Dependency-style Generic Haskell

Andres Löh

Dave Clarke

Johan Jeuring

institute of information and computing sciences, utrecht university

technical report UU-CS-2003-022

www.cs.uu.nl

Technical Report

Dependency-style Generic Haskell

Andres Löh Dave Clarke Johan Jeuring

Institute of Information and Computing Sciences
Utrecht University P.O. Box 80.089
3508 TB Utrecht, the Netherlands

Abstract

Generic Haskell is an extension of Haskell that supports the construction of generic programs. During the development of several applications, such as an XML editor and compressor, we encountered a number of limitations with the existing (Classic) Generic Haskell language, as implemented by the current Generic Haskell compiler. Specifically, generic definitions become disproportionately more difficult to write as their complexity increases, such as when one generic function uses another, because recursion is implicit in generic definitions. In the current implementation, writing such functions suffers the burden of a large administrative overhead and is at times counter-intuitive. Furthermore, the absence of type checking in the current implementation can make Generic Haskell hard to use.

In this paper we develop the foundations of *Dependency-style Generic Haskell* which addresses the above problems, shifting the burden from the programmer to the compiler. These foundations consist of a full type system for Dependency-style Generic Haskell's core language and appropriate reduction rules. The type system enables the programmer to write generic functions in a more natural style, taking care of dependency details which were previously the programmer's responsibility.

1 Introduction

Generic programming simplifies a programmer's job. No more need a programmer write functions such as *map* or *fold* or *pretty_print* for her data structures. These can be written once and for all as a generic definition and applied automatically to all of the programmer's datatypes, even as those datatypes evolve.

In Classic Generic Haskell, a Haskell extension based on ideas due to Hinze [13, 16, 18], generic functions can be written which are applicable to Haskell datatypes of all kinds. Generic functions are easy to write. Cases are supplied for sums and products,

for the unit datatype, for primitives, and for constructors. The type of each case follows a particular pattern. To produce a generic instance for some type, Hinze's theory dictates how a compiler assembles cases together following the structure of the type to produce well-typed code.

We have experimented quite a lot with the Classic Generic Haskell compiler and implemented a number of advanced generic programs such as digital searching, the zipper, and XML tools such as a compressor, an editor, and a database. For all Classic Generic Haskell's simplicity, however, some complexity remains to hamper us as soon as we write generic functions for these more involved applications. Rather than write generic definitions in the natural recursive style modern programmers are accustomed to, as in

$$map\langle Prod\ a\ b\rangle\ (a,b) = (map\langle a\rangle\ a, map\langle b\rangle\ b),$$

the theory underlying Classic Generic Haskell dictates that we supply additional parameters for the recursive invocations of generic functions, as in

$$map\langle Prod\rangle\ mapa\ mapb\ (a,b) = (mapa\ a, mapb\ b).$$

The additional arguments *mapa* and *mapb* are used to denote the recursive instances of *map* at the types for the arguments of the product. These function arguments are supplied with the appropriate value when the compiler generates code.

When using a generic function that depends upon another generic function, the number of parameters increases and functions must be tupled together, and unpacked when required. We must write something like

$$\begin{aligned} &foobar\langle Prod\rangle\ (fooa, bara)\ (foob, barb)\ (x,y) = \\ &\qquad (\text{definition of } foo,\ \text{definition of } bar) \end{aligned}$$

rather than the more natural

$$\begin{aligned} &foo\ \langle Prod\ a\ b\rangle\ (x,y) = \text{definition of } foo \\ &bar\langle Prod\ a\ b\rangle\ (x,y) = \text{definition of } bar. \end{aligned}$$

The reason for the complications is that we do not have access to the type arguments, only to the recursive calls of the function being defined. That means, for instance, that we cannot access $bar\langle a\rangle$ while defining $foo\langle Prod\rangle$ as a stand-alone function. The language forces the function into the structure of a catamorphism, but sometimes this fixed recursion pattern is not a good match for the algorithm one wants to implement, resulting in unnecessarily complex and nearly unmaintainable code.

Our present goal is to move this complexity from the user to the compiler. This shift enables the programmer to write her code in a natural style, leaving the dependency constraints to the type system. The result is not only a rational reconstruction of Classic Generic Haskell, which we dub Dependency-style Generic Haskell, in doing so we gain in expressivity, enabling examples which were previously only possible using unnatural coding practices.

From a larger perspective, we provide a firm foundation which not only goes beyond the foundation for Classic Generic Haskell, but opens the door to tackle problems which had previously seemed too far off. Future possibilities include better support for type-indexed types, higher-order and locally defined generic functions, dependency inference, type inference of kind-$\star$ type arguments, generic functions based on kinds other than $\star$, and pattern matching on type arguments.

The paper is organized as follows. Section 2 presents some motivating examples. Section 3 describes the core calculus. Section 4 presents type rules for checking well-formed expressions along with their dependency information. Section 5 gives the reduction semantics of the calculus. Section 6 describes related work. Section 7 discusses what we have achieved here, and points to future work.

In this paper we use *Classic Generic Haskell* to refer to earlier versions of Generic Haskell, as implemented in the Amber and Beryl [5] versions of the compiler. This paper describes the foundations of *Dependency-style Generic Haskell*. When no confusion will arise, we use Generic Haskell to refer to Dependency-style Generic Haskell.

2 Examples

This section introduces Dependency-style Generic Haskell through a number of examples.

2.1 Generic equality

The equality function takes two values of a datatype and compares them. In Haskell the equality function can be derived for a user-defined datatype, as in

$$\textbf{data}\ \mathit{Bush} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Bin}\ \mathit{Bush}\ \mathit{Bush}\ \textbf{deriving}\ \mathit{Eq}.$$

In Generic Haskell we can *define* the generic equality function. A datatype consists essentially of sums, products, base types like *Int*, and a unit type. Sums, products and unit can be viewed as Haskell datatypes as follows:

$$\begin{array}{l}\textbf{data}\ \mathit{Unit}\ \ \ \ = \mathit{Unit}\\ \textbf{data}\ \mathit{Sum}\ a\ b = \mathit{Inl}\ a \mid \mathit{Inr}\ b\\ \textbf{data}\ \mathit{Prod}\ a\ b = (a, b).\end{array}$$

We need to define equality only on these types to obtain an equality function for an arbitrary datatype. In Classic Generic Haskell, this is coded as follows.

$$\begin{array}{l}\textbf{type}\ \mathit{Eq}\langle\!\langle\star\rangle\!\rangle\ \ \ \ \ \ \ \ \ t = t \to t \to \mathit{Bool}\\ \textbf{type}\ \mathit{Eq}\langle\!\langle\kappa_1 \to \kappa_2\rangle\!\rangle\ t = \forall u.\mathit{Eq}\langle\!\langle\kappa_1\rangle\!\rangle\ u \to \mathit{Eq}\langle\!\langle\kappa_2\rangle\!\rangle\ (t\ u)\end{array}$$

$$
\begin{array}{llll}
eq\langle t :: \kappa \rangle & & & :: Eq\langle\!\langle \kappa \rangle\!\rangle\ t \\
eq\langle Int \rangle & i & j & = eqInt\ i\ j \\
eq\langle Unit \rangle & Unit & Unit & = True \\
eq\langle Sum \rangle\ eqa\ eqb & (Inl\ a) & (Inl\ a') & = eqa\ a\ a' \\
eq\langle Sum \rangle\ eqa\ eqb & (Inr\ b) & (Inr\ b') & = eqb\ b\ b' \\
eq\langle Sum \rangle\ eqa\ eqb & \_ & \_ & = False \\
eq\langle Prod \rangle\ eqa\ eqb & (a, b) & (a', b') & = eqa\ a\ a' \wedge eqb\ b\ b'
\end{array}
$$

This definition consists of a kind-indexed type, a type signature assigning the kind-indexed type to *eq*, and several *cases* defining *eq* for the basic datatypes. Using this definition, the Classic Generic Haskell compiler generates a definition of $eq\langle Bush \rangle$, which is used to compare two *Bush* values [18].

The function *eq* is a generic function which recurses over the type structure of its argument type. The recursion is implicit in the arguments *eqa* and *eqb* in the *Sum* and *Prod* cases. We would rather write the following definition, which is how we define equality in Dependency-style Generic Haskell.

$$
\begin{array}{llll}
eq\langle Int \rangle & i & j & = eqInt\ i\ j \\
eq\langle Unit \rangle & Unit & Unit & = True \\
eq\langle Sum\ \delta a\ \delta b \rangle & (Inl\ a) & (Inl\ a') & = eq\langle \delta a \rangle\ a\ a' \\
eq\langle Sum\ \delta a\ \delta b \rangle & (Inr\ b) & (Inr\ b') & = eq\langle \delta b \rangle\ b\ b' \\
eq\langle Sum\ \delta a\ \delta b \rangle & \_ & \_ & = False \\
eq\langle Prod\ \delta a\ \delta b \rangle & (a, b) & (a', b') & = eq\langle \delta a \rangle\ a\ a' \wedge eq\langle \delta b \rangle\ b\ b'
\end{array}
$$

The recursion over the type structure is explicit: the case $eq\langle Prod\ \delta a\ \delta b \rangle$ is expressed in terms of $eq\langle \delta a \rangle$ and $eq\langle \delta b \rangle$. We think this style is more readable, especially when a generic function depends on another generic function. Functions written in the this style can be translated to the former style, so no expressiveness is lost; the only difference is readability.

We write δa for a type variable that appears in a type index and call it a *dependency variable*. Thus we syntactically distinguish quantified type variables from type variables that may appear in a type index. It is not necessary to make this distinction, but it simplifies the terminology in later discussions. A dependency variable introduces a dependency. For example, if we write $eq\langle List\ \delta a \rangle$, then this function *depends on* the function $eq\langle \delta a \rangle$. "Depending on" means that in order to call function $eq\langle List\ \delta a \rangle$ on two lists, we need a function $eq\langle \delta a \rangle$ that determines equality of the values in the list, as in the following example,

$$
\begin{array}{l}
(\textbf{let}\ eq\langle \delta a \rangle = \lambda x \to \lambda y \to eqInt\ x\ y\ \textbf{in} \\
\quad eq\langle List\ \delta a \rangle\ [1,2,3]\ \ [1,2,3], \\
\ \textbf{let}\ eq\langle \delta a \rangle = \lambda c \to \lambda d \to toUpper\ c \equiv toUpper\ d\ \textbf{in} \\
\quad eq\langle List\ \delta a \rangle\ \texttt{"Hello"}\ \texttt{"HELLO"})
\end{array}
$$

which has value $(True, True)$.

The type of the Classic (i.e., the first) version of *eq* is the kind-indexed type *Eq*. Consider the case $eq\langle Prod \rangle$. The product type has kind $\star \to \star \to \star$, and hence $eq\langle Prod \rangle$

takes an equality function for the left component of the product, and an equality function for the right component of the product, and only then takes two product values. The second line of a kind-indexed type has the same structure for any implicitly recursive generic function. The type of equality on a type of kind $\kappa_1 \to \kappa_2$ is a function from the type of equality for kind κ_1 to the type of equality for kind κ_2. This structure is enforced by the translation method used in Generic Haskell. In the case of the dependency-style definition, a type index always has kind $\star$, but in turn may contain dependency variables, so a kind-indexed type in the classic sense is not an option. If we ignore dependency variables, we have the following type for function *eq* on types t of kind $\star$.

$$eq\langle t\rangle :: t \to t \to Bool$$

If a type argument of a generic function contains a dependency variable, and the generic function uses a (possibly different) generic function in its definition, then the type records a *dependency constraint* for the dependency variable. For example, the generic function *eq* only uses itself, so for the type $List\ \delta a$ we get the following type.

$$\begin{aligned} &eq\langle List\ \delta a\rangle :: \forall a.(eq\langle \delta a\rangle :: a \to a \to Bool) \Rightarrow \\ &\quad List\ a \to List\ a \to Bool \end{aligned}$$

It turns out that all information about the type of a generic function can be calculated from the base case for a single dependency variable of kind $\star$. For *eq* this is

$$eq\langle \delta a\rangle :: \forall a.(eq\langle \delta a\rangle :: a \to a \to Bool) \Rightarrow a \to a \to Bool.$$

The type of *eq* on a type with a kind other than $\star$, possibly containing many dependency variables, is a generalization of this type. We make this explicit by abstracting over the types a and δa using the **generalize** construct, and applying the expression obtained to the type on which we want an instance of the equality function.

$$\begin{aligned} &eq\langle t\rangle :: (\mathbf{generalize}\ \langle \delta a\rangle\ a \mapsto \\ &\quad (eq\langle \delta a\rangle :: a \to a \to Bool) \Rightarrow a \to a \to Bool)\ t \end{aligned}$$

In Section 4 we explain how to calculate instances of a generalized type.

