Type classes in Mercury

DRAFT

David Jeffery, Fergus Henderson and Zoltan Somogyi
Department of Computer Science, University of Melbourne
{dgj,fjh,zs}@cs.mu.oz.au

September 30, 1998

Abstract

The type systems of functional languages such as Haskell have recently become more powerful and expressive. They not only allow programmers to write code that works on values of any type (genericity), they also allow programmers to require that a particular type belongs to a given *type class* (constrained genericity). Such code may use any of the methods listed in the interface of the type class, since every type that is a member of the type class must implement those methods. This capability makes it significantly easier to express solutions to many common problems, and promotes code reuse.

Incorporating type classes in a logic programming language provides some new challenges. In this paper, we explain how we have extended Mercury's type system to include support for type classes. We give a formal semantics for this extension to our type system, adapting the typing rules used in functional languages to the differing demands of logic programming languages. We show that type classes integrate very nicely with Mercury's mode, uniqueness and determinism systems, and describe how our implementation works.

1 Introduction

Traditionally, logic programming languages such as Prolog have had weak, run-time typing. More recent logic programming languages such as Gödel[4] and Mercury[12] have embraced strong typing to increase the level of static checking that may be done and thus make their programs more reliable. Their type systems are based on the Mycroft-O'Keefe adaption [10] of the Hindley-Milner type system commonly used in functional languages. These type systems guarantee that "well-typed programs do not go wrong". This means that a well-typed program will never cause a run-time type error, even though the semantics of the typed program is equivalent to the semantics of the same program ignoring the types.

The type systems of modern functional languages such as Haskell[2] and Gofer[6], have significant extensions to the Hindley-Milner type system. Their most important extension is their support for constrained forms of genericity through *type classes*. A type class specifies an interface consisting of one or more methods. A type can be declared to be a member of a type class by specifying the implementations of the methods for that type class, and functions can be declared to take as arguments values of any type that is a member of a given type class.

Type classes give functional languages the kind of expressiveness that object-oriented constructs give to procedural languages. Just as a function in (say) C++ can operate on any object belonging to a given class, a function in Haskell can operate on any value whose type is a member of a given type class. Both the C++ and Haskell functions can invoke the operations listed in the interface of that class or type class. This means that the programmer does not have to write separate versions of the code of this function for each kind of object/value it needs to be able to operate on. This kind of capability has been proven to be indispensable in the construction of large software systems.

The difference between type classes in functional languages and classes in most object-oriented languages is that type classes provide inheritance only for interfaces, not for the implementations of methods. Even without inheritance of method implementations, the programmer can arrange for two implementations of a method to share code by having those implementations call the same functions, possibly in different contexts or with different arguments. This is because with type classes, the implementation of each method is explicitly under the control of the programmer. This explicit control also allows and in fact requires the programmer to resolve the kinds of problems that arise in object-oriented systems with multiple inheritance when a class inherits conflicting implementations of a method from two or more of its superclasses.

Mercury is a purely declarative logic programming language designed to support the construction of large software systems. The fact that the Hindley-Milner type system of Mercury does not support constrained genericity or the use of object-oriented programming techniques are significant drawbacks in this application area. We have therefore decided to integrate type classes into Mercury. Type classes are cleaner and have simpler semantics than previous proposals for adding object-oriented constructs to logic languages [1, 9], and unlike many of those proposals, they do not introduce any possibilities for run-time errors.

In section 2 we outline how type classes can be introduced into a generic strongly typed logic programming language. In section 3 we present a set of formal rules that define type correctness with respect to our system of type classes, rules that can be used for both type checking and type inference. These rules also describe transformations that provide both an implementation for type classes and a semantics for programs using type classes in terms of programs that do not. In section 5, we consider the interactions between the type class system and the rest of Mercury, and show that our type class mechanism integrates in a simple and elegant manner with Mercury's mode, determinism and uniqueness systems. In section 6 we then present some of the issues that have arisen from our implementation of type classes in the Mercury compiler.

2 Type classes in a typed logic programming language

We now consider the problem of adding type classes to a logic programming language with higher-order constructs and a Hindley-Milner type system [10]. We will use the syntax of Mercury, but for now we will restrict our attention to issues involving types. We defer discussion of the effects of Mercury's mode, determinism and uniqueness systems upon type classes until section 5.

Syntactically, the type class system introduces three new constructs to the language, namely `typeclass` declarations, `instance` declarations, and type class constraints.

2.1 Type class declarations

A `typeclass` declaration introduces a new type class, and defines the interface to which a vector of one or more types must conform in order to be considered a member of that type class. The interface is a set of *methods*, i.e. predicate declarations. (Mercury supports functions as well as predicates and hence our implementation of type classes supports functions too, but we don't discuss functions further in this paper. Internally the Mercury compiler transforms functions into predicates, so they do not cause any additional difficulties.)

For example, the declaration:

```
:- typeclass x_y_point(T) where [
        pred get_x(T, float),
        pred get_y(T, float),
        pred set_x(T, float, T),
        pred set_y(T, float, T)
].
```

introduces the type class `x_y_point/1` (where the "/1" indicates that the arity is one), and says that a type `T` belonging to this type class must provide four predicates with the specified signatures to implement the methods `get_x/2`, `get_y/2`, `set_x/3` and `set_y/3`.

Most type class declarations have one type variable as their parameter. However, we also allow a type class declaration to have two or more type variables as parameters. Such *multi-parameter* type classes allow the program to clearly specify relationships among several types, via one ore more operations in the type class whose signature includes more than one parameter of the type class. This capability increases the expressive power of the system, and is particularly useful in a language that supports constructor classes [5, 8]. (We do not address the issue of constructor classes in this paper.)

