Programming Patterns for Logtalk Parametric Objects

Paulo Moura

Dep. of Computer Science, University of Beira Interior, Portugal
Center for Research in Advanced Computing Systems,
INESC Porto, Portugal
pmoura@di.ubi.pt

Abstract. This paper presents a survey of programming patterns for Logtalk *parametric objects*. A parametric object is an object whose identifier is a compound term containing logical variables. These variables play the role of object parameters. Object predicates can be coded to depend on the parameter values. Parametric objects are a common feature of some other object-oriented logic programming languages and Prolog object-oriented extensions. Logtalk extends the usefulness of parametric objects by introducing the concept of *object proxies*. An object proxy is a compound term that can be interpreted as a possible instantiation of the identifier of a parametric object. Object proxies, when represented as predicate facts, allow application memory footprint to be minimized while still taking full advantage of Logtalk object-oriented features for representing and reasoning with taxonomic knowledge.

Keywords: logic programming, programming patterns, parametric objects, object proxies.

1 Introduction

Logtalk [1–3] is an object-oriented logic programming language that can use most Prolog implementations as a back-end compiler. Logtalk extends Prolog with versatile code encapsulation and code reuse constructs based on an interpretation of object-oriented concepts in the context of logic programming. Logtalk features *objects* (both *prototypes* and *classes*), *static* and *dynamic binding* (with predicate lookup caching), *single* and *multiple inheritance*, *protocols* (aka interfaces; sets of predicate declarations that can be implemented by any object), *categories* (fine-grained units of code reuse that can be used as object building blocks), *event-driven programming*, and *high-level multi-threading programming* (and-parallelism and competitive or-parallelism). Objects, protocols, and categories can be either static or dynamic. The use of static entities and static binding results in performance similar to plain Prolog.

Logtalk includes support for *parametric objects*, a feature common to some object-oriented logic programming languages and Prolog object-oriented extensions. A parametric object is an object whose identifier is a compound term containing logical variables. A simple example of a parametric object for representing two-dimensional geometric points could be:

```
:- object(point(_X, _Y)).

:- end_object.
```

The variables in the object identifier play the role of object parameters. Object predicates can be coded to depend on the parameter values. For example, assuming that we needed to compute the distance of a point to the origin, we could write:

```
:- object(point(_X, _Y)).

    :- public(distance/1).
    distance(Distance) :-
        parameter(1, X),
        parameter(2, Y),
        Distance is sqrt(X*X + Y*Y).

:- end_object.
```

After compiling this parametric object we could try queries such as:

```
| ?- point(3.0, 4.0)::distance(Distance).

Distance = 5.0
yes
```

Thus, a parametric object can be regarded as a generic object from which specific *instantiations* can be derived by instantiating the object parameters at runtime. Note, however, that instantiating object parameters does not create new objects. For example, the object identifiers `point(2.0, 1.2)` and `point(3.0, 4.0)` refer to the same parametric object. Parameter instantiation usually takes place when a message is sent to the object. By using logical variables, parameter instantiation is undone on backtracking. Object parameters can be any valid term: logical variables, constraint variables, atomic terms, or compound terms.[1]

Despite the simplicity of the concept, parametric objects proved a valuable asset in a diversity of applications. Parametric objects support several useful programming patterns, presented and illustrated in this paper. The remainder of the paper is organized as follows. Section 2 describes Logtalk built-in methods for accessing object parameters. Section 3 illustrates parameter passing within object hierarchies. Section 4 presents useful parametric object programming patterns, illustrated by examples. Section 5 presents and discusses the concept of *object proxies*, introducing additional programming patterns. Section 6 shows how to use both object proxies and regular objects to represent entities of the same *type* that differ on complexity or size. Section 7 discusses related work. Section 8 presents our conclusions.

[1] Logtalk also supports parametric categories, thus extending the advantages of parametric objects when using programming solutions based on category- based composition.