2.2 Huffman coding in XComprez

XComprez is a generic compressor for XML documents [12]. XComprez separates an XML document into its structure (the markup) and its contents (the strings). The DTD that describes the structure of the document is translated to a Haskell datatype, and the structure of the document is translated to a value of this datatype. Using knowledge about the DTD (and the datatype to which it is translated), the structure of an XML document can be compressed considerably; the contents are compressed by means of a standard compressor.

To improve the compression of the contents, we apply the following variant of Huffman coding. Given an input value, we calculate the number of occurrences of each

constructor. Given the number of occurrences of each constructor, we calculate the optimal Huffman encoding for the particular value, and we encode the value using this encoding.

The function *conCount* calculates the number of occurrences of each constructor in a value of a datatype:

$$
\begin{aligned}
&conCount\langle t\rangle :: (\textbf{generalize } \langle \delta a\rangle\ b \mapsto \\
&\quad (conCount\langle \delta a\rangle :: b \to [(ConDescr, Int)]) \\
&\qquad \Rightarrow b \to [(ConDescr, Int)])\ t.
\end{aligned}
$$

The function *conCount* is used in *encode*

$$
\begin{aligned}
&encode\langle t\rangle :: (\textbf{generalize } \langle \delta a\rangle\ b \mapsto \\
&\quad (conCount\langle \delta a\rangle :: b \to [(ConDescr, Int)], \\
&\quad\ encode'\langle \delta a\rangle \quad :: b \to [(ConDescr, Int)] \to [Bit]) \\
&\qquad \Rightarrow b \to [Bit])\ t \\
&encode\langle \delta a\rangle\ x = \textbf{let } table = conCount\langle \delta a\rangle\ x \textbf{ in} \\
&\qquad\qquad\qquad encode'\langle \delta a\rangle\ table\ x,
\end{aligned}
$$

where *encode′* is the generic function that encodes each constructor as a list of bits [12]. This is an example of a *generic abstraction* [6]: a generic function defined in terms of one or more other generic functions instead of recursively over the type structure. Function *encode* is an example where the dependency constraints contain dependencies on other generic functions, unlike function *eq* which only contains the constraint that is identical to the type itself.

2.3 Pretty-printing "important" information

Suppose we want to pretty-print a value of a datatype, but only parts which satisfy some condition. For example, we only want to print trees that have at least a certain height, or we only want to print balanced trees. Suppose we have a generic function *important* that determines whether or not to print a subtree.

$$important\langle t\rangle :: (\textbf{generalize } \langle \delta a\rangle\ b \mapsto (important\langle \delta a\rangle :: b \to Bool) \Rightarrow b \to Bool)\ t$$

The function *print* depends on function *important*.

```
print⟨Int⟩                 = printInt
print⟨Unit⟩                = printUnit
print⟨Sum δa δb⟩ (Inl a)   =
    if important⟨δa⟩ a then print⟨δa⟩ a else "..."
print⟨Sum δa δb⟩ (Inr b)   =
    if important⟨δb⟩ b then print⟨δb⟩ b else "..."
print⟨Prod δa δb⟩ (a, b)   =
    if important⟨δa⟩ a
    then if important⟨δb⟩ b then print⟨δa⟩ ++ print⟨δb⟩ b
                            else print⟨δa⟩ a ++ "..."
    else if important⟨δb⟩ b then "..." ++ print⟨δb⟩ b
                            else "..."
```

The dependency shows in the type of function *print*:

```
print⟨t⟩ :: (generalize ⟨δa⟩ b ↦
    (important⟨δa⟩ :: b → Bool, print⟨δa⟩ :: b → String)
        ⇒ b → String) t.
```

This is an example of a generic function defined recursively over the type structure, which depends on another generic function. In Classic Generic Haskell, we would have to pair the definitions of *print* and *important*, defining two essentially separate aspects at the same time.

2.4 Traversal functions

To illustrate the extensibility Generic Haskell provides, we present a series of modifications to a running example. Adapting from Lämmel and Peyton Jones [25], we use the following datatypes to represent the organizational structure of a company.

```
data Company = C [Dept]
data Dept    = D Name Manager [SubUnit]
data SubUnit = PU Employee | DU Dept
data Employee = E Person Salary
data Person  = P Name Address
data Salary  = S Float
type Manager = Employee
type Name    = String
type Address = String
```

Update Salaries We wish to update a *Company* value, which involves giving every *Person* a 15% pay rise. To do so requires visiting the entire tree and modifying every occurrence of *Salary*. The implementation requires pretty standard "boilerplate" code

which traverses the datatype, until it finds *Salary*, where it performs the appropriate update—itself one line of code—before reconstructing the result.

In Generic Haskell writing this function requires but a few lines. The code is based on the standard generic *map*:

$$
\begin{array}{lll}
map\langle t\rangle :: (\textbf{generalize}\ \langle \delta a\rangle\ a_1\ a_2 \mapsto (map\langle \delta a\rangle :: a_1 \to a_2) \Rightarrow a_1 \to a_2)\ t\ t \\
map\langle Unit\rangle & v & = v \\
map\langle Int\rangle & v & = v \\
map\langle Sum\ \delta a\ \delta b\rangle & (Inl\ a) & = Inl\ (map\langle \delta a\rangle\ a) \\
map\langle Sum\ \delta a\ \delta b\rangle & (Inr\ b) & = Inr\ (map\langle \delta b\rangle\ b) \\
map\langle Prod\ \delta a\ \delta b\rangle & (a, b) & = (map\langle \delta a\rangle\ a, map\langle \delta b\rangle\ b).
\end{array}
$$

The generalized type of *map* takes two arguments so that it can be used in the type modifying way we are used to. With only one argument it would degenerate to the generic identity function [14]. This is our first example of a generalized type that takes more than one type argument.

The code to perform the updating is given by the following three lines, the first of which is the necessary type signature, the second states that the function is based on *map*, though it introduces an additional parameter which is automatically threaded through the computation, and the third performs the update of the salary, using the threaded value. The **extends** construct denotes that the cases of *map* are copied into *update*. These are the *default cases* described in Clarke and Löh [6]. (This is, in effect, a preprocessing step and thus **extends** does not appear in the calculus.)

$$
\begin{array}{l}
update\langle t\rangle :: (\textbf{generalize}\ \langle \delta a\rangle\ a \mapsto \\
\quad (update\langle \delta a\rangle :: Float \to a \to a) \Rightarrow Float \to a \to a)\ t \\
update\langle \delta a\rangle\ p\ \textbf{extends}\ map\langle \delta a\rangle \\
update\langle Salary\rangle\ p\ (S\ s) = S\ (s \cdot (1 + p))
\end{array}
$$

The introduction of the new threaded variable p is impossible in Classic Generic Haskell. Being able to do this is a small but significant advance over our previous work.

Update "Update Salaries" Political forces require that the code be changed so that only marketing gets a raise, of 25%, as well as giving a certain manager a 10% raise. Being versed in generic programming enables the programmer to update the previous code as follows:

$$
\begin{array}{l}
updateNew\langle t\rangle :: (\textbf{generalize}\ \langle \delta a\rangle\ a \mapsto \\
\quad (updateNew\langle \delta a\rangle :: Float \to a \to a) \Rightarrow Float \to a \to a)\ t \\
updateNew\langle \delta a\rangle\ \textbf{extends}\ update\langle \delta a\rangle \\
updateNew\langle Dept\rangle\ p\ (D\ name\ manager\ subunits) = \\
\quad \textbf{let}\ newp = \textbf{if}\ name \equiv \texttt{"marketing"}\ \textbf{then}\ 0.25\ \textbf{else}\ p\ \textbf{in} \\
\qquad D\ name\ manager\ (updateNew\langle SubUnit\rangle\ newp\ subunits) \\
updateNew\langle Employee\rangle\ p\ (E\ person\ salary) = \\
\quad \textbf{let}\ newp = \textbf{if}\ person \equiv daevclarke\ \textbf{then}\ 0.1\ \textbf{else}\ p\ \textbf{in} \\
\qquad E\ person\ (updateNew\langle Salary\rangle\ newp\ salary).
\end{array}
$$

Separating Matching from Updating Understanding the social nature of management structures, the programmer writes her code more generally by separating the updating of salaries from the code which determines who gets what pay rise.

A match function performs a test on a part of the organizational structure. The result of the function is one of: *NoMatch* indicating that no work need be done to that part; *Value n* indicating that the value, presumably the amount of the pay rise, needs to be applied to this part of the tree; and finally *Inconsistent* indicating that different subparts of the same part require different inconsistent updates—which requires recursion to further distinguish the parts. This is captured using the following datatype and operation:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Data}\ a = \mathit{NoMatch} \mid \mathit{Value}\ a \mid \mathit{Inconsistent}\\
&(\bowtie) :: (\mathit{Eq}\ a) \Rightarrow \mathit{Data}\ a \to \mathit{Data}\ a \to \mathit{Data}\ a\\
&\mathit{NoMatch} \bowtie \mathit{NoMatch} = \mathit{NoMatch}\\
&(\mathit{Value}\ a) \bowtie (\mathit{Value}\ b) = \textbf{if}\ a \equiv b\ \textbf{then}\ \mathit{Value}\ a\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\ \mathit{Inconsistent}\\
&\_ \qquad\quad \bowtie \_ \qquad\quad = \mathit{Inconsistent}.
\end{aligned}
$$

Using the generic function known as *crush*, which is a generalization of *fold*,

$$
\begin{aligned}
&\mathit{crush}\langle t\rangle :: (\textbf{generalize}\ \langle \delta a\rangle\ a \mapsto\\
&\quad (\mathit{crush}\langle \delta a\rangle :: b \to (b \to b \to b) \to a \to b)\\
&\qquad \Rightarrow b \to (b \to b \to b) \to a \to b)\ t\\
&\mathit{crush}\langle \mathit{Unit}\rangle \quad e\ \mathit{op} \quad = e\\
&\mathit{crush}\langle \mathit{Int}\rangle \quad e\ \mathit{op} \quad = e\\
&\mathit{crush}\langle \mathit{Sum}\ \delta a\ \delta b\rangle\ e\ \mathit{op}\ (\mathit{Inl}\ a) = \mathit{crush}\langle \delta a\rangle\ e\ \mathit{op}\ a\\
&\mathit{crush}\langle \mathit{Sum}\ \delta a\ \delta b\rangle\ e\ \mathit{op}\ (\mathit{Inr}\ b) = \mathit{crush}\langle \delta b\rangle\ e\ \mathit{op}\ b\\
&\mathit{crush}\langle \mathit{Prod}\ \delta a\ \delta b\rangle\ e\ \mathit{op}\ (a, b) \quad = \mathit{op}\ (\mathit{crush}\langle \delta a\rangle\ e\ \mathit{op})\ (\mathit{crush}\langle \delta b\rangle\ e\ \mathit{op}\ b),
\end{aligned}
$$

the basis for all match functions is

$$
\begin{aligned}
&\mathit{matchBase}\langle t\rangle :: (\textbf{generalize}\ \langle \delta a\rangle\ a \mapsto\\
&\quad (\mathit{matchBase}\langle \delta a\rangle :: a \to \mathit{Data}\ b) \Rightarrow a \to \mathit{Data}\ b)\ t\\
&\mathit{matchBase}\langle \delta a\rangle\ \textbf{extends}\ \mathit{crush}\langle \delta a\rangle\ \mathit{NoMatch}\ (\bowtie).
\end{aligned}
$$

This match function can be specialized to perform the match that we are interested in, as follows.

