Poor man's generics and dynamics

JAMES CHENEY
Cornell University
Ithaca, NY 14853
(*e-mail:* jcheney@cs.cornell.edu)

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

Abstract

The recent years have seen a number of proposals for extending statically typed languages by dynamics or generics. Most proposals — if not all — require significant extensions to the underlying language. In this paper we show that this need not be the case. We propose a particularly lightweight extension that supports both dynamics and generics. Furthermore, the two features are smoothly integrated: dynamic values, for instance, can be passed to generic functions. Our proposal makes do with a standard Hindley-Milner type system augmented by existential types. Building upon these ideas we have implemented a small library that is readily usable both with Hugs and with the Glasgow Haskell compiler.

1 Introduction

A desirable feature of programming languages is *safety*. Broadly speaking, safe programming languages prevent untrapped errors at run time (Pierce, 2002). Safety can be achieved either by static checking, by dynamic checking, or by a combination of static and dynamic checks. Each approach has its pros and cons.

Dynamically typed languages, like Scheme and Java, preserve type information until run time, enabling concise definitions of utility functions (such as *show*, '`==`', *compare*) using dynamic casts, generic functions, or multimethods. However, type preservation introduces space and run-time overhead that it is up to the compiler to optimize away. Also, in dynamically typed languages, many type errors that could be caught at compile time are not detected until run time.

Statically typed languages, like Haskell and ML, are at the other extreme. Type-checking occurs at compile time and type information is discarded after compilation, so it is impossible to write functions whose behaviour depends on run-time type information. As a result, utility and communication functions cannot be defined once and for all. Instead, programmers must provide new versions of them for each new data type. Their definitions are essentially determined by type structure, but must be written out explicitly because this regular behaviour cannot be expressed using the Hindley-Milner type system.

Previous approaches to supporting generic programming, type-dependent optimizations, and dynamic casts within statically typed languages include:

- explicit dynamic typing (Leroy & Mauny, 1993; Abadi *et al.*, 1995), in which the language is augmented with a *Dynamic* type and a *typecase* or *cast* construct;
- polytypic programming (Jansson & Jeuring, 1997; Backhouse *et al.*, 1999; Hinze, 2000b), in which type-dependent functions written in a language extension are translated to pure polymorphic functions;
- ad-hoc polymorphism (Wadler & Blott, 1989; Hall *et al.*, 1996) (i.e. Haskell's type classes), in which types are associated with classes that indicate the presence of overloaded functions like '`==`';
- intensional polymorphism (Harper & Morrisett, 1995), in which type information is preserved throughout compilation and at run time, thus making the language effectively dynamically typed.

However, none of the above techniques is both easy to implement and powerful enough to support dynamics and generics. Explicit dynamic typing is nontrivial to implement and to prove type-safe, especially in the presence of polymorphism. Polytypic programming is typically implemented using source-to-source translation, and does not address dynamic typing. Type classes have long been present in Haskell, but are limited in expressiveness. Intensional polymorphism changes the language's semantics, precluding a simple type-erasing language implementation and complicating soundness proofs. Also, it is not clear how or whether dynamic typing and generic programming features can be safely combined; to date, each has been studied in isolation.

Recent work has addressed some of these shortcomings: Leroy and Mauny (1993) and Abadi et al. (1995) showed how to safely combine dynamics and polymorphism. Hinze and Peyton Jones (2001) have introduced derivable type classes, which permit polytypic definitions of type classes. Crary, Weirich, and Morrisett (1999) have reconciled intensional polymorphism with type erasure using explicit type representations. However, implementations of dynamics, derivable type classes, and type representations still seem to require substantial compiler modifications and correctness proofs.

In this paper, we show that this need not be the case. We present an encoding of type representations in Haskell 98 (Peyton Jones & Hughes, 1999) augmented with existential types, and show how to use type representations to define simple polytypic functions, type *Dynamic*, and finally generic functions of the same flavour as those definable by derivable type classes. This provides a simple and safe implementation for generics and dynamics in Haskell, with no additional compiler support or proofs of type soundness required. It also sheds light on the relationship between generics and dynamics and shows that they can coexist and interact peacefully.

The rest of the paper is structured as follows. Sec. 2 explains how to define type representations in Haskell and illustrates their use in programming functions that work for a family of types. Type representations are also at the heart of dynamics in Sec. 3. Sec. 4 shows how to achieve true genericity, the ability to define functions that work for all types. Finally, Sec. 5 reviews related work and Sec. 6 concludes. For reference, the complete implementation of the Haskell library is included in App. A.

2 Programming with type representations

Let's start modest. Assume that you have a possibly infinite family of types and you want to define a function, say, equality, that works for all types of this family (we call such a function *polytypic*). For concreteness, let us consider the family of types given by the following grammar.

$$\tau \quad ::= \quad \mathit{Int} \mid 1 \mid \tau + \tau \mid \tau \times \tau$$

We assume that the unit type, the sum type and the pair type are given by the following declarations (Haskell already offers isomorphic types but we introduce new types for reasons to become clear later).

$$\begin{array}{lcl} \mathbf{data}\ 1 & = & \mathit{Unit} \\ \mathbf{data}\ \alpha + \beta & = & \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta \\ \mathbf{data}\ \alpha \times \beta & = & \alpha :\!\times\!: \beta \end{array}$$

Now, in Haskell one would immediately fall back on the class system to implement such a type-indexed family of functions. The equality function is assigned the type[1]

$$(\texttt{==}) \quad :: \quad \forall \alpha \,.\, (\mathit{Eq}\ \alpha) \Rightarrow \alpha \to \alpha \to \mathit{Bool}.$$

The so-called class context '$(\mathit{Eq}\ \alpha) \Rightarrow$' makes explicit that the equality function does not work for all types but only for those types that are instances of the type class Eq.

The purpose of this section is to show that one can also do without type classes. The basic idea is to pass '`==`' an additional argument that *represents* the type at which the equality function is called. As a first try we could assign '`==`' the type $\forall \alpha \,.\, \mathit{Rep} \to \alpha \to \alpha \to \mathit{Bool}$, where Rep is the type of type representations. A moment's reflection, however, reveals that this won't work. The parametricity theorem (Wadler, 1989) implies that a function of this type must necessarily ignore its second and its third argument. The trick is to use a parametric type for type representations:

$$(\texttt{==}) \quad :: \quad \forall \alpha \,.\, \mathit{Rep}\ \alpha \to \alpha \to \alpha \to \mathit{Bool}.$$

Here $\mathit{Rep}\ \tau$ is the type representation of τ. The one-million-dollar question is, of course, how can we define such a type? There are at least two possibilities, both of which require extensions to Haskell's **data** construct.

[1] We will always write universal quantifiers explicitly.

2.1 Alternative 1: polymorphic data signatures

We inhabit $Rep\ \tau$ by defining for each type constructor a corresponding value constructor that represents the type. For the above family of types we introduce

$$\begin{array}{lll} R_{Int} & :: & Rep\ Int \\ R_1 & :: & Rep\ 1 \\ R_+ & :: & \forall\alpha\,.\,Rep\ \alpha \to (\forall\beta\,.\,Rep\ \beta \to Rep\ (\alpha+\beta)) \\ R_\times & :: & \forall\alpha\,.\,Rep\ \alpha \to (\forall\beta\,.\,Rep\ \beta \to Rep\ (\alpha\times\beta)). \end{array}$$

For instance, the type $1+(Int\times Int)$ is represented by the value $R_+\ R_1\ (R_\times\ R_{Int}\ R_{Int})$ of type $Rep\ (1+(Int\times Int))$.

Of course, the declarations above are not valid Haskell since data constructors can only be introduced via **data** declarations. But, let's accept this for the moment.

Given these prerequisites we can easily define a polytypic equality function that works for all representable types. We simply pattern match on the 'type' argument.

$$\begin{array}{lll} rEqual & :: & \forall\tau\,.\,Rep\ \tau \to \tau \to \tau \to Bool \\ rEqual\ (R_{Int})\ t_1\ t_2 & = & t_1 == t_2 \\ rEqual\ (R_1)\ t_1\ t_2 & = & \mathbf{case}\ (t_1,t_2)\ \mathbf{of} \\ & & \quad (Unit, Unit) \to True \\ rEqual\ (R_+\ r_\alpha\ r_\beta)\ t_1\ t_2 & = & \mathbf{case}\ (t_1,t_2)\ \mathbf{of} \\ & & \quad (Inl\ a_1, Inl\ a_2) \to rEqual\ r_\alpha\ a_1\ a_2 \\ & & \quad (Inl\ a_1, Inr\ b_2) \to False \\ & & \quad (Inr\ b_1, Inl\ a_2) \to False \\ & & \quad (Inr\ b_1, Inr\ b_2) \to rEqual\ r_\beta\ b_1\ b_2 \\ rEqual\ (R_\times\ r_\alpha\ r_\beta)\ t_1\ t_2 & = & \mathbf{case}\ (t_1,t_2)\ \mathbf{of} \\ & & \quad (a_1 :\times: b_1, a_2 :\times: b_2) \to rEqual\ r_\alpha\ a_1\ a_2 \\ & & \qquad\qquad\qquad\qquad\qquad \wedge\ rEqual\ r_\beta\ b_1\ b_2 \end{array}$$

Note that each equation defines a function whose type is a true instance of the declared type: $rEqual\ (R_{Int})$, for instance, has type $Int \to Int \to Bool$.

A polytypic function can also be defined in terms of other polytypic functions. The function *rElem*, for instance, implements polytypic list membership.

$$\begin{array}{lll} rElem & :: & \forall\tau\,.\,Rep\ \tau \to \tau \to [\tau] \to Bool \\ rElem\ r_\tau\ t\ x & = & or\ [rEqual\ r_\tau\ t\ a \mid a \leftarrow x] \end{array}$$

Remark 1

The function *rElem* hints at one essential difference to Haskell's class system. Consider the class-based variant of *rElem*:

$$elem\ ::\ \forall\alpha\,.\,(Eq\ \alpha) \Rightarrow \alpha \to \alpha \to Bool$$

The class context records precisely on which overloaded function *elem* depends. For instance, if *elem* additionally called *show* at type α, then $Show\ \alpha$ would be added to the class context. By contrast, the signature of the type-passing implementation would not change if it called a type-passing *rShow*. □

We have already noted that in Haskell data constructors can only be introduced by **data** declarations. Unfortunately, we cannot define $Rep\ \alpha$ via a **data** declaration since none of the data constructors has result type $Rep\ \alpha$. We require a slight extension that allows us to instantiate the declared type to some specific instance.

$$
\begin{array}{lcll}
\mathbf{data}\ Rep\ \tau & = & R_{Int} & \mathbf{with}\ \tau = Int \\
 & | & R_1 & \mathbf{with}\ \tau = 1 \\
 & | & \forall \alpha\, \beta\, .\, R_{+}\ (Rep\ \alpha)\ (Rep\ \beta) & \mathbf{with}\ \tau = \alpha + \beta \\
 & | & \forall \alpha\, \beta\, .\, R_{\times}\ (Rep\ \alpha)\ (Rep\ \beta) & \mathbf{with}\ \tau = \alpha \times \beta
\end{array}
$$

The idea is to assign R_{Int} the type $Rep\ \tau$ with the additional constraint that $\tau = Int$, recorded by the **with** clause. As an aside, note that a universal quantifier in front of a constructor name acts as an *existential quantifier*. So $R_{+}\ r_{\alpha}\ r_{\beta}$ has type $Rep\ \tau$ for *some* types α and β with $\tau = \alpha + \beta$. Existential types are supported both by Hugs and by the Glasgow Haskell compiler.

Remark 2

Equational type constraints have been studied in the context of module systems for ML (Leroy, 1994; Harper & Lillibridge, 1994) and typed closure conversion in the presence of intensional polymorphism (Minamide *et al.*, 1996). In those contexts, the problem is that important relationships between types are hidden by modular abstraction or existential quantification; it is solved using translucent sum types, singleton kinds, or restricted type equations in order to make enough type sharing information evident in the module or closure type.

Unfortunately, this prior work is not directly applicable, since it addresses type sharing between elements of a module or existential package, not type equations that depend on data type cases. For example, singleton kinds would not help because each case of *Rep* would require τ to have a different singleton kind. Perhaps this could be addressed using kind polymorphism, but this complicates the type system even further. □

Fortunately, we don't have to wait for yet another extension to Haskell's already quite impressive type system since **with** clauses can be simulated in Haskell 98 (augmented by existential types).

2.2 Alternative 2: equivalence types

In light of the Curry-Howard isomorphism we can view the constraint $\tau = Int$ as a proof obligation that the types τ and Int are equal. If types are propositions, then type equality amounts to propositional equivalence. Equivalence is typically defined as "if and only if": $A \equiv B$ means $(A \Rightarrow B) \wedge (B \Rightarrow A)$. Recall that conjunctions correspond to product types and implications to function types. Thus, we can define an *equivalence type* $\tau \leftrightarrow \tau'$ corresponding to the proposition that the types τ and τ' are equal as $(\tau \rightarrow \tau') \times (\tau' \rightarrow \tau)$. The function components can be thought of as casts from τ to τ' and back. In Haskell we declare

$$\mathbf{data}\ \alpha \leftrightarrow \beta \quad = \quad EP\{from :: \alpha \rightarrow \beta, to :: \beta \rightarrow \alpha\}.$$

An element ep of type $\tau \leftrightarrow \tau'$ is a proof that the two types are equal. Given a proof of equivalence, we can use the casts *from ep* and *to ep* to convert one into the other.