2.2 Instance declarations

An `instance` declaration says that a particular vector of types is a member of a given type class, and specifies the predicates that implement each of the methods listed in the interface of that type class. For example, given the definitions

```
:- type cartesian ---> cart(float, float).

:- pred cart_get_x(cartesian, float).
:- pred cart_get_y(cartesian, float).
:- pred cart_set_x(cartesian, float, cartesian).
:- pred cart_set_y(cartesian, float, cartesian).
```

the `instance` declaration

```
:- instance x_y_point(cartesian) where [
        pred(get_x/2) is cart_get_x,
        pred(get_y/2) is cart_get_y,
        pred(set_x/3) is cart_set_x,
        pred(set_y/3) is cart_set_y
].
```

makes the `cartesian` type a member of the `x_y_point` type class.

An `instance` declaration gives a type for each parameter of the type class. These types must have the form of a type constructor applied to zero or more type variables, and each of these type variables may occur only once in the whole type vector. A program may not contain more than one declaration declaring a particular vector of type constructors to be instances of a given type class. These restrictions ensure that there are no overlapping instance declarations, i.e. there is at most one instance declaration that may be applied to any vector of types.

2.3 Type class constraints

Polymorphic predicates have type variables in their signatures. In the absence of a type class constraint on the predicate declaration, the predicate may be called with actual values of any type. In the presence of one or more type class constraints on the predicate declaration, the predicate can only be called with values whose types satisfy all the constraints.

For example, in

```
:- pred draw_point(T, io__state, io__state) <= x_y_point(T).

draw_point(Point, IO0, IO) :-
        get_x(Point, X),
```

```
        get_y(Point, Y),
        move_pen(X, Y, IO0, IO1),
        pen_down(IO1, IO2),
        pen_up(IO2, IO).
```

the type class constraint (also called class context) `<= x_y_point(T)` says that the type variable `T` can only stand for types that are members of the `x_y_point` type class. This is why `draw_point` is allowed to invoke the methods of that type class on a value (`Point`) of a type that belongs to the type class. (The `io__state` arguments represent states of the world; Mercury uses them to ensure that I/O is referentially transparent.)

Each type class constraint is the name of a type class applied to one or more type variables. All these variables must appear in the types of the arguments of the predicate, but a variable may appear in more than one constraint and/or more than once in one constraint.

Type class constraints may also appear on `typeclass` and `instance` declarations. Constraints on type classes introduce the notion of "super-classes". Any vector of types declared to be an instance of a sub-class must also be declared to be an instance of the super-class. For example,

```
:- typeclass coloured_point(T) <= x_y_point(T) where [
        pred get_colour(T, colour)
].
```

introduces a new type class, `coloured_point`, describing points which have an associated colour, of which `x_y_point` is a super class. The methods defined by the `coloured_point` type class include all those defined by the `x_y_point` type class, as well as `get_colour`. Any type declared to be an instance of `coloured_point` must also be declared to be an instance of `x_y_point`.

Constraints appearing on `instance` declarations form an *implied instance*. For example, consider the following declaration of a type class of types that may be printed:

```
:- typeclass portrayable(T) where [
        pred portray(T, io__state, io__state)
].
```

The programmer could declare instances such as

```
:- instance portrayable(int) where [
        pred(portray/3) is io__write_int
].
```

(`io__write_int` is defined in the Mercury standard library.) However, when it comes to writing the instance declaration for a type such as `list(T)`, we want to be able to print out the list elements using the `portray/3` for the particular type of the list elements. This can be achieved by placing a type class constraint on the instance declaration, as in the following example:

```
:- instance portrayable(list(T)) <= portrayable(T) where [
        pred(portray/3) is portray_list
].

:- pred portray_list(list(T), io__state, io__state)
        <= portrayable(T).

portray_list([], IO, IO).
portray_list([X | Xs], IO0, IO) :-
        portray(X, IO0, IO1),
        io__write_char(' ', IO1, IO2),
        portray_list(Xs, IO2, IO).
```

This states that for every type `T` that is a member of `portayable`, `list(T)` is also a member.

2.4 Type correctness

We now briefly and informally discuss the notion of type correctness in our system. A more formal presentation is given in section 3. Essentially, a program is type correct if the program (ignoring type classes) is type correct under the Mycroft-O'Keefe type system[10], and the type class constraints of the program are satisfied.

The type class constraints for any predicate call are satisfied if and only if one of the following conditions holds for each such constraint:

1. The constraint is upon a vector of type variables, and the same constraint appears in the type signature of the caller.

2. The constraint is upon a vector of types where in each type at least the top type constructor bound, there is an instance declaration that declares this vector of type constructors to be a member of the desired type class, and all type class constraints on this instance declaration are also satisfied.

3. The constraint requires a vector of types to be a member of a type class, and this vector of types is known to be a member of a subclass of this type class.

2.5 Translation

For Haskell, the semantics of a program using type classes is defined by a simple source-to-source transformation to a program which does not use type classes[2, 13]. This transformation is also used in the implementations of the language. We take a similar approach here. In this section, we informally describe the translation, leaving a more formal approach until section 3.

Each constraint in the type signature of a predicate adds an extra argument to each clause of that predicate. Corresponding to this, each call to a predicate must supply one extra argument for each constraint on the called predicate. Each of these extra arguments contains a dictionary that specifies for each method in the interface of the relevant type class the implementation of that method for the actual type(s) corresponding to the vector of type variables in the type class constraint. The implementation will be represented by a higher-order constant. For a type class such as `portrayable` with one method, class methods are implemented by extracting the relevant field from the dictionary and performing a higher order call, as in

```
portray(DictPortray, X, IO0, IO) :-
        DictPortray = portrayable_dict(PortrayImplementation),
        call(PortrayImplementation, X, IO0, IO).
```

For each of the ways of satisfying a type class constraint listed in section 2.4, there is a corresponding method for obtaining the dictionary corresponding to that constraint.