$$
\begin{aligned}
&\mathit{match}\langle t\rangle :: (\textbf{generalize}\ \langle \delta a\rangle\ a \mapsto\\
&\quad (\mathit{match}\langle \delta a\rangle :: a \to \mathit{Data\ Float}) \Rightarrow a \to \mathit{Data\ Float})\ t\\
&\mathit{match}\langle \delta a\rangle\ \textbf{extends}\ \mathit{matchBase}\langle \delta a\rangle\\
&\mathit{match}\langle \mathit{Dept}\rangle\ (D\ \mathit{name}\ \_\ \_) \quad =\\
&\quad \textbf{if}\ \mathit{name} \equiv \texttt{"marketing"}\ \textbf{then}\ (\mathit{Value}\ 0.25)\ \textbf{else}\ \mathit{NoMatch}\\
&\mathit{match}\langle \mathit{Employee}\rangle\ p\ (E\ \mathit{person}\ \_) =\\
&\quad \textbf{if}\ \mathit{person} \equiv \mathit{daevclarke}\ \textbf{then}\ (\mathit{Value}\ 0.1)\ \textbf{else}\ \mathit{NoMatch}
\end{aligned}
$$

The update function can now refer to this match function, though their code can evolve separately. This is where the dependency information of Generic Haskell comes in.

$$
\begin{aligned}
&\mathit{updatePre}\langle t\rangle :: (\textbf{generalize } \langle \delta a\rangle\ a \mapsto \\
&\quad (\mathit{updatePre}\langle \delta a\rangle :: \mathit{Float} \to a \to a, \\
&\quad\ \mathit{update}\langle \delta a\rangle \quad :: \mathit{Float} \to a \to a, \\
&\quad\ \mathit{match}\langle \delta a\rangle \quad :: a \to \mathit{Data\ Float}) \\
&\qquad \Rightarrow \mathit{Float} \to a \to a)\ t \\
&\mathit{updatePre}\langle \delta a\rangle\ \textbf{extends}\ \mathit{update}\langle \delta a\rangle \\
&\mathit{updatePre}\langle \mathit{Con}\ \delta a\rangle\ p\ c@(\mathit{Con}\ a) = \\
&\quad \textbf{case}\ \mathit{match}\langle \mathit{Con}\ \delta a\rangle\ c\ \textbf{of} \\
&\qquad \mathit{Value\ newp} \to \mathit{Con}\ \$\ \mathit{update}\langle \delta a\rangle\ \mathit{newp}\ a \\
&\qquad \mathit{NoMatch} \to c \\
&\qquad \mathit{Inconsistent} \to \mathit{Con}\ \$\ \mathit{updatePre}\langle \delta a\rangle\ p\ a
\end{aligned}
$$

The *Con*-case is applied at the constructor positions in a value. This code can be converted to an ordinary Haskell value by specializing it to our *Company* datatype as follows:

$$
\begin{aligned}
&\mathit{updateGrand} :: \mathit{Float} \to \mathit{Company} \to \mathit{Company} \\
&\mathit{updateGrand} = \mathit{updatePre}\langle \mathit{Company}\rangle.
\end{aligned}
$$

2.5 Further Applications

Other applications which benefit from Dependency-style Generic Haskell include type-indexed datatypes [19, 17], such as generic dictionaries and the zipper: a data structure used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up, or down the tree. These navigation functions depend heavily upon each other, and create a heavy notational burden when expressed in Classic Generic Haskell.

3 Core language

Figure 1 presents the grammar for the core language underlying Dependency-style Generic Haskell. Grammar productions which specifically treat generic constructs are emphasized. The core language simplifies the language in which we present examples: all lambda abstractions are explicit, whereas pattern matching, constructors, infix operators, tuples and lists are not treated specially. Furthermore, **let** has been divided into **letrec** and **deplet**.

We use horizontal bars over subexpressions to denote potential repetition of constructs. For instance, $\bar{t}$ denotes a vector of (possibly different) types. Both primed (t') and indexed (t_0) elements denote independent elements, rather than the components of a vector, as we never need to refer to vector elements individually.

A Generic Haskell program consists of four components: programs, expressions, types and kinds.

Kinds

$\kappa ::= \star$ (kind of manifest types)
$\mid \bullet$ *(kind of type argument)*
$\mid \circ$ *(generic kind)*
$\mid \kappa_1 \to \kappa_2$ (functional kind)

Dependency kinds

$\rho ::= (\mathsf{K}) \Rightarrow \kappa$

Kind dependencies

$\mathsf{K} ::= \overline{\delta a :: \kappa}$

Types

$t ::= a, b, c, \ldots$ (variable)
$\mid \delta a, \delta b, \delta c, \ldots$ *(dependency variable)*
$\mid (t_1\ t_2)$ (application)
$\mid t_1 \to t_2$ (functional type)
$\mid$ **generalize** $\langle \delta a \rangle \mapsto \overline{\Lambda a}.\ s$
(generic type)

Type schemes/dependency types

$s ::= \overline{\forall a}.\ (\mathsf{D}) \Rightarrow t$

Dependency constraints

$\mathsf{D} ::= \overline{x\langle \delta a\ \overline{\delta b}\rangle :: s}$

Expressions

$e ::= x, y, z, \ldots$ (variable)
$\mid (e_1\ e_2)$ (application)
$\mid \lambda x \to e$ (lambda abstraction)
$\mid x\langle t \rangle$ *(type application)*
$\mid$ **letrec** $\overline{x = e}$ **in** e_0 (let(rec) binding)
$\mid$ **deplet** $x\langle \delta a\ \overline{\delta b}\rangle = e_1$ **in** e_2
(dependency binding)

Declarations

$D ::= x\langle a \rangle :: s = \overline{\langle b\ \overline{\delta c}\rangle \mapsto e}$
generic function

Main program

$M ::= \overline{D}; e$

Figure 1: Syntax of Generic Haskell core language

3.1 Programs

A program consists of definitions of generic functions, followed by a single expression to be evaluated. A generic function is a type-indexed value, an expression taking a type argument, consisting of an obligatory type signature followed by a number of cases defined over type patterns. Without loss of generality, we also assume that there is a set of global predefined datatypes and functions, rather than include these in the calculus.

3.2 Expressions

An expression may be a variable, a function application or abstraction, a recursive let binding (which is now marked explicitly as **letrec** instead of **let**), or one of two new features for generic programming. The first of these applies a generic function to a type to obtain the instance of that generic function on that type. Such an application of a type-indexed value might introduce a dependency constraint in the type of the expression, which will be propagated outwards until satisfied. This means that the generic function depends on an additional value which should be provided by the environment surrounding the expression. This is achieved using a **deplet**, the second generic programming feature. The **deplet** construct introduces dependency bindings which are used to satisfy dependency constraints arising from its body. The variable name associated with a **deplet** must refer to a known type-indexed value defined in the top-level program.

3.3 Types

Types are divided into two levels: types and type schemes, i.e. types with dependency constraints or universal quantification.

The first level, ordinary program types, consists of type variables (which also refer to globally known datatypes), type application and function types. In addition to these familiar concepts we have the novel notion of dependency type variable. Dependency type variables only appear within the special type parentheses $\langle\cdot\rangle$. Ordinary type variables are bound by a universal quantifier or a lambda abstraction at the type level, whereas dependency type variables are bound inside the type parentheses within a case of a generic function $\langle t\rangle \mapsto e$ or in a **deplet** construct, such as **deplet** $v\langle\delta a\ \overline{\delta b}\rangle = e_1$ **in** e_2. In the generic function case, the dependency variables scope over e; in the **deplet**, the variable δa scopes over e_1 and e_2, whereas the arguments $\overline{\delta b}$ are local to e_1. The type of a generic function is a generic type, built using **generalize** $\langle\delta a\rangle \mapsto \overline{\Lambda a}.\ s$. Lambda abstractions on the type level can occur only here. The generic type is instantiated at a particular type to obtain the type of an instance of the generic function for that type.

The second level consists of type schemes, which are types extended with universal quantification (to denote polymorphic types) and with dependency constraints (to denote dependencies on generic functions). A dependency constraint is a set of dependencies, each consisting of the name of a generic function, a dependency variable and a type. Dependencies are introduced by calls to generic functions and can be satisfied by **deplet**-bindings. We write $\overline{\forall a}.\ t$ if the set of dependency constraints is empty, and $(\mathsf{D}) \Rightarrow t$ if there are no quantified variables.

3.4 Kinds

Although they do not appear in the program text, kinds are used heavily in the internal machinery to control the well-formedness of types.

The kind of manifest types (i.e., types which correspond directly to values) is $\star$. Type constructors have functional kinds. Furthermore, we have two special kinds which play a role in the types of generic functions. The "new moon" $\bullet$ is used as the kind of the type argument in a generic function, and the "full moon" $\circ$ is the kind of the fully applied type of a generic function.

If a type contains one or more dependency variables, then this fact is reflected in its kind, $(\mathsf{K}) \Rightarrow \kappa$, in the form of kind dependencies. A kind dependency, K, is a set consisting of kind assignments for each dependency type variable. We use kind dependencies to check if the dependency structure of a type is well-formed, as well as to steer the computation of the generalized type of generic functions.

3.5 Scoping rules

Dependency variables are visible within a type signature, a case of a generic definition, or a complete expression. Within their scope, dependency variables have to be used with consistent kind. A **deplet** introduces local dependency variables, namely the ar-

guments to the main dependency variable, in the bound expression. Note that there is no intended assignment of dependency variables to specific types, and as such, usage of dependency variables does not need to be consistent in that sense. The important case in the definition of the function fdv which returns the free dependency variables is

$$\mathsf{fdv}(\mathbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \mathbf{in}\ e_2) = (\mathsf{fdv}(e_1) \setminus \{\overline{\delta b}\}) \cup \mathsf{fdv}(e_2) \cup \{\delta a\}.$$

Expression variables are bound by an enclosing lambda or **letrec**, but not by a **deplet**. The interesting cases of the function fev which returns the free expression variables are thus:

$$\begin{array}{ll}
\mathsf{fev}(\lambda x \to e) & = \mathsf{fev}(e) \setminus \{x\} \\
\mathsf{fev}(\mathbf{letrec}\ \overline{x = e}\ \mathbf{in}\ e_0) & = (\mathsf{fev}(\overline{e}) \cup \mathsf{fev}(e_0)) \setminus \{\overline{x}\} \\
\mathsf{fev}(\mathbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \mathbf{in}\ e_2) & = \mathsf{fev}(e_1) \cup \mathsf{fev}(e_2) \cup \{x\}.
\end{array}$$

Normal type variables are bound by an enclosing type lambda or universal quantifier. The interesting cases are:

$$\begin{array}{l}
\mathsf{ftv}(\forall a.s)\ = \mathsf{ftv}(s) \setminus \{a\} \\
\mathsf{ftv}(\Lambda a.s) = \mathsf{ftv}(s) \setminus \{a\}
\end{array}$$

Bound variables can be renamed freely.

4 Type and kind checking

The development of the type system we present in this section, and indeed the recent development of Generic Haskell, has been driven by how generic functions are used in actual programs. The original work of Hinze enabled independent generic functions to be written in a natural recursive style [15], though such functions could only be written for fixed kinds such as $\star$ or $\star \to \star$. Hinze [16] then lifted this restriction. As a result, generic functions in Generic Haskell are applicable to types of any kind. Unfortunately, the style in which functions are written is cumbersome to use. In this paper, we do things a little differently from Hinze, but gain significantly more ground. We enable generic functions to be written in a natural style, while retaining the ability to apply them at all kinds. The Classic Generic Haskell value $map\langle List\rangle$ becomes $map\langle List\ \delta a\rangle$ in the dependency style. That is, instead of applying *map* to the type constructor *List*, the arguments to the constructor are supplied with dependency variables, such as δa in $List\ \delta a$. In addition, the Classic Generic Haskell value $map\langle List\rangle$ receives the function which it applied to the list elements as a ordinary function argument, whereas in Dependency-style Generic Haskell, this argument is supplied via a **deplet** expression which provides a binding for $map\langle \delta a\rangle$.