2 Accessing Object Parameters

Logtalk provides a `parameter/2` built-in method for accessing object parameters. The first argument is the parameter position. The second argument is the object parameter. For example:

```
:- object(ellipse(_Rx, _Ry, _Color)).

    area(Area) :-
        parameter(1, Rx),
        parameter(2, Ry),
        Area is Rx*Ry*pi.
    ...
```

Logtalk compiles the calls to the method `parameter/2` in-line by unifying the second argument with the corresponding object identifier argument in the execution context argument in the translated clause head. A second built-in method, `this/1`, allows access to all object parameters at once:

```
    area(Area) :-
        this(ellipse(Rx, Ry, _)),
        Area is Rx*Ry*pi.
```

As with the `parameter/2` method, calls to the `this/1` method are compiled in-line.

An alternative to the `parameter/2` and `this/1` methods for accessing object parameters would be to interpret parameters as logical variables with global scope within the object. This solution was discarded as it forces the programmer to always be aware of parameter names when coding object predicates. It also collides with the local scope of Logtalk and Prolog variables in directives and predicate clauses.

3 Parameter Passing

Logtalk objects may *extend* (defining parent-prototype relations), *instantiate* (defining instance-class relations), or *specialize* (defining class-superclass relations) other objects. When using parametric objects, parameter passing is established through unification in the object opening directive. As an example, consider the following Logtalk version of a SICStus Objects [4] example:

```
:- object(ellipse(_RX, _RY, _Color)).

    :- public([color/1, rx/1, ry/1, area/1]).

    rx(Rx) :-
        parameter(1, Rx).

    ry(Ry) :-
        parameter(2, Ry).
    color(Color) :-
        parameter(3, Color).

    area(Area) :-
        this(ellipse(Rx, Ry, _)),
        Area is Rx*Ry*pi.

:- end_object.

:- object(circle(Radius, Color),              % circles are ellipses
    extends(ellipse(Radius, Radius, Color))).  % where Rx = Ry

    :- public(r/1).
    r(Radius) :-
        parameter(1, Radius).

:- end_object.

:- object(circle1(Color),
    extends(circle(1, Color))).

:- end_object.

:- object(red_circle(Radius),
    extends(circle(Radius, red))).

:- end_object.
```

A query such as:

```
| ?- red_circle(3)::area(Area).

Area = 28.274334
yes
```

will result in the following instantiation chain of the parametric object identifiers:

```
red_circle(3) -> circle(3, red) -> ellipse(3, 3, red)
```

Note that the predicate `area/1` is declared and defined in the object representing ellipses.

4 Programming Patterns

This section presents a survey of useful parametric object programming patterns. These patterns build upon the basic idea that a parametric object encapsulates a set of predicates for working with compound terms that share the same functor and arity. Additional programming patterns will be described later in this paper when presenting the Logtalk concept of parametric object proxies. The full source code of most examples is included in the current Logtalk distribution.

4.1 Using Parameters for Passing Typing Information

Parameters may be used to represent *type* information. This usage provides functionality akin to e.g. C++ templates or Java generics. There are, however, two important differences. First, the Logtalk runtime parameterization does not create a new object. Second, Logtalk provides runtime parameterization instead of compile-time parameterization. The canonical example is sorting a list of elements of a given type:

```
:- object(sort(_Type)).

    :- public(sort/2).
    sort([], []).
    sort([Pivot| Rest], Sorted) :-
        partition(Rest, Pivot, Small, Large),
        sort(Small, SortedSmall),
        sort(Large, SortedLarge),
        list::append(SortedSmall, [Pivot| SortedLarge], Sorted).

    partition([], _, [], []).
    partition([X| Xs], Pivot, Small, Large) :-
        parameter(1, Type),
        (   (Type::(X < Pivot)) ->
            Small = [X| Small1], Large = Large1
        ;   Small = Small1, Large = [X| Large1]
        ),
        partition(Xs, Pivot, Small1, Large1).

:- end_object.
```

Assuming a `rational` object implementing the Logtalk library `comparing` protocol, we can use queries such as:

```
| ?- sort(rational)::sort([1/8, 2/7, 6/5, 2/9, 1/3], Sorted).

Sorted = [1/8, 2/9, 2/7, 1/3, 6/5]
yes
```