Turning to the definition of *Rep* we replace each type constraint **with** $\tau = \tau'$ by the corresponding equivalence type $\tau \leftrightarrow \tau'$.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Rep}\ \tau & = & R_{\mathit{Int}}\ (\tau \leftrightarrow \mathit{Int}) \\
& | & R_1\ (\tau \leftrightarrow 1) \\
& | & \forall \alpha\, \beta\, .\, R_{+}\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta)\ (\tau \leftrightarrow (\alpha + \beta)) \\
& | & \forall \alpha\, \beta\, .\, R_{\times}\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta)\ (\tau \leftrightarrow (\alpha \times \beta))
\end{array}
$$

Additionally, we introduce so-called *smart constructors* corresponding to the R_τ constructors of the previous section that incorporate the reflexive equivalence value *self* as a proof of equivalence.

$$
\begin{array}{lcl}
\mathit{self} & :: & \forall \alpha\, .\, \alpha \leftrightarrow \alpha \\
\mathit{self} & = & \mathit{EP}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\} \\
r_{\mathit{Int}} & :: & \mathit{Rep}\ \mathit{Int} \\
r_{\mathit{Int}} & = & R_{\mathit{Int}}\ \mathit{self} \\
r_1 & :: & \mathit{Rep}\ 1 \\
r_1 & = & R_1\ \mathit{self} \\
r_{+} & :: & \forall \alpha\, .\, \mathit{Rep}\ \alpha \rightarrow (\forall \beta\, .\, \mathit{Rep}\ \beta \rightarrow \mathit{Rep}\ (\alpha + \beta)) \\
r_{+}\ r_\alpha\ r_\beta & = & R_{+}\ r_\alpha\ r_\beta\ \mathit{self} \\
r_{\times} & :: & \forall \alpha\, .\, \mathit{Rep}\ \alpha \rightarrow (\forall \beta\, .\, \mathit{Rep}\ \beta \rightarrow \mathit{Rep}\ (\alpha \times \beta)) \\
r_{\times}\ r_\alpha\ r_\beta & = & R_{\times}\ r_\alpha\ r_\beta\ \mathit{self}
\end{array}
$$

It remains to adapt the polytypic equality function to the new definition of *Rep*. The changes are straightforward: whenever we analyze a value e of type $\tau = \tau'$, we replace e by *from ep e* where $ep :: \tau \leftrightarrow \tau'$ is the corresponding proof of equivalence.

$$
\begin{array}{lcl}
\mathit{rEqual} & :: & \forall \tau\, .\, \mathit{Rep}\ \tau \rightarrow \tau \rightarrow \tau \rightarrow \mathit{Bool} \\
\mathit{rEqual}\ (R_{\mathit{Int}}\ \mathit{ep})\ t_1\ t_2 & = & \mathit{from}\ \mathit{ep}\ t_1 == \mathit{from}\ \mathit{ep}\ t_2 \\
\mathit{rEqual}\ (R_1\ \mathit{ep})\ t_1\ t_2 & = & \mathbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \mathbf{of} \\
& & \quad (\mathit{Unit}, \mathit{Unit}) \rightarrow \mathit{True} \\
\mathit{rEqual}\ (R_{+}\ r_\alpha\ r_\beta\ \mathit{ep})\ t_1\ t_2 & = & \mathbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \mathbf{of} \\
& & \quad (\mathit{Inl}\ a_1, \mathit{Inl}\ a_2) \rightarrow \mathit{rEqual}\ r_\alpha\ a_1\ a_2 \\
& & \quad (\mathit{Inl}\ a_1, \mathit{Inr}\ b_2) \rightarrow \mathit{False} \\
& & \quad (\mathit{Inr}\ b_1, \mathit{Inl}\ a_2) \rightarrow \mathit{False} \\
& & \quad (\mathit{Inr}\ b_1, \mathit{Inr}\ b_2) \rightarrow \mathit{rEqual}\ r_\beta\ b_1\ b_2 \\
\mathit{rEqual}\ (R_{\times}\ r_\alpha\ r_\beta\ \mathit{ep})\ t_1\ t_2 & = & \mathbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \mathbf{of} \\
& & \quad (a_1 :\times: b_1, a_2 :\times: b_2) \rightarrow \mathit{rEqual}\ r_\alpha\ a_1\ a_2 \\
& & \qquad\qquad\qquad \wedge\ \mathit{rEqual}\ r_\beta\ b_1\ b_2
\end{array}
$$

It is important to note that *rEqual* is polymorphically recursive: the recursive calls are at the existentially quantified types α and β. This means that the type signature is mandatory, otherwise the code would not typecheck.

The equality function takes values of the 'generic' type τ apart. When we *construct* a value of type $\tau = \tau'$, then we must wrap *to ep* around the constructed value. The polytypic function *rMinBound*, which constructs the least value of a type, serves as an example.

$$
\begin{array}{lcl}
\mathit{rMinBound} & :: & \forall \tau \,.\, \mathit{Rep}\ \tau \to \tau \\
\mathit{rMinBound}\ (R_{\mathit{Int}}\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{minBound}) \\
\mathit{rMinBound}\ (R_{1}\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{Unit}) \\
\mathit{rMinBound}\ (R_{+}\ r_\alpha\ r_\beta\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{Inl}\ (\mathit{rMinBound}\ r_\alpha)) \\
\mathit{rMinBound}\ (R_{\times}\ r_\alpha\ r_\beta\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{rMinBound}\ r_\alpha :\times: \mathit{rMinBound}\ r_\beta)
\end{array}
$$

The polytypic function *rMemo*, which memoizes a given function, is an intriguing example of a function that both analyzes and synthesizes values of the generic type.

$$
\begin{array}{lcl}
\mathit{rMemo} & :: & \forall \tau\ \nu \,.\, \mathit{Rep}\ \tau \to (\tau \to \nu) \to (\tau \to \nu) \\
\mathit{rMemo}\ (R_{\mathit{Int}}\ \mathit{ep})\ f & = & \lambda t \to f\ t \qquad \text{-- no memoization} \\
\mathit{rMemo}\ (R_{1}\ \mathit{ep})\ f & = & \lambda t \to \mathbf{case}\ \mathit{from}\ \mathit{ep}\ t\ \mathbf{of} \\
 & & \qquad \mathit{Unit} \to \mathit{fUnit} \\
\quad \mathbf{where}\ \mathit{fUnit} & = & f\ (\mathit{to}\ \mathit{ep}\ (\mathit{Unit})) \\
\mathit{rMemo}\ (R_{+}\ r_\alpha\ r_\beta\ \mathit{ep})\ f & = & \lambda t \to \mathbf{case}\ \mathit{from}\ \mathit{ep}\ t\ \mathbf{of} \\
 & & \qquad \mathit{Inl}\ a \to \mathit{fInl}\ a \\
 & & \qquad \mathit{Inr}\ b \to \mathit{fInr}\ b \\
\quad \mathbf{where}\ \mathit{fInl} & = & \mathit{rMemo}\ r_\alpha\ (\lambda a \to f\ (\mathit{to}\ \mathit{ep}\ (\mathit{Inl}\ a))) \\
\qquad\qquad \mathit{fInr} & = & \mathit{rMemo}\ r_\beta\ (\lambda b \to f\ (\mathit{to}\ \mathit{ep}\ (\mathit{Inr}\ b))) \\
\mathit{rMemo}\ (R_{\times}\ r_\alpha\ r_\beta\ \mathit{ep})\ f & = & \lambda t \to \mathbf{case}\ \mathit{from}\ \mathit{ep}\ t\ \mathbf{of} \\
 & & \qquad a :\times: b \to \mathit{fPair}\ a\ b \\
\quad \mathbf{where}\ \mathit{fPair} & = & \mathit{rMemo}\ r_\alpha\ (\lambda a \to \mathit{rMemo}\ r_\beta\ (\lambda b \to f\ (\mathit{to}\ \mathit{ep}\ (a :\times: b))))
\end{array}
$$

To see how *rMemo* works note that the helper definitions *fUnit*, *fInl*, *fInr*, and *fPair* do not depend on the actual argument of f. Thus, once f is given, they can be readily computed. Memoization relies critically on the fact that they are computed only on demand and then at most once. This is guaranteed if the implementation is *fully lazy*. The interested reader is referred to Hinze (2000a) for background information.[2]

2.3 A type class for type representations

When we call a polytypic function, we have to supply a type representation as an argument. Of course, if we are working in Haskell, we can use the class system for

[2] In (Hinze, 2000a) memoization is defined as the composition of a function that constructs a memo table and a function that queries the table. If we fuse the two functions thereby eliminating the memo data structure, we obtain the *rMemo* function above. Thanks are due to Koen Claessen for bringing this to our attention. Despite appearance, the memo data structures did not vanish into thin air. Rather, they are now built into the closures. For instance, the memo table for a disjoint union is a pair of memo tables. The closure for $\mathit{rMemo}\ (R_{+}\ r_\alpha\ r_\beta\ \mathit{ep})\ f$ consequently contains a pair of memoized functions, namely *fInl* and *fInr*.

this purpose. It is important to note, however, that this use of the class system is only a convenience and nothing essential.

$$
\begin{array}{l}
\textbf{class } \mathit{Representable}\ \tau\ \textbf{where} \\
\quad \mathit{rep} \quad :: \quad \mathit{Rep}\ \tau \\
\textbf{instance } \mathit{Representable}\ \mathit{Int}\ \textbf{where} \\
\quad \mathit{rep} \quad = \quad r_{\mathit{Int}} \\
\textbf{instance } \mathit{Representable}\ 1\ \textbf{where} \\
\quad \mathit{rep} \quad = \quad r_1 \\
\textbf{instance } (\mathit{Representable}\ \alpha, \mathit{Representable}\ \beta) \Rightarrow \mathit{Representable}\ (\alpha + \beta)\ \textbf{where} \\
\quad \mathit{rep} \quad = \quad r_{+}\ \mathit{rep}\ \mathit{rep} \\
\textbf{instance } (\mathit{Representable}\ \alpha, \mathit{Representable}\ \beta) \Rightarrow \mathit{Representable}\ (\alpha \times \beta)\ \textbf{where} \\
\quad \mathit{rep} \quad = \quad r_{\times}\ \mathit{rep}\ \mathit{rep}
\end{array}
$$

Building upon the type class we can define a variant of polytypic equality that automatically constructs the type representation.

$$
\begin{array}{lcl}
\mathit{cEqual} & :: & \forall \tau . (\mathit{Representable}\ \tau) \Rightarrow \tau \to \tau \to \mathit{Bool} \\
\mathit{cEqual}\ t_1\ t_2 & = & \mathit{rEqual}\ \mathit{rep}\ t_1\ t_2
\end{array}
$$

2.4 New types

So far we have interpreted $\tau \leftrightarrow \tau'$ as a proof of type equality. Actually, we can be more liberal and interpret an element of $\tau \leftrightarrow \tau'$ as a proof that the two types are *isomorphic* rather than equal. Coincidentally, Haskell offers a linguistic construct for introducing a new type that is isomorphic to an existing type. Consider as an example the type of triples.

$$\textbf{newtype } \mathit{Tri}\ \alpha\ \beta\ \gamma \quad = \quad \mathit{To}_{\mathit{Tri}}\{\mathit{from}_{\mathit{Tri}} :: \alpha \times (\beta \times \gamma)\}$$

The declaration defines $\mathit{Tri}\ \alpha\ \beta\ \gamma$ to be isomorphic to $\alpha \times (\beta \times \gamma)$ with the functions $\mathit{To}_{\mathit{Tri}}$ and $\mathit{from}_{\mathit{Tri}}$ witnessing the isomorphism. Since $\mathit{Tri}\ \alpha\ \beta\ \gamma$ is isomorphic to an existing type, we can represent it as follows.

$$
\begin{array}{lcl}
r_{\mathit{Tri}} & :: & \forall \alpha . \mathit{Rep}\ \alpha \to (\forall \beta . \mathit{Rep}\ \beta \to (\forall \gamma . \mathit{Rep}\ \gamma \to \mathit{Rep}\ (\mathit{Tri}\ \alpha\ \beta\ \gamma))) \\
r_{\mathit{Tri}}\ r_\alpha\ r_\beta\ r_\gamma & = & R_{\times}\ r_\alpha\ (r_{\times}\ r_\beta\ r_\gamma)\ (\mathit{EP}\ \mathit{from}_{\mathit{Tri}}\ \mathit{To}_{\mathit{Tri}})
\end{array}
$$

Now, if we pass a triple to a polytypic function, then the isomorphisms $\mathit{from}_{\mathit{Tri}}$ and $\mathit{To}_{\mathit{Tri}}$ are automatically called at the appropriate places. Furthermore, since $\mathit{Tri}\ \alpha\ \beta\ \gamma$ is *implemented* by the type on the right-hand side, both casts amount, in fact, to the identity function at run time. On the downside, this also means that a new type and its implementation type are treated alike. This may or may not what you want.

