Simulating

ABSTRACT

Defining nontrivial class instances for irregular and exponential datatypes in Haskell is challenging, and as a solution it has been proposed to extend the language with quantified class constraints of the form $\forall a.\, C\, a \Rightarrow C'\, (f\, a)$ in the contexts of instance declarations. We show how to express the equivalent of such constraints in vanilla Haskell 98, but

their utility in this language is limited. We also present a more flexible solution, which relies on a widely-supported language extension.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Haskell, type classes*

General Terms

Languages, Design

Keywords

Dictionaries, polymorphic types, type classes

1. INTRODUCTION

Contexts in Haskell instance declarations constrain type variables appearing in the defined instance type. As an example (adapted from Ralf Hinze and Simon Peyton Jones [4]) consider the class of types with representation in binary:

$$\mathbf{data}\ \mathit{Bit} \ = \ \mathit{Zero} \mid \mathit{One}$$

$$\begin{array}{l}\mathbf{class}\ \mathit{Binary}\ a\ \mathbf{where} \\ \quad \mathit{showBin}\ ::\ a \to [\mathit{Bit}]\end{array}$$

$$\begin{array}{l}\mathbf{instance}\ \mathit{Binary}\ \mathit{Bit}\ \mathbf{where} \\ \quad \mathit{showBin}\ =\ (:[\,])\end{array}$$

An instance of *Binary* for lists could be defined as follows:

$$\begin{array}{l}\textbf{instance}\ \mathit{Binary}\,a \Rightarrow \mathit{Binary}\,[a]\ \textbf{where}\\ \quad \mathit{showBin} = \mathit{concat}\ .\ \mathit{map}\,\mathit{showBin}\end{array}$$

Haskell'03, August 25, 2003, Uppsala, Sweden.

Quantified Class Constraints

Valery Trifonov
Department of Computer Science
Yale University
New Haven, CT, USA 06520-8285
trifonov@cs.yale.edu

This instance declaration actually represents an "instance generator," or a proof that the type $[a]$ is an instance of *Binary* whenever a is; hence the type variable a can be thought of as universally quantified, and the instance declaration as a proof of $\forall a.\, \mathit{Binary}\, a \Rightarrow \mathit{Binary}\, [a]$, where the quantification over a is made explicit.

Explicit quantification is not allowed in Haskell contexts, and typically it is unnecessary, because usually the construction of only finitely many class instances must be ensured in order to type-check a Haskell expression. In these cases, as in the above example, the range of the quantifier would include both the class context of the declaration and the type being declared as an instance.

However the ability to express polymorphic recursion in Haskell (directly or in the guise of recursive instance declarations) introduces some cases when specifying class contexts with local quantifiers appears to be the only solution. One of these cases occurs when in order to type-check an instance declaration, we have to prove the existence of infinitely many instances, as in instances of the following types (an example due to David Feuer and Simon Peyton Jones [6], arising in the context of Chris Okasaki's use of irregular datatypes to represent square matrices in [8]):

$$\begin{array}{lcl}\textbf{newtype}\ \mathit{Two}\ f\ a & = & \mathit{Two}\,(f\,(f\,a)) \\ \textbf{data}\ \mathit{Sq}\ f\ a & = & M\ a\,(f\ a) \mid E\,(\mathit{Sq}\,(\mathit{Two}\ f)\ a)\end{array}$$

In order to define the binary representation for expressions of type $\mathit{Sq}\ f\ a$ in terms of their components, we need types a and $f\ a$ to be instances of Binary:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Binary}\, a,\ \mathit{Binary}\,(f\ a), \ldots) \Rightarrow \mathit{Binary}\,(\mathit{Sq}\ f\ a) \\ \quad \textbf{where}\end{array}$$

$$
\begin{array}{lcl}
\mathit{showBin}\,(M\;x\;xs) & = & \mathit{showBin}\;x \mathbin{+\!\!+} \mathit{showBin}\;xs \\
\mathit{showBin}\,(E\;p) & = & \mathit{showBin}\;p
\end{array}
$$