Or simply use the pseudo-object *user*, thus using the Prolog built-in comparison predicates:

```
| ?- sort(user)::sort([3, 1, 4, 2, 9], Sorted).

Sorted = [1, 2, 3, 4, 9]
yes
```

4.2 Customizing Object Predicates

Parameters may be used to customize object predicates, improving code reusing by avoiding repeating similar predicate definitions. For example, assume that we want to define implementations of both *max-heaps* and *min-heaps* (also know as *priority queues*). Inserting a new element on a heap requires comparing it with the existing heap elements in order to ensure that the queue front element is always the one with the top priority. For a max-heap, this will be the element with the maximum priority while for a min-heap this will be the element with the minimum priority. We can use a parameter to represent *how* to compare heap elements:

```
:- object(heap(_Order)).

    % generic heap handling predicates containing calls such as:
    % ..., parameter(1, Order), compare(Order, X, Y), ...

:- end_object.
```

We could use this object directly but we can also define two handy shortcuts:

```
:- object(minheap,
    extends(heap(<))).

:- end_object.


:- object(maxheap,
    extends(heap(>))).

:- end_object.
```

This programing pattern illustrates how we can use parameters to customize object predicates by passing the *operations* to be invoked within those predicates. These operations can be either closures (as in the example above) or goals, linking this programming pattern to the concepts of meta-programming and meta-predicates.

4.3 Simplifying Object Interfaces

Parametric objects can simplify object interfaces by moving core object properties from predicate arguments to object parameters, that is, object parameters can be used to represent object *data*. This also makes the core properties visible without requiring the definition of *accessor* predicates to retrieve them. Consider the following parametric object representing rectangles:

```
:- object(rectangle(_Width, _Height)).

    :- public(area/1).
    area(Area) :-
        this(rectangle(Width, Height)),
        Area is Width*Height.

    :- public(perimeter/1).
    perimeter(Perimeter) :-
```

```
        this(rectangle(Width, Height)),
        Perimeter is 2*(Width + Height).

:- end_object.
```

The rectangle properties *width* and *height* are always accessible. The two rectangle predicates, `area/1` and `perimeter/1`, have a single argument returning the respective computed values.

Assume that our application would also need to compute areas and perimeters of circles, triangles, pentagons, and other (regular or non-regular) polygons. The alternative of using non parametric objects to encapsulate the same functionality would require adding extra arguments to both predicates that would depend on the type of shape. We could use instead a single *data* argument that would accept a compound term representing a shape but that would result in a awkward solution for encapsulating the knowledge about each type of shape in its own object. Another alternative would be to use predicates such as `area_circle/1` and `perimeter_triangle/1` but this solution is hard to extend to new shapes, to new predicates over shapes, and would hinder processing of heterogenous collections of shapes. Parametric objects provide us with a simple, clean, logical, and easily extensible knowledge representation solution.

4.4 Data-Centric Programming

Parametric objects can be seen as enabling a more data-centric programming style where data is represented by *instantiations* of parametric object identifiers. Instead of using a term as a predicate argument, predicates can be called by sending the corresponding message to the term itself.

To illustrate this pattern we will use the well-known example of symbolic differentiation and simplification of arithmetic expressions, which can be found in [5]. The L&O [6] system also uses this example to illustrate parametric theories. The idea is to represent arithmetic expressions as parametric objects whose name is the expression operator with greater precedence, and whose parameters are the operator sub-expressions (that are, themselves, objects). In order to simplify this example, the object methods will be restricted to symbolic differentiation of polynomials with a single variable and integer coefficients. In addition, we will omit any error-checking code. The symbolic simplification of arithmetic expressions could easily be programmed in a similar way.