Thus in the dependency style, whenever a type argument appears within the type parentheses $\langle\cdot\rangle$, it has kind $\star$, modulo dependencies. Our type system assigns the kind $(\delta a :: \star) \Rightarrow \star$ to the type $List\ \delta a$, which mirrors the $\star \to \star$ kind of the *List* type constructor. In the dependency style, generic functions still possess kind-indexed types,

but they can no longer be automatically applied to types of *all* kinds, rather types must be supplied with enough dependency variables to make them effectively kind $\star$. So ultimately nothing is lost.

The code of *map* (see Section 2) introduces a recursive dependency of *map* on itself at type δa. In addition to this common behavior, our system enables further dependencies on other generic functions. These are recorded in a generic type, as we also have seen in the examples. The goal of the type system, beyond usual type correctness, is to ensure that the correct dependency information is specified with generic functions, and that dependencies are correctly satisfied or propagated when using generic functions.

The kind checking rules are explained in Section 4.1. Before delving into the type system, we must first discuss in Section 4.2 the rather technical topic of kind-indexed types, which denote the signatures of generic functions and from which the types of instances are determined. Forming the basis for Hinze's elegant proposal [16], these now bear the brunt of the complexity with the addition of dependencies. The type checking rules will then be explained in Section 4.3.

$$\text{T-VAR}\ \frac{a :: \kappa \in \mathsf{A}}{\mathsf{A} \vdash a :: () \Rightarrow \kappa} \qquad \text{T-DVAR}\ \frac{}{\mathsf{A} \vdash \delta a :: (\delta a :: \kappa) \Rightarrow \kappa}$$

$$\text{T-SUB}\ \frac{\mathsf{A} \vdash t :: \rho_1 \qquad \rho_1 \leqslant \rho_2}{\mathsf{A} \vdash t :: \rho_2}$$

$$\text{T-APP}\ \frac{\mathsf{A} \vdash t_1 :: (\mathsf{K}) \Rightarrow (\kappa_2 \to \kappa_1) \qquad \mathsf{A} \vdash t_2 :: (\mathsf{K}) \Rightarrow \kappa_2}{\mathsf{A} \vdash (t_1\ t_2) :: (\mathsf{K}) \Rightarrow \kappa_1}$$

$$\text{T-FUN}\ \frac{\mathsf{A} \vdash s_1 :: (\mathsf{K}) \Rightarrow \star \qquad \mathsf{A} \vdash s_2 :: (\mathsf{K}) \Rightarrow \star}{\mathsf{A} \vdash (s_1 \to s_2) :: (\mathsf{K}) \Rightarrow \star}$$

$$\text{T-FORALL}\ \frac{\mathsf{A}, a :: \kappa_1 \vdash s :: () \Rightarrow \kappa_2 \qquad \kappa_2 \in \{\star, \bigcirc\}}{\mathsf{A} \vdash \forall a.s :: () \Rightarrow \kappa_2}$$

$$\text{T-LAMBDA}\ \frac{\mathsf{A}, a :: \kappa_1 \vdash s :: (\mathsf{K}) \Rightarrow \kappa_2}{\mathsf{A} \vdash \Lambda a.s :: (\mathsf{K}) \Rightarrow \kappa_1 \to \kappa_2}$$

$$\text{T-DEP}\ \frac{\begin{array}{c}\mathsf{A} \vdash s_1 :: \overline{\delta b :: \kappa} \Rightarrow \star \\ \kappa_0 \equiv \overline{\kappa \to} \star \qquad \mathsf{A} \vdash (\mathsf{D}) \Rightarrow s_2 :: (\delta a_0 :: \kappa_0, \mathsf{K}) \Rightarrow \star\end{array}}{\mathsf{A} \vdash (x\langle \delta a_0\ \overline{\delta b}\rangle :: s_1, \mathsf{D}) \Rightarrow s_2 :: (\delta a_0 :: \kappa_0, \mathsf{K}) \Rightarrow \star}$$

$$\text{T-GENERIC}\ \frac{\mathsf{A} \vdash \overline{\Lambda a.}\ s :: (\delta a :: \star) \Rightarrow \star^n}{\mathsf{A} \vdash \textbf{generalize}\ \langle \delta a \rangle \mapsto \overline{\Lambda a.}\ s :: () \Rightarrow \bigcirc^n}$$

Figure 2: Kind checking rules

$$\text{K-INST-REFL}\ \frac{}{\rho \leqslant \rho} \qquad \text{K-INST-DEP}\ \frac{(\mathsf{K}_1) \Rightarrow \kappa_1 \leqslant (\mathsf{K}_2) \Rightarrow \kappa_1}{(\mathsf{K}_1) \Rightarrow \kappa_1 \leqslant (\delta a :: \kappa_2, \mathsf{K}_2) \Rightarrow \kappa_1}$$

Figure 3: Instance relation on (dependency) kinds

4.1 Kind checking

Kind judgments are of the form $\mathsf{A} \vdash f :: (\mathsf{K}) \Rightarrow \kappa$. The environment A assigns kinds to global datatypes and local type variables. The kind-level dependencies K bind dependency variables to kinds. Dependencies are part of the syntax of the kinds. The intention is that the dependencies are inferred rather than previously known, which is why K does not appear on the left-hand side of the turnstile. The codomains of both A and K are restricted to plain kinds i.e., dependencies on the kind level are not nested. The full kind checking rules are presented in Figure 2.

The rules T-VAR, T-APP, T-FUN and T-LAMBDA are relatively standard, except that a collection of kind-level dependencies is threaded through the rules along with the kind.

The connection between kind-level dependencies and the use of dependency variables on the type level is established in the rules T-DVAR and T-DEP. A dependency variable introduces a dependency on itself by rule T-DVAR. Dependency variables must be used consistently at the same kind in any given scope. Type-level dependencies are also reflected at the kind level using rule T-DEP. If a type has a dependency of the form $(v\langle \delta a\ \overline{\delta b}\rangle :: s)$, then δa is the only dependency variable that is visible outside; arguments $\overline{\delta b}$ are local to the type scheme s. Thus s may depend on these argument variables (and only on these) and is thus of kind $(\overline{\delta a :: \kappa}) \Rightarrow \star$, where $\overline{\kappa}$ are the argument variables' kinds. The kind of δa is $\overline{\kappa \rightarrow} \star$, and this dependency is recorded in the final kind of the type.

Type signatures of generic functions are special in that they contain the **generalize** construct. This construct transforms a specific instance of the type of the generic function into a generalized type. This process is mirrored on the kind level. From rule T-GENERIC, the type to be generalized must have kind $\star^n$, where

$$\star^0 = \star \qquad \star^{n+1} = \star \rightarrow \star^n.$$

The arguments of this type are, after generalization, instantiated to the type argument of the generic function. To enforce this instantiation, the type argument of the generic function gets a special kind, the "new moon" $\bullet$. Accordingly, rule T-GENERIC assigns kind

$$\mathbf{◐}^0 = \circ \qquad \mathbf{◐}^{n+1} = \bullet \rightarrow \mathbf{◐}^n$$

to the type resulting from generalization. The fully applied generalized type then has "full moon" kind $\circ$. Only generalization introduces $\bullet$ and $\circ$ kinds, and P-GENERIC (discussed in Section 4.3) enforces that all type signatures of generic functions are of kind $\circ$.

Type signatures of generic functions must thus always consist of a fully applied generalized type, possibly involving universal quantification for type variables of kind $\circ$, as allowed by rule T-FORALL. The algorithm that computes the generalized kind-indexed type for the generic function makes heavy use of this fact.

The final component of kinding is the instance relation $\rho_1 \leqslant \rho_2$ between kinds. This is defined in rules K-INST-REFL and K-INST-DEP (Figure 3), and is used as a kind-level subsumption in T-SUB to extend the set of dependencies which appear in a kind.

4.2 Kind-indexed types

One of the key ideas of Hinze's theory, as implemented in Classic Generic Haskell, is that certain type-level constructs, such as recursion, abstraction and application, are *always* interpreted as their value-level counterparts. This leads to the reasoning that generic functions possess kind-indexed types [16]. The type of a generic function is tied to the kind of its type argument, establishing identities such as:

$$eq\langle Tree\ Int\rangle \equiv eq\langle Tree\rangle\ (eq\langle Int\rangle).$$

Here the application at the type level (of *Tree* to *Int*) can be replaced by the value level application of two instances of the generic function at those types.

In Dependency-style Generic Haskell, we no longer allow type arguments of higher kinds, but we keep the general idea: instead of a higher-kinded type, we use type arguments which contain dependency variables. This shift enables generic functions to depend on generic functions other than just itself. In the dependency style, the identity corresponding to the one above is:

$$eq\langle Tree\ Int\rangle \equiv \\ \quad \mathbf{deplet}\ eq\langle \delta a\rangle = eq\langle Int\rangle\ \mathbf{in}\ eq\langle Tree\ \delta a\rangle.$$

In Classic Generic Haskell, kind-indexed types have to be defined explicitly by the programmer. For instance, the type of generic equality reads

$$\begin{aligned} &Eq\langle\!\langle \star \rangle\!\rangle && a = a \to a \to Bool \\ &Eq\langle\!\langle \kappa_1 \to \kappa_2 \rangle\!\rangle && a = \forall b.Eq\langle\!\langle \kappa_1 \rangle\!\rangle\ b \to Eq\langle\!\langle \kappa_2 \rangle\!\rangle\ (a\ b). \end{aligned}$$

The function $eq\langle t\rangle$ then has type $Eq\langle\!\langle \kappa \rangle\!\rangle\ t$, where κ is the kind of t. The second case captures the fact that equality on kind $\kappa_1 \to \kappa_2$ takes an equality function on the argument type to an equality function on the resulting type.

Note that the second line of this type could be automatically derived by the compiler, because the interpretation of type application as value-level application is intrinsic to the theory. Because of the increased complexity due to dependencies, we opt for this solution in dependency-style Generic Haskell, requiring a signature only for kind $\star$, plus dependency information. Thus a generic type signature takes the following form (the general form of types of kind $\circ$):

$$x\langle c\rangle :: \overline{\forall a.}\ (\mathbf{generalize}\ \langle \delta a\rangle \mapsto \overline{\Lambda b.}\ (\overline{y\langle \delta a\rangle :: s}) \Rightarrow t_0)\ c \ldots c.$$

Three sorts of type variables occur within a type signature: the type argument c; outer quantified, *non-generic* variables $\overline{a}$; and abstracted variables $\overline{b}$. The latter kind are called *generic* variables and will be instantiated with the type argument when generating instances of the generic type. Note that there are as many applications to c at the end as there are variables in $\overline{b}$.

A generic function may depend on itself and on other functions. The dependencies $\overline{y}$ are the generic functions that x depends on (we also refer to the vector $\overline{y}$ as $\mathsf{dependencies}(x)$).

The pair $(\mathsf{length}(\overline{a}), \mathsf{length}(\overline{b}))$ is called the *signature* of x. We define the *base type* of x as

$$\mathsf{base}(x)\ (\overline{a} \mid \overline{b}) = t_0$$

The base type contains no dependency information.

Let us look at at the type schemes s that occur in the dependency constraint: for the generic type to be well-formed, each of these type schemes has to be an instance of the base type of the corresponding generic function y. However, as y might be another generic function (thus not x itself), it might be of a different signature, and we cannot be sure to which variables the base type is instantiated. All we know is that there must be such an instantiation. This instantiation is called the *constraint substitution* for y in x, so that we can rewrite each dependency constraint in the type to

$$y\langle\delta a\rangle :: \mathsf{base}(y)\ (\mathsf{cs}(y; x; \overline{a} \mid \overline{b}))$$

Hence, $\mathsf{cs}(y; x; \cdot)$ maps a vector of variables of the signature of x to a vector of variables of the signature of y. The whole type signature of x can thus be written as

$$\begin{aligned} x\langle c\rangle :: \overline{\forall a.}\ (&\mathbf{generalize}\ \langle\delta a\rangle \mapsto \overline{\Lambda b.}\ (\overline{y\langle\delta a\rangle :: \mathsf{base}(y)\ (\mathsf{cs}(y; x; \overline{a} \mid \overline{b}))}) \\ &\Rightarrow \mathsf{base}(x)\ (\overline{a} \mid \overline{b}))\ c \ldots c \end{aligned}$$

The constraint substitution $\mathsf{cs}(x; x; \cdot)$ for a function that depends on itself is always the identity.

