Research output for the designing of commonsense programming language for the views of Logtalk

* Dr. Paulo Jorge Lopes de Moura, Universidade da Beira Interior, Assistant Professor, Departamento de Inform´atica.

** Dr. Abel Jo˜ao Padr˜ao Gomes, Assistant Professor of the Department of Informatics of the University of Beira Interior, France.

Introduction

Why growing Logtalk? The closing answer is, of route, for the a laugh of it. But, some readers might count on a unique type of solution. There are a number of reasons, outcome of the modern-day state of affairs of both common sense programming and item-oriented programming.

Concerning good judgment programming, Prolog, its maximum considerable language, born in 1972 still lacks a popular library and a function set appropriate for programming inside the huge. The primary Prolog ISO popular. Concerning the core components of the language, turned into published in 1995, and is being followed at a sluggish tempo through the Prolog network. The second one ISO fashionable. Concerning the module system, turned into posted in 2000 and is being basically not noted by maximum of the Prolog community. Lengthy long gone are the days of the japanese fifth generation pc system venture targeted on the guarantees of common sense programming because the silver bullet. Nowadays, Prolog is a niche language. Logic became dropped as a programming paradigm from the first draft of the 2001 ACM/IEEE computer technology Curricula and turned into best re-brought within the curricula very last document after a great deal pressure from the good judgment programming community. There are, of course, many reasons for the present day country of good judgment programming in general, and Prolog specially. Although, sociologic and political motives aside, Prolog technical handicaps, together with the inexistence of popular libraries, popular foreign language interfaces, and powerful encapsulation mechanisms, make it an uphill battle to apply the language and the good judgment programming paradigm for coaching, discovering, or business software program development.

Concerning item-oriented programming, the sector is strongly biased closer to magnificence- based totally structures and the particular implementation of item-oriented principles in a few languages, considerably, C++ and Java. These languages, like another language, constitute a selected set of design selections on how to implement large item-orientated ideas.

Defining a brand-new item

we are able to define a brand new object in the same way we write Prolog code: via using a textual content editor. Item code (directives and predicates) is textually encapsulated among Logtalk directives: object/1-five and cease object/zero. The best object might be a self-contained prototype, no longer depending on some other Logtalk entity:

```
:- object(Object).
...
:- end_object.
```

The first argument of the opening directive is the object identifier. Object identifiers can be atoms or compound terms[1]. Objects share a single namespace with and categories.

Prototype hierarchies

Prototype hierarchies are constructed by defining extension relations between objects. To define an object as an extension of one or more objects we will write:

```
:-          object(Prototype,
        extends(Parents)).

       ...

  :- end_object.
```

The sequence of the parent prototypes in the object-opening directive determines the lookup order for predicate inheritance. The lookup is performed using a depth-first strategy.

Class hierarchies

Class hierarchies are constructed by defining instantiation and specialization relations between objects. To define an object as an instance of one or more classes, we will write:

```
:-          object(Object,
    instantiates(Classes)).

    ...

:-  end_object.
```

To define a class as a specialization of one or more classes (its superclass), we will write:

```
:-           object(Class,
    specializes(Superclass
    es)).

    ...

:-  end_object.
```

If we are defining a reflexive system where every class is also an object, we will be using the following pattern:

```
:-            object(Class,
    instantiates(Metaclasse
    s),
    specializes(Superclass
    es)).

    ...

:-  end_object.
```

Compiling objects

A stand-alone object is always compiled as a prototype, that is, a self-describing object. If we want to use classes and instances, then we will need to specify at least an initialization or a specialization relation. The best way to accomplish this is to define a set of objects that provides the basis of a reflective system. If a reflective system solution is not necessary, but we still want to construct class hierarchies, then we can simply turn a class into an instance of itself or, in other words, turn a class into its own metaclass. For example:

```
:-                    object(root,
    instantiates(root)).

    ...

:-  end_object.
```

An alternative solution would be to add a predefined root class to Logtalk, and then to use that class as the default root class when defining class-based hierarchies. However, it is a Logtalk design choice not to specify any predefined entities in order to keep the language simple and unbiased to particular solutions. We can always use an entity library if necessary.