For an arithmetic expression reduced to a single variable, x, we will have the following object:

```
:- object(x,
    implements(diffp)).  % protocol (interface) declaring a diff/1
                         % symbolic differentiation predicate
    diff(1).

:- end_object.
```

Arithmetic addition, $x + y$, can be represented by the parametric object `'+'(X, Y)` or, using operator notation, `X + Y`. Taking into account that the operands can either be numbers or other arithmetic expressions, a possible definition will be:

```
:- object(_ + _,
    implements(diffp)).

    diff(Diff) :-
        this(X + Y),
        diff(X, Y, Diff).

    diff(I, J, 0) :-
        integer(I), integer(J), !.

    diff(X, J, DX) :-
        integer(J), !, X::diff(DX).

    diff(I, Y, DY) :-
        integer(I), !, Y::diff(DY).

    diff(X, Y, DX + DY) :-
        X::diff(DX), Y::diff(DY).

:- end_object.
```

The object definitions for other simple arithmetic expressions, such as `X - Y` or `X * Y`, are similar. The expression x^n can be represented by the parametric object `X ** N` as follows:

```
:- object(_ ** _).

    :- public(diff/1).

    diff(N * X ** N2) :-
        this(X ** N),
        N2 is N - 1.

:- end_object.
```

By defining a suitable parametric object per arithmetic operator, any polynomial expression can be interpreted as an object identifier. For example, the polynomial `2*x**3 + x**2 - 4*x` $(2x^3 + x^2 - 4x)$ will be interpreted as `(2*x**3) + (x**2 - 4*x)` $((2x^3) + (x^2 - 4x))$. Thus, this expression is an *instantiation* of the `X + Y` parametric object identifier, allowing us to write queries such as:

```
| ?- (2*x**3 + x**2 - 4*x)::diff(D).

D = 2* (3*x**2)+2*x**1-4*1
yes
```

The resulting expression could be symbolically simplified using a predicate defined in the same way as the `diff/1` differentiation predicate.

4.5 Restoring Shared Constraint Variables

Object parameters can be used to restore shared variables between sets of constraints that are encapsulated in different objects. We illustrate this pattern using an example that is loosely based in the declarative process modeling research project ESProNa [7]. Each process can be described by a parametric object, implementing a common protocol (reduced here to a single predicate declaration for the sake of this example):

```
:- protocol(process_description).

    :- public(domain/2).
    :- mode(domain(-list(object_identifier), -callable), one).
    :- info(domain/2, [
        comment is 'Returns the process dependencies and constraints.',
        argnames is ['Dependencies', 'Constraints']]).

    % other process description predicates

:- end_ protocol.
```

Part of the process description is a set of finite domain constraints describing how many times the process can or must be executed. An object parameter is used to provide access to the process constraint variable. For example, the following process, `a(_)`, can be executed two, three, or four times:

```
:- object(a(_),
    implements(process_description)).

    domain([], (A #>= 2, A #=< 4)) :-
        parameter(1, A).

:- end_object.
```

Processes may depend on other processes. These dependency relations can be described using a list of parametric object identifiers. For example:

```
:- object(b(_),
    implements(process_description)).

    domain([a(A)], (B #>= A, B #=< 3)) :-
        parameter(1, B).

:- end_object.

:- object(c(_),
    implements(process_description)).

    domain([a(A), b(B)], (C #= B + 1, C #= A + 1)) :-
        parameter(1, C).

:- end_object.
```

Thus, the parametric object identifiers in the dependencies list allows us to relate the different process constraint variables.

The object parameters allows us to restore shared constraint variables at runtime in order to model sets of inter-dependent processes. For example (using Logtalk with GNU Prolog, with its native finite domain solver, as the back-end compiler):

```
| ?- process_model::solve([c(C)], Dependencies).

C = _#59(3..4)
Dependencies = [b(_#21(2..3)),a(_#2(2..3)),c(_#59(3..4))]
yes
```

The predicate `solve/2` (the code of the `process_model` object is omitted due to the lack of page space) computes this answer by retrieving and solving the conjunction of the constraints of processes `a(_)`, `b(_)`, and `c(_)`.

4.6 Logical Updates of Object State

Parametric objects provide an alternative to represent object dynamic state. This alternative supports logical updates, undone by backtracking, by representing object state using one or more object parameters. A similar solution was first used in the OL(P) [8]. Logtalk supports this solution using a pure logical subset implementation of *assignable variables* [9], available in the library `assignvars`.[2] This library defines *setter* and *getter* methods (`<=/2` and `=>/2`, respectively) and an initialization method (`assignable/2`) to work with assignable variables.

