Abstract

Type classes [19, 7] are, according to some authors, ‘the most beloved feature of Haskell’. They provide a clean, modular and elegant way of writing overloaded functions. Type classes are usually implemented using dictionaries—data structures containing functions and other dictionaries—which are passed as extra arguments to the overloaded functions. Thus, programs using type classes are translated into programs with dictionaries that are correct according to the Damas-Milner type system. Functional logic programming (FLP) has inherited the Damas-Milner type system from its functional part, so it may seem that using the same translation for these languages is also a good option. However, this translation presents some problems in FLP because of an undesired excess of sharing that appears with non-determinism.

In this paper we propose an alternative translation of type classes which uses type witnesses and type-indexed functions (in the sense of [11]) instead of dictionaries. The translated programs are correct according to a new simple extension of the Damas-Milner type system recently proposed for FLP [13]. Apart from solving the mentioned problem of undesired sharing, the proposed translation also obtains interesting efficiency results: in functional languages (GHC, Hugs) it can compete with the built-in translation, and in FLP (Toy, Curry) it obtains an important speedup—between 1.2 and 2.5 in the experiments—compared to the classical translation.

1 Introduction and Motivation

Type classes [19, 7] are one of the most successful features in Haskell. They provide an easy syntax to define overloaded functions—classes—and the implementation of those functions for different types—instances—. Type classes are usually implemented by means of a source-to-source transformation that introduce extra parameters to overloaded functions, generating Damas-Milner [3] correct programs. These arguments—called dictionaries—are data structures containing the implementation of overloaded functions for specific types and dictionaries for the superclasses. This translation produces efficient programs and combines well with modules and separate compilation, resulting in that nowadays it is the most used technique for implementing type classes in functional programming (FP).

Modern functional logic languages [8] like Curry [9] or Toy [15] have a strong resemblance to lazy functional languages like Haskell. A remarkable difference is that FLP supports non-deterministic functions [6], which are one of the most interesting features of the paradigm. In order to combine non-determinism with parameter passing the most natural way seems call-time choice [4]. It has the following intuitive meaning: given a function \( f e_1 \ldots e_n \), one first chooses some fixed (possibly partial) value for each of the actual parameters \( e_i \), before applying any rule for \( f \). Another intuitive view of
call-time choice is that duplicate occurrences of arguments in the right-hand side of a rule are shared, so they are reduced to the same values. Although non-deterministic functions are a big difference to FP, FLP have inherited the Damas-Milner type system from FP\(^1\). However, type classes have not been adopted yet in FLP, being the only attempts the experimental Zinc Compiler and a branch of the Münster Curry Compiler. One reason why they have not been integrated in FLP is that the dictionary approach produces a number of problems \([16, 17]\) that should be addressed.

In this paper we present an alternative source-to-source transformation for implementing type classes in FLP which uses type-indexed functions—functions with a different behavior for different types \([11]\)—and type witnesses—data structures representing types—instead of dictionaries. The proposed transformation solves the problem of undesired sharing exposed in \([17]\). This situation appears in the dictionary approach when type classes contain constant non-deterministic member functions. Consider the original program\(^2\) in Fig. 1-a). It contains a type class \(\text{arb}\) with a constant function \(\text{arb}\), and this class has an instance for \(\text{bool}\). Using the overloaded function \(\text{arb}\) we write the functions \(\text{arbP2}\), which returns a pair of (possibly different) instances of \(\text{arb}\), and \(\text{arbL2}\), which return a list of two elements of the same instance of \(\text{arb}\). Fig. 1-b) contains the translated program using the classical approach of dictionaries. The type class \(\text{arb}\) generates the data type of \(\text{arb}\) dictionaries \(\text{dictArb}\) and the projecting function \(\text{arb}\). The instance \(\text{arb bool}\) generates the particular \(\text{arb}\) function for booleans \(\text{arbBool}\) and the dictionary \(\text{dictArbBool}\). The functions \(\text{arbP2}\) and \(\text{arbL2}\) are translated into \(\text{arbP2'}\) and \(\text{arbL2'}\) resp. adding as many dictionaries as class constraints appear in their type contexts. The expression \(\text{arbP2'}::(\text{bool, bool})\) is translated into \(\text{arbP2'}\ \text{dictArbBool} \ \text{dictArbBool}\), whose evaluation generates the expected values: \((\text{true, true}), (\text{true, false}), (\text{false, true}),\) and \((\text{false, false})\). However, the expression \(\text{arbL2'}::[\text{bool}]\) is translated into \(\text{arbL2'}\ \text{dictArbBool}\) which generates only the values \([\text{true, true}]\) and \([\text{false, false}]\). The reason is that call-time choice semantics imposes that the copies of the function arguments made during parameter passing will share their values throughout the computation. In the first case two copies of \(\text{dictArbBool}\) are passed to \(\text{arbP2}\), so they can be reduced to the value \(\text{true}\) and \(\text{false}\) independently. On the contrary in the second case there is only one dictionary, so it can only be reduced to \(\text{true}\) or \(\text{false}\), value which must be shared in both positions of the resulting list.