Creating a new object at runtime

An object can be dynamically created at runtime by using the Logtalk built-in predicate create object/4:

```
| ?- create object(Object, Relations, Directives, Clauses).
```

The first argument, the identifier of the new object (a Prolog atom or compound term), must not match any existing entity identifier. The second argument corresponds to the relations described in the opening object directive. The third and fourth arguments are lists of directives and predicate clauses, respectively. For example, the following call:

```
| ?- create object(o1, [extends(o2)], [public(p/1)], [p(1), p(2)]).
```

is equivalent to compiling and loading the object:

```
:-  object(o1,

    extends(o2)).

    :-  dynamic.
```

```
:-
public(p/1).
p(1).

p(2).

:- end_object.
```

If we need to create many (dynamic) objects at runtime, then it is better to define a metaclass or a prototype with a predicate that calls this built-in predicate in order to create new objects. This predicate may provide automatic generation of object identifiers and accept object initialization options. The current Logtalk implementation contains example classes defining such predicates in its library.

Object initialization

We can define a goal to be executed as soon as an object is (compiled and) loaded in memory with the initialization/1 directive:

```
:- initialization(Goal).
```

The initialization goal can be any valid Prolog or Logtalk call. For example, the goal can be a call to a locally defined predicate:

```
:- object(foo).

    :- initialization(init).

    :-
    private(init/0).
    init :-

        ... .

    ...

:- end_object.
```

or a message to other object:

```
:- object(assembler).

    :- initialization(control::start).

:- end_object.
```

The ::/2 operator is used in Logtalk for message sending. The initialization goal can also be a message to *self* in order to call an inherited predicate. Assuming, for example, that we have an object named profiler defining a reset/0 predicate, we could write:

```
:-
    object(stopwat
    ch,
    extends(profiler
    )).

    :- initialization(::reset).

    ...

:- end_object.
```

The ::/1 operator is used in Logtalk for sending a message to *self*. Note that, in this context, *self* denotes the object containing the directive.

Descendant objects do not inherit initialization directives from ancestor objects. In addition, note that by initialization we do not necessarily mean setting an object's dynamic state.

Object dependencies

In addition to the relations declared in the object-opening directive, the predicate definitions contained in the object may imply other dependencies. These can be documented by using the directives calls/1 and uses/1.

The calls/1 directive can be used when a predicate definition sends some message that is declared in a specific protocol:

```
:- calls(Protocol).
```

When a predicate definition sends a message to a specific object, this dependence can be declared with the directive uses/1:

```
:- uses(Object).
```

These two directives may be used by the Logtalk runtime engine to ensure that all necessary entities are loaded when running an application. The directive uses/1 is also the basis for a planned extension of the Logtalk language to support object namespaces.

Object documentation

An object can be documented with arbitrary user-defined information by using the directive info/1:

```
:- info(List).
```

Assuming, for example, that we have defined an object containing list predicates, it could be documented as follows:

```
:- info([
    version is 1.0,
    author is 'Paulo
    Moura', date is
    2000/7/24,
    comment is 'List predicates.']).
```

Finding defined objects

We can enumerate, using backtracking, all defined objects by calling the Logtalk built-in predicate current object/1 with a non-instantiated variable:

```
| ?- current_object(Object).
```

This predicate can also be used to test whether an object is defined by calling it with a valid object identifier (either an atom or a compound term).

Towers of Hanoi

This example shows how to use an object to encapsulate a solution for the well-known "Towers of Hanoi" problem. The object will be a self-contained prototype with its interface resuming to a single predicate whose argument will be the number of disks for which we want to solve the problem:

```
:- object(hanoi).
:- public(run/1).

run(Disks) :-

     move(Disks, left, middle, right).

move(1, Left, _, Right):-

     !,

     report(Left, Right).

     Move

          (Disks,   Left,   Aux,   Right):-
          Disks2     is     Disks     -     1,
          move(Disks2,     Left,     Right,
          Aux),    report(Left,    Right),
          move(Disks2,     Aux,      Left,
          Right).

     report(Pole1,              Pole2):-
          write('Move a disk from '),

          writeq(Pole1),  write('  to  '),  writeq(Pole2),
          write('.'), nl.

:- end_object.
```

Note that, if we remove the Logtalk directives (the opening and closing object directives, and the predicate directive), the remaining code consists of Prolog-compliant predicate clauses. This is an important feature of Logtalk: Prolog code can be easily encapsulated by Logtalk objects with little or no modifications.