1. The dictionary to be passed to the callee can come from one of the extra arguments added to the head of the caller.

2. The dictionary can be produced by a goal that corresponds to an instance declaration.

3. The dictionary for a type class can be extracted from the dictionary for one of its subclasses.

The first way ought to be self-explanatory. To illustrate the second way, consider the two instance declarations of the type class portrayable above. The declaration asserting that integers are portrayable has no type class constraints on it, so the goal corresponding to this declaration is simply

```
IntDict = portrayable_dict(io__write_int)
```

where `io__write_int` is syntactic sugar for

```
lambda([X, IO0, IO], io__write_int(X, IO0, IO))
```

Instance declarations with constraints on them correspond to goals with one input argument for each constraint. The declaration asserting that a list of `T`s is portrayable has a type class constraint requiring `T` to be portrayable. Therefore its goal has one input argument giving the dictionary for `T` from which it constructs the dictionary for `list(T)`.

```
ListDict = portrayable_dict(portray_list(TDict))
```

where `portray_list(TDict)` is syntactic sugar for

```
lambda([X, IO0, IO], portray_list(TDict, X, IO0, IO))
```

This would be used in situations such as the following, where we build a list whose elements have a type known to be portrayable, a type whose dictionary is in `ElemDict`:

```
        ...
        Xs = [Elem1, Elem2, Elem3],
        ListDict = portrayable_dict(portray_list(ElemDict)),
        portray(ListDict, Xs, IO0, IO),
        ...
```

To illustrate the third way of getting hold of dictionaries, consider the following predicate whose implementation extracts the dictionary corresponding to the constraint `<= x_y_point(T)` from the dictionary for the constraint `<= coloured_point(T)`:

```
:- pred draw_colour_point(T, io__state, io__state)
        <= coloured_point(T).

draw_colour_point(ColourDict, CP, IO0, IO) :-
        get_colour(ColourDict, CP, Colour),
        set_pen_colour(Colour, IO0, IO1),
        ColourDict = coloured_point_dict(XYDict, _GetColour),
        draw_x_y_point(XYDict, CP, IO1, IO).
```

3 Typing and translation rules

In this section we give a set of rules which both define the notion of type-correctness in our type system, and give a translation from our source language (a typed, higher order logic programming language) to our target language (an untyped, higher order logic programming language). In section 5, we consider the problem of adding type classes to the full Mercury programming language, and examine the challenges posed by Mercury's strong mode, determinism and uniqueness systems. The rules defined here are used by the Melbourne Mercury compiler for both type checking and type inference.

We begin by defining the type system of our source language, and defining a syntax for our source and target languages. We then give a set of rules which define type correctness of source programs in our system, and which simultaneously define a translation into our target language. This formalism is in the spirit of [10], but our rules resemble those for Haskell in [2] more closely. In section 4 we compare our proposal to the Haskell type class system.

Note: we use vector notation to represent a sequence of entities. For example, to represent a sequence of x, we write $\overline{x}$, which is equivalent to $< x_1, \ldots, x_n >$.

We begin by defining the syntax of our source language. Figure 1 shows the syntax of types, which are an extension of the Mycroft-O'Keefe[10] type system with higher order types and type class constraints. Figure 2 shows the declaration syntax, while figure 3 shows the syntax of the clauses themselves. Since our target language has no types or declarations, its syntax is confined to that shown in figure 3.

Our typing rules are presented with respect to an "environment", which represents the assumptions and information collected during type checking. The

$$
\begin{array}{llll}
\textit{Type variable} & \alpha & & \\
\textit{Type constructor} & \chi & & \\
\textit{Class name} & \kappa & & \\
\textit{Simple type} & \tau & \rightarrow & \alpha \\
 & & | & \chi\overline{\tau} \\
 & & | & pred(\overline{\tau}) \\
\textit{Context} & \theta & \rightarrow & < \kappa_1\overline{\alpha}_1, \ldots, \kappa_n\overline{\alpha}_n > \\
\textit{Qualified type} & \sigma & \rightarrow & \forall\overline{\alpha}.pred(\overline{\tau}) \Leftarrow \theta
\end{array}
$$

Figure 1: Syntax of types in the source language

$$
\begin{array}{lll}
\textit{Declaration} & \rightarrow & \textit{Pred Declaration} \\
 & | & \textit{Type Declaration} \\
 & | & \textit{Type Class Declaration} \\
 & | & \textit{Instance Declaration} \\
\textit{Pred Declaration} & \rightarrow & \texttt{pred}\ \pi(\overline{\tau}) \Leftarrow \theta \\
\textit{Type Declaration} & \rightarrow & \texttt{type}\ \chi\overline{\alpha} \rightarrow f_1(\overline{\tau}_1); \ldots; f_n(\overline{\tau}_n) \\
\textit{Type Class Declaration} & \rightarrow & \texttt{typeclass}\ \kappa\overline{\alpha} \Leftarrow \theta\ \texttt{where}\ [\\
 & & \quad (\textit{Pred Declaration})_1, \ldots, \\
 & & \quad (\textit{Pred Declaration})_n] \\
\textit{Instance Declaration} & \rightarrow & \texttt{instance}\ \kappa\overline{\chi\overline{\alpha}} \Leftarrow \theta\ \texttt{where}\ [\\
 & & \quad \pi_1/n_1\ \texttt{is}\ \pi'_1, \ldots, \\
 & & \quad \pi_m/n_m\ \texttt{is}\ \pi'_m]
\end{array}
$$

Figure 2: Syntax of declarations in the source language

$$
\begin{array}{llll}
\textit{Variable} & v & & \\
\textit{Functor} & f & & \\
\textit{Predicate name} & \pi & & \\
\textit{Term} & t & \rightarrow & v \\
 & & | & f(\overline{t}) \\
 & & | & \lambda\overline{v}.G \\
\textit{Atom} & A & \rightarrow & \pi(\overline{t}) \\
\textit{Goal} & G & \rightarrow & G, G \\
 & & | & \neg\ G \\
 & & | & call(v, \overline{t}) \\
 & & | & A \\
 & & | & \texttt{true} \\
\textit{Clause} & C & \rightarrow & A \leftarrow G \\
\textit{Program} & P & \rightarrow & \overline{C}
\end{array}
$$