Consider the following simple example, where a parametric object is used to describe a geometric rectangle. The object state is comprised by its read-only dimensions, *width* and *height*, and by its dynamically updatable *position*:

```
:- object(rectangle(_Width, _Height, _Position),
    imports(private::assignvars)).

    :- public([init/0, area/1, move/2, position/2]).

    init :-
        parameter(3, Position),
        ::assignable(Position, (0, 0)).

    area(Area) :-
        this(rectangle(Width, Height, _)),
        Area is Width*Height.

    move(X, Y) :-
        parameter(3, Position),
        ::Position <= (X, Y).
```

[2] This library is implemented as a *category* [1], a fine-grained unit of code reuse that can be virtually imported by any object without code duplication.

```
    position(X, Y) :-
        parameter(3, Position),
        ::Position => (X, Y).

:- end_object.
```

A sample query could be:

```
| ?- rectangle(2, 3, _)::(init, area(Area), position(X0, Y0),
     move(1, 1), position(X1, Y1), move(2, 2), position(X2, Y2)).

Area = 6
X0 = Y0 = 0
X1 = Y1 = 1
X2 = Y2 = 2
yes
```

Using object parameters for representing mutable state provides a solution that supports clean and logical application semantics, backtracking over state changes, and better performance than using assert and retract operations to update an object dynamic database.

5 Parametric Object Proxies

Compound terms with the same functor and arity as a parametric object identifier may act as *proxies* to a parametric object. Proxies may be stored on the database as Prolog facts and be used to represent different *instantiations* of a parametric object identifier. This representation can be ideal to minimize application memory requirements in the presence of a large number of data objects whose state is immutable. As a simple example, assume that our data represents geometric circles with attributes such as an identifier, the circle radius and the circle color:

```
% circle(Id, Radius, Color)
circle('#1', 1.23, blue).
circle('#2', 3.71, yellow).
circle('#3', 0.39, green).
circle('#4', 5.74, black).
```

We can define the following parametric object for representing circles:

```
:- object(circle(_Id, _Radius, _Color)).

    :- public([id/1, radius/1, color/1, area/1, perimeter/1]).

    id(Id) :-
        parameter(1, Id).

    radius(Radius) :-
```

```
        parameter(2, Radius).

    color(Color) :-
        parameter(3, Color).

    area(Area) :-
        parameter(2, Radius),
        Area is 3.1415927*Radius*Radius.

    perimeter(Perimeter) :-
        parameter(2, Radius),
        Perimeter is 2*3.1415927*Radius.

:- end_object.
```

The `circle/3` parametric object provides a simple solution for encapsulating a set of predicates that perform computations over circle properties, as illustrated by the `area/1` and `perimeter/1` predicates.

Logtalk provides a convenient notational shorthand for accessing proxies represented as Prolog facts when sending a message:

```
| ?- {Proxy}::Message.
```

In this case, Logtalk proves `Proxy` as a Prolog goal and sends the message to the resulting term, interpreted as the identifier of a parametric object.[3] This construct can either be used with a proxy argument that is sufficiently instantiated in order to unify with a single Prolog fact or with a proxy argument that unifies with several facts. Backtracking over the proxy goal is supported, allowing all matching object proxies to be processed by e.g. a simple failure-driven loop. For example, in order to construct a list with the areas of all the circles we can write:

```
| ?- findall(Area, {circle(_, _, _)}::area(Area), Areas).

Areas = [4.75291, 43.2412, 0.477836, 103.508]
yes
```

In non-trivial applications, data objects, represented as object proxies, and the corresponding parametric objects, are tied to large hierarchies representing taxonomic knowledge about the application domain. An example is the LgtSTEP application [10] used for validating STEP files, which is a format for sharing data models between CAD/CAM applications. A typical data model can contain hundreds of thousands of geometrical objects, which are described by a set of hierarchies representing geometric concepts, data types, and consistency rules. In its current version, the LgtSTEP application hierarchies define 252 geometric or related entities, 78 global consistency rules, 68 data types, and 69 support functions. STEP files use a simple syntax for geometrical and related objects. Consider the following fragment:

