Type classes for logic programming languages

David Jeffery
Supervisor: Zoltan Somogyi

A motivating “object-oriented” example

```
interface x_y_point {

    public abstract float get_x();
    public abstract float get_y();

    public abstract void set_x();
    public abstract void set_y();
}
```

A motivating example

Can write functions which take parameters of some *abstract base class*. We don't care how they're implemented.

```
void draw_x_y_point(x_y_point the_point)
{
    float x, y;

    x = the_point.get_x();
    y = the_point.get_x();

    /* code to draw the point */
    ...
}
```

Instantiating the class with a representation

```
class cartesian implements x_y_point {

    public float get_x() { return x; }
    public float get_y() { return y; }

    public void set_x(float newx) { x = newx; }
    public void set_y(float newy) { y = newy; }

    private float x;
    private float y;
}
```

Another instantiation

```
class polar implements x_y_point {

    public float get_x() { return r*cos(theta); }
    public float get_y() { return r*sin(theta); }

    public void set_x(float newx) { ... }
    public void set_y(float newy) { ... }

    private float r;
    private float theta;
}
```

How can we do that in Mercury?

The Hindley-Milner type system

Used in ML, Miranda, Mercury, Gödel, NU-Prolog type checker, etc.

```
    % declaring a new type
:- type tree(T)
        --->  null
        ;     node(tree(T), T, tree(T)).

    % declaring a predicate's type
:- pred insert_into_tree(T, tree(T), tree(T)).
```

Polymorphism under Hindley-Milner

```
:- pred append(list(T), list(T), list(T)).

append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :-
        append(Xs, Ys, Zs).
```

Values with polymorphic types (eg. `X` above) cannot have any operations performed on them.

Type classes

Haskell, a pure functional language, has a more expressive type system. It has *type classes*:

- allow us to perform operations on variables whose type is polymorphic by allowing type variables to be *constrained* to belong to a "class".

- still strongly typed

- can write things similar to our motivating example above (and more!)

Type classes in Mercury

```
:- 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)
].
```

`instance` declarations

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

:- 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
].

:- pred cart_get_x(cartesian, float).
cart_get_x(cart(X, _), X).

:- pred cart_get_y(cartesian, float).
cart_get_y(cart(_, Y), Y).

:- pred cart_set_x(cartesian, float, cartesian).
cart_set_x(cart(_, Y), X, cart(X, Y)).

:- pred cart_set_y(cartesian, float, cartesian).
cart_set_y(cart(X, _), Y, cart(X, Y)).
```

Constraints on predicate types:

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

draw_x_y_point(Point, ...) :-
    get_x(Point, X),
    ...
```

Constraints on `typeclass` decls
(super classes):

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

Constraints on `instance` decls (implied instances):

```
:- instance printable(list(T)) <= printable(T)
        where [
    pred(print/3) is print_list
].

:- pred print_list(list(T), ...) <= printable(T).

print_list([], ...).
print_list([X|Xs], ...) :-
    print(X),
    write(", "),
    print_list(Xs, ...).
```

When we print a list , the correct print implementation for the elements is chosen at run-time.

Type correctness of type classes

Type class constraints for a call can be satisfied in three ways:

1. Constraint is on type variable(s), and the constraint also appears in the type signature of the caller.

2. Constraint is on type(s) (not type variable(s)), and an instance declaration can be applied. Implied constraints must also be satisfied.

3. Constraint has a super class constraint that can be satisfied

A source-to-source transformation

Pass "dictionaries" or "type-class-infos" which contain the implementations of all the class' methods to be called.

1. Constraint is satisfied by caller's type signature: pass the dictionary directly from the head variables

2. Constraint is satisfied by applying instance declaration: get the dictionary corresponding to the instance declaration

3. Constraint is satisfied by using super class relation: extract super class dictionary from sub class

Transformation of our example

The `draw_x_y_point` example above gets transformed to:

```
draw_x_y_point(Dictionary, Point, ...) :-
    get_x(Dictionary, Point, X),
    ...

get_x(Dictionary, Point, X) :-
    Dictionary = dictionary(GetX, _GetY, ...),
    call(GetX, Point, X).
```

Transformation of `instance` declarations

```
instance_cartesian_0_x_y_point(
        dictionary(
            cart_get_x,
            cart_get_y,
            cart_set_x,
            cart_set_y
        )
    ).
```

This gets called in situations such as:

```
    ...
    A = cart(1.3, 7.4),
    instance_cartesian_0_x_y_point(Dictionary),
    draw_x_y_point(Dictionary, A, ...),
    ...
```

Transformation of implied instances

```
instance_list_1_printable(ElementDictionary,
        dictionary(
            print_list(ElementDictionary)
        )
    ).
```

This gets called in situations such as:

```
    ...
    Xs = [Elem1, Elem2, Elem3],
    instance_list_1_printable(ElemDictionary,
        Dictionary),
    print(Dictionary, List, ...),
    ...
```

Transformation of super classes

```
superclass_coloured_point_x_y_point(
        dictionary(
            XYPointDictionary,
            _get_colour
        ),
        XYPointDictionary
    ).
```

This gets called in situations such as:

```
draw_colour_point(ColourDict, CP, ...) :-
    get_colour(ColourDict, CP, Colour),
    set_pen_colour(Colour, ...),
    superclass_coloured_point_x_y_point(
        ColourDict, XYDict),
    draw_x_y_point(XYDict, CP, ...).
```

Type classes, modes and determinism

- Need to ensure that instance methods have the correct *modes* defined on them, and that the corresponding *determinism* is correct.

- The dictionary contains *procedures* not *predicates*.

- Since we do the transformation as a separate step to type checking, we retain a "proof" of constraint satisfaction so that the transformation can be done without redoing type checking

Conclusion

- Type systems for LP languages can be made more expressive by including type classes, and the mechanism integrates well with the rest of the language

- Provides *interface inheritance* that is found in "object-oriented" languages
  - Abstraction mechanism
  - Code re-use (of predicates which have constrained polymorphic arguments)

- Semantics is simple

- All of Mercury's static checks are still possible