Phantom Types

James Cheney[1], Ralf Hinze[2]

[1] Cornell University
Ithaca, NY 14850
jcheney@cs.cornell.edu
[2] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de

Abstract. Phantom types are data types with type constraints associated with different cases. Examples of 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 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, 19], defining generic functions by translation from "polytypic" languages to existing languages [15, 13], and implementing staged computation with run-time type checking [8, 23] or compile-time computation [22]. Recently, 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 "proofs" of type equality.

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 the identity function;
- it requires a different equality type definition for each kind; and
- it is limited by the constraint-solving abilities of the underlying type-checker.

The last limitation is particularly vexing because it places many interesting potential applications just out of reach, and imposes unnecessary run-time overhead in others. 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 type checker does not know certain valid properties of types that are not important for normal type checking. For example, because type equality in Haskell is syntactic, $\tau_1 = \tau_1'$ follows from $\tau_1 \times \tau_2 = \tau_1' \times \tau_2'$, but in Haskell, there is no useful way to "prove" this implication using equality types. We will discuss this point in more detail in Sections 2.3 and 4.

In this paper we present a programming construct for defining so-called *phantom types*. In our formalism, data type 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 type checking case bodies for the constructor. In combination with unrestricted existential quantifiers, data type 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.

The term *phantom type* was originally coined by Leijen and Meijer [17] to denote parameterized types that do not use their type argument(s). They use phantom types for type-safe embeddings of domain specific languages into Haskell. Their approach is, however, strictly more limited: while they can enforce type constraints when constructing values of a phantom type, they cannot make use of these constraints when decomposing a value. Our proposal exactly fills this gap.

In the rest of this paper, we describe our proposal in more detail, discuss applications, present a type system for a Haskell-like language that has phantom types, and prove some of its properties. Finally, we discuss related and future work and conclude.

2 Introducing 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

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

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, Haskell's linguistic construct for introducing constructors. 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 $Rep\ \tau$ but not $Rep\ Int$.

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* data type as follows:

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

The **with** clause that is attached to each constructor records its type constraints. For instance, R_{Int} has type $Rep\ \tau$ with the additional constraint $\tau = Int$. 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 $Rep\ [Int]$ and is illegal in a context where an element of type $Rep\ Double$ 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. These variables can be seen as being *existentially quantified.*

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

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

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 $\forall\tau\,.\, Rep\ \tau \to \tau \to \tau \to Bool$, each equation has a more specific type as dictated by the type constraints. As an example, the first equation has type $Rep\ Int \to Int \to Int \to Bool$ 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 an equation may relate two type expressions of the same kind. In particular, an equation may relate two types of a higher-order kind. We will encounter examples of more advanced phantom types in Section 2.3.

2.2 Generic traversals

Building upon the *Rep* type of the previous 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. Its type is

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

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.

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

The simplest traversal is, of course, *copy*, which does nothing.

$$\begin{array}{ll}
\mathit{copy} & :: \mathit{Traversal} \\
\mathit{copy}\ r_\tau & = \mathit{id}
\end{array}$$

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

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

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.

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

The function *imap* can be seen as a 'traversal transformer'. It enjoys functor-like properties: $imap\ copy = copy$ and $imap\ (f \circ g) = imap\ f \circ imap\ g$.

$$
\begin{array}{ll}
everywhere, everywhere' & :: Traversal \rightarrow Traversal \\
everywhere\ f & = f \circ imap\ (everywhere\ f) \\
everywhere'\ f & = imap\ (everywhere'\ f) \circ f
\end{array}
$$

Actually, there are two flavours of the combinator: *everywhere* f applies f *after* the recursive calls (it proceeds bottom-up), whereas *everywhere'* applies f *before* (it proceeds top-down).

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 [16].

2.3 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 data types* [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 data types 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.

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

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 type* [5], as the definition involves 'recursive calls', e.g. $Trie\ \kappa_1\ (Trie\ \kappa_2\ \nu)$, that are substitution instances of the defined type.

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

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

Note that *lookup* (as well as *equal*) requires a non-schematic form of recursion called *polymorphic recursion* [21]: the recursive calls are at types which are substitution instances of the declared type (see also Section 6).

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

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

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 type checks. 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 type check 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 three out of nine combinations are covered. The reason is simply that, for instance, the case $\mathit{merge}\ c\ (T_+\ \mathit{ta}\ \mathit{tb})\ (T_\times\ \mathit{ta}')$ 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 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 $\mathit{Trie}\ \kappa$ of kind $\star \to \star$.

