Inheritance

The inheritance mechanisms found on object-oriented programming languages allow us the specialization of previously defined objects, avoiding the unnecessary repetition of code. In the context of logic programming, we can interpret inheritance as a form of theory extension: an object will virtually contain, besides its own predicates, all the predicates inherited from other objects that are not redefined by itself.

Protocol inheritance

Protocol inheritance refers to the inheritance of predicate declarations (scope directives). These can be contained in objects, in protocols, or in categories. Logtalk supports single and multi-inheritance of protocols: an object or a category may implement several protocols and a protocol may extend several protocols.

Search order for prototype hierarchies

The search order for predicate declarations is first the object, second the implemented protocols (and the protocols that these may extend), third the imported categories (and the protocols that they may implement), and last the objects that the object extends. This search is performed in a depth-first way. When an object inherits two different declarations for the same predicate, by default only the first one will be considered.

Search order for class hierarchies

The search order for predicate declarations starts in the object classes. Following the classes declaration order, the search starts in the classes implemented protocols (and the protocols that these may extend), third the classes imported categories (and the protocols that they may implement), and last the superclasses of the object classes. This search is performed in a depth-first way. If the object inherits two different declarations for the same predicate, by default only the first one will be considered.

Implementation inheritance

Implementation inheritance refers to the inheritance of predicate definitions. These can be contained in objects or in categories. Logtalk supports multi-inheritance of implementation: an object may import several categories or extend, specialize, or instantiate several objects.

Search order for prototype hierarchies

The search order for predicate definitions is similar to the search for predicate declarations except that implemented protocols are ignored (they can only contain predicate directives).

Search order for class hierarchies

The search order for predicate definitions is similar to the search for predicate declarations except that implemented protocols are ignored (they can only contain predicate directives).

Inheritance versus predicate redefinition

When we define a predicate that is already inherited from other object, the inherited definitions are hidden by the new definitions. This is called overriding inheritance: a local definition overrides any inherited ones. For example, assume that we have the following two objects:

```
:- object(root).

    :- public(bar/1).
    :- public(foo/1).

    bar(root).

    foo(root).

:- end_object.


:- object(descendant,
    extends(root)).

    foo(descendant).

:- end_object.
```

After compiling and loading these objects, we can check the overriding behavior by trying the following queries:

```
| ?- root::(bar(Bar), foo(Foo)).

Bar = root
Foo = root
yes


| ?- descendant::(bar(Bar), foo(Foo)).

Bar = root
Foo = descendant
yes
```

However, we can explicitly program other behaviors. Let us see a few examples.

Specialization inheritance

Specialization of inherited definitions: the new definition uses the inherited definitions, adding to this new code. This is accomplished by calling the `^^/1` operator in the new definition.

```
:- object(root).

    :- public(init/0).

    init :-
        write('root init'), nl.

:- end_object.


:- object(descendant,
    extends(root)).

    init :-
        write('descendant init'), nl,
        ^^init.

:- end_object.


| ?- descendant::init.

root init
descendant init

yes
```

Union inheritance

Union of the new with the inherited definitions: all the definitions are taken into account, the calling order being defined by the inheritance mechanisms. This can be accomplished by writing a clause that just calls, using the `^^/1` operator, the inherited definitions. The relative position of this clause among the other definition clauses sets the calling order for the local and inherited definitions.

```
:- object(root).

    :- public(foo/1).

    foo(1).
    foo(2).

:- end_object.


:- object(descendant,
    extends(root)).

    foo(3).
    foo(Foo) :-
        ^^foo(Foo).

:- end_object.


| ?- descendant::foo(Foo).

Foo = 3 ;
Foo = 1 ;
Foo = 2 ;

no
```

Selective inheritance

Hiding some of the inherited definitions, or differential inheritance: this form of inheritance is normally used in the representation of exceptions to generic definitions. Here we will need to use the `^^/1` operator to test and possibly reject some of the inherited definitions.

```
:- object(bird).

    :- public(mode/1).

    mode(walks).
    mode(flies).

:- end_object.


:- object(penguin,
    extends(bird)).

    mode(swims).
    mode(Mode) :-
        ^^mode(Mode),
        Mode \= flies.

:- end_object.


| ?- penguin::mode(Mode).

Mode = swims ;
Mode = walks ;

no
```

Public, protected, and private inheritance

To make all public predicates declared via implemented protocols, importeds categories, or inherited objects protected or to make all public and protected predicates private we prefix the entity's name with the corresponding keyword. For instance:

```
:- object(Object,
    implements(private::Protocol)).       % all the Protocol public and protected
    ...                                   % predicates become Object's private
:- end_object.                            % predicates
```

or:

```
:- object(Class,
    specializes(protected::Superclass)).  % all the Superclass public predicates
    ...                                   % become Object's protected predicates
:- end_object.
```

Omitting the scope keyword is equivalent to using the public scope keyword. For example:

```
:- object(Object,
    imports(public::Category)).
    ...
:- end_object.
```

This is the same as:

```
:- object(Object,
    imports(Category)).
    ...
:- end_object.
```

This way we ensure backward compatibility with older Logtalk versions and a simplified syntax when protected or private inheritance are not used.

Composition versus multiple inheritance

It is not possible to discuss inheritance mechanisms without referring to the long and probably endless debate on single versus multiple inheritance. The single inheritance mechanism can be implemented in an very efficient way, but it imposes several limitations on reusing, even if the multiple characteristics we intend to inherit are orthogonal. On the other hand, the multiple inheritance mechanisms are attractive in their apparent capability of modeling complex situations. However, they include a potential for conflict between inherited definitions whose variety does not allow a single and satisfactory solution for all the cases.

Until now, no solution that we might consider satisfactory for all the problems presented by the multiple inheritance mechanisms has been found. From the simplicity of some extensions that use the Prolog search strategy like [McCabe 92] or [Moss 94] and to the sophisticated algorithms of CLOS [Bobrow 88], there is no adequate solution for all the situations. Besides, the use of multiple inheritance carries some complex problems in the domain of software engineering, particularly in the reuse and maintenance of the applications. All these problems are substantially reduced if we preferably use in our software development composition mechanisms instead of specialization mechanisms [Taenzer 89]. Multiple inheritance can and should be seen more as a useful analysis and project abstraction, than as an implementation technique [Shan 93]. Logtalk provides first-class support for software composition using *categories*.

Nevertheless, Logtalk supports multi-inheritance by enabling an object to extend, instantiate, or specialize more than one object. The current Logtalk release provides a predicate directive, `alias/3`, which may be used to solve some multi-inheritance conflicts. Lastly, it should be noted that the multi-inheritance support does not compromise performance when we use single-inheritance.