The **generalize** construct generalizes the type for a special case (the type of $x\langle\delta a\rangle$, where δa is of kind $(\delta a :: \star) \Rightarrow \star$) to a function tapp which produces a correct type for all type arguments of kind $(\mathsf{K}) \Rightarrow \star$. This function is used during type checking and is defined in terms of the kind-indexed type kapp as follows:

$$\mathsf{tapp}(x; t) = \overline{\forall a.}\ \mathsf{kapp}(x; \rho; \overline{a} \mid t \ldots t).$$

where ρ is the dependency kind of t. The function kapp, which is similar to the kind-indexed type in Classic Generic Haskell, depends on both the non-generic and the generic variables that occur in the type signature. The full definition of kapp is:

$$\begin{aligned} &\mathsf{kapp}(x; \star; && \overline{a} \mid \overline{t}) = \mathsf{base}(x)\ (\overline{a} \mid \overline{t}) \\ &\mathsf{kapp}(x; () \Rightarrow \star; && \overline{a} \mid \overline{t}) = \mathsf{kapp}(x; \star; \overline{a} \mid \overline{t}) \\ &\mathsf{kapp}(x; (\delta a :: \overline{\kappa \rightarrow} \star, \mathsf{K}) \Rightarrow \star; \overline{a} \mid \overline{t}) = && \\ &\quad \overline{\forall d.}\ (\overline{y\langle\delta a\ \overline{\delta b}\rangle :: \mathsf{kapp}(y; (\overline{\delta b :: \kappa}) \Rightarrow \star; \mathsf{cs}(y; x; \overline{a} \mid \overline{d\ \overline{\delta b}}))}) && \\ &\qquad \Rightarrow \mathsf{kapp}(x; (\mathsf{K}) \Rightarrow \star; \overline{a} \mid \overline{[\delta a \mapsto d]t}). && \end{aligned}$$

For kind $\star$ types (without dependency variables), the resulting type is the type specified in the type signature, without the dependencies. The third case expresses the intuition that a generic function on a type that depends on a variable of kind $\overline{\kappa \to} \star$ introduces dependencies for the functions that x depends on, using kapp on that kind and the corresponding constraint substitution from the type signature. If δa is not of kind $\star$, then local dependency variables, $\overline{\delta b}$, are introduced in the dependency constraints and used in the recursive calls of kapp to reduce $\overline{\kappa \to} \star$ to the form $(\mathsf{K}) \Rightarrow \star$.

One important thing to observe is that with these definitions of tapp and kapp, the type specified by the user is indeed the special case for a dependency variable δa of kind $(\delta a :: \star) \Rightarrow \star$. Expanding the definition yields

$$\begin{aligned}
&\mathsf{tapp}(x; \delta a) \\
&\quad = \overline{\forall a.}\ \mathsf{kapp}(x; (\delta a :: \star) \Rightarrow \star; \overline{a} \mid \overline{\delta a}) \\
&\quad = \overline{\forall a.}\ \overline{\forall b.}\ (\overline{y\langle \delta a \rangle :: \mathsf{kapp}(y; \star; \mathsf{cs}(y; x; \overline{a} \mid \overline{b}))}) \Rightarrow \mathsf{kapp}(x; \star; \overline{a} \mid \overline{b}) \\
&\quad = \overline{\forall a.}\ \overline{\forall b.}\ (y\langle \delta a \rangle :: \mathsf{base}(y)\ (\mathsf{cs}(y; x; \overline{a} \mid \overline{b}))) \Rightarrow \mathsf{base}(x)\ (\overline{a} \mid \overline{b})
\end{aligned}$$

which, if one abstracts from the quantified variables, is exactly the part that can be found within the **generalize** construct.

As a concrete example, recall the type of generic equality from Section 2:

$$eq\langle a \rangle :: (\textbf{generalize}\ \langle \delta a \rangle \mapsto \Lambda b.(eq\langle \delta a \rangle :: b \to b \to Bool) \Rightarrow b \to b \to Bool)\ a.$$

Figure 4 contains some example values of the tapp function for this generalized type. The last case, involving a dependency variable of kind $\star \to \star$, is an example of a nested dependency constraint, making use of a local dependency variable argument.

$$\begin{array}{lll}
t :: \star & \mathsf{tapp}(eq; t) & = t \to t \to Bool \\
t :: \star \to \star & \mathsf{tapp}(eq; t\ \delta a) & = \forall a.(eq\langle \delta a \rangle :: a \to a \to Bool) \Rightarrow t\ a \to t\ a \to Bool \\
t :: \star \to (\star \to \star) & \mathsf{tapp}(eq; t\ \delta a\ \delta b) & = \forall a.\forall b.(eq\langle \delta a \rangle :: a \to a \to Bool, eq\langle \delta b \rangle :: b \to b \to Bool) \\
 & & \qquad \Rightarrow t\ a\ b \to t\ a\ b \to Bool \\
t :: (\star \to \star) \to \star & \mathsf{tapp}(eq; t\ \delta a) & = \forall a.(eq\langle \delta a\ \delta b \rangle :: \forall b.(eq\langle \delta b \rangle :: b \to b \to Bool) \\
 & & \qquad \Rightarrow a\ b \to a\ b \to Bool) \\
 & & \qquad \Rightarrow t\ a \to t\ a \to Bool
\end{array}$$

Figure 4: Example instances of tapp for generic equality

4.3 Type checking

Type judgments have the form $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: s$, where the environment V assigns types to variables and generic function names, whereas the other two environments support kind checking. The type judgments differ from a standard Hindley-Milner system mainly due to the handling of dependency constraints. As dependency constraints represent hidden arguments, and dependency constraints can be nested and contain universal quantifiers, we also have to deal with rank-n polymorphic types to some extent. We have drawn from Odersky and Läufer [28] to deal with this aspect.

$$\textsc{E-Var}\ \frac{x :: s \in \mathsf{V}}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash x :: s}$$

$$\textsc{E-Sub}\ \frac{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: s_1 \qquad \mathsf{A}\ \mathsf{K} \vdash s_1 \leqslant s_2}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: s_2} \qquad \textsc{E-Gen}\ \frac{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: s \qquad a \notin \mathsf{ftv}(\mathsf{V})}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: \forall a.s}$$

$$\textsc{E-App}\ \frac{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e_1 :: (\mathsf{D}) \Rightarrow (t_2 \to t_1) \qquad \mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e_2 :: (\mathsf{D}) \Rightarrow t_2}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash (e_1\ e_2) :: (\mathsf{D}) \Rightarrow t_1}$$

$$\textsc{E-Lambda}\ \frac{\mathsf{A}\ \mathsf{K}\ \ \mathsf{V}, x :: t_1 \vdash e :: (\mathsf{D}) \Rightarrow t_2}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \lambda x \to e :: (\mathsf{D}) \Rightarrow t_1 \to t_2}$$

$$\textsc{E-TApp}\ \frac{\begin{array}{c}\mathsf{A} \vdash s_1 :: \circ \\ x :: s_1 \in \mathsf{V} \qquad \mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \mathsf{tapp}(x; t) = s_2\end{array}}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash x\langle t \rangle :: s_2}$$

$$\textsc{E-Letrec}\ \frac{\mathsf{V}' \equiv \mathsf{V}, \overline{x :: s} \qquad \mathsf{A}\ \mathsf{K}\ \mathsf{V}' \vdash \overline{e :: s} \qquad \mathsf{A}\ \mathsf{K}\ \mathsf{V}' \vdash e_0 :: s_0}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \mathbf{letrec}\ \overline{x = e}\ \mathbf{in}\ e_0 :: s_0}$$

$$\textsc{E-Deplet}\ \frac{\begin{array}{c}\mathsf{K}' \equiv \mathsf{K}, \overline{\delta b :: \kappa} \qquad \mathsf{A} \vdash \delta a\ \overline{\delta b} :: (\mathsf{K}') \Rightarrow \star \\ \mathsf{A}\ \mathsf{K}'\ \mathsf{V} \vdash e_1 :: \overline{\forall a.}\ (\mathsf{D}_1, \mathsf{D}_2) \Rightarrow t' \qquad \overline{a \notin \mathsf{ftv}(\mathsf{D}_2)} \qquad \mathsf{A}\ \mathsf{K} \vdash (\mathsf{D}_3) \leqslant (\mathsf{D}_2) \\ \mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e_2 :: (x\langle \delta a\ \overline{\delta b} \rangle :: \overline{\forall a.}\ (\mathsf{D}_1) \Rightarrow t', \mathsf{D}_3) \Rightarrow t\end{array}}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \mathbf{deplet}\ x\langle \delta a\ \overline{\delta b} \rangle = e_1\ \mathbf{in}\ e_2 :: (\mathsf{D}_2) \Rightarrow t}$$

Figure 5: Type checking rules

$$\textsc{T-Inst-Refl}\ \frac{}{\mathsf{A}\ \mathsf{K} \vdash s \leqslant s}$$

$$\textsc{T-Inst-Dep}\ \frac{\mathsf{A}\ \mathsf{K} \vdash (\mathsf{D}_1) \leqslant (\mathsf{D}_2)}{\mathsf{A}\ \mathsf{K} \vdash (\mathsf{D}_1) \Rightarrow t \leqslant (\mathsf{D}_2) \Rightarrow t}$$

$$\textsc{T-Inst-Univ-1}\ \frac{\mathsf{A}\ \mathsf{K} \vdash [a \mapsto t]s_1 \leqslant s_2}{\mathsf{A}\ \mathsf{K} \vdash \forall a.s_1 \leqslant s_2}$$

$$\textsc{T-Inst-Univ-2}\ \frac{\mathsf{A}\ \mathsf{K} \vdash s_1 \leqslant s_2 \qquad a \notin \mathsf{ftv}(s_1)}{\mathsf{A}\ \mathsf{K} \vdash s_1 \leqslant \forall a.s_2}$$

$$\textsc{D-Inst-1}\ \frac{\mathsf{A} \vdash \delta a\ \overline{\delta b} :: (\overline{\delta b :: \kappa}, \mathsf{K}) \Rightarrow \star \qquad \mathsf{A} \vdash s :: (\mathsf{K}) \Rightarrow \star}{\mathsf{A}\ \mathsf{K} \vdash () \leqslant (\overline{x\langle \delta a\ \overline{\delta b} \rangle :: s})}$$

$$\textsc{D-Inst-2}\ \frac{\mathsf{A}\ \mathsf{K} \vdash s_2 \leqslant s_1 \qquad \mathsf{A}\ \mathsf{K} \vdash (\mathsf{D}_1) \leqslant (\mathsf{D}_2)}{\mathsf{A}\ \mathsf{K} \vdash (x\langle \delta a\ \overline{\delta b} \rangle :: s_1, \mathsf{D}_1) \leqslant (x\langle \delta a\ \overline{\delta b} \rangle :: s_2, \mathsf{D}_2)}$$

Figure 6: Instance relation on types and dependency constraints

The type rules E-VAR, E-GEN, E-APPL, and E-LAMBDA are pretty standard. Type schemes and thus dependency constraints are only introduced by the following constructs: a variable that has been bound in a **letrec**, or a type application of a generic function.

The **letrec** construct is the usual recursive let binding corresponding to **let** in Haskell. All variables bound in a **letrec** binding are assumed to be mutually dependent. Dependency constraints are treated analagously to class constraints or implicit parameters in that a value having dependencies may be bound to a variable in a **letrec**. This gives the feel of dynamically scoped dependency variables. For example, the program

$$\begin{array}{l}\textbf{letrec } x = \mathit{map}\langle \mathit{List}\ \delta a\rangle \textbf{ in}\\ \ (\textbf{deplet } \mathit{map}\langle\delta a\rangle = \lambda x \to x + 1 \textbf{ in } x\ [1,2,3],\\ \ \ \textbf{deplet } \mathit{map}\langle\delta a\rangle = \mathit{null}\ x \textbf{ in } x\ [\texttt{"one"},\texttt{"two"},\texttt{"three"}])\end{array}$$

has type $(\mathit{List}\ \mathit{Int}, \mathit{List}\ \mathit{Bool})$. We made this choice for the typing rule of **letrec** because it matches the idea that the dependencies are part of the type of a value. An alternative is discussed in Section 7.