Figure 3: Syntax of programs in the source and target languages

components of the environment are as follows:

- P: the types of predicate symbols: $P = \{\pi/n : \sigma\}$
- F: the types of functors: $F = \{f/n : \tau_1 \to \ldots \to \tau_n \to \tau\}$
- LIE: local instance environment, which is a mapping from type class constraints to the local variables which represent their dictionaries: $LIE = \{\kappa\overline{\alpha} : v\}$
- IE: instance environment, which is a mapping from type class constraints to the goals which are used to generate them and the variables which contain their input and output variables: $IE = \{\forall\overline{\alpha}.\kappa\overline{\tau} \Leftarrow \theta : \lambda\overline{v}.G\}$
- CE: class environment, which is a mapping from type class name and arity to the super classes and predicates and arities of the methods of the type class: $CE = \{\forall\overline{\alpha}.\kappa/k : \overline{\pi/n : \theta} \Leftarrow \theta\}$
- V: the types of variables in a clause: $V = \{v : \tau\}$

These environment components are stored in a single environment: $E = (P, F, LIE, IE, CE, V)$.

Definition: (ENV of E) means selection of the ENV environment component from the environment E.

Definition: $ENV_1 \oplus ENV_2$ means the union of ENV_1 and ENV_2. It is an error if the domains of ENV_1 and ENV_2 are not disjoint.

We also write $E \oplus P_0$ to mean the environment E but with the elements of P_0 added to its P field.

Definition: $A \cong B$ means that A and B are equal up to variable renaming.

We now present the typing/transformation rules. In these rules, we use the notation $E \vdash G \leadsto G'$ to indicate that G is well typed with respect to E, and it is transformed into G'. These rules may either be viewed as performing type *checking* (by looking up a predicate's declared type in the environment E for each predicate call) or as performing type *inference* (by calculating the types and constraints for each predicate where predicate type declaration was given).

The first rule, $\mathcal{CLAUSE}$ (in figure 4) is concerned with the typing of a clause. Intuitively, the rule says that:

- the head of the clause is well typed if the types of all the arguments are the same as the types of the arguments of the predicate according to the environment, and the type class constraints on the predicate are exactly those that occur in the LIE which is used to type and transform the body;
- the body of the clause is judged according to the environment containing the types of the variables in the head;
- the head and body of the clause are transformed independently, both with respect to the LIE containing one variable for each constraint on the predicate in the head.

Note that during the transformation the predicate name is annotated with its original arity so that in adding extra arguments we do not make two predicates which were originally of the same name but different arity into predicates of the same name and same arity.

Rules $\mathcal{NOT}$, $\mathcal{TRUE}$ and $\mathcal{CONJ}$ handle negated goals, `true`, and conjunctions respectively.

The $\mathcal{ATOM}$ rule is concerned with an atomic goal (a predicate call). Essentially, checking the type correctness of a predicate call amounts to

- looking up the type of the called predicate and finding a suitable type substitution which maps the types of the callee's arguments to those of the caller;
- checking that the constraints on the callee, with the substitution applied so they are expressed on the arguments of the caller, are satisfied.

The transformation for a predicate call is:

- Add any goals produced as part of the transformation of the type class constraints as a conjunction in front of the goal.
- Add the variables which contain the type class dictionaries for the type class constraints to the front of the atom being called.

$\mathcal{HO} - \mathcal{CALL}$ checks a higher order call. This rule is quite simple, since higher-order types do not have any type class constraints. The reason for that is that in Mercury, just as in Haskell and most similar languages, higher order terms are monomorphic, not polymorphic, so there are no type variables to constrain. Extending our system to support nested universal type quantifiers (i.e. rank-2 polymorphism) remains a topic for future work.

The final rules concerned with goals themselves are $\mathcal{TERM}$, $\mathcal{VAR}$ and $\mathcal{HO} - \mathcal{CONST}$ in figure 5. These check the types of terms, variables and higher order constants. Only $\mathcal{HO} - \mathcal{CONST}$ requires an explanation. When checking the lambda goal, we augment the environment with one variable of the appropriate type for each argument of the lambda expression, since these variables are visible from inside the lambda goal but not from the outside. The transformation of the lambda goal may introduce new variables representing type class dictionaries. Some of these may be nonlocal to the lambda goal, i.e. they may occur outside the lambda goal as well. This is OK, since this may happen even without the transformation; all these nonlocal variables will be included in the closure generated for the lambda expression.

Figure 6 shows the typing and translation of type class constraints themselves. The simplest method of constraint satisfaction comes from the rule $\mathcal{DICT} - \mathcal{TAUT} - \mathcal{LIE}$, which says that if there is a type class dictionary for a constraint in the local variables of a clause, then the constraint is solved and the dictionary is found in the variable.

The rule $\mathcal{DICT} - \mathcal{TAUT} - \mathcal{IE}$ satisfies a type class constraint by using an instance declaration or the super class relation from the IE.

$\mathcal{DICT} - \mathcal{SPEC}$ merely allows special cases of more general constraints.

The rule $\mathcal{DICT} - \mathcal{REL}$ allows a constraint to be satisfied if it can be satisfied under a set of constraints (using an implied instance or super class, the proof of which will involve $\mathcal{DICT} - \mathcal{TAUT} - \mathcal{IE}$), and in turn those constraints can be solved.

Finally, $\mathcal{DICT} - \mathcal{SEQ}$ satisfies a sequence of constraints by satisfying them individually.