[3] The `{}/1` functor was chosen as it is already used as a control construct to bypass the Logtalk compiler in order to call Prolog predicates.

```
#1=CARTESIAN_POINT('',(0.E0,0.E0,-3.38E0));
#2=DIRECTION('',(0.E0,0.E0,1.E0));
#3=DIRECTION('',(1.E0,0.E0,0.E0));
#4=AXIS2_PLACEMENT_3D('',#1,#2,#3);
```

The first line above defines a cartesian point instance, whose identifier is `#1`, followed by an empty comment and the point coordinates in a 3D space. Similar for the other lines. This syntax is easily translated to Prolog predicate facts that are interpreted by Logtalk as object proxies:

```
cartesian_point('#1', '', (0.0, 0.0, -3.38)).
direction('#2', '', (0.0, 0.0, 1.0)).
direction('#3', '', (1.0, 0.0, 0.0)).
axis2_placement_3d('#4', '', '#1', '#2', '#3').
```

Complemented by the corresponding parametric objects, object proxies provide a source level representation solution whose space requirements are roughly equal to those of the original STEP file. An alternative representation using one Logtalk object per STEP object results in space requirements roughly equal to 2.2 times the size of the original STEP file in our experiments.[4] For example, the cartesian point above could be represented by the object:

```
:- object('#1',
    instantiates(cartesian_point)).

    comment('').
    coordinates(0.0, 0.0, -3.38).

:- end_object.
```

For simple, immutable data objects, Logtalk object representation provides little benefit other than better readability. Object proxies and parametric objects allows us to avoid using a Logtalk object per data object, providing a bridge between the data objects and the hierarchies representing the knowledge used to reason about the data, while optimizing application memory requirements.

6 Using Both Object Proxies and Regular Objects

While object proxies provide a compact representation, complex domain objects are often better represented as regular objects. Moreover, while in object proxies the meaning of an argument is implicitly defined by its position in a compound term, predicates with meaningful names can be used in regular objects. In some applications it is desirable to be able to choose between object proxies and regular objects on a case-by-case basis. An example is the L-FLAT [11] application, a full rewrite in Logtalk of P-FLAT [12], a Prolog toolkit for teaching Formal Languages and Automata Theory. L-FLAT is a work in progress where one of

[4] Logtalk object representation is currently optimized for execution time performance, not memory space requirements.

the goals is to allow users to use both object proxies and regular objects when representing Turing machines, regular expressions, finite automata, context free grammars, and other concepts and mechanisms supported by L-FLAT. This flexibility allows users to represent e.g. a simple finite automaton with a small number of states and transitions as an object proxy:

```
% fa(Id, InitialState, Transitions, FinalStates)
fa(fa1, 1, [1/a/1, 1/a/2, 1/b/2, 2/b/2, 2/b/1], [2]).
```

It also allows users to represent e.g. a complex Turing machine with dozens of transitions as a regular object:

```
:- object(aibiciTM,
    instantiates(tm)).

    initial(q0).

    transitions([
        q0/'B'/'B'/'R'/q1,
        q1/a/'X'/'R'/q2,   q1/'Y'/'Y'/'R'/q5,  q1/'B'/'B'/'R'/q6,
        q2/a/a/'R'/q2,     q2/'Y'/'Y'/'R'/q2,  q2/b/'Y'/'R'/q3,
        q3/b/b/'R'/q3,     q3/'Z'/'Z'/'R'/q3,  q3/c/'Z'/'L'/q4,
        q4/a/a/'L'/q4,     q4/b/b/'L'/q4,      q4/'Y'/'Y'/'L'/q4,
        q4/'Z'/'Z'/'L'/q4, q4/'X'/'X'/'R'/q1,
        q5/'Y'/'Y'/'R'/q5, q5/'Z'/'Z'/'R'/q5,  q5/'B'/'B'/'R'/q6
    ]).

    finals([q6]).

:- end_object.
```