The proposed translation solves the previous problem translating each overloaded function into a type-indexed function, instead of a parametrically polymorphic function which extracts concrete implementations from dictionaries. Moreover, the proposed translation produces FLP programs which run faster than those using dictionaries. This speedup has been measured experimentally in Toy and Curry (PAKCS \([10]\)). Since the proposed translation does not rely on any specific feature of FLP, we have also measure the speedup of the translation in FP (GHC, Hugs). Although these systems have optimizations for type classes, the results obtained are encouraging: the proposed translation compete in efficiency with the built-in translation, and runs faster compared to a handwritten dictionary translation.

2 Preliminaries

The target functional logic language of the translation appears in Fig. 2. The syntax of the expressions is very similar to FP, with the exception of \(\lambda\)-abstractions, which are not supported. Patterns—our notion of values—are a subset of the expressions. Notice that constructor and function symbols partially ap-
Definition 1 We say a rule \( f \vdash e \) is well-typed wrt. a set of assumptions \( \mathcal{A} \) iff:

- \( \mathcal{A} \vdash \{ X_n : \sigma_n \} \models f \vdash e : \tau_L[\pi_L] \)
- \( \mathcal{A} \vdash \{ \alpha_n : \beta_n \} \models e : \tau_R[\pi_R] \)
- \( \exists \pi. (\tau_L, \sigma_n \pi_L) = (\tau_R, \beta_n \pi_R) \pi \)

where \( X_n \) are the variables in \( \Gamma \) and \( \alpha_n, \beta_n \) are fresh type variables.
Intuitively, a rule is well-typed if the types inferred for the right-hand side—rhs—and its variables are more general than the types inferred for its left-hand side—lhs—and its variables. Notice that programmers must provide an explicit type for every function symbol, otherwise the first point of the definition fails to infer the type for the expression $f$. Therefore Def. 1 cannot be used to infer the types of the functions, but to typecheck that the types provided for the functions are correct.

The previous definition assures subject reduction (type preservation) and progress under a variation of the let-rewriting [14] semantics (see [13] for more details). However, the most remarkable feature is that this notion of well-typedness allows the programmer to define type-indexed functions in a very easy way. Consider Fig. 3, where $z$ and $s$ are the constructors of Peano natural numbers. The three first rules for \textit{size} are well-typed because the type inferred for the rhs (\textit{nat}) is more general than the inferred in the lhs (\textit{nat} again). In the fourth rule the types inferred for the lhs and the variable $X$ are \textit{nat} and \textit{nat} resp., and in the rhs the inferred types are \textit{nat} and $\beta$ resp., so the rule is well typed since (\textit{nat}, $\beta$) is more general than (\textit{nat}, $\textit{nat}$). The same happens in the fourth rule of \textit{eq}, where (\textit{bool}, $\beta$, $\beta$) inferred for the rhs is more general than (\textit{bool}, \textit{nat}, \textit{nat}) inferred for the lhs. The rest of rules for \textit{eq} are well-typed for similar reasons.