We conclude our formal treatment of the system by considering the effects that the various declarations of our source language have upon the environment E. The meaning of these rules, presented in figures 7, 8 and 9, is rather different to that of our previous rules; rather than *using* the elements of the environment E, these rules *build* the environment. Each declaration rule has a translation which is the new environment which results from processing the declaration. When processing the whole program, we start with an empty environment ($E = (\{\}, \{\}, \{\}, \{\}, \{\}, \{\})$), then process all the declarations, threading the environment through as it is built. The environment which results from processing all of the declarations is the environment from which the clauses are judged.

In rule $\mathcal{PRED}$, we see that the effect of a `pred` declaration is to ensure that the type of a predicate is as declared. Note that, when there is no `pred` declaration for a predicate, we can view the rules given above as a means for inferring the predicate's type. Similarly, rule $\mathcal{TYPE}$ shows that a `type` declaration annotates the environment with the types of all the functors of the type.

The final two rules deal with `typeclass` and `instance` declarations. The $\mathcal{TYPECLASS}$ rule shows that the `typeclass` declaration adds to the environment

- an entry in CE for the new type class;
- an entry in P for each class method;
- an entry in IE for each super class of the class.

The declaration also adds a predicate for each class method to the target program. The implementations of the predicates for selecting class methods and super class dictionaries from dictionaries are implemented here in terms of a predicate $\texttt{project}^m_n\texttt{/2}$, which takes its first (input) argument (which is a dictionary with m entries), and binds its second (output) argument to the nth entry.

The rule for `instance` declarations ($\mathcal{INSTANCE}$) checks that the types of the instance methods are those required by the type class, and that any type class constraints on an instance methods are implied by the constraints on the instance declaration itself. This rule also checks that all super class constraints are satisfied, and adds the dictionaries for the super classes to the dictionary for this instance. It then adds to IE an entry which contains a goal that generates this dictionary.

4 Comparison with the Haskell type class system

The chief difference between the rules we have presented here and those for Haskell presented in [2] is that the Haskell rules deal chiefly with expressions whereas we deal with predicates and goals. One consequence of that is that our goals in many cases must explicitly name (and record the types of) the variables of each clause, whereas the the Haskell rules use application and lambda abstraction which makes the rules somewhat simpler. Rather than inferring or checking the type of a single expression, we traverse conjunctions of goals, building up (or checking against) a set of constraints as we go.

Due to the similarity between the two formalisms, the theoretical results about Haskell style type classes, proven in papers such as [7, 11] should carry over quite simply to Mercury.

Nonetheless, there are still some differences between the "flavour" of type classes we have chosen here and the Haskell flavour. The most obvious difference is that we do not (yet) support constructor classes in Mercury, although we do support multi-parameter type classes, a feature notably lacking from Haskell.

Some of the *design space* for type classes is explored in [8]. For most of the design questions raised in that paper, we have made the same decisions as the current incarnation of Haskell. The exceptions are the following.

- Constraints on an instance declaration must all be on type variables, but the type variables need not be distinct.
- Constraints on a type class declaration must also all be on type variables, and once again the type variables need not be distinct.
- The type signature of class methods can take any form that is legal for any other predicate.

5 Type Classes in Mercury

Our treatment so far has been concerned only with type checking and inference. We now consider the problem of integrating the type system presented above into Mercury, a purely declarative logic programming language that has not only a strong type system but strong static mode, determinism and uniqueness systems as well[3]. In the context of the full language, our system of type classes must be a little more complicated.

Our main concern is the need to ensure that a call to a class method, where the actual code called is not chosen until runtime, does not result in a mode, determinism or uniqueness error. To make this possible, we extend the syntax of `typeclass` declarations:

```
:- typeclass x_y_point(T) where [
        pred get_x(T, float),
```

```
        mode get_x(in, out) is det,

        ...
].
```

Each method of the type class also has a `mode` declaration which gives information about the modes and uniqueness of the arguments and the determinism of that mode of the predicate. As with all Mercury predicates, there may be more than one `mode` declaration for each method. We call each mode of a predicate a *procedure*.

Checking the correctness of the methods in an `instance` declaration is no longer just a matter of checking the type correctness of the method with respect to the corresponding type class method; it is also necessary to check that the instance method is mode, uniqueness and determinism correct. Doing so turns out quite elegantly in our system; type checking the instance declaration gives us these things "for free".

To type check an instance method, we introduce an auxiliary predicate whose type is the type of the corresponding class method with the instance types substituted for the class type variables, and whose type class constraints are the union of the class method's constraints and the constraints from the head of the instance declaration. For example, given the following declarations:

```
:- typeclass c1(T) where [
        pred p(T, T2) <= c2(T2),
        mode p(in, out) is det,
        mode p(out, in) is nondet
].

:- instance c1(list(T)) <= c(T) where [
        pred(p/2) is some_predicate
].
```

the correctness of `p` is checked by introducing the following auxiliary predicate (although the name mangling used in practice is more robust):

```
:- pred p_prime(list(T), T2) <= (c2(T2), c(T)).

p_prime(X, Y) :-
        p(X, Y).
```

In fact, it is `p_prime` that is added to the dictionary, not `p`. This allows `p` to be less constrained (or more polymorphic) than `p_prime` is expected to be.

The introduction of `p_prime` also allows us to check the mode, determinism and uniqueness correctness of `p`. All we need to do is add the following mode declarations:

```
:- mode p_prime(in, out) is det.
:- mode p_prime(out, in) is nondet.
```

The compiler treats introduced predicate just like any other, so even though we introduced the predicate to do *type* checking, the appropriate phases of the compiler also perform mode, determinism and uniqueness checking. In addition, just as this allows the type of `p` to be more general than `p_prime`'s, the mode, determinism and uniqueness may also be more general. For modes, this means that an *implied mode* may be used. For determinism, it means that the procedure may have a more strict determinism. (eg. it may be known never to fail even though it is allowed to). For uniqueness, is means that an argument may be returned that it guaranteed to be unique, even though this is not required.

