigma'}{\Delta; \Psi; \Gamma \vdash \lambda x{:}\sigma.e : \sigma \to \sigma'} \qquad \frac{\Delta; \Psi; \Gamma \vdash e_1 : \sigma \to \sigma' \quad \Delta; \Psi; \Gamma \vdash e_2 : \sigma}{\Delta; \Psi; \Gamma \vdash e_1\ e_2 : \sigma'}$$

$$\frac{\Delta; \Psi; \Gamma, f{:}\sigma \vdash e : \sigma}{\Delta; \Psi; \Gamma \vdash \mathbf{fix}\ f{:}\sigma.e : \sigma} \qquad \frac{\Delta; \Psi; \Gamma \vdash e_1 : \sigma_1 \quad \Delta; \Psi; \Gamma \vdash e_2 : \sigma_2}{\Delta; \Psi; \Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Delta; \Psi; \Gamma \vdash e : \sigma_1 \times \sigma_2}{\Delta; \Psi; \Gamma \vdash \pi_1\ e : \sigma_1} \qquad \frac{\Delta; \Psi; \Gamma \vdash e : \sigma_1 \times \sigma_2}{\Delta; \Psi; \Gamma \vdash \pi_2\ e : \sigma_2}$$

$$\frac{\Delta, \alpha{:}\kappa; \Psi; \Gamma \vdash e : \sigma}{\Delta; \Psi; \Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\sigma} \qquad \frac{\Delta; \Psi; \Gamma \vdash e : \forall\alpha{:}\kappa \quad \Delta \vdash \tau : \kappa}{\Delta; \Psi; \Gamma \vdash e[\tau] : \sigma[\tau/\alpha]}$$

$$\frac{\begin{array}{l}\Sigma_{T.C\ \overline{\alpha}} = \exists\overline{\beta{:}\kappa}.C\ \overline{\sigma}\ \mathbf{with}\ \overline{\epsilon{:}\kappa'} \qquad \Delta; \Psi; \Gamma \vdash e_i : \sigma_i[\overline{\tau}'/\overline{\alpha}, \overline{\tau}/\overline{\beta}] \\ \Delta; \Psi \vdash \epsilon_i[\overline{\tau}'/\overline{\alpha}, \overline{\tau}/\overline{\beta}] : \kappa'_i \qquad \Delta \vdash \tau_i : \kappa_i\end{array}}{\Delta; \Psi; \Gamma \vdash C\ [\overline{\tau}]\ \overline{e} : T\ \overline{\tau}'}$$

$$\frac{\Delta; \Psi; \Gamma \vdash e : T\ \overline{\tau} \quad \Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma}{\Delta; \Psi; \Gamma \vdash \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms : \sigma} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash \mathbf{fail} : \sigma}$$

$$\frac{\Delta; \Psi; \Gamma \vdash e : \sigma_1 \quad \Delta; \Psi \vdash \sigma_1 \equiv \sigma_2}{\Delta; \Psi; \Gamma \vdash e : \sigma_2}$$

$\boxed{\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma}$

$$\frac{}{\Delta; \Psi; \Gamma \vdash \cdot : T\ \overline{\tau} \Rightarrow \sigma}$$

$$\frac{\begin{array}{l}\Sigma_{T.C\ \overline{\alpha}} = \exists\overline{\beta}.C\ \overline{\sigma}\ \mathbf{with}\ \overline{\epsilon} \qquad \Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma' \\ \Delta, \overline{\beta}; \Psi, \overline{\epsilon}[\overline{\tau}/\overline{\alpha}, \overline{\gamma}/\overline{\beta}]; \Gamma, \overline{x{:}\sigma}[\overline{\tau}/\overline{\alpha}, \overline{\gamma}/\overline{\beta}] \vdash e : \sigma'\end{array}}{\Delta; \Psi; \Gamma \vdash C\ [\overline{\gamma}]\ \overline{x} \to e \mid ms : T\ \overline{\tau} \Rightarrow \sigma'}$$