2.5 Genuine type equality

We have seen in the previous section that the equivalence type $\tau \leftrightarrow \tau'$ does not guarantee that τ and τ' are actually equal (and we have used this to good effect).

An intriguing question is whether we can devise an equivalence type that only admits equal types. Perhaps surprisingly, this is indeed possible and even more surprisingly, the underlying idea goes back to Leibniz—see Wadler (2001) for a related application of Leibniz's idea. According to Leibniz, two terms are equal if one may be substituted for the other. Adapting this principle to types, we define,

$$\textbf{newtype}\ \alpha \leftrightarrow \beta \quad = \quad \mathit{EP}\{\,\mathit{unEP} :: \forall \phi\,.\,\phi\ \alpha \rightarrow \phi\ \beta\,\}.$$

Note that the universally quantified type variable ϕ ranges over type constructors of kind $\star \rightarrow \star$. Thus, an element of $\alpha \leftrightarrow \beta$ is a function that converts an element of type $\phi\ \alpha$ into an element of $\phi\ \beta$ for *any* type constructor ϕ. Clearly, such a function is impossible to define unless $\alpha = \beta$.

$$\begin{array}{lcl} \mathit{self} & :: & \forall \alpha\,.\,\alpha \leftrightarrow \alpha \\ \mathit{self} & = & \mathit{EP}\ \mathit{id} \end{array}$$

To provide the same functionality as before we have to define functions that given a proof of type equality cast one type into the other. The *from* function is easy: we substitute the identity type for ϕ.

$$\begin{array}{lcl} \textbf{newtype}\ \mathit{Id}\ a & = & \mathit{Id}\{\,\mathit{unId} :: \alpha\,\} \\ \mathit{from} & :: & \forall \alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \\ \mathit{from}\ \mathit{ep} & = & \mathit{unId} \cdot \mathit{unEP}\ \mathit{ep} \cdot \mathit{Id} \end{array}$$

The *to* function is more subtle: we first substitute the type constructor $\psi\ x = x \rightarrow \alpha$ for ϕ. This gives us a function of type $(\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \alpha)$, which we then apply to the identity function.

$$\begin{array}{lcl} \textbf{newtype}\ \alpha \leftarrow \beta & = & \mathit{Inv}\{\,\mathit{unInv} :: \beta \rightarrow \alpha\,\} \\ \mathit{to} & :: & \forall \alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \rightarrow (\beta \rightarrow \alpha) \\ \mathit{to}\ \mathit{ep} & = & \mathit{unInv}\ (\mathit{unEP}\ \mathit{ep}\ (\mathit{Inv}\ \mathit{id})) \end{array}$$

Interestingly, both definitions of $\alpha \leftrightarrow \beta$ have precursors in the literature: they underly, for instance, Weirich's implementation of type-safe cast (2000). Though the above definition of $\alpha \leftrightarrow \beta$ is attractive, we will stick to the first variant as we rely on the broader interpretation of type equivalence as types being isomorphic.

3 Dynamics

3.1 The type Dynamic

A dynamic value is a pair consisting of the value itself and a representation of its type. Using the type of type representations *Rep* we can readily define

$$\begin{array}{lcl} \textbf{data}\ \mathit{Dynamic} & = & \forall \alpha\,.\,\mathit{Dyn}\ (\mathit{Rep}\ \alpha)\ \alpha \\ \mathit{dynamic} & :: & \forall \alpha\,.\,(\mathit{Representable}\ \alpha) \Rightarrow \alpha \rightarrow \mathit{Dynamic} \\ \mathit{dynamic}\ a & = & \mathit{Dyn}\ \mathit{rep}\ a. \end{array}$$

The existential quantifier (which is written as a universal quantifier) effectively hides the type of the dynamic value. It goes without saying that dynamic values thus defined are first-class citizens: we can, for instance, construct a list of dynamic values.

$$[\mathit{dynamic}\ (4711 :: \mathit{Int}), \mathit{dynamic}\ \texttt{'F'}, \mathit{dynamic}\ (\lambda n \to n + 1 :: \mathit{Int})] :: [\mathit{Dynamic}]$$

Now, what can we do with a value $\mathit{Dyn}\ r_\alpha\ a$ of type *Dynamic*? Not that much so far. If we have a polytypic function f of type $\forall\tau\,.\,\mathit{Rep}\ \tau \to \tau \to \nu$ where τ does not appear in ν, then we can call $f\ r_\alpha\ a$. The polytypic function analyzes the type representation r_α and takes the appropriate action (the existentially quantified type variable cannot escape, since τ does not appear in ν). However, we cannot take the equality of two dynamic values or cast a dynamic value into a static value of a given type. For both applications we have to check two type representations for equality, which is what we tackle next.

The function *unify* takes two type representations and possibly returns a proof of their equivalence.

$$\mathit{unify} \quad :: \quad \forall\tau_1\ \tau_2\,.\,\mathit{Rep}\ \tau_1 \to \mathit{Rep}\ \tau_2 \to [\tau_1 \leftrightarrow \tau_2]$$

The result type is $[\tau_1 \leftrightarrow \tau_2]$ rather than $\mathit{Maybe}\ (\tau_1 \leftrightarrow \tau_2)$ so that we can use list comprehensions for defining *unify* (see below). The run-time unification essentially combines proofs of equivalence. Assume, for instance, that *unify* is called with $R_{\mathit{Int}}\ \mathit{ep}_1 :: \mathit{Rep}\ \tau_1$ and $R_{\mathit{Int}}\ \mathit{ep}_2 :: \mathit{Rep}\ \tau_2$. The proofs $\mathit{ep}_1 :: \tau_1 \leftrightarrow \mathit{Int}$ and $\mathit{ep}_2 :: \tau_2 \leftrightarrow \mathit{Int}$ can be combined into a proof of $\tau_1 \leftrightarrow \tau_2$ by applying the laws of symmetry and transitivity. These laws correspond to the functions *inv* and '$\diamond$', respectively.

$$\begin{array}{lcl}
\mathit{inv} & :: & \forall\alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \to (\beta \leftrightarrow \alpha) \\
\mathit{inv}\ f & = & \mathit{EP}\{\mathit{from} = \mathit{to}\ f, \mathit{to} = \mathit{from}\ f\} \\
(\diamond) & :: & \forall\alpha\ \beta\ \gamma\,.\,(\beta \leftrightarrow \gamma) \to (\alpha \leftrightarrow \beta) \to (\alpha \leftrightarrow \gamma) \\
f \diamond g & = & \mathit{EP}\{\mathit{from} = \mathit{from}\ f \cdot \mathit{from}\ g, \mathit{to} = \mathit{to}\ g \cdot \mathit{to}\ f\}
\end{array}$$

The unification procedure is slightly more complicated for parametric types. Assume, for instance, that *unify* is passed the representations $R_+\ r_{\alpha_1}\ r_{\beta_1}\ \mathit{ep}_1 :: \mathit{Rep}\ \tau_1$ and $R_+\ r_{\alpha_2}\ r_{\beta_2}\ \mathit{ep}_2 :: \mathit{Rep}\ \tau_2$. The arguments provide proofs $\mathit{ep}_1 :: \tau_1 \leftrightarrow (\alpha_1 + \beta_1)$ and $\mathit{ep}_2 :: \tau_2 \leftrightarrow (\alpha_2 + \beta_2)$. Through recursive invocations of *unify* we possibly obtain proofs of $\alpha_1 \leftrightarrow \alpha_2$ and $\beta_1 \leftrightarrow \beta_2$. Using the law of congruence we can then construct a proof of $(\alpha_1 + \beta_1) \leftrightarrow (\alpha_2 + \beta_2)$. Finally, the subproofs are combined into a proof of $\tau_1 \leftrightarrow \tau_2$ by applying symmetry and transitivity. The law of congruence corresponds to the function '$\oplus$' defined below.

$$\begin{array}{lcl}
(+) & :: & \forall\alpha\ \beta\,.\,(\alpha \to \beta) \to (\forall\gamma\ \delta\,.\,(\gamma \to \delta) \to ((\alpha + \gamma) \to (\beta + \delta))) \\
(f + g)\ (\mathit{Inl}\ a) & = & \mathit{Inl}\ (f\ a) \\
(f + g)\ (\mathit{Inr}\ b) & = & \mathit{Inr}\ (g\ b) \\
(\oplus) & :: & \forall\alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \to (\forall\gamma\ \delta\,.\,(\gamma \leftrightarrow \delta) \to ((\alpha + \gamma) \leftrightarrow (\beta + \delta))) \\
f \oplus g & = & \mathit{EP}\{\mathit{from} = \mathit{from}\ f + \mathit{from}\ g, \mathit{to} = \mathit{to}\ f + \mathit{to}\ g\}
\end{array}$$

Remark 3

As an aside for the more theoretically minded reader, types and equivalence types form the objects and arrows of a category. The identity arrow is *self* and '$\diamond$' implements the composition of arrows. Furthermore, the sum type is a bifunctor in this category; its action on arrows is given by '$\oplus$'. In fact, since each arrow *ep* has an inverse *inv ep*, every parametric data type including the arrow type can be turned into a *covariant* functor. $\square$

Given these prerequisites *unify* can be defined as follows.

$$
\begin{array}{lll}
\mathit{unify} & :: & \forall \tau_1\ \tau_2 \,.\, \mathit{Rep}\ \tau_1 \to \mathit{Rep}\ \tau_2 \to [\tau_1 \leftrightarrow \tau_2] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\mathit{Int}}\ \mathit{ep}_1)\ (R_{\mathit{Int}}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond \mathit{ep}_1] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_1\ \mathit{ep}_1)\ (R_1\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond \mathit{ep}_1] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_+\ r_{\alpha_1}\ r_{\beta_1}\ \mathit{ep}_1)\ (R_+\ r_{\alpha_2}\ r_{\beta_2}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond (\mathit{ep}_\alpha \oplus \mathit{ep}_\beta) \diamond \mathit{ep}_1 \mid \mathit{ep}_\alpha \leftarrow \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}, \mathit{ep}_\beta \leftarrow \mathit{unify}\ r_{\beta_1}\ r_{\beta_2}] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_\times\ r_{\alpha_1}\ r_{\beta_1}\ \mathit{ep}_1)\ (R_\times\ r_{\alpha_2}\ r_{\beta_2}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond (\mathit{ep}_\alpha \otimes \mathit{ep}_\beta) \diamond \mathit{ep}_1 \mid \mathit{ep}_\alpha \leftarrow \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}, \mathit{ep}_\beta \leftarrow \mathit{unify}\ r_{\beta_1}\ r_{\beta_2}] \\
\multicolumn{3}{l}{\mathit{unify}\ \_\ \_} \\
 & = & [\,]
\end{array}
$$

Using run-time unification we can easily cast a dynamic value into a static value of a given representable type.

$$
\begin{array}{lll}
\mathit{rCast} & :: & \forall \tau \,.\, \mathit{Rep}\ \tau \to \mathit{Dynamic} \to \tau \\
\mathit{rCast}\ r_\tau\ (\mathit{Dyn}\ r_\alpha\ a) & = & \textbf{case}\ \mathit{unify}\ r_\tau\ r_\alpha\ \textbf{of} \\
 & & \quad [\mathit{ep}] \to \mathit{to}\ \mathit{ep}\ a \\
 & & \quad \_ \to \mathit{error}\ \texttt{"cast: type mismatch"} \\
\mathit{cast} & :: & (\mathit{Representable}\ \tau) \Rightarrow \mathit{Dynamic} \to \tau \\
\mathit{cast}\ d & = & \mathit{rCast}\ \mathit{rep}\ d
\end{array}
$$

Another useful function is dynamic function application, the application of a dynamic function to a dynamic value.

$$
\begin{array}{lll}
\mathit{apply} & :: & \mathit{Dynamic} \to \mathit{Dynamic} \to \mathit{Dynamic} \\
\multicolumn{3}{l}{\mathit{apply}\ (\mathit{Dyn}\ (R_\to\ r_\alpha\ r_\beta\ \mathit{ep})\ f)\ (\mathit{Dyn}\ r_{\alpha'}\ x)} \\
 & = & \textbf{case}\ \mathit{unify}\ r_\alpha\ r_{\alpha'}\ \textbf{of} \\
 & & \quad [\mathit{ep}'] \to \mathit{Dyn}\ r_\beta\ ((\mathit{from}\ \mathit{ep}\ f)\ (\mathit{to}\ \mathit{ep}'\ x)) \\
 & & \quad \_ \to \mathit{error}\ \texttt{"apply: type mismatch"} \\
\mathit{apply}\ \_\ \_ & = & \mathit{error}\ \texttt{"apply: not a function"}
\end{array}
$$

Example 1
Using dynamic values we can, for instance, implement a heterogeneous symbol table—a finite map from strings to values of any type.