Armed with the knowledge that class method calls themselves cannot cause mode, determinism or uniqueness errors, it is simple to show that the transformations described by our rules preserve mode, determinism and uniqueness correctness. The reasoning for this is as follows.

Our transformation does not alter the mode or uniqueness properties of any of the source program's variables. To show that the transformation preserves mode and uniqueness correctness, it is therefore sufficient to show that the *introduced* variables have no such errors.

First, the introduced variables are not required to be unique, which means uniqueness errors cannot arise. Second, for all the introduced variables, the producer of the variable is inserted before all consumers. This is clear from an examination of the rules themselves. The only rules which introduce variables are $\mathcal{CLAUSE}$, which adds variables to the head of the clause, and $\mathcal{ATOM}$, which adds variables to calls in the body of the clause.

$\mathcal{CLAUSE}$ assumes that its introduced variables are bound by its caller. $\mathcal{ATOM}$ makes sure that this is so, since it binds every variable before it is used by the call to which it is added. In terms of the $\mathcal{ATOM}$ rule, $v_1, \ldots, v_k$ are always bound by G. This is definitely so since each of the dictionary rules either produces the dictionary variable without taking any input variables (when the constraint comes from LIE or from an `instance` declaration with no constraints), or by taking some dictionary variables (which must already be ground, since their constraints must have already been satisfied) and producing a new dictionary from them.

Any variables introduced by the transformation inside a negated context are either bound inside the negation and never used outside of the negation, or are input to the negated goal. Both alternatives preserve mode correctness. This also applies to the conditions of if-then-elses. (Our discussion above omitted the if-then-else and disjunction connectives of Mercury in the interests of brevity. Their rules are very similar to the $\mathcal{CONJ}$ rule.)

All the goals introduced by our transformation (those which construct dictionaries and those that select methods from dictionaries) have exactly one solution. Inserting these goals before other goals has no direct effect on determinism, since the goal G_1, G_2 has exactly the same determinism as G_2 if G_1 always has exactly one solution. It could possibly have an indirect effect if the inserted goal defines the value of a variable that is visible from outside of the conjunction, since this might suppress Mercury's implicit pruning of nondeterministic goals with no outputs. However, the goals introduced by the transformation either produce no variables or produce variables whose only consumer is the transformed original goal, so this cannot happen. Therefore the determinism of a procedure is never changed by our transformation.

In the current version of Mercury, a higher-order value does not stand for a predicate; it stands for a procedure, one mode of a predicate. Therefore the dictionaries we produce have one entry not for each predicate in the interface of the type class, but for each mode of each predicate in the interface of the type class. Therefore, the code that performs a class method call must choose the correct procedure of the called method from the dictionary. This is not a problem, since the mode of the method call is known at compile time.

6 Type checking: proofs and transformations

The Melbourne Mercury compiler has separate phases for type analysis, mode analysis, determinism analysis and uniqueness analysis (in that order), where each phase checks all the declarations of that kind that are present and infers the ones that are missing.

Since, as we have just argued, the transformations called for by the type class rules cannot introduce mode, determinism or uniqueness errors, we could perform these transformation at type checking time. However, this would make it more difficult to issue good error messages during the mode, determinism and uniqueness analysis phases, since messages that refer to a transformed program are hard to interpret. We therefore postpone the type class transformation until after all the semantic analysis phases of the compiler. (However, the introduction of auxiliary predicated for instance methods, described in section 5, is performed *before* any semantic analysis passes so that all semantic analyses are performed on it). Hence while the type analysis phase uses the rules of section 3 to check or infer membership of type classes, it does not perform the associated transformations. Instead, it records a "proof" of each constraint as it is satisfied. This proof records which rule was used to satisfy the constraint, the details of the application of that rule (e.g. which instance declaration was used), and what other constraints were used as assumptions in the proof. Another phase of the compiler, after type, mode determinism and uniqueness analysis, uses these proofs to perform the transformations described by proofs. We have ensured that the phases of the compiler between the creation of these proofs and their use cannot invalidate the proofs.

7 Conclusion

We have proposed a system of type classes for logic programming languages based on the Hindley-Milner type system, providing them with significantly greater expressive power. To provide a semantics for this system of type classes, we have adapted the typing and transformation rules of [2], which work on nested expressions, to the flat structure of logic programs. Although we have not yet done the required proofs, most of the results proven for type classes in functional languages in papers such as [7, 11] should have corresponding results in our system as well.

We have integrated our proposed system of type classes into Mercury. This required resolving the interactions between type classes and Mercury's mode, determinism and uniqueness systems. The solution turned out to be quite elegant. The transformation use to check type correctness of methods in instance declarations turned out to be exactly that required to check mode, determinism and uniqueness correctness.

The type class system described in this paper has been implemented in the Melbourne Mercury compiler, and will be part of version 0.8 of the Mercury distribution. In the meantime, it is available in the releases-of-the-day from `<ftp://turiel.cs.mu.oz.au/pub/mercury/beta-releases>`. Further information about obtaining the Mercury distribution is available at the Mercury home page, `<http://www.cs.mu.oz.au/mercury>`.

References

[1] Andrew Davison. A survey of logic programming-based object-oriented languages. Technical Report 92/3, Department of Computer Science, University of Melbourne, Melbourne, Australia, January 1992.

[2] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[3] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1996.

[4] Pat M. Hill and John W. Lloyd. The Gödel report. Technical Report 91-02, Department of Computer Science, University of Bristol, Bristol, England, 1991.

[5] Mark Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Conference on Functional Programming Languages and Computer Architecture*, June 1993.

[6] Mark Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, May 1994.