### 3 Translation using Type-Indexed Functions

As we have said in Sect. 1, the proposed translation uses type-indexed functions and type witnesses instead of dictionaries of the classical translation. The possibility of a translation using type representations was already mentioned in [2], although it was not further developed. The main idea is to replace each overloaded symbol by a type-indexed function with the rules of all the instances, and use a type-witness to determine which rules can be applied in a certain situation. As an example consider the program in Fig. 4-a)—where we have used a syntax for classes and instances similar to Haskell—and its translation in Fig. 4-b). Notice how the first argument of \textit{eq}' (the type witness) determines which rules can be used: if it is $\#\textit{nat}$ then the rules of the instance \textit{eq nat} are used, and if $\#\textit{list A}$ is passed then the rules of the instance \textit{eq (list A)} are used. The steps of the translation are:

1. Each data type is extended with a \textit{type witness} for that type. The type witness of the data type $C$ is $\forall \alpha. C$ with type $\forall \alpha. C$. In the example program, the witness generated for \textit{nat} is simply $\#\textit{nat}$ of type \textit{nat}, and for \textit{list A} is $\#\textit{list}$ of type $\forall \alpha. \textit{list A}$. Type witnesses are a way of representing types as values which follows the same idea of type representations [2, 11].

2. Member functions and functions using overloaded symbols are extended with extra arguments representing type witnesses. The information about the number and type of the type witnesses is extracted from the context—class constraints—of the type. In the example, \textit{eq :: eq A \Rightarrow A \Rightarrow A \Rightarrow \textit{bool}} is translated into \textit{eq' :: A \Rightarrow A \Rightarrow A \Rightarrow \textit{bool}}. The new first argument is the type witness of the elements to compare. Like in the translation using dictionaries, each occurrence of an $\textit{eq}$ in...
overloaded function in the right-hand side must be completed with the corresponding type witness. In the example, the rule \( \text{eq} (s \; X) \; (s \; Y) \rightarrow \text{eq} \; X \; Y \) is translated into \( \text{eq}' \; #\text{nat} \; (s \; X) \; (s \; Y) \rightarrow \text{eq}' \; #\text{nat} \; X \; Y \). The function \( f \; X \; Y \rightarrow \text{eq} \; X \; Y \).

As in the translation using dictionaries, goals are translated introducing concrete type witnesses. For example the goal \( f \; \text{nil} \; (\text{cons} \; z \; \text{nil}) \) is translated into \( f' \; (#\text{list} \; #\text{nat}) \; \text{nil} \; (\text{cons} \; z \; \text{nil}) \).

A point in favor of this translation is that we need less type witnesses than dictionaries in the classical translation\(^4\). For example a function \( g :: (\text{eq} \; A, \text{show} \; A) \rightarrow A \rightarrow \text{bool} \) needs two different dictionaries (for \text{eq} and \text{show}) however it only needs one type-witness (for the \( A \) type) since both class constraints affect \( A \). Another advantage is that with dictionaries, member functions must be extracted from the dictionaries before applying them, and it can mean traversing a chain of superclasses dictionaries\(^5\). With the proposed translation, the rules of the instance to apply are directly determined by the type witness argument regardless of the subclass depth.