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

The type *Rep* can be seem as a type representation type that additionally incorporates the type-indexed data type. 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).

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

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

3 Typing Rules

We now give the static semantics of phantom types. We model the source language $\lambda_=$ as a lazy, explicitly typed and kinded variant of F_ω with data types. Figure 1 shows the syntax modifications introduced in $\lambda_=$; 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 type checked. 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. It should be clear how to extend this approach to deep pattern matching, **let** bindings, and so on.

Type equations are of the form $\tau = \tau'$, often abbreviated ϵ when τ, τ' are not important. Phantom data type 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.

The operational semantics of data type and case expressions is completely standard; type equations have no effect on expression evaluation. To avoid complicating the typing rules or operational semantics, we do not check that patterns are exhaustive or non-overlapping. If multiple patterns match, only the

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

Fig. 1. Syntax modifications for $\lambda_=$

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
$\Delta; \Psi \vdash \tau_1 = \tau_2 : \kappa$	constructor equivalence
$\Delta; \Psi \vdash \sigma_1 = \sigma_2$	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. We expect all equation contexts in source programs to unify, although for technical reasons we have to relax this restriction in order to prove type soundness.

The type equivalence rules are mostly standard, including reflexivity, transitivity, symmetry, and congruence rules. The new rules are the *hypothesis* rule:

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

and the *decomposition rules* that invert the congruence rule of equational logic:

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

We treat the base type constructors $\times, \to, T$ as constants with the appropriate kinds, e.g. $(\times), (\to) : \star \to \star \to \star$. Thus, we need only one congruence rule and two decomposition rules, for application.

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 type checks 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.

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

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

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

4 Applications of the decomposition rule

Type equivalence and unification are closely related. In particular, the decomposition rules correspond to unification steps that we know must have occurred to prove the congruence. That is, if we know that two pair types unify, then we also know that their respective components must also unify; more generally, in Haskell, if two applications unify, then the function and argument components must also unify respectively.

We have already seen one situation where the decomposition rule is needed: in the functional encoding of generalized tries of Section 2.3. 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, **Typerec** $\tau\ \tau_{Int}\ \tau_\times$ is equivalent to τ_{Int} if $\tau = Int$, and $\tau_\times\ \tau_1\ \tau_2$ (**Typerec** $\tau_1\ \tau_{Int}\ \tau_\times$) (**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 [18], array flattening, and automatic marshaling [10]. Using phantom types, we can simulate **Typerec** as follows:

$$\begin{array}{l} \mathbf{data}\ \mathit{Typerec}\ \alpha\ \tau\ \phi = T_{Int}\ \tau\ \mathbf{with}\ \alpha = \mathit{Int} \\ \qquad\qquad\qquad\qquad\ \mid\ T_\times\ (\phi\ \beta\ \gamma\ (\mathit{Typerec}\ \beta\ \tau\ \phi)\ (\mathit{Typerec}\ \gamma\ \tau\ \phi)) \\ \qquad\qquad\qquad\qquad\quad\ \mathbf{with}\ \alpha = (\beta,\gamma) \end{array}$$

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

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

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

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

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

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

The *unInt* function type checks even without decompositions, since we already know that $\mathit{Int} = \mathit{Int}$. But without the decomposition rules, the pair case does not type check, because the type of p is $(\mathit{Rev}\ \beta', \mathit{Rev}\ \gamma')$ for some existentially quantified β', γ' such that $(\beta, \gamma) = (\beta', \gamma')$, while the desired type is $(\mathit{Rev}\ \beta, \mathit{Rev}\ \gamma)$ with β, γ universally quantified. Without decompositions, we cannot conclude that $\beta = \beta'$, $\gamma = \gamma'$ or that $(\mathit{Rev}\ \beta, \mathit{Rev}\ \gamma) = (\mathit{Rev}\ \beta', \mathit{Rev}\ \gamma')$, 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 data type, 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:

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

Unions are like standard phantom data types, except that the run-time data type 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 $\lambda_=$, a type system for a Haskell-like language with phantom types. In the rest of this section, we outline the proofs of soundness and decidability of type checking for $\lambda_=$. The full version of this paper [**?**] contains complete proofs of these results.

We first establish some properties of type equivalence. First, in the absence of hypotheses, type equivalence is syntactic equality.