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

[8] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop*, volume 788 of *Lecture Notes in Computer Science*. Springer Verlag, June 1997.

[9] Juan José Moreno-Navarro, Julio Mariño, Andrés del Pozo, Angel Herranz-Nieva, and Julio García-Martín. Adding type classes to functional-logic languages. In *Joint Conference on Declarative Programming*, July 1996.

[10] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[11] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, January 1993.

[12] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29 (1-3), October 1996.

[13] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, January 1989.

$$\mathcal{CLAUSE}\ \frac{\begin{array}{l} A = \pi(t_1, \ldots, t_n) \\ (P \text{ of } E)\pi/n = \sigma_\pi \\ E \vdash t_i : \tau_i \rightsquigarrow t'_i \\ \sigma_\pi \cong pred(\tau_i, \ldots, \tau_n) \Leftarrow < \kappa_1\overline{\alpha}_1, \ldots, \kappa_m\overline{\alpha}_m > \\ LIE = \{\kappa_1\overline{\alpha}_1 : v_1, \ldots, \kappa_n\overline{\alpha}_n : v_n\} \\ E \oplus LIE \vdash G \rightsquigarrow G' \\ A' = \pi_n(v_i, \ldots, v_m, t'_1, \ldots, t'_n) \end{array}}{E \vdash A \leftarrow G \rightsquigarrow A' \leftarrow G'}$$

$$\mathcal{NOT}\ \frac{E \vdash G \rightsquigarrow G'}{E \vdash \neg G \rightsquigarrow \neg G'}$$

$$\mathcal{TRUE}\ \frac{}{E \vdash \texttt{true} \rightsquigarrow \texttt{true}}$$

$$\mathcal{CONJ}\ \frac{\begin{array}{c} E \vdash G_1 \rightsquigarrow G'_1 \\ E \vdash G_2 \rightsquigarrow G'_2 \end{array}}{E \vdash G_1, G_2 \rightsquigarrow G'_1, G'_2}$$

$$\mathcal{ATOM}\ \frac{\begin{array}{l} (P \text{ of } E)\pi/n = \forall\alpha_1, \ldots, \alpha_m.pred(\tau_{\pi 1}, \ldots, \tau_{\pi n}) \Leftarrow \theta_\pi \\ E \vdash t_i : \tau_i \rightsquigarrow t'_i \\ < \tau_1, \ldots, \tau_n > = < \tau_{\pi 1}, \ldots, \tau_{\pi n} > [\tau_{a1}/\alpha_1, \ldots, \tau_{am}/\alpha_m] \\ E \vdash \theta_\pi[\tau_{a1}/\alpha_1, \ldots, \tau_{am}/\alpha_m] \rightsquigarrow \lambda v_1, \ldots, v_k.G \end{array}}{E \vdash \pi(t_1, \ldots, t_n) \rightsquigarrow G, \pi_n(v_1, \ldots, v_k, t'_1, \ldots, t'_n)}$$

$$\mathcal{HO}-\mathcal{CALL}\ \frac{\begin{array}{c} E \vdash t_i : \tau_i \rightsquigarrow t'_i \\ E \vdash v_i : pred(\tau_1, \ldots, \tau_n) \end{array}}{E \vdash \texttt{call}(v, t_1, \ldots, t_n) \rightsquigarrow \texttt{call}(v, t'_1, \ldots, t'_n)}$$

Figure 4: Typing/transformation rules for goals

$$\mathcal{TERM}\frac{\begin{array}{l}(F \text{ of } E)f = \tau_1 \to \ldots \to \tau_n \to \tau \\ E \vdash t_i : \tau_i \leadsto t'_i\end{array}}{E \vdash f(t_1, \ldots, t_n) : \tau \leadsto f(t'_1, \ldots, t'_n)}$$

$$\mathcal{VAR}\frac{(Vof\ E)v = \tau}{E \vdash v : \tau \leadsto v}$$

$$\mathcal{HO}-\mathcal{CONST}\frac{\begin{array}{l}VE = \{v_1 : \tau_1, \ldots, v_n : \tau_n\} \\ E \oplus VE \vdash G \leadsto G'\end{array}}{E \vdash \lambda v_1 \ldots v_n.G : pred(\tau_1, \ldots \tau_n) \leadsto \lambda v_1 \ldots v_n.G'}$$

Figure 5: Typing/transformation rules for values

$$\mathcal{DICT}-\mathcal{TAUT}-\mathcal{LIE}\frac{(LIE \text{ of } E)\kappa\overline{\alpha} = v}{E \vdash \kappa\overline{\alpha} \leadsto \lambda v'.v = v'}$$

$$\mathcal{DICT}-\mathcal{TAUT}-\mathcal{IE}\frac{(IE \text{ of } E)\forall\overline{\alpha}.\kappa\overline{\tau} \Leftarrow \theta \cong \lambda\overline{v}.G}{E \vdash \forall\overline{\alpha}.\kappa\overline{\tau} \Leftarrow \theta \leadsto \lambda\overline{v}.G}$$

$$\mathcal{DICT}-\mathcal{SPEC}\frac{E \vdash \forall\alpha_1 \ldots \alpha_n.\kappa\overline{\tau} \Leftarrow \theta \leadsto \lambda\overline{v}.G}{E \vdash \kappa\overline{\tau} \Leftarrow \theta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \leadsto \lambda\overline{v}.G}$$

$$\mathcal{DICT}-\mathcal{REL}\frac{\begin{array}{l}E \vdash \kappa\overline{\tau} \Leftarrow < \kappa_1\overline{\tau}_1, \ldots, \kappa_n\overline{\tau}_n > \leadsto \lambda v_1, \ldots, v_n, v.G \\ E \vdash \kappa_i\overline{\tau}_i \leadsto \lambda v_i.G_i\end{array}}{E \vdash \kappa\overline{\tau} \leadsto \lambda v.(G_1, \ldots, G_n, G)}$$

$$\mathcal{DICT}-\mathcal{SEQ}\frac{E \vdash \kappa_i\overline{\tau}_i \leadsto \lambda v_i.G_i}{E \vdash < \kappa_1\overline{\tau}_1, \ldots, \kappa_n\overline{\tau}_n > \leadsto \lambda v_1, \ldots, v_n.(G_1, \ldots, G_n)}$$