Let us ignore the fact that the constraint on $f\,a$ is not in Haskell 98 (allowing type expressions in class constraints hinders instance inference); several implementations support such constraints. This instance declaration is not yet complete—additional context, denoted by ellipsis, is needed for the case when the term is constructed with E: we need an instance of Binary for $\mathit{Sq}\,(\mathit{Two}\,f)\,a$, which is a substitution instance of the very type for which we are defining the current instance. The instance inference algorithm assumes the current declaration is already available, so we can instantiate the known thus far context of the declaration to find out that we need the instance $\mathit{Binary}\,(\mathit{Two}\,f\,a)$:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Binary}\,(f\,(f\,a)) \Rightarrow \mathit{Binary}\,(\mathit{Two}\,f\,a)\ \mathbf{where} \\
\quad \mathit{showBin}\,(\mathit{Two}\,x) \;=\; \mathit{showBin}\,x
\end{array}
$$

Now we run into the real problem in defining the instance $\mathit{Binary}\,(\mathit{Sq}\,f\,a)$: Since the constraint $\mathit{Binary}\,(f\,(f\,a))$ must be included in the ellipsis, the instantiation with $\mathit{Two}\,f$ for f in turn requires the instance $\mathit{Binary}\,(\mathit{Sq}\,(\mathit{Two}\,(\mathit{Two}\,f))\,a)$, hence the constraint $\mathit{Binary}\,(f\,(f\,(f\,(f\,a))))$ must be added, etc., ad infinitum[1]—meaning that no finite proof exists that an instance $\mathit{Binary}\,(\mathit{Sq}\,f\,a)$ can be constructed.

In other cases a finite number of instances would suffice for checking the instance declaration, but no finite instance construction is possible. This happens in the following example (adapted from Peyton Jones' message [5] and his paper with

Hinze [4]):

$$\textbf{data}\ \mathit{GRose}\ f\ a\ =\ \mathit{GBranch}\ a\ (f\ (\mathit{GRose}\ f\ a))$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Binary}\ a,\ \mathit{Binary}\ (f\ (\mathit{GRose}\ f\ a)))\ \text{-- illegal}\\ \qquad\Rightarrow\ \mathit{Binary}\ (\mathit{GRose}\ f\ a)\ \textbf{where}\\ \quad \mathit{showBin}\ (\mathit{GBranch}\ x\ \mathit{xs})\ =\ \mathit{showBin}\ x\ +\!\!+\ \mathit{showBin}\ \mathit{xs}\end{array}$$

The constraint $\mathit{Binary}\ (f\ (\mathit{GRose}\ f\ a))$, required by the second application of *showBin*, is not legal in Haskell 98. Implementations which allow it (in extensions) accept this declaration; however, except in degenerate cases for f, no instances $\mathit{Binary}\ (\mathit{GRose}\ f\ a)$ can actually be created. Consider the example of $f = [\,]$. To create an instance of *Binary* for $\mathit{GRose}\ [\,]\ a$, the compiler must first create one for $[\mathit{GRose}\ [\,]\ a]$; however according to the instance declaration for $\mathit{Binary}\ [a]$ it must first create an instance for $\mathit{GRose}\ [\,]\ a$, causing the instance generation to diverge.

The problems with generating an unbounded number of instances and with mutually dependent instances could be resolved if, instead of trying to describe them all, we could describe a *recipe* for creating them. Hinze and Peyton Jones observe this in [4], and point out that "no ordinary Haskell context will do" and that a solution would be to allow "polymorphic predicates" of the form

$$\mathit{ctx}\ ::=\ \forall\overline{a}.\,(\mathit{ctx}_1,\ \ldots,\ \mathit{ctx}_n)\ \Rightarrow\ C\ t$$

where C is a class name and t is a type, in instance contexts. (We use the term "quantified constraints" instead of "polymorphic predicates.") In the above examples the necessary constraint is

$$\forall a.\, \mathit{Binary}\, a \;\Rightarrow\; \mathit{Binary}\,(f\, a)$$

Thus the instance declaration for GRose takes the form

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Binary}\, a, \forall b.\, \mathit{Binary}\, b \;\Rightarrow\; \mathit{Binary}\,(f\, b)) \\
\qquad\quad \Rightarrow\; \mathit{Binary}\,(\mathit{GRose}\, f\, a)\ \textbf{where} \\
\quad \mathit{showBin}\,(\mathit{GBranch}\, x\, \mathit{xs}) \;=\; \mathit{showBin}\, x \mathbin{+\!\!+} \mathit{showBin}\, \mathit{xs}
\end{array}$$

Now the required instance for $f\,(\mathit{GRose}\, f\, a)$ can be constructed by instantiating the quantified constraint with the type $\mathit{GRose}\, f\, a$ and applying the result to the current instance. Similarly the instance $\mathit{Binary}\,(f\, a)$, needed in the declaration of $\mathit{Binary}\,(\mathit{Sq}\ f\, a)$ in the earlier example, can be constructed from $\mathit{Binary}\, a$ (instead of required in the context), thus cutting the infinite chain of required instances.