Lemma 1 (Syntactic Equality). *If $\Delta \vdash \tau_i : \kappa$, then $\Delta; \cdot \vdash \tau_1 = \tau_2 : \kappa$ 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 $\sigma_1 =_\alpha \sigma_2$.*

Second, in the presence of a unifying context, type equivalence is the same as syntactic equality after applying a unifier.

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

The proofs of these properties are straightforward.

The proof of type soundness is similar in structure to those for related systems λ_i^{ML} [20] and λ_R [7]. We prove the standard substitution lemmas, as well as additional cases for substituting types into equivalence judgments and substituting equivalence derivations into equivalence and typing derivations. 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 2 (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* $\Delta; \Psi; \Gamma \vdash ms : T\ \overline{\tau} \Rightarrow \sigma$

Lemma 3 (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.

Lemma 4 (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 $\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ C\ [\overline{\beta}]\ \overline{x} \rightarrow e \mid ms$ 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 type checks.

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''.*

Now we turn to the question of the decidability of type checking for $\lambda_=$. Type equivalences are a special case of general equational implication, which is undecidable. However, we showed earlier that if Ψ unifies, then $\Delta; \Psi \vdash \tau = \tau' : \kappa$ 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 *unifiable* if all contexts in it unify.

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

1. $\Delta; \Psi \vdash \tau_1 = \tau_2 : \kappa$
2. $\Delta; \Psi \vdash \sigma_1 = \sigma_2$
3. *there exists σ' such that $\Delta \vdash \sigma'$ and $\Delta; \Psi; \Gamma \vdash e : \sigma'$ has a unifiable derivation*

4. $\Delta;\Psi;\Gamma \vdash ms : T\,\overline{\tau} \Rightarrow \sigma$ has a unifiable derivation

The idea of the proof is to define normal forms for types, expressions and match derivations. The normal form of a type σ (relative to Ψ) is $\Theta\sigma$, where Θ is the most general unifier of Ψ. Two types are equivalent relative to Ψ if and only if their normal forms are equal. Normal derivations are unifiable derivations in which types are normalized as much as possible. With these definitions it is straightforward to show that all unifiable derivations can be converted to normal derivations, and finally show that normal derivation search is syntax-directed.

We believe that it is reasonable to restrict source programs to typecheck with unifiable derivations. Programs that typecheck but do not have unifiable derivations contain *dead code* that can never be executed because its equation context can never be satisfied. This (contrived) typecheckable program

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

would be rejected by the unifiability restriction because the $R_{\times}$ case involves a nonunifiable context $\mathit{Int} = (\beta, \gamma)$. However, the programmer can always get an equivalent program that does typecheck by deleting the dead code or generalizing the type signature.

6 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 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. For example, the expression $\lambda x \to \textbf{case}\ x\ \textbf{of}\ R_{\mathit{Int}} \to 0$ can have type $\mathit{Rep}\ \alpha \to \alpha$, but it can also have type $\mathit{Rep}\ \alpha \to \mathit{Int}$. These two types correspond to two distinct ways of solving $\alpha = \mathit{Int} \Rightarrow \beta = \mathit{Int}$ for β: the first sets $\beta = \alpha$ and uses the hypothesis, the second sets $\beta = \mathit{Int}$ 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. One solution is to require result-type annotations for all phantom-type **case** expressions (or propagate type signatures to these places). 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." We leave the study of (partial) type inference in the presence of equations to future work.

7 Related Work

Part of the inspiration for equality types and type equations comes from attempting to implement the type representations and **typecase** of Crary, Weirich, and Morrisett's λ_R [7]. 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 $\lambda_=$. Type λ-abstractions do not seem to be problematic 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 requires higher-order unification, which is undecidable.

Xi, Chen, and Chen [24] have introduced *guarded recursive data types*, or data types whose type arguments may be constrained in each case. Guarded recursive data types can be also be used to implement type representations, 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 more general than guarded recursive data types, since the guards can be interpreted as equations of the form $\alpha = \tau : \star$, whereas phantom type equations can be of any form and have any kind.

Another closely 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 $\Delta \vdash \tau = \tau'$. Singleton kinds can encode hypotheses of the form $\alpha = \tau$ as $\alpha : S(\tau)$. We plan to study the relationship between singleton kinds and phantom type-style equational constraints.

8 Future Work

Our system has several limitations: for example, as we described earlier, we cannot quite implement a first class **Typerec**. We believe that something like our proposed "phantom union" construct (essentially, a phantom data type without run-time tags) could solve this problem. 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, 19]. A third direction for future work is broadening the scope of phantom type constraints beyond type equations. Some related systems, like Cayenne's dependent type system [3] and Dependent ML [25], can handle other constraint domains such as integer inequalities in order to capture many interesting data-structure invariants.