Figure 6: Typing/transformation rules for type class constraints

$$\mathcal{PRED}\frac{\begin{array}{l}\overline{\alpha} = \text{type variables in } \tau_1, \ldots \tau_n \\ P = \{\pi/n = \forall\overline{\alpha}.pred(\tau_1, \ldots \tau_n) \Leftarrow \theta\}\end{array}}{\begin{array}{l}E \vdash \texttt{pred } \pi(\tau_1, \ldots, \tau_n) \Leftarrow \theta \\ \leadsto \\ E \oplus P\end{array}}$$

$$\mathcal{TYPE}\frac{\begin{array}{l}\text{all type variables in } \tau_{1,1} \ldots, \tau_{n,m_n} \text{ are in } \overline{\alpha} \\ F = \{f_i/m_i = \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to \chi\overline{\alpha}\}\end{array}}{\begin{array}{l}E \vdash \texttt{type } \chi\overline{\alpha} \to f_1(\tau_{1,1}, \ldots \tau_{1,m_1}); \ldots; f_n(\tau_{n,1}, \ldots \tau_{n,m_n}) \leadsto \\ E \oplus F\end{array}}$$

Figure 7: Typing/transformation rules for `type` and `pred` declarations

$$\mathcal{TYPECLASS}\frac{\begin{array}{l}\sigma_i = pred(\overline{\tau_i}) \Leftarrow \theta_i \\ \overline{\tau}_i \text{ is a sequence of } m_i \text{ types} \\ CE = \{\kappa/k : \\ \quad \forall\overline{\alpha}. < \pi_1/m_1 : \sigma_1, \ldots, \pi_n/m_n : \sigma_n > \Leftarrow (\theta_{\kappa 1}, \ldots \theta_{\kappa l})\} \\ P = \{\pi_i/m_i = pred(\overline{\tau_i}) \Leftarrow (\theta_i, \kappa\overline{\alpha})\} \\ IE = \{\theta_{\kappa i} \Leftarrow \kappa\overline{\alpha} : \lambda v_1, v_2.\texttt{project}_i^{l+n}(v_1, v_2)\} \\ \theta_i \text{ contains } k_i \text{ constraints}\end{array}}{\begin{array}{l}E \vdash \texttt{typeclass } \kappa\overline{\alpha} \Leftarrow (\theta_{\kappa 1}, \ldots \theta_{\kappa l}) \texttt{ where [} \\ \quad \texttt{pred } \pi_1(\overline{\tau}_1) \Leftarrow \theta_1, \ldots, \texttt{pred } \pi_n(\overline{\tau}_n) \Leftarrow \theta_n \\ ] \\ \leadsto \\ E \oplus CE \oplus P \oplus IE, \\ \quad \{\pi_i(v, v_1, \ldots, v_{k_i}, v'_1, \ldots, v'_{m_i}) \leftarrow \quad \texttt{project}_{l+i}^{l+n}(v, v_{imp}), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{call}(v_{imp}, v_1, \ldots, v_{k_i}, v'_1, \ldots, v'_{m_i})\}\end{array}}$$

Figure 8: Typing/transformation rules for `typeclass` declarations

$$
\mathcal{INSTANCE}\;\frac{\begin{array}{l}
(CE \text{ of } E)\kappa/k = \\
\quad \forall \alpha_1 \ldots \alpha_k. < \pi_1/n_1 : \sigma_1, \ldots, \pi_m/n_m : \sigma_m > \Leftarrow (\theta'_1, \ldots, \theta'_{l'}) \\
\sigma_i = pred(\overline{\tau}_i) \Leftarrow \theta_i \\
(P \text{ of } E)\pi'_i/n_i = pred(\overline{\tau}'_i) \Leftarrow \theta_{\pi'_i} \\
\overline{\tau}'_i = \overline{\tau}_i[\chi_1\overline{\alpha}_1/\alpha_1, \ldots, \chi_k\overline{\alpha}_k/\alpha_k] \\
LIE = \{v_1 : \theta_1, \ldots, v_l : \theta_l\} \\
E \oplus LIE \vdash \lambda v'_i, \ldots, v'_{n_i}.\pi'_i(v'_i, \ldots, v'_{n_i}) \rightsquigarrow G_i \\
E \oplus LIE \vdash \theta'_i[\chi_1\overline{\alpha}_1/\alpha_1, \ldots, \chi_k\overline{\alpha}_k/\alpha_k] \rightsquigarrow (G_{si}, < v_{si} >) \\
IE = \{\kappa\chi_1\overline{\alpha}_1 \ldots \chi_k\overline{\alpha}_k : \\
\quad \lambda v_1, \ldots, v_l, v_{out}. \\
\qquad G_{s1}, \ldots G_{sn}, \\
\qquad v_{out} = \texttt{dictionary}(v_{s1}, \ldots, v_{sl'}, G_1, \ldots, G_m).\}
\end{array}}{\begin{array}{l}
E \vdash \texttt{instance } \kappa\chi_1\overline{\alpha}_1 \ldots \chi_k\overline{\alpha}_k \Leftarrow (\theta_1, \ldots, \theta_l) \texttt{ where [} \\
\quad \pi_1/n_1 \texttt{ is } \pi'_1, \ldots, \quad \pi_m/n_m \texttt{ is } \pi'_m \\
] \\
\rightsquigarrow E \oplus IE
\end{array}}
$$

Figure 9: Typing/transformation rules for `instance` declarations