In this paper we show that Haskell's constructor classes offer a way to express the equivalent of quantified constraints in vanilla Haskell 98. The full compliance with the language comes at the price of non-local flow-based program transformations, which limit its scope of applicability. We also show how to achieve closer simulation of the uses of quantified constraints when programming with the widely-supported lan-

[1] In contrast, other uses of polymorphic recursion require a statically unbounded number of instances to be constructed at run time, but only a finite number of class constraints, so they are correct programs in Haskell 98.

guage extension with variable constructor heads in declared instance types. While not covering the range of applications targeted by the proposal for direct language support, these solutions can be applied successfully to problems which have received attention in the community [4, 5, 6, 7, 10], and yield programs in supported Haskell.

2. DICTIONARIES AND TYPE EQUIVALENCES

A quantified constraint $\forall a.\, C\, a \;\Rightarrow\; C\,(f\, a)$ indicates the requirement that an "instance generator" for class C is available for type f. Our goal is thus to allow the context of one instance generator (e.g. for the instance $\mathit{Binary}\,(\mathit{GRose}\, f\, a))$ to request the existence of another instance generator (e.g. that for $\forall b.\, \mathit{Binary}\, b \;\Rightarrow\; \mathit{Binary}\,(f\, b)$), whereas Haskell only allows instances to be requested. With this observation, let us look at the semantics of instances and generators and try to find a correspondence.

The standard semantics of Haskell type classes is given by translating the language into one with explicit passing of *dictionaries* [9] — records containing the methods of the required instances. Thus a class declaration

$$\textbf{class}\; C\, a\; \textbf{where}\; \overline{m_i \;::\; \mathit{cty}_i}$$

gives rise to a type constructor $C \;=\; \lambda a :: \kappa.\, \{\overline{m_i :: [\![\mathit{cty}_i]\!]_a}\}$ in a standard extension of F_ω [2] with records (tuples):

kinds $\quad \kappa ::= * \mid \kappa \;\rightarrow\; \kappa'$

$$\begin{array}{lrcl} \text{types} & \tau & ::= & a \mid \lambda a :: \kappa.\, \tau \mid \tau\, \tau' \mid \tau \rightarrow \tau' \mid \forall a :: \kappa.\, \tau \\ & & \mid & \{\overline{m_i :: \tau_i}\} \mid \ldots \\ \text{terms} & e & ::= & x \mid \lambda x :: \tau.\, e \mid e\, e' \mid \Lambda a :: \kappa.\, e \mid e\, [\tau] \\ & & \mid & \{\overline{m_i \;=\; e_i}\} \mid e.m \mid \ldots \end{array}$$

where x ranges over term variables, a and b range over type variables, and m ranges over labels.[2] The types $[\![cty_i]\!]_a$ are the translations of cty_i, which are Haskell types with contexts; the most important feature of the translation is that it turns class contexts into types of dictionaries as arguments, and quantifies over all free type variables other than a:

$$[\![(\overline{C_j\ (a_j\ \overline{t_j})}) \Rightarrow t]\!]_a \quad = \quad \forall \overline{b}.\, \overline{C_j\ (a_j\ \overline{t_j})} \rightarrow t$$

where $\overline{b}$ is a sequence of all type variables in the set $\{\overline{a_j}\} \cup FV(\overline{t_j}) \cup FV(t) - \{a\}$, i.e. all type variables free in the type (including the contexts) but a. Note that the metavariable t ranges over simple Haskell types; we gloss over the details of their translation by assuming they are a subset of the target language types τ.