4 Advantages of the proposed translation

The first advantage of the translation is that it solves the mentioned problem of undesired sharing. In Fig. 1 we showed a program which does not produce the expected values for the evaluation of \( \text{arbL2} :: [\text{bool}] \) —translated into \( \text{arbL2}' \; \text{dictArbBool} \)—if the classical translation of dictionaries is used. The problem is that the call-time choice semantics forces the two copies of the dictionary \text{dictArbBool} to be shared in the rhs of the rule for \( \text{arbL2}' \). Therefore the concrete function that \text{dictArbBool} contains will be evaluated to \text{true} or \text{false} and that shared value will appear twice in the resulting list. Fig. 5-a) shows the translation of the program in Fig. 1 using type-indexed functions. The code for \text{arbL2}' is the same as the code generated by the dictionary approach, however \text{arb} is different: it does not extract a concrete function from a dictionary but accepts a type witness and use it to choose which rules—i.e., which instance—we are using. Therefore different rules for \text{arb} may be used in each element of the list, avoiding the

\(^4\)We do not consider the optimizations for the classical translation presented in [1, 18].

\(^5\)Although this problem can be solved flattening the dictionary structure [1].
arb :: A -> A
arb #bool -> true
arb #bool -> false
arbP2' :: A -> B -> (A,B)
arbP2' WitA WitB -> (arb WitA, arb WitB)
arbL2' :: A -> [A]
arbL2' Wit -> [arb Wit, arb Wit]
arbL2' #bool

a) Translation of the program in Fig 1
b) Reduction of arbL2 #bool

Figure 5: The goal arbL2 :: [bool] with the proposed translation

undesired sharing. Fig. 5-b) shows the reduction (using \(\rightarrow^q\) [13]) of the translation of arbL2 :: [bool] to obtain the missing answer [true, false].

Apart from solving the problem of undesired sharing, the proposed translation also presents a real improvement in efficiency compared to the classical translation using dictionaries.

We have measured the real gain in efficiency obtained using type witnesses and type-indexed functions instead of handwritten dictionaries over the FLP systems Toy and Curry. As a matter of curiosity, we have also measured the gain in efficiency over FP systems—GHC, GHCi and Hugs—using handwritten dictionaries and built-in type classes.

Translated programs using dictionaries are valid in all the systems since they only need a Damas-Milner type system. Programs using type witnesses and type-indexed functions require an implementation with the type system proposed in [13]. None of the tested systems support it, so we have followed two solutions. For TOY 2.3.1 we have used an special version disabling type checking. This does not distort the measures since once compiled, programs do not carry any type information at run time, so compiled programs are the same regardless of the chosen type system. For the rest of the systems—GHC 6.10.4, Hugs Sept. 2006 and PAKCS 1.9.1(7)—we have included all the data constructors inside the same universal data type. Thus, only one instance for each type class is needed and all the rules fit inside, making the program well-typed.

We have measured the speedup in programs with different levels of subclasses (Fig. 6). For FLP systems, the obtained speedup grows almost linearly with the subclass depth, with an speedup between 1.1–1.3 when one subclass is used to 1.5–2.5 when nine subclasses are used. For FP the speedup grows also almost linearly in all the cases except in GHC using handwritten dictionaries. In general terms, for FP systems the translation using type witnesses performs faster. The only exception is in GHCi using the built-in type classes, where the translation using type witnesses runs about 10% slower for one subclass, and for higher subclass depth the time used in both translation is the same.

These results encourage us to use type witnesses instead of dictionaries when implementing type classes in FLP languages, since the obtained efficiency is better than the one obtained with the classical—with optimizations—dictionary technique.

5 Conclusions and Future Work

In this paper we have proposed an alternative translation of type classes for FLP which solves one problem of the classical dictionary approach and also generates faster programs (Sect. 4). This translation relies in type witnesses and type-indexed functions, and creates well-typed programs according to a simple extension of the Damas-Milner type system recently proposed [13]. However this translation is also interesting in FP, as the efficiency results show. For this, the type system in [13] could be applied directly to FP (it does not rely on any FLP specific feature), or GADTs could be used to implement type witnesses and type-indexed functions [2, 11].

However, further work is needed in order to
develop completely this translation. First, we need to formalize it in detail, step which is essential for the implementation and integration into Toy. Second, it would be necessary to study the behavior of this translation with the rest of problems exposed in [16]. Finally, it would be interesting to study how this translation combines with modules and separate compilation, features appearing in modern FP and FLP compilers.

References