The transparent use of both object proxies and regular objects requires that all predicates expecting e.g. a regular expression object accept both an object identifier and an object proxy as argument. While this may sound as an additional hurdle, the solution is simple. Given that an object proxy is also an instantiation of the identifier of a parametric object, it suffices to define the corresponding parametric object. If, for example, we want to represent some finite automata as instances of a class `fa` and other finite automata as object proxies, we simply define a parametric object as an instance of the class `fa`. The object parameters will be the properties that might be unique for a specific finite automaton. The parametric object define the predicates that give access to these properties. These predicates would be the same used in class `fa` to define all predicates that must access to finite automaton properties:

```
:- object(fa(_Id, _Initial, _Transitions, _Finals),
    instantiates(fa)).

    initial(Initial) :-
        parameter(2, Initial).
    transitions(Transitions) :-
        parameter(3, Transitions).

    finals(Finals) :-
        parameter(4, Finals).

:- end_object.
```

Thus, using both object proxies and regular objects is simple, fully transparent, and just a matter of defining the necessary parametric objects.

7 Related Work

Logtalk parametric objects are based on L&O [6] parametric theories and SICStus Objects [4] parametric objects, with some input from the OL(P) [8] representation of object instances. The two main differences are the concept of object proxies (introduced in Logtalk and discussed in this paper) and parameter accessing. While Logtalk parameter values are accessed using the `parameter/1` and `this/1` built-in methods, the scope of the parameters in L&O and SICStus Objects is the whole object. The generic concepts of parametric objects and object parameters are also similar to, respectively, the concepts of *units* and *unit arguments* found on some implementations of Contextual Logic Programming [13]. We take a brief look to each one of these systems in this section.

7.1 L&O Parametric Theories

In L&O, parametric objects are known as parametric theories. A theory is identified by a *label*, a term that is either a constant or a compound term with variables. L&O uses as example a parametric theory describing trains:

```
    train(S, Cl, Co):{
        colour(Cl).
        speed(S).
        country(Co).
        journey_time(Distance, T) :-
            T = Distance/S.
    }
```

The label variables are universally quantified over the theory. A specific train can be described by instantiating the label variables:

```
    train(120, green, britain)
```

Messages can be sent to labels, which act as object identifiers. For example, the following message:

```
    train(120, green, britain):journey_time(1000, Time)
```

will calculate a journey time using the value of the label first parameter as the speed of the train.

7.2 SICStus Parametric Objects

SICStus parametric objects are similar to L&O parametric theories, with parameters acting as global variables for the parametric object. The SICStus Objects manual contains the following example, describing ellipses and circles:

```
ellipse(RX, RY, Color) :: {
    color(Color) &
    area(A) :-
        :(A is RX*RY*3.14159265)
}.

circle(R, Color) :: {
    super(ellipse(R, R, Color))
}.

red_circle(R) :: {
    super(circle(R, red))
}.
```

SICStus Objects uses the predicate `super/1` to declare the ancestors of an object. This example illustrates parameter-passing between related objects in a hierarchy, a feature also found in L&O and Logtalk.

7.3 OL(P) Object Instances

OL(P) is a Prolog object-oriented extension that represents object instances using a notation similar to parametric objects. An instance `I` of an object named `Object` is represented as `Object(I)`. The term `I` is a list of attributes and attribute-value pairs. Instance state changes can be accomplished by constructing a new list with the updated and unchanged attributes. The OL(P) system documentation offers the following example:

```
| ?- rect(I)::area(A), rect(I)::move(5, 5, J).
```

The method `move/3` will return, in its third argument, the attribute list `J` resulting from the update of the attribute list `I`. In addition, OL(P) provides a nice notation for accessing and updating attributes. This solution for object state changes implies the use of extra arguments for methods that update attributes. Nevertheless, it is an interesting technique, which preserves the declarative semantics found on pure Prolog programs. We can easily apply this solution to Logtalk programs by using parametric objects. Moreover, we are not restricted to using a list of attributes. If the number of attributes is small, an identifier with the format `Object(V1, V2, ..., Vn)` will provide a more efficient solution.

7.4 GNU Prolog/CX Parametric Units

Contextual Logic Programming defines a modularity concept, *unit*, which supports parameterization in some implementations, such as GNU Prolog/CX [14].