An "instance generator" declaration, of the form

$$\textbf{instance}\ (\overline{C_j\ a_j}) \Rightarrow C\, \tau\ \textbf{where}\ \overline{m_i \;=\; e_i}$$

can be translated as a "dictionary generator" term

$$dg \;=\; \Lambda \overline{b}.\, \lambda \overline{d_j :: C_j\ a_j}.\, \{\overline{m_i \;=\; [\![e_i]\!]^{d_j}_{C_j\ a_j}}\}$$

[2] We use an overloaded notation $\overline{A}$ for sequences of terms of the syntactic category ranged over by A: the separators between the terms in the sequence should be inferred from the

context. If each of the terms A has component subterms we need to refer to, they are all indexed with the same subscript; in some cases these subterms are themselves sequences.

where the translation $[\![\cdot]\!]^{d_j}_{C_j\, a_j}$ replaces all uses of the instances $C_j\, a_j$ by operations on the corresponding dictionaries d_j. Details of this translation are omitted, because they are not important for us at this point; important is the type of the term dg:

$$dg \;::\; \forall\overline{b}.\, \overline{C_j\, a_j} \;\rightarrow\; \{\overline{m_i :: \tau_i}\}$$

Thus, in terms of this translation, our goal is to make dictionary generators like dg take parameters of the type of dg. However the translation obviously only allows dictionaries as parameters of dictionary generators.

But perhaps it is possible to have a dictionary parameter whose type is *isomorphic* to the type of a generator?

The difference between dictionaries and dictionary generators is that the former are records of values, while the latter are polymorphic functions producing records of values. However there are well-known isomorphisms which we can use to construct maps between the two types, namely the distributivity laws

$$\begin{array}{rcl} \tau \;\rightarrow\; \{\overline{m_i :: \tau_i}\} & \leftrightarrow & \{\overline{m_i :: \tau \;\rightarrow\; \tau_i}\} \\ \forall a :: \kappa.\, \{\overline{m_i :: \tau_i}\} & \leftrightarrow & \{\overline{m_i :: \forall a :: \kappa.\, \tau_i}\} \end{array}$$

So we have

$$\forall\overline{a}.\, \overline{C_j\, a_j} \;\rightarrow\; \{\overline{m_i :: \tau_i}\} \;\leftrightarrow\; \{\overline{m_i :: \forall\overline{a}.\, \overline{C_j\, a_j} \;\rightarrow\; \tau_i}\}$$

This is a result in our variant of F_ω, but not in Haskell yet—not all F_ω types can be represented in Haskell. In particular, the argument types we must push under the record type constructor correspond to dictionaries, and hence to contexts in Haskell—that is, they cannot be represented as parameters of Haskell functions. Luckily, however, methods in Haskell 98 can have *local contexts* in addition to the context of the instance declaration, and the prenex universal quantification on method types corresponds exactly to the quantification in the type on the right hand side.

3. A REPRESENTATION IN HASKELL 98

Returning to Haskell, suppose we have a class declaration

$$\textbf{class}\ C\, a\ \textbf{where}\ \overline{m_i\ ::\ \overline{ctx_i}\ \Rightarrow\ t_i},$$

and in the context of some instance declaration we need the quantified constraint $\forall b.\, \overline{ctx'}\ \Rightarrow\ C\,(f\,t)$, where the type variable b appears in $\overline{ctx'}$ and the type t:

$$\begin{array}{l}\textbf{instance}\ (\forall b.\, \overline{ctx'}\ \Rightarrow\ C\,(f\,t),\, \overline{ctx''})\ \Rightarrow\ C'\,t'\ \textbf{where} \\ \quad \overline{m'_j\ =\ e_j}\end{array}$$

We introduce a "functorial class" C_f declared as

$$\textbf{class}\ C\_f\ f\ \textbf{where}\ \overline{m\_f_i\ ::\ \forall b.\, [f\,t/a]((\overline{ctx'},\, ctx_i)\ \Rightarrow\ t_i)}$$

where $[t/a]cty$ denotes the type obtained by substituting t for a in cty; the quantification over b is implicit in Haskell 98 but shown here for emphasis, while other implicitly quantified variables are not shown. Then we use the constraint $C\_f\ f$ instead of the desired quantified constraint, and we use the method names m_f_i instead of m_i in the expressions in the dynamic scope of this constraint, e.g.

$$\begin{array}{l}\textbf{instance}\ (C\_f\ f,\ \overline{ctx''})\ \Rightarrow\ C'\ t'\ \textbf{where} \\ \quad \overline{m'_j\ =\ [m\_f_i/m_i]e_j}\end{array}$$

This syntactic transformation is in general non-local: The requirement to cover the dynamic scope of the constraint implies that all overloaded functions with the constraint $C\ a$ in the type, instantiated with $f\ t$ for a in applications reachable from e_j, must be cloned, and the names of these functions and m_i substituted by their clones' names in the cloned code.