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Table} & = & [(\mathit{String}, \mathit{Dynamic})] \\
\mathit{rLookup} & :: & \forall \tau \,.\, \mathit{Rep}\ \tau \to \mathit{String} \to \mathit{Table} \to \mathit{Maybe}\ \tau \\
\mathit{rLookup}\ r_\tau\ s\ t & = & \mathit{fmap}\ (\mathit{rCast}\ r_\tau)\ (\mathit{lookup}\ s\ t)
\end{array}
$$

For alternative solutions the reader is referred to Weirich (2000). $\square$

Example 2

Dynamic values also provide a simple way of implementing C's *printf* function in a type-safe manner (see also (Shields *et al.*, 1998)). The *printf* function allows the programmer to show an arbitrary number of arguments of different types. Both the number of arguments and their types are specified by a so-called format string which is passed as a first argument to *printf*. Since the format string may be unknown at compile time (because it may be read from an external source), *printf* cannot be statically type-checked.

$$
\begin{array}{lcl}
\mathit{printf} & :: & \mathit{String} \rightarrow [\mathit{Dynamic}] \rightarrow \mathit{ShowS} \\
\mathit{printf}\ \texttt{""}\ \_ & = & \mathit{showString}\ \texttt{""} \\
\mathit{printf}\ (\texttt{'\%'} : \texttt{'d'} : cs)\ (d : ds) & = & \mathit{shows}\ (\mathit{cast}\ d :: \mathit{Int}) \cdot \mathit{printf}\ cs\ ds \\
\mathit{printf}\ (\texttt{'\%'} : \texttt{'s'} : cs)\ (d : ds) & = & \mathit{showString}\ (\mathit{cast}\ d :: \mathit{String}) \cdot \mathit{printf}\ cs\ ds \\
\mathit{printf}\ (\texttt{'\%'} : \texttt{'\%'} : cs)\ ds & = & \mathit{showChar}\ \texttt{'\%'} \cdot \mathit{printf}\ cs\ ds \\
\mathit{printf}\ (\texttt{'\%'} : c : cs)\ [\,] & = & \mathit{error}\ \texttt{"printf: missing argument"} \\
\mathit{printf}\ (c : cs)\ ds & = & \mathit{showChar}\ c \cdot \mathit{printf}\ cs\ ds
\end{array}
$$

Note that the code assumes that the *String* type is representable. □

3.2 Closing the circle

We add *Dynamic* to the family of representable types, so that we can also pass a dynamic value to a polytypic function

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Rep}\ \tau & = & \ldots \\
 & | & R_{\mathit{Dynamic}}\ (\tau \leftrightarrow \mathit{Dynamic})
\end{array}
$$

Note that *Rep* τ and *Dynamic* are now defined by mutual recursion.

Of course, we have to extend the polytypic function definitions to take the new case into account. Here is how we take equality of two dynamic values.

$$
\begin{array}{l}
\mathit{rEqual}\ (R_{\mathit{Dynamic}}\ \mathit{ep})\ d_1\ d_2 \\
\quad = \ \ \textbf{case}\ (\mathit{from}\ \mathit{ep}\ d_1, \mathit{from}\ \mathit{ep}\ d_2)\ \textbf{of} \\
\qquad\qquad (\mathit{Dyn}\ r_{\alpha_1}\ a_1, \mathit{Dyn}\ r_{\alpha_2}\ a_2) \rightarrow \textbf{case}\ \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}\ \textbf{of} \\
\qquad\qquad\qquad [\mathit{ep}'] \rightarrow \mathit{rEqual}\ r_{\alpha_1}\ a_1\ (\mathit{to}\ \mathit{ep}'\ a_2) \\
\qquad\qquad\qquad \_ \rightarrow \mathit{False}
\end{array}
$$

We first determine whether the types of the dynamic values are equal. If this is the case, *rEqual* is called recursively to check whether the values are equal, as well.

4 Generics

4.1 Generic representation types

What have we achieved so far? Using type representations we can program a function that works uniformly for all *representable* types. Let's become more ambitious now. We aim at broadening the scope of the polytypic functions so that they work for *all* types including types that the programmer is yet to define. We won't achieve this goal in its full glory—this requires an external tool or some support from the compiler—but we will get pretty close. The programmer only has to do a bit of extra work for each newly introduced type.

The principal idea is to make every type representable. Consider as a first simple example the data type of Booleans.

$$\mathbf{data}\ \mathit{Bool} \quad = \quad \mathit{False} \mid \mathit{True}$$

Now, the type *Bool* is isomorphic to the sum type $1 + 1$, which is already representable.[3] Here are functions that convert to and fro.

$$\begin{array}{lcl}
\mathit{from}_{\mathit{Bool}} & :: & \mathit{Bool} \rightarrow 1 + 1 \\
\mathit{from}_{\mathit{Bool}}\ \mathit{False} & = & \mathit{Inl}\ \mathit{Unit} \\
\mathit{from}_{\mathit{Bool}}\ \mathit{True} & = & \mathit{Inr}\ \mathit{Unit} \\
\mathit{to}_{\mathit{Bool}} & :: & 1 + 1 \rightarrow \mathit{Bool} \\
\mathit{to}_{\mathit{Bool}}\ (\mathit{Inl}\ \mathit{Unit}) & = & \mathit{False} \\
\mathit{to}_{\mathit{Bool}}\ (\mathit{Inr}\ \mathit{Unit}) & = & \mathit{True}
\end{array}$$

Using these isomorphisms we can represent the type *Bool* as follows.

$$\begin{array}{lcl}
r_{\mathit{Bool}} & :: & \mathit{Rep}\ \mathit{Bool} \\
r_{\mathit{Bool}} & = & R_{+}\ r_1\ r_1\ (\mathit{EP}\ \mathit{from}_{\mathit{Bool}}\ \mathit{to}_{\mathit{Bool}})
\end{array}$$

The type $1 + 1$ is the so-called *generic representation type* of *Bool*, see (Hinze & Peyton Jones, 2001).

Since Haskell's **data** construct introduces a sum of products and both sums and products are representable, it looks as if we are done. But not quite, data types may be recursively defined and they may be parametric (possibly abstracting over higher-order kinded type constructors). We will address each point in turn.

Haskell's list data type serves as a nice example for a data type that is both recursive and parametric.

$$\mathbf{data}\ [\alpha] \quad = \quad [\,] \mid \alpha : [\alpha]$$

Its generic representation type is $1 + (\alpha \times [\alpha])$ with the isomorphisms given by

$$\begin{array}{lcl}
\mathit{from}_{[\,]} & :: & \forall \alpha\,.\,[\alpha] \rightarrow 1 + (\alpha \times [\alpha]) \\
\mathit{from}_{[\,]}\ [\,] & = & \mathit{Inl}\ \mathit{Unit} \\
\mathit{from}_{[\,]}\ (a : as) & = & \mathit{Inr}\ (a :\!\times\!: as) \\
\mathit{to}_{[\,]} & :: & \forall \alpha\,.\,1 + (\alpha \times [\alpha]) \rightarrow [\alpha] \\
\mathit{to}_{[\,]}\ (\mathit{Inl}\ \mathit{Unit}) & = & [\,] \\
\mathit{to}_{[\,]}\ (\mathit{Inr}\ (a :\!\times\!: as)) & = & a : as.
\end{array}$$

Thus, the following representation of $[\alpha]$ suggests itself.

$$\begin{array}{lcl}
r_{[\,]} & :: & \forall \alpha\,.\,\mathit{Rep}\ \alpha \rightarrow \mathit{Rep}\ [\alpha] \\
r_{[\,]}\ r_\alpha & = & R_{+}\ r_1\ (r_{\times}\ r_\alpha\ (r_{[\,]}\ r_\alpha))
\end{array}$$

[3] Strictly speaking, the types *Bool* and $1 + 1$ are not isomorphic in Haskell since 1 contains an additional bottom element. We simply ignore this complication here.

Two points are worth noting. First, the type representation of the list type constructor is a *function* taking a representation of τ to a representation of $[\tau]$. Second, the representation of $[\tau]$ is given by an *infinite* term as type recursion is mapped onto value recursion. Since Haskell is a lazy language, this is not a problem as far as the polytypic functions are concerned: the call $\mathit{rEqual}\ (r_{[\,]}\ r_{\mathit{Int}})\ [0\,..\,8]\ [0\,..\,9]$, for instance, happily evaluates to *False*. However, the presence of infinite terms renders the unification of representations impossible: quite annoyingly the call $\mathit{cast}\ (\mathit{dynamic}\ [0\,..\,9 :: \mathit{Int}]) :: [\mathit{Int}]$ does not terminate.

There are at least two solutions to this problem. We could represent type recursion explicitly by introducing a fixed point operator—this is the approach taken in PolyP (Jansson & Jeuring, 1997). However, this is technically rather awkward and never completely general as type recursion may span over several types (mutual recursion) and as it may involve type constructors rather than types (so-called *nested data types*, see (Bird & Meertens, 1998)). Thus, we would need an infinite family of fixed point operators. Furthermore, checking equality of higher-order kinded type constructors is undecidable. Haskell avoids the latter problem by using *name equivalence* rather than *structural equivalence*, which motivates the second solution.

We continue to represent recursive types by infinite terms but additionally label the representation by its Haskell name. The following data type is sufficient for capturing closed Haskell type terms (including higher-order kinded types).

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Term} & = & \mathit{App}\ \mathit{String}\ [\mathit{Term}] \\
 & & \mathbf{deriving}\ (\mathit{Eq})
\end{array}
$$

For instance, $[\mathit{Int}]$ is represented by $\mathit{App}\ \texttt{"[]"}\ [\mathit{App}\ \texttt{"Int"}\ [\,]]$. We augment $\mathit{Rep}\ \tau$ by one additional constructor.

$$
\begin{array}{lcll}
\mathbf{data}\ \mathit{Rep}\ \tau & = & R_{\mathit{Int}} & (\tau \leftrightarrow \mathit{Int}) \\
 & | & R_{\mathit{Dynamic}} & (\tau \leftrightarrow \mathit{Dynamic}) \\
 & | & R_1 & (\tau \leftrightarrow 1) \\
 & | & \forall \alpha\, \beta\,.\, R_{+}\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta) & (\tau \leftrightarrow (\alpha + \beta)) \\
 & | & \forall \alpha\, \beta\,.\, R_{\times}\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta) & (\tau \leftrightarrow (\alpha \times \beta)) \\
 & | & \forall \alpha\ \ .\, R_{\mathit{Type}}\ \mathit{Term}\ (\mathit{Rep}\ \alpha) & (\tau \leftrightarrow \alpha)
\end{array}
$$

On the face of it we now have two types for type representations: $\mathit{Rep}\ \tau$ captures the structural information and *Term* captures the naming information. The function $\mathit{term}_{\star}$ extracts the latter information from the former.

$$
\begin{array}{lcl}
\mathit{term}_{\star} & :: & \forall \tau\,.\, \mathit{Rep}\ \tau \to \mathit{Term} \\
\mathit{term}_{\star}\ (R_{\mathit{Int}}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"Int"}\ [\,] \\
\mathit{term}_{\star}\ (R_{\mathit{Dynamic}}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"Dynamic"}\ [\,] \\
\mathit{term}_{\star}\ (R_1\ \mathit{ep}) & = & \mathit{App}\ \texttt{"1"}\ [\,] \\
\mathit{term}_{\star}\ (R_{+}\ r_{\alpha}\ r_{\beta}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"(+)"}\ [\mathit{term}_{\star}\ r_{\alpha}, \mathit{term}_{\star}\ r_{\beta}] \\
\mathit{term}_{\star}\ (R_{\times}\ r_{\alpha}\ r_{\beta}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"(*)"}\ [\mathit{term}_{\star}\ r_{\alpha}, \mathit{term}_{\star}\ r_{\beta}] \\
\mathit{term}_{\star}\ (R_{\mathit{Type}}\ t\ r_{\alpha}\ \mathit{ep}) & = & t
\end{array}
$$

We change the definition of $r_{[\,]}$ to incorporate the list type name.

$$\begin{array}{lcl} r_{[\,]} & :: & \forall \alpha \,.\, Rep\ \alpha \rightarrow Rep\ [\alpha] \\ r_{[\,]}\ r_\alpha & = & R_{Type}\ (App\ \texttt{"[]"}\ [term_\star\ r_\alpha]) \\ & & \qquad (r_+\ r_1\ (r_\times\ r_\alpha\ (r_{[\,]}\ r_\alpha)))\ (EP\ from_{[\,]}\ to_{[\,]}) \end{array}$$

It remains to extend the definition of *unify*. To unify two labelled representations $R_{Type}\ t_1\ r_{\alpha_1}\ ep_1$ and $R_{Type}\ t_2\ r_{\alpha_2}\ ep_2$, we simply test t_1 and t_2 for equality.

$$\begin{array}{l} unify\ (R_{Type}\ t_1\ r_{\alpha_1}\ ep_1)\ (R_{Type}\ t_2\ r_{\alpha_2}\ ep_2) \\ \quad = \quad [inv\ ep_2 \diamond head\ (unify\ r_{\alpha_1}\ r_{\alpha_2}) \diamond ep_1 \mid t_1 == t_2] \end{array}$$