By rule E-TAPP, a generic function is declared with a type signature of kind $\circ$. From the generic type signature and the type at which the generic function is applied, the result type of the expression is computed using the auxiliary function tapp, described above.

A **deplet** expression is used to satisfy dependency constraints arising from the use of a generic function in its body. The type s (without dependencies) of the expression e_1 which is used to satisfy the dependency must match the type occurring in the appropriate dependency constraint of the body expression e_2. As **deplet** is somewhat like locally defining an additional case for a generic function, we use local dependency variables as type arguments when the dependency which arises is not of kind $\star$. Thus, the argument variables $\overline{\delta b}$ in the rule E-DEPLET are local to the expression e_1. Finally, the remaining dependencies D_1 and D_2 of the bound expression and the body must be compatible as well.

As with kinds, we have a subtyping relation on types (Figure 6). The relation $s_1 \leqslant s_2$ expresses the intention that expressions of type s_1 are also of type s_2. The relation is exploited in the subsumption rule E-SUB. The rules in Figure 6 modify those of Odersky and Läufer to handle dependency constraints. Rule T-INST-DEP establishes a connection between the instance relation on type schemes and the instance relation on constraints. Dependency constraints represent hidden parameters. Unused, but well-kinded dependencies can be added to a type (D-INST-1). Furthermore, because they are arguments, dependencies become more general as the types in them become more specific (D-INST-2). This is analogous to the contravariance of the argument position of the function arrow.

The final rules are for type checking the well-formedness of generic definitions and programs. These are given in Figure 7. By P-GENERIC, a case of a generic function is type correct if the type of its right-hand side matches the type declared for the generic function. This is achieved using the function tapp to compute the type of the generic

$$\textsc{P-Generic}\ \frac{\dfrac{\mathsf{A}, a :: \bullet \vdash s :: () \Rightarrow \circ}{\mathsf{A} \vdash b\ \overline{\delta c} :: (\mathsf{K}) \Rightarrow \star} \qquad \dfrac{}{\mathsf{A\ K\ V} \vdash e :: \mathsf{tapp}(x; b\ \overline{\delta c})}}{\mathsf{A\ V} \vdash x\langle a \rangle :: s = \overline{\langle b\ \overline{\delta c} \rangle \mapsto e}\ \text{well-formed}}$$

$$\textsc{P-Prog}\ \frac{\dfrac{}{\mathsf{A\ V} \vdash x\langle a \rangle :: s = \overline{\langle b\ \overline{\delta c} \rangle \mapsto e}\ \text{well-formed}} \qquad \mathsf{A\ K\ V} \vdash e :: s'}{\mathsf{A\ V} \vdash x\langle a \rangle :: s = \overline{\langle b\ \overline{\delta c} \rangle \mapsto e}; e\ \text{well-formed}}$$

Figure 7: Program checking rules

function at the specific type of the case in question, using the type signature of the generic function. Finally, by P-PROG, a program in the core language is type correct if all cases of all generic functions are correct, and if the final expression is typeable. The environments A and V contain global datatypes and global functions, respectively.

5 Reduction semantics

Figure 8 presents the reduction rules for the core language. It remains implicit in the rules that bound variables should generally be renamed in such a way that no variables are captured during substitution. The style in which these rules are presented keeps **letrec** and **deplet** definitions around to act like environments for the expressions being evaluated. Reduction proceeds against a fixed environment S containing globally known values and the bindings for each case of each generic function that is declared in the program. The environment therefore contains entries of the form $x \mapsto e$ which map a variable to an expression and $x(\delta a\ \overline{\delta b}) \mapsto e$ which define a case of a generic function or a generic function's *specialization* to a specific type. The Generic Haskell compiler computes embedding and projections of all datatypes into types constructed from *Unit*, *Sum*, and *Prod*, and specializations for such types. This mechanism is orthogonal to the system described in this paper, and has been described elsewhere [14]. We assume that specializations are provided for all globally known type constants. (A more detailed analysis is possible that provides only the specializations that are really needed.) To be able to perform the reduction, we need to keep information about dependencies from the type checking process. The reason is that we have to know when it is safe to reduce a **deplet** or a lambda expression. Therefore, we assume that the domain of an expression's dependency constraint is accessible via the deps function. It is easy to verify that we can maintain this information during the reduction process. We also assume that the kinds of all type arguments in generic applications and the dependencies of all generic functions are available via functions arity, which gives the arity of a type, and dependencies, which gives the names of generic functions on which it depends. The function length denotes the length of a vector of variables.

The reduction rules R-APP and R-LETREC are standard, except for the fact that in

Variables and application

(R-GLOBAL) $x \rightsquigarrow e$
where $x \mapsto e \in \mathsf{S}$

(R-APP) $(\lambda x \to e_1)\ e_2 \rightsquigarrow [x \mapsto e_2]e_1$
where $\mathsf{deps}(e_2) = \emptyset$

letrec

(R-LETREC) $\textbf{letrec}\ x_0 = e_0; \overline{x = e}\ \textbf{in}\ e' \rightsquigarrow \textbf{letrec}\ x_0 = e_0; \overline{x = e}\ \textbf{in}\ [x_0 \mapsto e_0]e'$

(R-LETREC-ELIM-1) $\textbf{letrec}\ \overline{x = e}; \overline{y = e'}\ \textbf{in}\ e_0 \rightsquigarrow \textbf{letrec}\ \overline{y = e'}\ \textbf{in}\ e_0$
where $\overline{x} \cap \mathsf{fev}(\textbf{letrec}\ \overline{y = e'}\ \textbf{in}\ e_0) = \emptyset$

(R-LETREC-ELIM-2) $\textbf{letrec}\ \varepsilon\ \textbf{in}\ e_0 \rightsquigarrow e_0$

(R-LETREC-LAMBDA) $\textbf{letrec}\ \overline{x = e}\ \textbf{in}\ \lambda y \to e_0 \rightsquigarrow \lambda y \to \textbf{letrec}\ \overline{x = e}\ \textbf{in}\ e_0$

deplet

(R-DEPLET) $\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e\ \textbf{in}\ x\langle \delta a\ \overline{\delta b}\rangle \rightsquigarrow e$
where $\mathsf{fdv}(e) \setminus \{\overline{\delta b}\} = \emptyset$

(R-DEPLET-ELIM) $\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \textbf{in}\ e_2 \rightsquigarrow e_2$
where $x\langle \delta a\rangle \notin \mathsf{deps}(e_2)$

(R-DEPLET-LAMBDA) $\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \textbf{in}\ \lambda y \to e_2 \rightsquigarrow \lambda y \to \textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \textbf{in}\ e_2$

(R-DEPLET-APP) $\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \textbf{in}\ (e_2\ e_3) \rightsquigarrow (\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle\ \textbf{in}\ e_2)\ (\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle\ \textbf{in}\ e_3)$

(R-DEPLET-LETREC) $\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_0\ \textbf{in}\ (\textbf{letrec}\ \overline{x = e}\ \textbf{in}\ e') \rightsquigarrow \textbf{letrec}\ \overline{x = e}\ \textbf{in}\ (\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_0\ \textbf{in}\ e')$

(R-DEPLET-DEPLET) $\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_0\ \textbf{in}\ (\textbf{deplet}\ y\langle \delta a'\ \overline{\delta b'}\rangle = e_1\ \textbf{in}\ e_2) \rightsquigarrow \textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_0\ \textbf{in}\ (\textbf{deplet}\ y\langle \delta a'\ \overline{\delta b'}\rangle = (\textbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_0\ \textbf{in}\ e_1)\ \textbf{in}\ e_2)$

Type application

(R-CASE) $x\langle a\ \overline{\delta b}\rangle \rightsquigarrow e$
where $x\langle a\ \overline{\delta b}\rangle \mapsto e \in \mathsf{S}$

(R-TAPP) $x\langle t_0\ t_1\ \overline{\delta a}\rangle \rightsquigarrow \overline{\textbf{deplet}\ y\langle \delta a_1\ \overline{\delta b}\rangle = y\langle t_1\ \overline{\delta b}\rangle\ \textbf{in}}\ x\langle t_0\ \delta a_1\ \overline{\delta a}\rangle$
where $\delta a, \overline{\delta b}$ are fresh
$\mathsf{arity}(t_1) = \mathsf{length}(\overline{\delta b})$
$\mathsf{dependencies}(x) = \overline{y}$

Figure 8: Reduction rules

R-APP, all dependencies of the argument have to be resolved before the application can be reduced. This is because lambda-bound variables are always of simple types, not type schemes, and thus are dependency-free. A **letrec** is reduced by substitution in the underlying expression. A **letrec** can only be eliminated if the bound variables do not occur in its body anymore. Unlike **letrec**, a **deplet** construct is not reduced by performing substitution on its complete body. Rather, it is pushed down the expression, as shown for example in R-DEPLET-LETREC. A **deplet** is directly propagated to the body of a **letrec** here, not affecting the expressions in the bound variables. This implements the dynamically scoped behavior of dependency constraints. There are similar rules that push a **deplet** through applications or lambda abstractions.

The three rules R-CASE, R-DEPLET and R-TAPP handle various forms of generic function application. These depend on whether the form of the function's type argument is, respectively, a type constructor applied to some number of dependency variables, a dependency variable applied to some number of variables, or a type of some other form. By rule R-CASE a generic function applied to a type constructor with dependency arguments is reduced to the corresponding value in the environment. Rule R-DEPLET handles a local dependency declaration for the case where a dependency variable is in the constructor position, reducing to the value defined in the appropriate **deplet** clause. Finally, rule R-TAPP reduces the other cases, where the type appearing in an argument position is not a dependency variable. The idea is to transform away the rightmost of these, t_1, into dependencies on a new dependency variable δa_1 which replaces t_1 in the argument type. The dependencies introduced by attempting to reduce the generic function x are those which x depends upon. A consequence of applying this rule is that the type arguments in the resulting expression are simpler than the original; multiple applications of this rule eventually simplify all type arguments so that the rules R-CASE and R-DEPLET become applicable.

Expressions are reduced to values, where values are defined by the grammar

$$\begin{aligned} v ::= {} & \lambda x \to e \\ \mid {} & \textbf{letrec } \overline{x = e} \textbf{ in } v \end{aligned}$$

Values are lambda expressions, because there are no constants in our language. If the function body refers to local variables, surrounding let-bindings cannot always be eliminated. However, we can strengthen the condition and demand that each variable in the $\overline{x}$ occurs free in either the body or the other bound expressions.

Theorem 1 (Progress). *If e is a well-typed expression with $\mathsf{fev}(e) \in \mathsf{domain}(\mathsf{S})$ and $\mathsf{deps}(e) = \emptyset$, then e is either a value or there is an expression e' such that $e \rightsquigarrow e'$.*

Proof. By induction on the type derivation for e. We distinguish cases based on the final judgment:

Case E-VAR: If $e \equiv x$, then x is a global function (because $x \in \mathsf{V}$ and $x \in \mathsf{S}$), and R-GLOBAL applies.

Case E-SUB: Follows directly from the induction hypothesis.

Case E-APP: Thus $e \equiv (e_1\ e_2)$. If e_1 is not a value, then the induction hypothesis applies. If it is a value, then it contains a lambda expression, possibly within **letrec** statements by the grammar for values. If the lambda expression is nested, then R-LETREC-LAMBDA applies until the lambda is on the surface of the expression. Then we can apply R-APP, because $\mathsf{depvars}(e) = \emptyset$ implies $\mathsf{depvars}(e_2) = \emptyset$.

Case E-LAMBDA: This is a value.

Case E-GEN: Follows directly from the induction hypothesis.

Case E-TAPP: Because there are no dependencies, the type argument is of the form $a\ \bar{t}$. Either all the t are dependency variables—then R-CASE applies—or we can apply R-TAPP.