We also provide instances of the form

$$\textbf{instance}\ C\_f\ T\ \textbf{where}\ \overline{m\_f_i\ =\ m_i}$$

for each type constructor T for which we would need an instance of the desired quantified constraint. The methods in these instances are (modulo the type isomorphisms of Section 2) essentially trampolines to the defined as usual methods in instances of C for applications of T.

Although we omit the kind specifications of type variables for brevity, it should be clear that this transformation is valid for arbitrary consistent kinds; however different "functorial classes" must be provided for type constructors of different kinds. Since this scheme supports the cases when the type t in the quantified constraint $\forall b.\ \overline{ctx'}\ \Rightarrow\ C\ (f\ t)$ is not simply the variable b, if the constraints $\forall b.\ ctx_1\ \Rightarrow\ C\ (f_1\ t_1)$

and $\forall b.\, ctx_2 \;\Rightarrow\; C\,(f_2\,t_2)$ are both needed in instance declarations, and $t_1 \neq t_2$, we would have different "functorial classes" for them; this is also the case in particular when the kinds of t_1 and t_2 (hence of f_1 and f_2) are different.

In the example of the *GRose* type in the introduction, Hinze and Peyton Jones suggest the use of the quantified constraint $\forall a.\, Binary\,a \;\Rightarrow\; Binary\,(f\,a)$ to define an instance of *Binary*. We instead declare the class

$$\begin{array}{l} \textbf{class}\ Binary\_f\ f\ \textbf{where} \\ \quad showBin\_f\ ::\ Binary\,a \Rightarrow\ f\,a\ \rightarrow\ [Bit] \end{array}$$

Then an instance of *Binary* can be constructed for *GRose* as follows:

$$\begin{array}{l} \textbf{instance}\ (Binary\,a,\ Binary\_f\ f) \\ \qquad\quad \Rightarrow\ Binary\,(GRose\,f\,a)\ \textbf{where} \\ \quad showBin\,(GBranch\,x\,xs)\ = \\ \qquad\quad showBin\,x\ +\!\!+\ showBin\_f\ xs \end{array} \tag{1}$$

Additionally, for the construction of *Binary* instances for $GRose\,[\,]\,Bit$ we need also the declaration

$$\begin{array}{c} \textbf{instance}\ Binary\_f\ [\,]\ \textbf{where} \\ showBin\_f\ =\ showBin \end{array}$$

assuming we already have the instance $Binary\,[a]$, shown in the introduction.

The simplicity of the auxiliary declarations is due to the type inference and dictionary conversion, which automatically insert the type and dictionary applications. As an illustration, the translation of the above code into a variant of F_ω is shown in Figure 1; the calculus is enriched with pattern matching on function arguments and a fixpoint expression $\mathbf{rec}\, x :: \tau = e$ to allow the translation of recursive instances, and we assume the standard definitions of *List*, *concat*, *map*, *append*, and *Bit* are available. Note that the definition of *Binary_f_List* implements half of the isomorphism between $\mathit{Binary\_f}\ f$ and $\forall a.\, \mathit{Binary}\, a \to \mathit{Binary}\,(f\, a)$, while the other half is inlined in the last three lines of the figure and evident in the order of the selection from and applications of d_f. (An implementation based on Hinze and Peyton Jones' proposal would just avoid these shuffles.)

This approach, however, is limited by the non-local aspects of the transformation. To apply it, we must be able to locate and clone statically all functions which are invoked from the translated instance declaration and have types with the constraints we are replacing. Since Haskell 98 does not