If t_1 and t_2 are equal, we know that r_{α_1} and r_{α_2} must be unifiable. The call $head\ (unify\ r_{\alpha_1}\ r_{\alpha_2})$ immediately returns the proof. Note that the following definition of *unify* does not work, as it is too eager.

$$\begin{array}{l} unify\ (R_{Type}\ t_1\ r_{\alpha_1}\ ep_1)\ (R_{Type}\ t_2\ r_{\alpha_2}\ ep_2) \qquad \text{-- WRONG} \\ \quad = \quad [inv\ ep_2 \diamond ep_\alpha \diamond ep_1 \mid t_1 == t_2, ep_\alpha \leftarrow unify\ r_{\alpha_1}\ r_{\alpha_2}] \end{array}$$

Turning to the treatment of higher-order kinded data types let us first take a look at one popular example. The following definition introduces so-called *generalized rose trees*.

$$\mathbf{data}\ Tree\ \phi\ \alpha \quad = \quad Node\ \alpha\ (\phi\ (Tree\ \phi\ \alpha))$$

Since the first argument of *Tree* ranges over type constructors of kind $\star \rightarrow \star$, *Tree* has kind $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$. We have already seen that a type constructor ϕ of kind $\star \rightarrow \star$ is represented by a function of type $\forall \alpha \,.\, Rep\ \alpha \rightarrow Rep\ (\phi\ \alpha)$. Since *Tree* abstracts over type constructors of this kind it has type

$$r_{Tree} \quad :: \quad \forall \phi \,.\, (\forall \alpha \,.\, Rep\ \alpha \rightarrow Rep\ (\phi\ \alpha)) \rightarrow (\forall \alpha \,.\, Rep\ \alpha \rightarrow Rep\ (Tree\ \phi\ \alpha)).$$

Note that r_{Tree} possesses a so-called *rank-2* type. In general, a type constructor τ of kind κ is represented by an element of $Rep_\kappa\ \tau$ with

$$\begin{array}{lcl} Rep_\star\ \tau & = & Rep\ \tau \\ Rep_{\kappa_1 \rightarrow \kappa_2}\ \tau & = & \forall \alpha \,.\, Rep_{\kappa_1}\ \alpha \rightarrow Rep_{\kappa_2}\ (\tau\ \alpha). \end{array}$$

As the latest version of the Glasgow Haskell compiler (GHC 5.03) supports rank-n types, we can, in fact, represent types of arbitrary kinds.

So far so good. One small problem remains though. To define the representation of $Tree\ \phi\ \alpha$, we have to construct type terms from the representations of ϕ and α. Since the representation of ϕ of type $Rep_{\star \rightarrow \star}\ \phi$ is a *function* this sounds like a hard nut to crack. Fortunately, Haskell's type language is a multi-sorted term algebra as Haskell does not offer general type abstraction. Each type term is of the form $App\ s\ [t_1, \ldots, t_n]$; type application is purely syntactic: applying $App\ s\ [t_1, \ldots, t_n]$ to t yields $App\ s\ [t_1, \ldots, t_n, t]$. Consequently, from the result of an application we can reconstruct both the original function and its argument.

$$\begin{array}{lcl} term_{\star \rightarrow \star} & :: & \forall \phi \,.\, (\forall \alpha \,.\, Rep\ \alpha \rightarrow Rep\ (\phi\ \alpha)) \rightarrow Term \\ term_{\star \rightarrow \star}\ r_\phi & = & \mathbf{case}\ term_\star\ (r_\phi\ r_1)\ \mathbf{of}\ App\ t\ ts \rightarrow App\ t\ (init\ ts) \end{array}$$

Given these prerequisites we define

$$
\begin{array}{lcl}
r_{Tree}\ r_\phi\ r_\alpha & = & R_{Type}\ (App\ \texttt{"Tree"}\ [term_{\star\to\star}\ r_\phi, term_\star\ r_\alpha]) \\
 & & \qquad (r_\times\ r_\alpha\ (r_\phi\ (r_{Tree}\ r_\phi\ r_\alpha)))\ (EP\ from_{Tree}\ to_{Tree}) \\
from_{Tree} & :: & \forall \phi\ \alpha\,.\ Tree\ \phi\ \alpha \to \alpha \times (\phi\ (Tree\ \phi\ \alpha)) \\
from_{Tree}\ (Node\ a\ ts) & = & (a :\times: ts) \\
to_{Tree} & :: & \forall \phi\ \alpha\,.\ \alpha \times (\phi\ (Tree\ \phi\ \alpha)) \to Tree\ \phi\ \alpha \\
to_{Tree}\ (a :\times: ts) & = & Node\ a\ ts
\end{array}
$$

4.2 Constructor names

To be able to write polytypic functions that show or read elements of some data type, we add one more constructor to the *Rep* data type.

$$
\begin{array}{lcl}
\mathbf{data}\ Rep\ \tau & = & \ldots \\
 & | & R_{Con}\ String\ (Rep\ \tau)
\end{array}
$$

The R_{Con} constructor is intended to record the string representation of a data constructor. Of course, in practice one would replace *String* by a more elaborate data type that contains further information such as fixity, see (Hinze & Peyton Jones, 2001). The updated definition of r_{Bool} illustrates the use of R_{Con}.

$$
\begin{array}{lcl}
r_{Bool} & :: & Rep\ Bool \\
r_{Bool} & = & R_{Type}\ (App\ \texttt{"Bool"}\ [\,]) \\
 & & \qquad (r_+\ (R_{Con}\ \texttt{"False"}\ r_1)\ (R_{Con}\ \texttt{"True"}\ r_1)) \\
 & & \qquad (EP\ from_{Bool}\ to_{Bool})
\end{array}
$$

Here is a simplified version of Haskell's *show* function, which converts an element of any data type to its string representation. To understand the definition keep in mind that the sum type is used to represent the cases of a **data** declaration while the product type is used to represent the arguments of a single constructor. The unit type 1 signals that a constructor has no arguments.

$$
\begin{array}{lcl}
rShows & :: & \forall \tau\,.\ Rep\ \tau \to \tau \to ShowS \\
rShows\ (R_{Int}\ ep)\ t & = & shows\ (from\ ep\ t) \\
rShows\ (R_{Dynamic}\ ep)\ t & = & \mathbf{case}\ from\ ep\ t\ \mathbf{of} \\
 & & \quad Dyn\ r_\alpha\ x \to showChar\ \texttt{'('} \cdot showString\ \texttt{"dynamic "} \\
 & & \qquad\qquad \cdot\ rShows\ r_\alpha\ x \cdot showChar\ \texttt{')'} \\
 & & \\
rShows\ (R_1\ ep)\ t & = & showString\ \texttt{""} \\
rShows\ (R_+\ r_\alpha\ r_\beta\ ep)\ t & = & \mathbf{case}\ from\ ep\ t\ \mathbf{of} \\
 & & \quad Inl\ a \to rShows\ r_\alpha\ a \\
 & & \quad Inr\ b \to rShows\ r_\beta\ b \\
rShows\ (R_\times\ r_\alpha\ r_\beta\ ep)\ t & = & \mathbf{case}\ from\ ep\ t\ \mathbf{of} \\
 & & \quad (a :\times: b) \to rShows\ r_\alpha\ a \cdot showString\ \texttt{" "} \cdot rShows\ r_\beta\ b \\
rShows\ (R_{Type}\ e\ r_\alpha\ ep)\ t & = & rShows\ r_\alpha\ (from\ ep\ t) \qquad \text{-- nullary constructor} \\
rShows\ (R_{Con}\ s\ (R_1\ ep))\ t & = & showString\ s \\
rShows\ (R_{Con}\ s\ r_\alpha)\ t & = & showChar\ \texttt{'('} \cdot showString\ s \cdot showChar\ \texttt{' '} \\
 & & \cdot\ rShows\ r_\alpha\ t \cdot showChar\ \texttt{')'}
\end{array}
$$

Since type representations are ordinary values, we can separate special cases simply by pattern matching. The second but last equation of *rShows*, for instance, handles nullary constructors while the last equation takes care of the remaining cases.

5 Related work

The polymorphic Horn clause language of Hanus (1991) generalizes the untyped Horn clause resolution semantics of Prolog to typed and polymorphically typed terms. Our initial definition of *Rep* in Section 2.1 is legal in this language, which appears to be strictly more powerful than Haskell. Interestingly, the semantics requires the presence of types at run time. Optimizations are possible for so-called *type preserving* functions where the type variables of the argument types also occur in the result type (note that the *Rep* constructors are type-preserving).

Jansson and Jeuring (1997) developed PolyP, a variant of Haskell that includes a polytypic function construct permitting definitions by primitive recursion on the structure of regular data types, but did not support higher-order kinded type arguments. Hinze (2002) proposed an approach based on indexing values by types and types by kinds. This made it possible to write definitions of functions like *map* that work for arbitrary polymorphic data structures. This approach has been implemented in Generic Haskell (Clarke *et al.*, 2001), a successor to PolyP. Hinze and Peyton Jones (2001) introduced derivable type classes, which can define type-indexed values within classes but are limited to kind $\star$. Clean's generics system (Alimarine & Plasmeijer, 2001) generalizes derivable type classes to allow generic type classes defined at arbitrary kinds rather than just $\star$.

Abadi et al. (1991) first considered rigorously the problem of adding a *Dynamic* type and type pattern matching *typecase* to a monomorphic ML-like language. Leroy and Mauny (1993) studied the interaction of *Dynamic* with implicit polymorphism and implemented a restricted form of polymorphic type pattern matching with both $\forall$ and $\exists$ quantifiers. Abadi et al. (1995) considered dynamics with explicit and implicit polymorphism, and showed how to generalize *typecase* to arbitrary polymorphic patterns. We believe our *Dynamic* also can supports making values of closed polymorphic types dynamic, although we have yet to experiment with unifying and pattern-matching polymorphic type representations.

GHC's *Dynamic* library contains *TypeRep* and *Dynamic* types and a *Typeable* class that are weaker, 'untyped' versions of our *Dynamic*, *Rep*, and *Representable*. Type representations are abstract, and it is impossible to, for example, unpack component *TypeRep*s from a product *TypeRep*. On the other hand, our typed versions constitute a safe implementation of these constructs.

Clean also includes support in development for a richer type *Dynamic* (Pil, 1999) that includes *typecase* with pattern matching on (possibly polymorphic) types, in the style of Leroy and Mauny, and also supports type-dependent functions. Clean's dynamics employ a type class $TC\ \alpha$ that says that it is possible to make α dynamic, so, unlike earlier approaches, values with partially abstract types containing free type variables can be cast to *Dynamic* as long as all the type variables are of class

TC. Our *Dynamic* supports exactly this behavior, whence Clean's TC is analogous to our *Representable* class.

Shields, Sheard, and Peyton Jones (1998) present an alternative implementation of dynamics based on staged type inference. In staged computation (Davies & Pfenning, 2001; Taha & Sheard, 1997), compilation of parts of a program may be delayed, so functions may be specialized to arguments available at compile time. Staged type inference delays type inference and type checking until run time as well. This makes it possible to avoid many of the difficulties of explicit polymorphic type pattern matching encountered in previous approaches, since unification occurs at run time when concrete type information is available. Our approach also employs run-time unification, if only for monomorphic types, but our *unify* is a user-level program rather than compiler-generated code. It would be interesting to see whether our approach generalized to polymorphic unifcation. Staged computation may also be useful in optimizing representation-passing by specializing generic functions to particular representations. We have also experimented with using GHC's rewrite rules (Peyton Jones *et al.*, 2001) to automatically rewrite representation-based functions when type information is known at compile time.

Weirich (2000) showed how to implement type-safe cast and a form of *Dynamic* in Haskell using type classes. The two Haskell implementations of *cast* employ mutually recursive type classes $\mathit{CastTo}\ \alpha$ and $\mathit{CastFrom}\ \beta$; the first implementation of these classes interprets cast as coercion $(\alpha \to \beta)$, whereas the second interprets cast as unrestricted substitution $(\phi\ \alpha \to \phi\ \beta)$. These interpretations of cast correspond exactly to our interpretations of type equivalence.

Implicit, compiler-generated type information has been studied in many languages and proposals for implementing statically-typed dynamics and implementing intensional polymorphism. Explicit type representations are not new either: they were introduced by Crary, Weirich, and Morrisett (1999); the authors also observed that representations could be used to implement an explicit *Dynamic* type. Crary and Weirich (1999) and Weirich (2001) have also considered encodings of type representations in the more powerful type systems LX, which includes function, sum, product, and recursive kinds, and LU^{-}, which includes impredicative kind polymorphism.

Baars and Swierstra (2002) have independently discovered the type representation encoding presented in Section 2. However, instead of starting with representation-passing generic functions and attempting to implement representations, they start with dynamic types and postulate a type family $\mathit{TypeRep}\ \alpha$ that contains enough information for dynamic typechecking, and then derive an implementation for it. Baars and Swierstra address dynamic typing issues beyond those considered here, such as dynamic typing and compilation of expressions. In contrast, we have considered both generics and dynamics, and interrelated them. We have also shown how to represent a more general class of types, including polymorphic and recursive types.