Unit arguments play a similar role as object parameters. As in other systems, the scope of the unit arguments is the whole unit. The GNU Prolog/CX authors present the following example of a simple dictionary implementation using a list of key-value pairs:

```
:- unit(dict(ST)).

dict(ST).

lookup(KEY, VALUE) :- ST=[KEY=VALUE|_].
lookup(KEY, VALUE) :- ST=[_|STx], dict(STx) :> lookup(KEY, VALUE).
```

Instead of augmenting all dictionary predicates with an argument for passing the dictionary term, an unit argument is used to hold this term. Thus, this example is an instance of the programming pattern presented in section 4.3.

8 Conclusions

The main contributions of this paper are a survey of useful parametric object programming patterns and the Logtalk concept of object proxies. The use of these programming patterns and the implementation of objects proxies in other object-oriented logic programming languages and Prolog object-oriented extensions is straightforward.

Parametric objects enable useful programming patterns that complement Logtalk encapsulation and reuse features. Object parameters allow passing *data*, *type information*, and *operations* to object predicates. Parametric objects may be used to encapsulate a set of predicates for reasoning about compound terms sharing the same functor and arity, to simplify object interfaces, to restore shared variables between sets of constraints stored in different objects, to enable a more data-centric style of programming, and to provide a logical alternative to the use of an object dynamic database for representing mutable state. It is also possible to use the *instantiations* of a parametric object identifier to represent the history of object state changes.

Logtalk extends the usefulness of parametric objects by introducing the concept of object proxies, which provide a bridge between Logtalk objects and compact plain Prolog representations. Some applications need to represent a large number of immutable objects. These objects typically represent data that must be validated or used for data mining. This kind of data is often exported from databases and must be converted into objects for further processing. The application domain logic is usually represented by a set of hierarchies with the data objects at the bottom. Although data conversion is most of the time easily scriptable, the resulting objects take up more space than a straightforward representation as plain Prolog facts. By using object proxies, application memory footprint can be minimized while still taking full advantage of Logtalk object-oriented features for representing and reasoning with taxonomic knowledge. The concept of object proxies can be easily adopted and used in other languages supporting parametric objects.

Acknowledgements. This work is partially supported by the FCT research project MOGGY – PTDC/EIA/70830/2006. We are grateful to Paul Crocker for helpful suggestions in improving the Logtalk support for object proxies. We thank also José Silva, Sara C. Madeira, and the anonymous reviewers for their comments and help in revising this paper.

References

1. Moura, P.: Logtalk 2.6 Documentation. Technical Report DMI 2000/1, University of Beira Interior, Portugal (July 2000)
2. Moura, P.: Logtalk - Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
3. Moura, P.: Logtalk 2.39.0 User and Reference Manuals (February 2010)
4. Swedish Institute for Computer Science: SICStus Prolog 4.0 User Manual (April 2009)
5. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer, New York (1987)
6. McCabe, F.G.: Logic and Objects. Computer Science. Prentice Hall, Englewood Cliffs (1992)
7. Igler, M., Joblonski, S.: ESProNa – Engine for Semantic Process Navigation (2009), `http://www.ai4.uni-bayreuth.de/`
8. Fromherz, M.: OL(P): Object Layer for Prolog (1993), `ftp://parcftp.xerox.com/ftp/pub/ol/`
9. Kino, N.: Logical assignment of Prolog terms (2005), `http://www.kprolog.com/en/logical_assignment/`
10. Moura, P., Marchetti, V.: Logtalk Processing of STEP Part 21 Files. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 453–454. Springer, Heidelberg (2006)
11. Moura, P., Dias, A.M.: L-FLAT: Logtalk Toolkit for Formal Languages and Automata (2009), `http://code.google.com/p/lflat/`
12. Wermelinger, M., Dias, A.M.: A Prolog Toolkit for Formal Languages and Automata. In: ITiCSE 2005: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, pp. 330–334. ACM, New York (2005)
13. Monteiro, L., Porto, A.: A language for contextual logic programming. In: Logic Programming Languages: Constraints, Functions, and Objects, pp. 115–147. MIT Press, Cambridge (1993)
14. Abreu, S., Diaz, D.: Objective: In minimum context. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 128–147. Springer, Heidelberg (2003)