After compiling and loading this object, we can test our code by sending the message run/1 to the object. Message sending is performed using the infix operator: :/2. An example call will be:

```
?- hanoi::run(3).

Move a disk from left to right.

Move a disk from left to middle.

Move a disk from right to middle.

Move a disk from left to right.
Move a disk from middle to left.
Move a disk from middle to right.
Move a disk from left to right.
yes
```

This case provides an object solution that is essentially equal to a module answer. In addition, it illustrates how, in Logtalk, we will easily outline stand-alone objects that are not connected to any hierarchy, as in any prototype-based totally language.

Logtalk integration of lessons and prototypes in a unmarried language lets in us to use the identical message sending mechanisms and the equal techniques for dynamically growing, disposing, and enumerating items. Despite the fact that we can not blend prototypes with training and instances inside the same hierarchy, we will freely change messages among them.

Logtalk prototypes may be used as a replacement for modules in Prolog programs, imparting several vital capabilities now not available in modern-day Prolog module structures. Particularly, prototypes offer data hiding, a function missing in module systems. In addition, Logtalk affords a degree of compatibility with Prolog compilers now not matched by using any module machine. This makes a Logtalk program lots greater portable than a Prolog program that makes use of modules.

Conclusions

Logtalk may be defined as a multi-paradigm language that helps logic application, item-orientated programming, and event-pushed programming. However, Logtalk purpose turned into no longer to best guide those programming paradigms but to combine them. The mixing changed into made by using, first, reinterpreting item standards within the context of logic programming and, 2nd, via reinterpreting event concepts inside the context of item-orientated programming.

This starts off evolved via evaluating Logtalk as a Prolog item-orientated extension and as an interpreted, interactive object-orientated programming language. Secondly, the relevance of occasion-driven programming within the context of item-oriented languages is described. Thirdly, class-primarily based composition, and its assist for issue-based programming, is summarized. Fourthly, Logtalk native assist for mirrored image is tested. Then, the Logtalk help for automated software documentation is supplied. Next, the experience of the use of Logtalk in the school room is described, followed with the aid of a few information at the Logtalk distribution numbers. Ultimately, the roadmap for destiny improvement is provided.

Logtalk as a Prolog item-orientated extension

Logtalk reinterprets the concept of item as a hard and fast of predicate directives (declarations) and clauses (definitions). Consequently, message sending is reinterpreted as evidence creation the usage of the predicates defined for the receiving item. Inheritance mechanisms allow us to define the complete database of an item. A method is then clearly the predicate definition decided on from an object entire database so one can answer a message. Through reinterpreting the principles of object, message, and technique in common sense programming phrases, a simple mapping is set up among Logtalk semantics and the acquainted Prolog semantics.

In trendy terms, this reinterpretation of object standards is shared via maximum Prolog item-oriented extensions. To assess and evaluate Logtalk with other item-orientated extensions and Prolog module structures, the following standards could be used: compatibility with Prolog compilers, language syntax, interpretation of the concept of item, feature set, and working environment

Logtalk compatibility

Logtalk is the best Prolog object-orientated extension available these days that has been de- signed from scratch for compatibility with maximum compilers and with the ISO Prolog popular. This layout intention sets it aside from other Prolog extensions. The preprocessor answer followed for the Logtalk implementation lets in it to run on most computer systems and working structures for which a present-day Prolog compiler is available. Especially, the contemporary Logtalk version is compatible with thirty-one versions of twenty Prolog compilers.

Logtalk syntax

Logtalk makes use of, whenever feasible, popular Prolog syntax, and defines fashionable language constructs, consistent with the cutting-edge practice and expectancies of Prolog programmers. This facilitates to easy the mastering curve for Prolog programmers. That is extra than a syntactic sugar difficulty. As an example, Logtalk allows existing Prolog code to be encapsulated in gadgets without any modifications. Best when a predicate needs to call different object predicates, would minimum adjustments be required. For this reason, clean conversion of vintage Prolog packages is ensured.

The position of objects in logic programming

The primary motive of objects in Logtalk is the encapsulation and reusing of code, for that reason decoupling this functionality from the theoretical troubles of dynamic nation alternate in common sense programming. As such, Logtalk affords a realistic view, rather than a theoretical view, of the role of gadgets in logic programming in fashionable, and in Prolog in particular. Via focusing at the encapsulation and code reuse proprieties of items, Logtalk objectives to be an effective device for fixing software program engineering troubles in Prolog programming.