$$
\begin{array}{l}
\mathit{Binary} :: * \to * \\
\quad = \lambda a :: *.\, \{\mathit{showBin} :: a \to \mathit{List}\ \mathit{Bit}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Binary\_f} :: (* \to *) \to * \\
\quad = \lambda f :: * \to *. \\
\qquad \{\mathit{showBin\_f} :: \forall a.\, \mathit{Binary}\, a \to f\, a \to \mathit{List}\ \mathit{Bit}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Binary\_List} :: \forall a :: *.\, \mathit{Binary}\, a \to \mathit{Binary}\,(\mathit{List}\, a) \\
\quad = \Lambda a :: *.\, \lambda d_a :: \mathit{Binary}\, a. \\
\qquad \{\mathit{showBin} = \lambda xs :: \mathit{List}\, a. \\
\qquad\quad \mathit{concat}\,[\mathit{Bit}] \\
\qquad\qquad (\mathit{map}\,[a]\,[\mathit{List}\ \mathit{Bit}]\,(d_a.\mathit{showBin})\, xs)\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Binary\_f\_List} :: \mathit{Binary\_f}\ \mathit{List} \\
\quad = \{\mathit{showBin\_f} \ = \ \Lambda a :: *.\, \lambda d_a :: \mathit{Binary}\, a. \\
\qquad\qquad (\mathit{Binary\_List}\, [a]\, d_a).\mathit{showBin}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Binary\_GRose\_type} :: * \\
\quad = \ \forall a. \forall f.\, \mathit{Binary}\, a \ \rightarrow \ \mathit{Binary\_f}\ f \ \rightarrow \ \mathit{Binary}\, (f\, a)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Binary\_GRose} :: \mathit{Binary\_GRose\_type} \\
\quad = \mathbf{rec}\ d :: \mathit{Binary\_GRose\_type} \\
\qquad\qquad = \ \Lambda a.\, \Lambda f.\, \lambda d_a.\, \lambda d_f. \\
\qquad\qquad\quad \{\mathit{showBin} \ = \\
\qquad\qquad\qquad \lambda(\mathit{GBranch}\, (x :: a)\, (\mathit{xs} :: f\ (\mathit{GRose}\ f\ a))). \\
\qquad\qquad\qquad\quad \mathit{append}\, [\mathit{Bit}] \\
\qquad\qquad\qquad\qquad (d_a.\mathit{showBin}\, x) \\
\qquad\qquad\qquad\qquad (d_f.\mathit{showBin\_f}\ [\mathit{GRose}\ f\ a] \\
\qquad\qquad\qquad\qquad\qquad\qquad (d\, [a]\, [f]\, d_a\ d_f) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{xs})\}
\end{array}
$$

Figure 1: Translation of instances of *Binary*.

allow constraints to be nested in types, it may appear that these functions are not first class, hence their invocations are always direct and their reachability can be determined statically. This is not the case, because these functions may be methods of another class; then their types may contain constraints,[3] and their invocations are not only indirect—they are invisible in the Haskell code. Consider the types

Sq and Two, introduced earlier. Defining an instance of $Binary$ for Sq is now straightforward by replacing the quantified constraint on f with $Binary\_f\ f$. We must then define an instance of $Binary\_f$ for $Two\ f$ under the assumption of $Binary\_f\ f$. However, this is impossible, because (following the algorithm) we need to replace with $Binary\_f\ f$ the constraint $Binary\ a$ in the type of $showBin\_f$ in the assumed instance $Binary\_f\ f$, which cannot be determined statically.

4. A MORE FLEXIBLE APPROACH

Suppose we also need an instance of $Binary$ for the type $GRose\ (GRose\ [\,])\ Bit$. To satisfy the constraints in declaration (1), we have to declare an instance of $Binary\_f$ for $GRose\ [\,]$. Naturally we can obtain it from the more general

$$\begin{array}{l}\textbf{instance}\ Binary\_f\ f\ \Rightarrow\ Binary\_f\ (GRose\ f)\ \textbf{where}\\ \quad showBin\_f\ =\ showBin\end{array}$$

Just as in the case of lists above, this definition exploits the existence of an instance of $Binary$ for $GRose\ f\ a$. However we have to provide these (trivial) declarations, each defining $showBin\_f$ in terms of $showBin$, for each type constructor required to satisfy the quantified constraint encoded by $Binary\_f$, and as we showed they introduce a major weakness, because their invocations cannot be replaced statically.

An alternative is to define $showBin$ in terms of $showBin\_f$

---

[3] Ironically this is exactly the Haskell feature that made possible the approach in the first place.

(to illustrate this we have to ignore the code shown above, including and following (1), as well as the earlier instance declaration for $\mathit{Binary}\,[a]$). It turns out that a single declaration suffices:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Binary}\,a,\ \mathit{Binary\_f}\ f)\ \Rightarrow\ \mathit{Binary}\,(f\,a)\ \textbf{where}\\
\quad \mathit{showBin}\ =\ \mathit{showBin\_f}
\end{array}
$$

Unfortunately, due to the type variable f in the head of the instance type, this declaration is not in Haskell 98; however, at least two implementations support extensions allowing such declarations. The list type constructor is now handled by one additional declaration:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Binary\_f}\,[\,]\ \textbf{where}\\
\quad \mathit{showBin\_f}\ =\ \mathit{concat}\ .\ \mathit{map}\,\mathit{showBin}
\end{array}
$$