Case E-LETREC: Here, $e \equiv$ **letrec** $\overline{x = e}$ **in** e_0. If e_0 is a value, then so is e. If it is not a value, then the interesting question is whether $\mathsf{fev}(e_0) \cap \{\overline{x}\} = \emptyset$. If yes, then this implies $\mathsf{fev}(e_0) = \mathsf{fev}(e)$, and thus the induction hypothesis applies. If not, the R-LETREC is applicable.

Case E-DEPLET: Here, $e \equiv$ **deplet** $x\langle \delta a\ \overline{\delta b}\rangle = e_1$ **in** e_2. We will show inductively on the structure of an expression that we can always reduce this expression to a form where no deplet-binding occurs at the outside.

If $x\langle \delta a\rangle \notin \mathsf{deps}(e_2)$, then R-DEPLET-ELIM applies. Otherwise, we have to look closely at e_2. If e_2 is a variable y, then y is free in e and thus defined in S. Therefore, R-GLOBAL applies. If e_2 is an application, a lambda abstraction or a letrec binding, we can apply the appropriate rule that pushes the deplet down. If e_2 is another deplet, then the induction hypothesis applies.

Finally, e_2 might be a type application. The type argument must contain δa, otherwise $x\langle \delta a\rangle$ would not have been in $\mathsf{deps}(e_2)$. Thus one of R-DEPLET, R-CASE or R-TAPP is applicable. □

For the subject reduction theorem we need the following lemma:

Lemma 1. *If* $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash (\lambda x \to e) :: (\mathsf{D}) \Rightarrow t$*, then there is a derivation tree for this judgment where the last rule that is applied is* E-LAMBDA.

Proof. The only other rule that could be last is E-SUB. In this situation, the end of the derivation is E-LAMBDA, followed by one or more applications of E-GEN or E-SUB. Informally, E-SUB could be used here for one of two things: getting rid of universal quantification, or extending the set of dependency constraints. If E-SUB is used to remove universal quantifiers, then they must previously have been introduced by another application of E-SUB, or by E-GEN. Then these two applications can be canceled out. If E-SUB is used to extend the context of dependency constraints, then the application of this rule can be pushed down the derivation tree, so that it occurs before the E-LAMBDA. This informal argument can be formalized into another inductive proof, but we omit this proof here. □

Theorem 2 (Preservation/subject reduction). *If* $\mathsf{A\ K\ V} \vdash e :: s$*, and* $e \rightsquigarrow e'$ *is a reduction rule that can be applied to* e*, then* $\mathsf{A\ K\ V} \vdash e' :: s$*.*

Proof. We need to distinguish cases based on the reduction rule in question. Often, we assume without loss of generality that a certain rule has been applied last to derive $\mathsf{A\ K\ V} \vdash e :: s$: in many cases, these rules could have been followed by one or more applications of E-SUB or E-GEN. If we remove these rules, we have $\mathsf{A\ K\ V} \vdash e :: s'$, where the last rule applied is the rule in question, and we can change the type into s by multiple applications of subtyping or generalization. We then can apply this theorem to the modified typing judgment and get $\mathsf{A\ K\ V} \vdash e' :: s'$. Because subtyping and generalization only affect the type, not the expression, we can now apply the same subtyping and generality rules that we previously removed and have $\mathsf{A\ K\ V} \vdash e' :: s$, which proves the general case.

Case R-GLOBAL: We assume global definitions to be type correct.

Case R-APP: Here, $e \equiv ((\lambda x \to e_1)\ e_2)$. We know by E-APP that $s \equiv (\mathsf{D}) \Rightarrow t_1$ and that there is a t_2 such that

$$\mathsf{A\ K\ V} \vdash (\lambda x \to e_1) :: (\mathsf{D}) \Rightarrow t_2 \to t_1$$

and

$$\mathsf{A\ K\ V} \vdash e_2 :: (\mathsf{D}) \Rightarrow t_2.$$

Because $\mathsf{deps}(e_2) = \emptyset$, we know that even $\mathsf{A\ K\ V} \vdash e_2 :: t_2$ is derivable.

Using the lemma, we can assume that the last rule in the derivation of the first judgment is E-LAMBDA. Thus,

$$\mathsf{A\ K\ V}, x :: t_2 \vdash (\mathsf{D}) \Rightarrow t_1,$$

from which we immediately get that $\mathsf{A\ K\ V} \vdash [x \mapsto e_2]e_1 :: s$.

Case R-LETREC: This is immediately implied by E-LETREC.

Case R-LETREC-ELIM-1: This is also immediately implied by E-LETREC, as the unused variables can be eliminated from V'.

Case R-LETREC-ELIM-2: We have $e \equiv \mathbf{letrec}\ \varepsilon\ \mathbf{in}\ e_0$. From E-LETREC we learn that $\mathsf{V}' \equiv \mathsf{V}$ in this situation and therefore immediately that $\mathsf{A\ K\ V} \vdash e_0 :: s_0$.

Case R-LETREC-LAMBDA: Thus $e \equiv \mathbf{letrec}\ \overline{x = e}\ \mathbf{in}\ \lambda y \to e_0$. From E-LETREC we get that if $\mathsf{V}' \equiv \mathsf{V}, \overline{x :: s'}$ for a suitable vector of type schemes $\overline{s'}$, then

$$\mathsf{A\ K\ V}' \vdash \overline{e :: s'}$$

and

$$\mathsf{A\ K\ V}' \vdash (\lambda y \to e_0) :: s.$$

If we use the lemma, we can infer from E-LAMBDA that $s \equiv (\mathsf{D}) \Rightarrow t_1 \to t_2$ and

$$\mathsf{A}\ \mathsf{K}\ \ \mathsf{V}', y :: t_1 \vdash e_0 :: (\mathsf{D}) \Rightarrow t_2.$$

We can extend the variable environment and get

$$\mathsf{A}\ \mathsf{K}\ \ \mathsf{V}', y :: t_1 \vdash \overline{e :: s'}$$

(here we assume that y has been renamed so that it does not capture free variables in the let-bound expressions). Now we can apply E-LETREC again to the last two judgments, which yields

$$\mathsf{A}\ \mathsf{K}\ \ \mathsf{V}, y :: t_1 \vdash \mathbf{letrec}\ \overline{x = e}\ \mathbf{in}\ e_0 :: (\mathsf{D}) \Rightarrow t_2.$$

A final application of E-LAMBDA results in the proposition.

Case R-DEPLET: In this case, $e \equiv \mathbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_0\ \mathbf{in}\ x\langle \delta a\ \overline{\delta b}\rangle$. We can derive

$$\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e_0 :: \overline{\forall a.}\ (\mathsf{D}_2) \Rightarrow t'$$

and that

$$\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash x\langle \delta a\ \overline{\delta b}\rangle :: (x\langle \delta a\ \overline{\delta b}\rangle :: \overline{\forall a.}\ t', \mathsf{D}_3) \Rightarrow t$$

with $\mathsf{A}\ \mathsf{K} \vdash \overline{\forall a.}\ (\mathsf{D}_2) \Rightarrow t' \leqslant (\mathsf{D}_2) \Rightarrow t$. Here we use not only rule E-DEPLET, but also E-TAPP and knowledge about tapp.

Case R-DEPLET-ELIM: We have $e \equiv \mathbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \mathbf{in}\ e_2$ and we know that $x\langle \delta a\rangle \notin \mathsf{deps}(e_2)$. Therefore $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e_2 :: (\mathsf{D}_3) \Rightarrow s$ with $\mathsf{D}_3 \leqslant \mathsf{D}_2$ and $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: (\mathsf{D}_2) \Rightarrow s$. The claim follows from E-SUB.

Case R-DEPLET-LAMBDA: The situation is that $e \equiv \mathbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \mathbf{in}\ \lambda y \to e_2$, and because of E-DEPLET we know that

$$\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: (\mathsf{D}_2) \Rightarrow t$$

and

$$\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \lambda y \to e_2 :: (x\langle \delta a\ \overline{\delta b}\rangle :: s, \mathsf{D}_3) \Rightarrow t$$

with $\mathsf{A}\ \mathsf{K} \vdash \mathsf{D}_3 \leqslant \mathsf{D}_2$. From E-LAMBDA (using the lemma once more), we get that $t \equiv t_1 \to t_2$ and that

$$\mathsf{A}\ \mathsf{K}\ \mathsf{V}, y :: t_1 \vdash e_2 :: (x\langle \delta a\ \overline{\delta b}\rangle :: s, \mathsf{D}_3) \Rightarrow t_2.$$

From this we can conclude by E-DEPLET that

$$\mathsf{A}\ \mathsf{K}\ \mathsf{V}, y :: t_1 \vdash \mathbf{deplet}\ x\langle \delta a\ \overline{\delta b}\rangle = e_1\ \mathbf{in}\ e_2 :: (\mathsf{D}_2) \Rightarrow t_2.$$

Another application of E-LAMBDA yields the proposition.

Cases R-DEPLET-APP, R-DEPLET-LETREC, and R-DEPLET-DEPLET: These cases are analogous to the case R-DEPLET-LAMBDA. We need lemmas corresponding to the lemma for lambda abstraction: if a derivation exists for an application (or a **letrec**, or a **deplet**), then there is also a derivation tree where E-APP (or E-LETREC, or E-DEPLET) is the last judgment that is applied.

Case R-CASE: Because the case of the generic function has been checked to be well-formed by rule P-GENERIC, we know that if $x\langle a\ \overline{\delta b}\rangle \mapsto e \in \mathsf{S}$, then $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash e :: \mathsf{tapp}(x; a\ \overline{\delta b})$. Rule E-TAPP yields $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash x\langle a\ \overline{\delta b}\rangle :: \mathsf{tapp}(x; a\ \overline{\delta b})$.

Case R-TAPP: Let us first look at the type of $x\langle t_0\ t_1\ \overline{\delta a}\rangle$. We know from E-TAPP that $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash x\langle t_0\ t_1\ \overline{\delta a}\rangle :: \mathsf{tapp}(x; t_0\ t_1\ \overline{\delta a})$. On the other hand, we also know from E-TAPP that $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash x\langle t_0\ \delta a_1\ \overline{\delta a}\rangle :: \mathsf{tapp}(x; t_0\ \delta a_1\ \overline{\delta a})$. If $\mathsf{A} \vdash t_1 :: \kappa$, then $\mathsf{A} \vdash t_0\ \delta a_1\ \overline{\delta a} :: (\delta a_1 :: \kappa, \mathsf{K}) \Rightarrow \kappa'$ whereas $\mathsf{A} \vdash t_0\ t_1\ \overline{\delta a} :: (\mathsf{K}) \Rightarrow \kappa'$. Thus, we can apply the appropriate case in the definition of kapp and see that

$$
\begin{aligned}
&\mathsf{tapp}(x; t_0\ \delta a_1\ \overline{\delta a}) \\
&\quad\equiv \overline{\forall a.}\ \mathsf{kapp}(x; (\delta a_1 :: \kappa, \mathsf{K}) \Rightarrow \kappa'; \overline{a} \mid \overline{t_0\ \delta a_1\ \overline{\delta a}}) \\
&\quad\equiv \overline{\forall a.}\ \overline{\forall d.}\ (y\langle \delta a_1\ \overline{\delta b}\rangle :: \mathsf{kapp}(y; (\overline{\delta b :: \kappa}) \Rightarrow \star; \mathsf{cs}(y; x; \overline{a} \mid \overline{d\ \overline{\delta b}})))) \\
&\qquad\qquad \Rightarrow \mathsf{kapp}(x; \mathsf{K} \Rightarrow \kappa'; \overline{a} \mid \overline{t_0\ d\ \overline{\delta a}}) \\
&\quad\leqslant \overline{\forall a.}\ (y\langle \delta a_1\ \overline{\delta b}\rangle :: \mathsf{kapp}(y; (\overline{\delta b :: \kappa}) \Rightarrow \star; \mathsf{cs}(y; x; \overline{a} \mid \overline{t_1\ \overline{\delta b}}))) \\
&\qquad\qquad \Rightarrow \mathsf{kapp}(x; \mathsf{K} \Rightarrow \kappa'; \overline{a} \mid \overline{t_0\ t_1\ \overline{\delta a}})
\end{aligned}
$$