6 Conclusion

Previous approaches to implementing generic programming and dynamic typing in high-level statically typed languages have involved substantial language modifications and substantial proofs of type safety for the modified language. Dynamics and generics have been studied separately, leaving unresolved the question of whether dynamic values can be used with generic functions and vice versa. We have shown that generic programming and dynamic typing features can be derived simultaneously and compatibly from type representations. Moreover, type representations can be encoded in Haskell using existentials and equivalence types, so they can be implemented and given a semantics by translation. As far as we know this is the first approach to make this connection between statically typed generics and dynamics explicit.

Unlike prior approaches to implementing type representations, we encode representations as ordinary data types and *typecase*s as ordinary cases. As a result, all of the existing pattern matching constructs and optimizations carry over to representation patterns with no effort. Mutually recursive functions and multiple type arguments also pose no problems. Representation-passing functions can be compiled separately from their calling contexts; in contrast, other approaches to statically typed generic programming cannot compile generic function uses and definitions separately. Dynamic types and run-time type checking can be defined in terms of representations, but since type representations are explicitly typed program data, this implementation of *Dynamic* is more flexible than implementations in which type information is compiler-generated and abstract.

There are several directions for improving our approach. For example, it is not possible to override the behaviour of a polytypic function at some specific instance (without changing the definition of *Rep*). In contrast, overriding is easy using type classes. Extensible data types might alleviate this problem. Our encoding also incurs unnecessary run-time type dispatch overhead when types are available at compile time. Furthermore, our encoding translates arbitrary tuples and data types to a universal data type consisting of binary products, sums, and constructors, which may incur additional overhead. These drawbacks could be addressed through specialization, run-time code generation, or deforestation optimizations. Finally, while it is possible to define polytypic functions such as *map* that analyze type constructors of kind other than $\star$ (using a different *Rep* type), constructing functions that work for all types of all kinds seems out of reach within the Hindley-Milner type system. These are all important directions for future work.

References

Abadi, M., Cardelli, L., Pierce, B. and Rémy, D. (1995) Dynamic typing in polymorphic languages. *Journal of Functional Programming* **5**(1):111–130.

Abadi, M., Cardelli, L., Pierce, B. and Plotkin, G. (1991) Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems* **13**(2):237–268.

Alimarine, A. and Plasmeijer, R. (2001) A generic programming extension for Clean.

Arts, T. and Mohnen, M. (eds), *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01* pp. 257–278.

Baars, A. I. and Swierstra, S. D. (2002) Typing dynamic typing. Peyton Jones, S. (ed), *Proceedings of the 2002 International Conference on Functional Programming, Pittsburgh, PA, USA, October 4-6, 2002*. ACM Press. To appear.

Backhouse, R., Jansson, P., Jeuring, J. and Meertens, L. (1999) Generic Programming — An Introduction —. Swierstra, S. D., Henriques, P. R. and Oliveira, J. N. (eds), *3rd International Summer School on Advanced Functional Programming, Braga, Portugal*. Lecture Notes in Computer Science 1608, pp. 28–115. Springer-Verlag.

Bird, R. and Meertens, L. (1998) Nested datatypes. Jeuring, J. (ed), *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*. Lecture Notes in Computer Science 1422, pp. 52–67. Springer-Verlag.

Clarke, D., Hinze, R., Jeuring, J., Löh, A. and de Wit, J. (2001) *The Generic Haskell user's guide*. Tech. rept. UU-CS-2001-26. Universiteit Utrecht.

Crary, K. and Weirich, S. (1999) Flexible type analysis. *ACM SIGPLAN Notices* **34**(9):233–248. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

Crary, K., Weirich, S. and Morrisett, G. (1999) Intensional polymorphism in type-erasure semantics. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, MD*. ACM SIGPLAN Notices (**34**)1, pp. 301–312. ACM Press.

Davies, R. and Pfenning, F. (2001) A modal analysis of staged computation. *Journal of the ACM* **48**(3):555–604.

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

Hanus, M. (1991) Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science* **89**(1):63–106.

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

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

Hinze, R. (2000a) Memo functions, polytypically! Jeuring, J. (ed), *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal* pp. 17–32. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

Hinze, R. (2000b) A new approach to generic functional programming. Reps, T. W. (ed), *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 00), Boston, Massachusetts, January 19-21* pp. 119–132.

Hinze, R. (2002) Polytypic values possess polykinded types. *Science of Computer Programmming*. To appear.

Hinze, R. and Peyton Jones, S. (2001) Derivable type classes. Hutton, G. (ed), *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, vol. 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science. The preliminary proceedings appeared as a University of Nottingham technical report.

Jansson, P. and Jeuring, J. (1997) PolyP—a polytypic programming language extension. *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France* pp. 470–482. ACM Press.

Leroy, X. (1994) Manifest types, modules, and separate compilation. *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94), Portland, Oregon* pp. 109–122. ACM, New York, NY.

Leroy, X. and Mauny, M. (1993) Dynamics in ML. *Journal of Functional Programming* **3**(4):431–463.

Minamide, Y., Morrisett, G. and Harper, R. (1996) Typed closure conversion. *Conference Record of the* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)* pp. 271–283.

Peyton Jones, S. and Hughes, J. (eds). (1999) *Haskell 98 — A Non-strict, Purely Functional Language.* Available from `http://www.haskell.org/definition/`.

Peyton Jones, S., Tolmach, A. and Hoare, T. (2001) Playing by the rules: Rewriting as an optimization technique in GHC. Hinze, R. (ed), *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop (HW'2001), 2nd September 2001, Firenze, Italy.* Electronic Notes in Theoretical Computer Science, Vol. 59, pp. 203–233. The preliminary proceedings appeared as a Universiteit Utrecht technical report, UU-CS-2001-62.

Pierce, B. C. (2002) *Types and programming languages.* MIT Press.

Pil, M. (1999) Dynamic types and type dependent functions. Hammond, K., Davie, A. J. T. and Clack, C. (eds), *Implementation of Functional Languages, 10th International Workshop, IFL'98, London, UK, September 9-11, Selected Papers.* Lecture Notes in Computer Science 1595, pp. 169–185. Springer.

Shields, M., Sheard, T. and Jones, S. P. (1998) Dynamic typing as staged type inference. *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)* pp. 289–302. Association for Computing Machinery.

Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97).* ACM SIGPLAN Notices (**32**)12, pp. 203–217. ACM Press.

Wadler, P. (1989) Theorems for free! *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK* pp. 347–359. Addison-Wesley Publishing Company.

Wadler, P. (2001) The Girard-Reynolds isomorphism. Kobayashi, N. and Pierce, B. C. (eds), *Proc. of 4th Int. Symp. on Theoretical Aspects of Computer Science, TACS 2001, Sendai, Japan, 29–31 Oct. 2001.* Lecture Notes in Computer Science 2215, pp. 468–491. Springer-Verlag.

Wadler, P. and Blott, W. (1989) How to make *ad-hoc* polymorphism less *ad hoc*. *Proceedings of the 16th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)* pp. 60–76. ACM Press.

Weirich, S. (2000) Type-safe cast: functional pearl. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '00).* ACM SIGPLAN Notices (**35**)9, pp. 58–67. ACM Press.

Weirich, S. (2001) Encoding intensional type analysis. Sands, D. (ed), *Proceedings of the 10th European Symposium on Programming, ESOP 2001.* Lecture Notes in Computer Science 2028, pp. 92–106.

A Listing

A.1 Generic representation types

$$
\begin{array}{lcl}
\textbf{data}\ 1 & = & \mathit{Unit} \\
\textbf{data}\ \alpha + \beta & = & \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta \\
\textbf{data}\ \alpha \times \beta & = & \alpha :\!\times\!: \beta
\end{array}
$$

Standard mapping functions on the above types (and on the function type).

$$
\begin{array}{lcl}
(+) & :: & \forall \alpha\ \beta\,.\,(\alpha \to \beta) \to (\forall \gamma\ \delta\,.\,(\gamma \to \delta) \to ((\alpha + \gamma) \to (\beta + \delta))) \\
(f + g)\ (\mathit{Inl}\ a) & = & \mathit{Inl}\ (f\ a) \\
(f + g)\ (\mathit{Inr}\ b) & = & \mathit{Inr}\ (g\ b) \\
(\times) & :: & \forall \alpha\ \beta\,.\,(\alpha \to \beta) \to (\forall \gamma\ \delta\,.\,(\gamma \to \delta) \to ((\alpha \times \gamma) \to (\beta \times \delta))) \\
(f \times g)\ (a :\!\times\!: b) & = & f\ a :\!\times\!: g\ b \\
(\to) & :: & \forall \alpha\ \beta\,.\,(\alpha \to \beta) \to (\forall \gamma\ \delta\,.\,(\gamma \to \delta) \to ((\beta \to \gamma) \to (\alpha \to \delta))) \\
(f \to g)\ h & = & g \cdot h \cdot f
\end{array}
$$

A.2 Type equivalence

$$
\textbf{data}\ \alpha \leftrightarrow \beta \quad = \quad \mathit{EP}\{\mathit{from} :: \alpha \to \beta, \mathit{to} :: \beta \to \alpha\}
$$

Reflexivity, symmetry and transitivity.

$$
\begin{array}{lcl}
\mathit{self} & :: & \forall \alpha\,.\,\alpha \leftrightarrow \alpha \\
\mathit{self} & = & \mathit{EP}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\} \\
\mathit{inv} & :: & \forall \alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \to (\beta \leftrightarrow \alpha) \\
\mathit{inv}\ f & = & \mathit{EP}\{\mathit{from} = \mathit{to}\ f, \mathit{to} = \mathit{from}\ f\} \\
\multicolumn{3}{l}{\textbf{infixr}\ 9\ \diamond} \\
(\diamond) & :: & \forall \alpha\ \beta\ \gamma\,.\,(\beta \leftrightarrow \gamma) \to (\alpha \leftrightarrow \beta) \to (\alpha \leftrightarrow \gamma) \\
f \diamond g & = & \mathit{EP}\{\mathit{from} = \mathit{from}\ f \cdot \mathit{from}\ g, \mathit{to} = \mathit{to}\ g \cdot \mathit{to}\ f\}
\end{array}
$$

Mapping functions for generic representation types (and for the function type) implementing the laws of congruence.

$$
\begin{array}{lcl}
(\oplus) & :: & \forall \alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \to (\forall \gamma\ \delta\,.\,(\gamma \leftrightarrow \delta) \to ((\alpha + \gamma) \leftrightarrow (\beta + \delta))) \\
f \oplus g & = & \mathit{EP}\{\mathit{from} = \mathit{from}\ f + \mathit{from}\ g, \mathit{to} = \mathit{to}\ f + \mathit{to}\ g\} \\
(\otimes) & :: & \forall \alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \to (\forall \gamma\ \delta\,.\,(\gamma \leftrightarrow \delta) \to ((\alpha \times \gamma) \leftrightarrow (\beta \times \delta))) \\
f \otimes g & = & \mathit{EP}\{\mathit{from} = \mathit{from}\ f \times \mathit{from}\ g, \mathit{to} = \mathit{to}\ f \times \mathit{to}\ g\} \\
(\ominus) & :: & \forall \alpha\ \beta\,.\,(\alpha \leftrightarrow \beta) \to (\forall \gamma\ \delta\,.\,(\gamma \leftrightarrow \delta) \to ((\alpha \to \gamma) \leftrightarrow (\beta \to \delta))) \\
f \ominus g & = & \mathit{EP}\{\mathit{from} = \mathit{to}\ f \to \mathit{from}\ g, \mathit{to} = \mathit{from}\ f \to \mathit{to}\ g\}
\end{array}
$$

A.3 Type representations

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{Rep}\ \tau & = & R_{\mathit{Int}} & (\tau \leftrightarrow \mathit{Int}) \\
 & \mid & R_{\mathit{Char}} & (\tau \leftrightarrow \mathit{Char}) \\
 & \mid & R_{\mathit{Dynamic}} & (\tau \leftrightarrow \mathit{Dynamic}) \\
 & \mid & \forall \alpha\, \beta\, .\, R_{\rightarrow}\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta) & (\tau \leftrightarrow (\alpha \rightarrow \beta)) \\
 & \mid & R_{1} & (\tau \leftrightarrow 1) \\
 & \mid & \forall \alpha\, \beta\, .\, R_{+}\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta) & (\tau \leftrightarrow (\alpha + \beta)) \\
 & \mid & \forall \alpha\, \beta\, .\, R_{\times}\ (\mathit{Rep}\ \alpha)\ (\mathit{Rep}\ \beta) & (\tau \leftrightarrow (\alpha \times \beta)) \\
 & \mid & \forall \alpha\ \ .\, R_{\mathit{Type}}\ \mathit{Term}\ (\mathit{Rep}\ \alpha) & (\tau \leftrightarrow \alpha) \\
 & \mid & R_{\mathit{Con}}\ \mathit{String}\ (\mathit{Rep}\ \tau) &
\end{array}
$$

Smart constructors.