An analogous declaration would do it for GRose, but its kind suggests that a more general definition is useful:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Binary\_f3}\ (g :: (* \to *) \to * \to *)\ \textbf{where}\\
\quad \mathit{showBin\_f3}\ ::\ (\mathit{Binary}\,a,\ \mathit{Binary\_f}\ f)\ \Rightarrow\ g\,f\,a\ \to\ [\mathit{Bit}]\\[1ex]
\textbf{instance}\ (\mathit{Binary\_f}\ (f :: * \to *),\ \mathit{Binary\_f3}\ g)\\
\qquad\quad \Rightarrow\ \mathit{Binary\_f}\ (g\ f)\ \textbf{where}\\
\quad \mathit{showBin\_f}\ =\ \mathit{showBin\_f3}\\[1ex]
\textbf{instance}\ \mathit{Binary\_f3}\ \mathit{GRose}\ \textbf{where}\\
\quad \mathit{showBin\_f3}\,(\mathit{GBranch}\,x\,\mathit{xs})\ =\ \mathit{showBin}\,x\ +\!\!+\ \mathit{showBin}\,\mathit{xs}
\end{array}
$$

The kind annotations are shown for clarity, but they are inferred unambiguously. The strong similarity between the instance declarations for $\mathit{Binary\_f}\ (g\ f)$ and $\mathit{Binary}\ (f\ a)$, as well as those for other function kinds, cannot be taken advantage of in Haskell, because they refer to classes with different (names and) types of methods.[4] On the other hand we only have to define one class and one instance for every kind of type constructor for which we need instances of *Binary*, and its subkinds (i.e. syntactic subterms of the kind expression), and in a typical Haskell program their number is very small.

With this approach the type Sq, shown in the introduction, is just as easy to handle:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Binary\_f3}\ \mathit{Two}\ \textbf{where} \\
\quad \mathit{showBin\_f3}\ (\mathit{Two}\ x)\ =\ \mathit{showBin}\ x
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Binary\_f3}\ \mathit{Sq}\ \textbf{where} \\
\quad \mathit{showBin\_f3}\ (M\ x\ \mathit{xs})\ =\ \mathit{showBin}\ x\ +\!\!+\ \mathit{showBin}\ \mathit{xs} \\
\quad \mathit{showBin\_f3}\ (E\ s)\qquad\ =\ \mathit{showBin}\ s
\end{array}
$$

In another example, that of an "exponential" type

$$\textbf{data}\ T\ f\ a\ =\ A\ a\ (f\ a)\ \mid\ T\ (T\ f\ (T\ f\ a))$$

the encoding works together with Haskell's recursive instances:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Binary\_f3}\ T\ \textbf{where} \\
\quad \mathit{showBin\_f3}\ (A\ x\ \mathit{xs})\ =\ \mathit{showBin}\ x\ +\!\!+\ \mathit{showBin}\ \mathit{xs} \\
\quad \mathit{showBin\_f3}\ (T\ u)\qquad\ =\ \mathit{showBin}\ u
\end{array}
$$

The syntax of quantified constraints allows for an empty list of premises, as in for instance $\forall a.\ C\ (f\ a)$. A case when

[4] An extension of Clean which allows sharing the code for such instances is presented in [1]; it can be supported by compiling to the language of [3].

this sort of constraint is useful was demonstrated by Ashley Yakeley in [10]: The class of bifunctors

$$\begin{array}{l}\textbf{class}\ \mathit{Bifunctor}\ f\ \textbf{where}\\ \quad \mathit{bimap} :: (a \to a') \to (b \to b') \to f\,a'\,b \to f\,a\,b'\end{array}$$

can be synthesized from the classes of functors and cofunctors:

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\ \text{-- standard}\\ \quad \mathit{fmap} :: (a \to b) \to f\,a \to f\,b\\ \textbf{class}\ \mathit{Cofunctor2}\ f\ \textbf{where}\\ \quad \mathit{comap2} :: (a \to a') \to f\,a'\,b \to f\,a\,b\end{array}$$

if one could write the instance declaration

$$\begin{array}{l}\textbf{instance}\ (\mathit{Cofunctor2}\ f, \forall a.\, \mathit{Functor}\,(f\,a))\\ \qquad \Rightarrow \mathit{Bifunctor}\ f\ \textbf{where}\\ \quad \mathit{bimap}\,\mathit{fa}\,\mathit{fb} = \mathit{comap2}\,\mathit{fa}\ .\ \mathit{fmap}\,\mathit{fb}\end{array}$$