Here, $\mathsf{dependencies}(x) = \overline{y}$. In the last step we have instantiated the variables $\overline{d}$ according to the subtype relation on type schemes. We are now going to apply E-DEPLET multiple times for the **deplet** statements that are introduced by the reduction. Each of these is of the form

$$\mathbf{deplet}\ y\langle \delta a_1\ \overline{\delta b}\rangle = y\langle t_1\ \overline{\delta b}\rangle\ \mathbf{in} \ldots$$

By E-TAPP, we know that $\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash y\langle t_1\ \overline{\delta b}\rangle :: \mathsf{tapp}(y; t_1\ \overline{\delta b})$. Here,

$$
\begin{aligned}
&\mathsf{tapp}(y; t_1\ \overline{\delta b}) \\
&\quad\equiv \overline{\forall a.}\ \mathsf{kapp}(y; (\overline{\delta b :: \kappa}) \Rightarrow \star; \overline{a} \mid \overline{t_1\ \overline{\delta b}}) \\
&\quad\leqslant \mathsf{kapp}(y; (\overline{\delta b :: \kappa}) \Rightarrow \star; \mathsf{cs}(y; x; \overline{a} \mid \overline{t_1\ \overline{\delta b}}))
\end{aligned}
$$

We therefore indeed can apply R-DEPLET repeatedly to get

$$
\begin{aligned}
&\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \overline{\mathbf{deplet}\ y\langle \delta a_1\ \overline{\delta b}\rangle = y\langle t_1\ \overline{\delta b}\rangle\ \mathbf{in}}\ x\langle t_0\ \delta a_1\ \overline{\delta a}\rangle \\
&\qquad\qquad :: \mathsf{kapp}(x; \mathsf{K} \Rightarrow \kappa'; \overline{a} \mid \overline{t_0\ t_1\ \overline{\delta a}})
\end{aligned}
$$

or

$$\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \overline{\mathbf{deplet}\ y\langle \delta a_1\ \overline{\delta b}\rangle = y\langle t_1\ \overline{\delta b}\rangle\ \mathbf{in}}\ x\langle t_0\ \delta a_1\ \overline{\delta a}\rangle :: \mathsf{tapp}(x; t_0\ t_1\ \overline{\delta a})$$

by E-GEN. □

6 Related Work

The field of generic programming has grown considerably in recent years. Much of this work stems from so-called theories of datatypes, for example [3], which are often based on category theoretic notions such as initial algebras. These approaches focus on generic control constructs, such as folds [31], which are derived from datatypes.

Based on initial algebra semantics, Charity [7] automatically generates folds and so forth for datatypes on demand, but otherwise does not allow the programmer to write her own generic functions. PolyP [21] extends Haskell with a special construct for defining generic functions, also based on initial algebras. Although PolyP permits type inference and functions which make use of the pattern functor of a datatype, which we cannot do, it supports only regular datatypes, not mutually recursive, multiparameter, nested types, or types containing function spaces.

In Functorial ML [23] the algorithm for *map*, for example, is defined using combinators to find the data upon which argument function will apply. While supporting type inference, Functorial ML programming is a rather low level language and lacks the simplicity of Hinze's approach. Functorial ML and other work on shape theory has resulted in the programming language FISh [22].

Weirich and others [32] (and the earlier work on intensional polymorphism [8]) employ a **typecase** construct which performs run-time tests on types to implement polytypic functions of an expressiveness similar to Hinze's. Ruehr's structural polymorphism [30] adopts similar type tests. By avoiding type interpretation at run-time, Generic Haskell distinguishes itself from these approaches.

The work of Hinze [18, 16, 13], upon which Generic Haskell is based, is a major improvement over the other approaches, because it allows the instantiation of generic functions on mutually recursive, multiparameter, nested types, or types containing function spaces. Clarke and Löh [6] present a few extensions to Hinze's framework which are compatible with the framework presented here, and will be incorporated into the compiler for Dependency-style Generic Haskell.

Closest to the work described in the present paper are both Weirich's original higher-order intensional type analysis [32] and her recent type-erasure approach [33]. This approach offers the advantage of support for separate compilation, dynamic loading, and polymorphic recursion. In addition, by carrying around a representation of type information, she can successfully treat universal and existential quantification. The implementation of our approach requires no type information at run-time, enables separate compilation, and can certainly handle polymorphic recursion, as our cases must have a certain degree of polymorphism. Our calculus does however carry run-time information around, using it for dispatch purposes. The big difference between our system and Weirich's is that we handle the dependency of one generic on another, and treat generic functions by name, rather than as anonymous case statements.

Altenkirch and McBride [2] present a rather complicated encoding of much of our style of generic programming (excluding the dependency extension) into a system of dependent types. While this approach is highly expressive, we argue that there is a long way to go before it is usable by mere mortals as a programming language. We

expect that using type theory one can do anything we do in this paper and probably much more, but we suspect that programming the examples given in Section 2 will be very difficult.

The programming language Haskell supports **deriving** clauses for a certain number of built-in classes (*Eq*, *Show*, etc) [29]. This facilitates the automatic overloading of certain functions for datatypes which have the deriving clause specified. Hinze and Peyton Jones explore an extension, following Hinze's earlier ideas [15], which integrated generics with Haskell's type class system [20]. This system suffers from some limitations due to the interaction with the type class system. G'Caml [9, 10] presents a generic programming extension for O'Caml [26]. The proposal does not aim to cover all datatypes, and as such can be seen as a way of achieving Haskell-style overloading in O'Caml. The generic extension for Clean is also based on Hinze's work [1]. This proposal is more closely integrated with the type class system, but does not include any of the extensions described here. Generic Haskell, on the other hand, takes the approach of exploring generic programming in isolation, as an extension to the Haskell language.

Lämmel and Peyton Jones present a very neat idea which enables generic programs to be written in almost pure Haskell (Haskell with rank-2 types and a form of type coercion operator). We believe that our approach is more general. One reason supporting this is that our definitions are open, in that they can be extended or updated, whereas once a generic function has been constructed in Lämmel and Peyton Jones's proposal, its behavior is fixed. They also lack the ability for one generic function to depend on another generic function, as our dependencies permit. Furthermore, they generate generics only at kind $\star$. Their approach is based on combinators, whereas ours is based on functions which are defined indexed on the structure of types. In their favor, their approach is better integrated into Haskell.

A dependency resembles a qualified type: a type with restrictions that require certain type variables to be an instance of a particular class [24]. Inferring the context requirements for qualified types is, however, easier, because the dependencies for a generic function appear at multiple kinds, and thus have types with differing forms. Dependencies are similar in spirit to implicit parameters [27], and **deplet** is similar to the **with** construct (now an ordinary **let**) in that the functions referred to are not explicitly given as arguments. Implicit parameters are supplied at a fixed monomorphic type, whereas a dependency refers (often) to an entire generic function, which includes cases of polymorphic type.

The development of Generic Haskell is example driven. We encountered the need for dependencies when we implemented the digital searching and zipper examples in our work on type-indexed datatypes [19]. Most other approaches we have seen only use simple examples such as *map* and *zip* as their motivating examples.

7 Discussion

We have made design choices at several points when developing the language. We discuss two here. Firstly, distinguishing dependency variables from ordinary variables

has been useful in making the dependency concept explicit, although this distinction is not necessary. Secondly, we could have adopted an alternative formulation of **letrec** which implements a statically scoped view, as opposed to the more dynamic view taken in our system, using the following rule.

$$\frac{\mathsf{A}\ \mathsf{K}\ \mathsf{V}' \vdash \overline{e :: (\mathsf{D}) \Rightarrow t'} \qquad \mathsf{V}' \equiv \mathsf{V}, \overline{x :: t'} \qquad \mathsf{A}\ \mathsf{K}\ \mathsf{V}' \vdash e_0 :: (\mathsf{D}) \Rightarrow t}{\mathsf{A}\ \mathsf{K}\ \mathsf{V} \vdash \mathbf{letrec}\ \overline{x = e}\ \mathbf{in}\ e_0 :: (\mathsf{D}) \Rightarrow t}$$

Here the types for the let-bound variables are added to the environment without their dependencies. Thus the bound variables have a dependency-free type when used in the body. The dependencies of the bound variables as well as of the body must be compatible and become the common dependencies of the whole expression.

As presented here, generic functions are explicitly annotated for types of kind $\star$. Rank-n types appear, in conjunction with dependency constraints, in the types of instances of generic functions on particular types. It is easy to extend the language presented here to allow rank-n types everywhere. If we are given explicit type annotations for all rank-n types, we can infer the types of the expressions in the language, following along the lines of Odersky and Läufer [28], also adapting ideas from Lewis *et al* [27] and Chitil [4].

A prototype implementation of the system described in this paper including type inference exists, although we have not yet incorporated it into the Generic Haskell compiler. The prototype can be obtained by contacting the authors.

In conclusion, we have introduced a core language, a type system, and a reduction semantics for a new style of generic programming language which we have called Dependency-style Generic Haskell. The development of this language has been strongly motivated by practical concerns, derived from our experience programming in Generic Haskell. The difference with Classic Generic Haskell is significant, and the number of new concepts may seem overwhelming. However, we feel the added complexity is justified, as the system presented in this paper provides foundations which not only go beyond Classic Generic Haskell, but also opens the door to tackle problems which previously seemed too far off. Possibilities include better support for type-indexed types, higher-order and locally-defined generic functions, dependency inference, type inference of kind-$\star$ type arguments, generic functions based on kinds other than $\star$, and pattern matching on type arguments. Initial experiments have shown promising results in these areas. We feel that dependency inference, i.e. allowing the programmer to omit dependencies in generic type signatures, is our most pressing concern.

References

[1] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01*, pages 257–278, 2001.

[2] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Gibbons and Jeuring [11], pages 1–20.

[3] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[4] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP '01*, pages 193–204, September 2001.

[5] Dave Clarke, Johan Jeuring, and Andres Löh. The Generic Haskell User's Guide – Beryl release. Technical Report UU-CS-2002-047, Utrecht University, 2002.

[6] Dave Clarke and Andres Löh. Generic Haskell, specifically. In Gibbons and Jeuring [11], pages 21–48.

[7] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.

[8] Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98*, pages 301–312. ACM Press, 1998.

[9] Jun Furuse. G'Caml – O'Caml with extensional polymorphism extension. Project home page at `http://pauillac.inria.fr/~furuse/generics/`, 2001.

[10] Jun Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, January 2001.

[11] Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming*, volume 243 of *IFIP*. Kluwer Academic Publishers, January 2003.

[12] Paul Hagg. A framework for developing generic XML Tools. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.

[13] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, Technical report of Utrecht University, UU-CS-1999-28, 1999.

[14] Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University.

[15] Ralf Hinze. A new approach to generic functional programming. In *POPL'00*, pages 119–132. ACM Press, 2000.

[16] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.

[17] Ralf Hinze and Johan Jeuring. Generic Haskell: applications, 2003. To appear.

[18] Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory, 2003. To appear.

[19] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th Mathematics of Program Construction Conference, MPC'02*, volume 2386 of *LNCS*, pages 148–174, 2002.

[20] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001.

[21] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.

[22] C. B. Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2:307–315, 1999.

[23] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

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

[25] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, 2003.

[26] Xavier Leroy et al. *The Objective Caml system release 3.02, Documentation and user's manual*, December 2001. Available from `http://caml.inria.fr/ocaml/htmlman/`.

[27] Jeffrey Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit parameters: Dynamic scoping with static types. In *POPL '00*, pages 108–118, Jan 2000.

[28] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *POPL '96*, pages 54–67, New York, N.Y., 1996.

[29] Simon Peyton Jones, John Hughes, et al. Haskell 98 — A non-strict, purely functional language. Available from `http://haskell.org`, February 1999.

[30] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types*. PhD thesis, University of Michigan, 1992.

[31] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture FPCA '93*, pages 233–242. ACM Press, June 93.

[32] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *Programming Languages and Systems: 11th European Symposium on Programming, ESOP 2002*, 2002.

[33] Stephanie Weirich. Higher-order intensional type analysis in type erasure semantics. Available from `http://www.cis.upenn.edu/~sweirich/`, December 2002.