$$
\begin{array}{lcl}
r_{\mathit{Int}} & :: & \mathit{Rep}\ \mathit{Int} \\
r_{\mathit{Int}} & = & R_{\mathit{Int}}\ \mathit{self} \\
r_{\mathit{Char}} & :: & \mathit{Rep}\ \mathit{Char} \\
r_{\mathit{Char}} & = & R_{\mathit{Char}}\ \mathit{self} \\
r_{\mathit{Dynamic}} & :: & \mathit{Rep}\ \mathit{Dynamic} \\
r_{\mathit{Dynamic}} & = & R_{\mathit{Dynamic}}\ \mathit{self} \\
r_{\rightarrow} & :: & \forall \alpha\, .\, \mathit{Rep}\ \alpha \rightarrow (\forall \beta\, .\, \mathit{Rep}\ \beta \rightarrow \mathit{Rep}\ (\alpha \rightarrow \beta)) \\
r_{\rightarrow}\ r_{\alpha}\ r_{\beta} & = & R_{\rightarrow}\ r_{\alpha}\ r_{\beta}\ \mathit{self} \\
r_{1} & :: & \mathit{Rep}\ 1 \\
r_{1} & = & R_{1}\ \mathit{self} \\
r_{+} & :: & \forall \alpha\, .\, \mathit{Rep}\ \alpha \rightarrow (\forall \beta\, .\, \mathit{Rep}\ \beta \rightarrow \mathit{Rep}\ (\alpha + \beta)) \\
r_{+}\ r_{\alpha}\ r_{\beta} & = & R_{+}\ r_{\alpha}\ r_{\beta}\ \mathit{self} \\
r_{\times} & :: & \forall \alpha\, .\, \mathit{Rep}\ \alpha \rightarrow (\forall \beta\, .\, \mathit{Rep}\ \beta \rightarrow \mathit{Rep}\ (\alpha \times \beta)) \\
r_{\times}\ r_{\alpha}\ r_{\beta} & = & R_{\times}\ r_{\alpha}\ r_{\beta}\ \mathit{self}
\end{array}
$$

A class for representable types.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Representable}\ \tau\ \textbf{where} \\
\quad \mathit{rep}\ \ ::\ \ \mathit{Rep}\ \tau \\
\textbf{instance}\ \mathit{Representable}\ \mathit{Int}\ \textbf{where} \\
\quad \mathit{rep}\ \ =\ \ r_{\mathit{Int}} \\
\textbf{instance}\ \mathit{Representable}\ \mathit{Char}\ \textbf{where} \\
\quad \mathit{rep}\ \ =\ \ r_{\mathit{Char}} \\
\textbf{instance}\ \mathit{Representable}\ \mathit{Dynamic}\ \textbf{where} \\
\quad \mathit{rep}\ \ =\ \ r_{\mathit{Dynamic}} \\
\textbf{instance}\ (\mathit{Representable}\ a, \mathit{Representable}\ b) \Rightarrow \mathit{Representable}\ (a \rightarrow b)\ \textbf{where} \\
\quad \mathit{rep}\ \ =\ \ r_{\rightarrow}\ \mathit{rep}\ \mathit{rep} \\
\textbf{instance}\ (\mathit{Representable}\ \alpha, \mathit{Representable}\ \beta) \Rightarrow \mathit{Representable}\ (\alpha + \beta)\ \textbf{where} \\
\quad \mathit{rep}\ \ =\ \ r_{+}\ \mathit{rep}\ \mathit{rep} \\
\textbf{instance}\ (\mathit{Representable}\ \alpha, \mathit{Representable}\ \beta) \Rightarrow \mathit{Representable}\ (\alpha \times \beta)\ \textbf{where} \\
\quad \mathit{rep}\ \ =\ \ r_{\times}\ \mathit{rep}\ \mathit{rep}
\end{array}
$$

A.4 Dynamics

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Dynamic} & = & \forall\alpha\,.\,\mathit{Dyn}\ (\mathit{Rep}\ \alpha)\ \alpha \\
\mathit{dynamic} & :: & \forall\alpha\,.\,(\mathit{Representable}\ \alpha) \Rightarrow \alpha \to \mathit{Dynamic} \\
\mathit{dynamic}\ x & = & \mathit{Dyn}\ \mathit{rep}\ x
\end{array}
$$

Run-time unification of types.

$$
\begin{array}{lcl}
\mathit{unify} & :: & \forall\tau_1\ \tau_2\,.\,\mathit{Rep}\ \tau_1 \to \mathit{Rep}\ \tau_2 \to [\tau_1 \leftrightarrow \tau_2] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\mathit{Int}}\ \mathit{ep}_1)\ (R_{\mathit{Int}}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond \mathit{ep}_1] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\mathit{Char}}\ \mathit{ep}_1)\ (R_{\mathit{Char}}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond \mathit{ep}_1] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\mathit{Dynamic}}\ \mathit{ep}_1)\ (R_{\mathit{Dynamic}}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond \mathit{ep}_1] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\to}\ r_{\alpha_1}\ r_{\beta_1}\ \mathit{ep}_1)\ (R_{\to}\ r_{\alpha_2}\ r_{\beta_2}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond (\mathit{ep}_\alpha \ominus \mathit{ep}_\beta) \diamond \mathit{ep}_1 \mid \mathit{ep}_\alpha \leftarrow \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}, \mathit{ep}_\beta \leftarrow \mathit{unify}\ r_{\beta_1}\ r_{\beta_2}] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_1\ \mathit{ep}_1)\ (R_1\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond \mathit{ep}_1] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{+}\ r_{\alpha_1}\ r_{\beta_1}\ \mathit{ep}_1)\ (R_{+}\ r_{\alpha_2}\ r_{\beta_2}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond (\mathit{ep}_\alpha \oplus \mathit{ep}_\beta) \diamond \mathit{ep}_1 \mid \mathit{ep}_\alpha \leftarrow \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}, \mathit{ep}_\beta \leftarrow \mathit{unify}\ r_{\beta_1}\ r_{\beta_2}] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\times}\ r_{\alpha_1}\ r_{\beta_1}\ \mathit{ep}_1)\ (R_{\times}\ r_{\alpha_2}\ r_{\beta_2}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond (\mathit{ep}_\alpha \otimes \mathit{ep}_\beta) \diamond \mathit{ep}_1 \mid \mathit{ep}_\alpha \leftarrow \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}, \mathit{ep}_\beta \leftarrow \mathit{unify}\ r_{\beta_1}\ r_{\beta_2}] \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\mathit{Con}}\ s_1\ r_{\alpha_1})\ (R_{\mathit{Con}}\ s_2\ r_{\alpha_2})} \\
 & = & \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2} \\
\multicolumn{3}{l}{\mathit{unify}\ (R_{\mathit{Type}}\ t_1\ r_{\alpha_1}\ \mathit{ep}_1)\ (R_{\mathit{Type}}\ t_2\ r_{\alpha_2}\ \mathit{ep}_2)} \\
 & = & [\mathit{inv}\ \mathit{ep}_2 \diamond \mathit{head}\ (\mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}) \diamond \mathit{ep}_1 \mid t_1 == t_2] \\
\mathit{unify}\ \_\ \_ & = & [\,]
\end{array}
$$

Type-safe cast and dynamic function application.

$$
\begin{array}{lcl}
\mathit{rCast} & :: & \forall\tau\,.\,\mathit{Rep}\ \tau \to \mathit{Dynamic} \to \tau \\
\mathit{rCast}\ r_\tau\ (\mathit{Dyn}\ r_\alpha\ a) & = & \mathbf{case}\ \mathit{unify}\ r_\tau\ r_\alpha\ \mathbf{of} \\
 & & \quad [\mathit{ep}] \to \mathit{to}\ \mathit{ep}\ a \\
 & & \quad \_ \to \mathit{error}\ \texttt{"cast: type mismatch"} \\
\mathit{cast} & :: & (\mathit{Representable}\ \tau) \Rightarrow \mathit{Dynamic} \to \tau \\
\mathit{cast}\ d & = & \mathit{rCast}\ \mathit{rep}\ d \\
\mathit{apply} & :: & \mathit{Dynamic} \to \mathit{Dynamic} \to \mathit{Dynamic} \\
\multicolumn{3}{l}{\mathit{apply}\ (\mathit{Dyn}\ (R_{\to}\ r_\alpha\ r_\beta\ \mathit{ep})\ f)\ (\mathit{Dyn}\ r_{\alpha'}\ x)} \\
 & = & \mathbf{case}\ \mathit{unify}\ r_\alpha\ r_{\alpha'}\ \mathbf{of} \\
 & & \quad [\mathit{ep}'] \to \mathit{Dyn}\ r_\beta\ ((\mathit{from}\ \mathit{ep}\ f)\ (\mathit{to}\ \mathit{ep}'\ x)) \\
 & & \quad \_ \to \mathit{error}\ \texttt{"apply: type mismatch"} \\
\mathit{apply}\ \_\ \_ & = & \mathit{error}\ \texttt{"apply: not a function"}
\end{array}
$$

A.5 Type terms

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Term} & = & \mathit{App}\ \mathit{String}\ [\mathit{Term}] \\
 & & \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \\
\mathit{term}_{\star} & :: & \forall \tau\,.\,\mathit{Rep}\ \tau \to \mathit{Term} \\
\mathit{term}_{\star}\ (R_{\mathit{Int}}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"Int"}\ [\,] \\
\mathit{term}_{\star}\ (R_{\mathit{Char}}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"Char"}\ [\,] \\
\mathit{term}_{\star}\ (R_{\mathit{Dynamic}}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"Dynamic"}\ [\,] \\
\mathit{term}_{\star}\ (R_{\to}\ r_{\alpha}\ r_{\beta}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"->"}\ [\mathit{term}_{\star}\ r_{\alpha}, \mathit{term}_{\star}\ r_{\beta}] \\
\mathit{term}_{\star}\ (R_{1}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"1"}\ [\,] \\
\mathit{term}_{\star}\ (R_{+}\ r_{\alpha}\ r_{\beta}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"(+)"}\ [\mathit{term}_{\star}\ r_{\alpha}, \mathit{term}_{\star}\ r_{\beta}] \\
\mathit{term}_{\star}\ (R_{\times}\ r_{\alpha}\ r_{\beta}\ \mathit{ep}) & = & \mathit{App}\ \texttt{"(*)"}\ [\mathit{term}_{\star}\ r_{\alpha}, \mathit{term}_{\star}\ r_{\beta}] \\
\mathit{term}_{\star}\ (R_{\mathit{Type}}\ t\ r_{\alpha}\ \mathit{ep}) & = & t \\
\mathit{term}_{\star\to\star} & :: & \forall \phi\,.\,(\forall \alpha\,.\,\mathit{Rep}\ \alpha \to \mathit{Rep}\ (\phi\ \alpha)) \to \mathit{Term} \\
\mathit{term}_{\star\to\star}\ r_{\phi} & = & \textbf{case}\ \mathit{term}_{\star}\ (r_{\phi}\ r_{1})\ \textbf{of}\ \mathit{App}\ t\ \mathit{ts} \to \mathit{App}\ t\ (\mathit{init}\ \mathit{ts})
\end{array}
$$

A.6 Generics

Examples of type representations.

$$
\begin{array}{lcl}
r_{\mathit{Bool}} & :: & \mathit{Rep}\ \mathit{Bool} \\
r_{\mathit{Bool}} & = & R_{\mathit{Type}}\ (\mathit{App}\ \texttt{"Bool"}\ [\,]) \\
 & & \qquad (r_{+}\ (R_{\mathit{Con}}\ \texttt{"False"}\ r_{1})\ (R_{\mathit{Con}}\ \texttt{"True"}\ r_{1})) \\
 & & \qquad (\mathit{EP}\ \mathit{from}_{\mathit{Bool}}\ \mathit{to}_{\mathit{Bool}}) \\
\mathit{from}_{\mathit{Bool}} & :: & \mathit{Bool} \to 1 + 1 \\
\mathit{from}_{\mathit{Bool}}\ \mathit{False} & = & \mathit{Inl}\ \mathit{Unit} \\
\mathit{from}_{\mathit{Bool}}\ \mathit{True} & = & \mathit{Inr}\ \mathit{Unit} \\
\mathit{to}_{\mathit{Bool}} & :: & 1 + 1 \to \mathit{Bool} \\
\mathit{to}_{\mathit{Bool}}\ (\mathit{Inl}\ \mathit{Unit}) & = & \mathit{False} \\
\mathit{to}_{\mathit{Bool}}\ (\mathit{Inr}\ \mathit{Unit}) & = & \mathit{True}
\end{array}
$$

$$
\begin{array}{lcl}
r_{[\,]} & :: & \forall \alpha\,.\,\mathit{Rep}\ \alpha \to \mathit{Rep}\ [\alpha] \\
r_{[\,]}\ r_{\alpha} & = & R_{\mathit{Type}}\ (\mathit{App}\ \texttt{"[]"}\ [\mathit{term}_{\star}\ r_{\alpha}]) \\
 & & \qquad (r_{+}\ (R_{\mathit{Con}}\ \texttt{"[]"}\ r_{1})\ (R_{\mathit{Con}}\ \texttt{"(:)"}\ (r_{\times}\ r_{\alpha}\ (r_{[\,]}\ r_{\alpha})))) \\
 & & \qquad (\mathit{EP}\ \mathit{from}_{[\,]}\ \mathit{to}_{[\,]}) \\
\mathit{from}_{[\,]} & :: & \forall \alpha\,.\,[\alpha] \to 1 + (\alpha \times [\alpha]) \\
\mathit{from}_{[\,]}\ [\,] & = & \mathit{Inl}\ \mathit{Unit} \\
\mathit{from}_{[\,]}\ (a : \mathit{as}) & = & \mathit{Inr}\ (a :\times: \mathit{as}) \\
\mathit{to}_{[\,]} & :: & \forall \alpha\,.\,1 + (\alpha \times [\alpha]) \to [\alpha] \\
\mathit{to}_{[\,]}\ (\mathit{Inl}\ \mathit{Unit}) & = & [\,] \\
\mathit{to}_{[\,]}\ (\mathit{Inr}\ (a :\times: \mathit{as})) & = & a : \mathit{as}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Representable}\ \mathit{Bool}\ \textbf{where} \\
\quad \mathit{rep} \quad = \quad r_{\mathit{Bool}} \\
\textbf{instance}\ (\mathit{Representable}\ \alpha) \Rightarrow \mathit{Representable}\ [\alpha]\ \textbf{where} \\
\quad \mathit{rep} \quad = \quad r_{[\,]}\ \mathit{rep}
\end{array}
$$