9 Conclusions

We have presented a language extension called phantom types that generalizes phantom type encoding tricks found in the literature in that it allows efficient construction and analysis of constrained values, whereas in prior approaches efficiency must be sacrificed for the capability to analyze values or vice versa. 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. Our construct also seems much simpler and easier to implement than other proposed language modifications that support similar features.

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 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM Press, October 2002.
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. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. Available from `http://research.microsoft.com/~simonpj/papers/hmap/`, 2002.
17. 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.
18. 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.
19. Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.
20. Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, 1995. Published as CMU Technical Report CMU-CS-95-226.
21. 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.
22. 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.
23. 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.
24. Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. Online draft, 2002. http://www.cs.bu.edu/fac/hwxi/GRecTypecon/PAPER/main.ps.
25. 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.

A $\lambda_=$ details

A.1 Syntax and Operational Semantics

$$
\begin{array}{rrcl}
(\textit{type contexts}) & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \\
(\textit{equation contexts}) & \Psi & ::= & \cdot \mid \Psi, \epsilon{:}\kappa \\
(\textit{substitutions}) & \Theta & ::= & \cdot \mid \Theta, \alpha = \tau \\
(\textit{data type contexts}) & \Sigma & ::= & \cdot \mid \Sigma; \mathbf{data}\ T\ \overline{\alpha{:}\kappa} = \Sigma_T \\
(\textit{data type signatures}) & \Sigma_T & ::= & \cdot \mid \Sigma_T \mid \exists \overline{\beta{:}\kappa}.C\ \overline{\sigma}\ \mathbf{with}\ \overline{\epsilon{:}\kappa'} \\
(\textit{term contexts}) & \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma
\end{array}
$$

$$
\begin{array}{rrcl}
(\textit{kinds}) & \kappa & ::= & \star \mid \kappa \to \kappa \\
(\textit{constructors}) & \tau & ::= & \alpha \mid \tau_1\ \tau_2 \mid \mathit{Int} \mid \to \mid \times \mid T \\
(\textit{equations}) & \epsilon & ::= & \tau_1 = \tau_2 \\
(\textit{types}) & \sigma & ::= & \tau \mid \mathit{Int} \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall \alpha{:}\kappa.\sigma \\
(\textit{expressions}) & e & ::= & i \mid x \mid \lambda x : \sigma.e \mid \mathbf{fix}\ f{:}\sigma.e \mid e_1\ e_2 \mid \langle e_1, e_2 \rangle \\
 & & \mid & \pi_1 e \mid \pi_2 e \mid e[\tau] \\
 & & \mid & C\ [\overline{\tau}]\ \overline{e} \mid \mathbf{case}[\sigma]\ e\ \mathbf{of}\ ms \mid \mathbf{fail} \\
(\textit{matches}) & ms & ::= & \cdot \mid C\ [\overline{\beta}]\ \overline{x} \to e \mid ms
\end{array}
$$

$$
\begin{array}{rrcl}
(\textit{values}) & v & ::= & i \mid \lambda x{:}\sigma.e \mid \langle e_1, e_2 \rangle \mid \Lambda \alpha{:}\kappa.e \\
 & & \mid & C\ [\overline{\tau}]\ \overline{e} \mid \mathbf{fail} \\
(\textit{evaluators}) & E & ::= & [\cdot] \mid E\ e \mid \pi_1\ E \mid \pi_2\ E \mid E[c] \mid \mathbf{case}[\sigma]\ E\ \mathbf{of}\ ms
\end{array}
$$

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

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

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

$$\mathbf{case}[\sigma]\ C\ [\overline{\tau}]\ \overline{e}\ \mathbf{of}\ \cdot \mapsto \mathbf{fail} \qquad E[\mathbf{fail}] \mapsto \mathbf{fail}$$

A.2 Well-Formedness

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

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

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

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

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

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

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

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

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

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

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

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

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

A.3 Unification

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

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

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

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

A.4 Type Equivalence

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

$$\frac{}{\Delta; \Psi, \tau_1 = \tau_2{:}\kappa \vdash \tau_1 = \tau_2 : \kappa} \quad \frac{\Delta \vdash \tau : \kappa}{\Delta; \Psi \vdash \tau = \tau : \kappa}$$

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

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

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

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

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

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

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

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

A.5 Typing

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

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

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

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

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

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

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

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

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

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

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

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