Following the approach, we write instead

$$\begin{array}{l}\textbf{class}\ \mathit{Functor\_f}\ f\ \textbf{where}\\ \quad \mathit{fmap\_f} :: (a \to a') \to f\,b\,a \to f\,b\,a'\\ \textbf{instance}\ \mathit{Functor\_f}\ f \Rightarrow \mathit{Functor}\,(f\,a)\ \textbf{where}\\ \quad \mathit{fmap} = \mathit{fmap\_f}\end{array}$$

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Cofunctor2}\ f,\ \mathit{Functor\_f}\ f) \\
&\qquad \Rightarrow\ \mathit{Bifunctor}\ f\ \textbf{where} \\
&\quad \mathit{bimap}\ \mathit{fa}\ \mathit{fb}\ =\ \mathit{comap2}\ \mathit{fa}\ .\ \mathit{fmap}\ \mathit{fb}
\end{aligned}
$$

which completes a program valid in Haskell with extensions for variable head instances and overlapping instances.

5. RELATED WORK

Ralf Hinze and Simon Peyton Jones describe in [4] the utility of quantified constraints in the context of automatic derivation of instances. They propose extending the language with quantified constraints, and provide semantics for the extension. Artem Alimarine and Rinus Plasmejier [1] present extensions to Clean, which allow the use of induction on the structure of kinds in the definition of classes and instances in the style of [3].

In contrast the simulations outlined in our paper are not intended as a substitute for a language extension for the purpose of providing compiler support for other features (for example automatic instance derivation), although our second approach can be used as a basis for a preprocessor. Our goal is to offer a solution for problems involving a limited set of kinds, for which it is feasible to code the required class and instance declarations. Such problems, requiring quantified constraints, have been discussed multiple times on the Haskell mailing lists in recent years. Conor McBride [7] has independently outlined the essence of the solution presented here; unfortunately subsequent discussions on the same topic indicate that his description was not interpreted to suggest a solution within the existing language.

6. CONCLUSION

Of the two presented approaches to simulating quantified constraints, the first has the advantages that it can be used in Haskell 98, and it does not require changes in the way instances of the original classes are constructed. However its dependence on the ability to perform (a restricted form of) flow analysis of the program prevents it from handling some cases of irregular types. The second approach requires an extension of Haskell allowing a type variable in the head of an instance declaration, and forces some changes in the style of coding of instances; in return it is much more flexible. While not a substitute for a language extension, the second approach appears quite useful in solving typical problems involving quantified constraints.

7. ACKNOWLEDGMENTS

Thanks to Paul Hudak and the anonymous referees for their suggestions on improving the presentation and the technical content.

This research was supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-0208618, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the author and do not reflect the views of these agencies.

8. REFERENCES

[1] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In *Implementation of Functional Languages, 13th International Workshop (IFL 2001)*, volume 2312 of *LNCS*, pages 168–186, Stockholm, Sweden, September 2001. Springer.

[2] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur*. PhD thesis, University of Paris VII, 1972.

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

[4] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *Electronic Notes in Theoretical Computer Science*, Montréal, Canada, September 2000. Elsevier. http://www.elsevier.nl/inca/publications/store/5/0/5/6/2/5/.

[5] Simon Peyton Jones. Re: Rank-2 polymorphism & type inference, December 2000. Message posted on the Haskell mailing list, available at http://haskell.cs.yale.edu/pipermail/haskell/2000-December/006303.html.

[6] Simon Peyton Jones. Deriving excitement, February 2002. Message posted on the Haskell mailing list, available at http://haskell.cs.yale.edu/pipermail/haskell/2002-February/005426.html.

[7] Conor T. McBride. Re: higher-kind deriving…or not, February 2002. Message posted on the Haskell-café mailing list, available at http://www.haskell.org/pipermail/haskell-cafe/2002-February/002785.html.

[8] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 28–35, Paris, France, September 1999.

[9] John Peterson and Mark P. Jones. Implementing type classes. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 227–236, Albuquerque, New Mexico, June 1993.

[10] Ashley Yakeley. Re: Arrow classes, July 2003. Message posted on the Haskell-café mailing list, available at http://haskell.cs.yale.edu/pipermail/haskell-cafe/2003-July/004659.html.