Generic functions: generic equality.

$$
\begin{array}{lcl}
\mathit{rEqual} & :: & \forall\tau\,.\,\mathit{Rep}\ \tau \to \tau \to \tau \to \mathit{Bool} \\
\mathit{rEqual}\ (R_{\mathit{Int}}\ \mathit{ep})\ t_1\ t_2 & = & \mathit{from}\ \mathit{ep}\ t_1 == \mathit{from}\ \mathit{ep}\ t_2 \\
\mathit{rEqual}\ (R_{\mathit{Char}}\ \mathit{ep})\ t_1\ t_2 & = & \mathit{from}\ \mathit{ep}\ t_1 == \mathit{from}\ \mathit{ep}\ t_2 \\
\mathit{rEqual}\ (R_{\mathit{Dynamic}}\ \mathit{ep})\ d_1\ d_2 & = & \textbf{case}\ (\mathit{from}\ \mathit{ep}\ d_1, \mathit{from}\ \mathit{ep}\ d_2)\ \textbf{of} \\
 & & \quad (\mathit{Dyn}\ r_{\alpha_1}\ v_1, \mathit{Dyn}\ r_{\alpha_2}\ v_2) \\
 & & \qquad \to \textbf{case}\ \mathit{unify}\ r_{\alpha_1}\ r_{\alpha_2}\ \textbf{of} \\
 & & \qquad\qquad [\,\mathit{ep}'\,] \to \mathit{rEqual}\ r_{\alpha_1}\ v_1\ (\mathit{to}\ \mathit{ep}'\ v_2) \\
 & & \qquad\qquad \_ \to \mathit{False} \\
\mathit{rEqual}\ (R_{\to}\ r_\alpha\ r_\beta\ \mathit{ep})\ t_1\ t_2 & = & \mathit{error}\ \texttt{"rEqual: equality of functions"} \\
\mathit{rEqual}\ (R_1\ \mathit{ep})\ t_1\ t_2 & = & \textbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \textbf{of} \\
 & & \quad (\mathit{Unit}, \mathit{Unit}) \to \mathit{True} \\
\mathit{rEqual}\ (R_{+}\ r_\alpha\ r_\beta\ \mathit{ep})\ t_1\ t_2 & = & \textbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \textbf{of} \\
 & & \quad (\mathit{Inl}\ a_1, \mathit{Inl}\ a_2) \to \mathit{rEqual}\ r_\alpha\ a_1\ a_2 \\
 & & \quad (\mathit{Inl}\ a_1, \mathit{Inr}\ b_2) \to \mathit{False} \\
 & & \quad (\mathit{Inr}\ b_1, \mathit{Inl}\ a_2) \to \mathit{False} \\
 & & \quad (\mathit{Inr}\ b_1, \mathit{Inr}\ b_2) \to \mathit{rEqual}\ r_\beta\ b_1\ b_2 \\
\mathit{rEqual}\ (R_{\times}\ r_\alpha\ r_\beta\ \mathit{ep})\ t_1\ t_2 & = & \textbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \textbf{of} \\
 & & \quad (a_1 :\times: b_1, a_2 :\times: b_2) \to \mathit{rEqual}\ r_\alpha\ a_1\ a_2 \\
 & & \qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{rEqual}\ r_\beta\ b_1\ b_2 \\
\mathit{rEqual}\ (R_{\mathit{Type}}\ e\ r_\alpha\ \mathit{ep})\ t_1\ t_2 & = & \mathit{rEqual}\ r_\alpha\ (\mathit{from}\ \mathit{ep}\ t_1)\ (\mathit{from}\ \mathit{ep}\ t_2) \\
\mathit{rEqual}\ (R_{\mathit{Con}}\ s\ r_\alpha)\ t_1\ t_2 & = & \mathit{rEqual}\ r_\alpha\ t_1\ t_2
\end{array}
$$

Generic minimum.

$$
\begin{array}{lcl}
\mathit{rMinBound} & :: & \forall\tau\,.\,\mathit{Rep}\ \tau \to \tau \\
\mathit{rMinBound}\ (R_{\mathit{Int}}\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{minBound}) \\
\mathit{rMinBound}\ (R_{\mathit{Char}}\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{minBound}) \\
\mathit{rMinBound}\ (R_{\mathit{Dynamic}}\ \mathit{ep}) & = & \mathit{error}\ \texttt{"rMinBound: dynamic"} \\
\mathit{rMinBound}\ (R_{\to}\ r_\alpha\ r_\beta\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\lambda a \to \mathit{rMinBound}\ r_\beta) \\
\mathit{rMinBound}\ (R_1\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{Unit}) \\
\mathit{rMinBound}\ (R_{+}\ r_\alpha\ r_\beta\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{Inl}\ (\mathit{rMinBound}\ r_\alpha)) \\
\mathit{rMinBound}\ (R_{\times}\ r_\alpha\ r_\beta\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{rMinBound}\ r_\alpha :\times: \mathit{rMinBound}\ r_\beta) \\
\mathit{rMinBound}\ (R_{\mathit{Type}}\ t\ r_\alpha\ \mathit{ep}) & = & \mathit{to}\ \mathit{ep}\ (\mathit{rMinBound}\ r_\alpha) \\
\mathit{rMinBound}\ (R_{\mathit{Con}}\ s\ r_\alpha) & = & \mathit{rMinBound}\ r_\alpha
\end{array}
$$

Generic unparsing.

$$
\begin{array}{lcl}
\mathit{rShows} & :: & \forall\tau\,.\,\mathit{Rep}\;\tau \to \tau \to \mathit{ShowS} \\
\mathit{rShows}\;(R_{\mathit{Int}}\;\mathit{ep})\;t & = & \mathit{shows}\;(\mathit{from}\;\mathit{ep}\;t) \\
\mathit{rShows}\;(R_{\mathit{Char}}\;\mathit{ep})\;t & = & \mathit{shows}\;(\mathit{from}\;\mathit{ep}\;t) \\
\mathit{rShows}\;(R_{\mathit{Dynamic}}\;\mathit{ep})\;t & = & \mathbf{case}\;\mathit{from}\;\mathit{ep}\;t\;\mathbf{of} \\
 & & \quad \mathit{Dyn}\;r_\alpha\;x \to \mathit{showChar}\;\texttt{'('} \cdot \mathit{showString}\;\texttt{"dynamic "} \\
 & & \qquad\qquad\qquad\quad \cdot\;\mathit{rShows}\;r_\alpha\;x \cdot \mathit{showChar}\;\texttt{')'} \\
\mathit{rShows}\;(R_{\to}\;r_\alpha\;r_\beta\;\mathit{ep})\;t & = & \mathit{showString}\;\texttt{"<function>"} \\
\mathit{rShows}\;(R_1\;\mathit{ep})\;t & = & \mathit{showString}\;\texttt{""} \\
\mathit{rShows}\;(R_{+}\;r_\alpha\;r_\beta\;\mathit{ep})\;t & = & \mathbf{case}\;\mathit{from}\;\mathit{ep}\;t\;\mathbf{of} \\
 & & \quad \mathit{Inl}\;a \to \mathit{rShows}\;r_\alpha\;a \\
 & & \quad \mathit{Inr}\;b \to \mathit{rShows}\;r_\beta\;b \\
\mathit{rShows}\;(R_{\times}\;r_\alpha\;r_\beta\;\mathit{ep})\;t & = & \mathbf{case}\;\mathit{from}\;\mathit{ep}\;t\;\mathbf{of} \\
 & & \quad (a :\times: b) \to \mathit{rShows}\;r_\alpha\;a \cdot \mathit{showString}\;\texttt{" "} \cdot \mathit{rShows}\;r_\beta\;b \\
\mathit{rShows}\;(R_{\mathit{Type}}\;e\;r_\alpha\;\mathit{ep})\;t & = & \mathit{rShows}\;r_\alpha\;(\mathit{from}\;\mathit{ep}\;t) \\
\mathit{rShows}\;(R_{\mathit{Con}}\;s\;(R_1\;\mathit{ep}))\;t & = & \mathit{showString}\;s \\
\mathit{rShows}\;(R_{\mathit{Con}}\;s\;r_\alpha)\;t & = & \mathit{showChar}\;\texttt{'('} \cdot \mathit{showString}\;s \cdot \mathit{showChar}\;\texttt{' '} \\
 & & \cdot\;\mathit{rShows}\;r_\alpha\;t \cdot \mathit{showChar}\;\texttt{')'}
\end{array}
$$

Generic memoization.

$$
\begin{array}{lcl}
\mathit{rMemo} & :: & \forall\tau\;\nu\,.\,\mathit{Rep}\;\tau \to (\tau \to \nu) \to (\tau \to \nu) \\
\mathit{rMemo}\;(R_{\mathit{Int}}\;\mathit{ep})\;f & = & \lambda t \to f\;t \qquad \text{-- no memoization} \\
\mathit{rMemo}\;(R_{\mathit{Char}}\;\mathit{ep})\;f & = & \lambda t \to f\;t \qquad \text{-- no memoization} \\
\mathit{rMemo}\;(R_{\mathit{Dynamic}}\;\mathit{ep})\;f & = & \lambda t \to f\;t \qquad \text{-- no memoization} \\
\mathit{rMemo}\;(R_{\to}\;r_\alpha\;r_\beta\;\mathit{ep})\;f & = & \lambda t \to f\;t \qquad \text{-- no memoization} \\
\mathit{rMemo}\;(R_1\;\mathit{ep})\;f & = & \lambda t \to \mathbf{case}\;\mathit{from}\;\mathit{ep}\;t\;\mathbf{of} \\
 & & \qquad\qquad \mathit{Unit} \to \mathit{fUnit} \\
\quad \mathbf{where}\;\mathit{fUnit} & = & f\;(\mathit{to}\;\mathit{ep}\;(\mathit{Unit})) \\
\mathit{rMemo}\;(R_{+}\;r_\alpha\;r_\beta\;\mathit{ep})\;f & = & \lambda t \to \mathbf{case}\;\mathit{from}\;\mathit{ep}\;t\;\mathbf{of} \\
 & & \qquad\qquad \mathit{Inl}\;a \to \mathit{fInl}\;a \\
 & & \qquad\qquad \mathit{Inr}\;b \to \mathit{fInr}\;b \\
\quad \mathbf{where}\;\mathit{fInl} & = & \mathit{rMemo}\;r_\alpha\;(\lambda a \to f\;(\mathit{to}\;\mathit{ep}\;(\mathit{Inl}\;a))) \\
\qquad\qquad \mathit{fInr} & = & \mathit{rMemo}\;r_\beta\;(\lambda b \to f\;(\mathit{to}\;\mathit{ep}\;(\mathit{Inr}\;b))) \\
\mathit{rMemo}\;(R_{\times}\;r_\alpha\;r_\beta\;\mathit{ep})\;f & = & \lambda t \to \mathbf{case}\;\mathit{from}\;\mathit{ep}\;t\;\mathbf{of} \\
 & & \qquad\qquad a :\times: b \to \mathit{fPair}\;a\;b \\
\quad \mathbf{where}\;\mathit{fPair} & = & \mathit{rMemo}\;r_\alpha\;(\lambda a \to \mathit{rMemo}\;r_\beta\;(\lambda b \to f\;(\mathit{to}\;\mathit{ep}\;(a :\times: b)))) \\
\mathit{rMemo}\;(R_{\mathit{Type}}\;e\;r_\alpha\;\mathit{ep})\;f & = & \lambda t \to \mathit{rMemo}\;r_\alpha\;(\lambda a \to f\;(\mathit{to}\;\mathit{ep}\;a))\;(\mathit{from}\;\mathit{ep}\;t) \\
\mathit{rMemo}\;(R_{\mathit{Con}}\;s\;r_\alpha)\;f & = & \mathit{rMemo}\;r_\alpha\;f
\end{array}
$$