Implementation solutions for object-orientated standards

Logtalk shows how to put into effect the principle object-oriented ideas in Prolog. Those encompass ideas no longer discovered in my view on most Prolog item-oriented extensions along with: support for each classes and prototypes; met instructions; protocols and protocol hierarchies; public, protected, and personal predicates; and public, protected, and personal inheritance. Consequently, Logtalk is in all likelihood one of the most whole Prolog object-orientated ex- anxiety available today. Further, Logtalk indicates the way to implement other essential ideas that aren't to be had on other object-oriented languages, such as classes and event-pushed programming. In contrast to other item-oriented extensions which are both proprietary or depend closely on the specifics of the native module systems, Logtalk implementation solutions are absolutely well suited with any compiler that complies with the element I of the ISO Prolog trendy.

Gadgets as a substitute for modules

Logtalk objects offer an alternative to using Prolog modules, either as implemented in maximum Prolog compilers or as specified within the part II of the ISO Prolog popular. Like Prolog modules, Logtalk prototypes can be defined as stand-on my own encapsulation entities. Similarly, although Logtalk does not provide a right away replacement for module import and export directives, the extension relation between prototypes, together with protocol implementation and class importation relations, lets in equivalent characteristically. Logtalk items have numerous vital benefits over Prolog modules:

Logtalk predicate scope directives make sure statistics hiding, a missing function in the ISO widespread for Prolog modules. Logtalk message sending mechanism, built-in methods, and built-in predicates enforce predicate scope directives. The ISO well known specifies that any module predicate may be called the usage of explicit module qualification; it considers that mechanisms to put into effect facts hiding are implementation-established features.

Compatibility with current Prolog compilers.

Logtalk is well suited with nearly all contemporary Prolog compilers. The ISO well known for Prolog modules is still to being followed by means of most Prolog providers, in component due to differences with current and broadly used module structures. The ISO trendy specifies incompatible approaches of affirming met predicates. No such nuisances exist in Logtalk. A few matters cannot be special within a general and must be considered as implementation established capabilities. The syntax for maintaining met predicates is certainly now not one of them. Logtalk provides a number of treasured capabilities inside the improvement of big-scale tasks, which can be outdoor the scope of module structures. Those include predicate reuse and specialization via inheritance and composition, occasion-driven programming, mirrored image, and automatic era of documentation.

Operating environment and other sensible matters

Logtalk operating surroundings is constrained to the subset of common functions of the well-matched Prolog compilers. As an instance, there's no commonplace set of predicates for operating gadget access (so limiting the functionality of the Logtalk compiler) or a common general for building graphical consumer interfaces. Proprietary object-oriented extensions, developed to paintings with a single Prolog compiler, are capable of provide richer operating surroundings, taking advantage of precise functions of a compiler and its web hosting operating gadget. Although, within its compatibility regulations, Logtalk tries to provide a studying and working environment just like different Prolog compilers and object-oriented extensions that use textual content-primarily based development gear. For modifying supply files, the present day Logtalk distribution includes syntax-coloring configuration files for famous textual content editors, together with textual content templates for defining new entities and entity predicates. This improves the programming revel in, especially by helping to avoid syntax mistakes when writing entity and predicate directives. Additionally blanketed are a number of programming examples and great documentation, which comprises a consumer manual, a reference manual, and programming tutorials.

Out of doors the educational international, those practical subjects are as critical because the technical features and medical achievements of the language itself. As such, they're fundamental in building a consumer network, who will use the Logtalk language as a device to solve real troubles.

Logtalk as an item-orientated programming language

Logtalk extends Prolog, in the equal way as CLOS extends LISP or objective-C extends C. As a programming language in its personal proper, Logtalk shares features with common object-orientated languages. But, there are also some crucial differences because of its Prolog roots. The maximum large one is that Logtalk eliminates a few dichotomies deeply established on maximum item-oriented languages. Those dichotomies are frequently used as a manner of characterizing and classifying item-orientated languages. Specially, Logtalk makes no difference among variables and methods, allows maximum language elements to be both dynamic or static, and integrates classes and prototypes within the identical language. In spite of the primary factors are not unusual to a few Prolog object-orientated extensions, they're well worth summarizing right here. Unlike Logtalk, almost all item-oriented languages are strictly described as both magnificence-based or prototype-based totally languages. Most of them are characterized by way of a clean distinction among variables and techniques, what is static and what's dynamic, and what need to be carried out at compile time or may be performed at runtime.

Predicates as both variables and methods

Logtalk item predicates unify the concepts of item strategies and object variables, therefore simplifying the language semantics. Predicates remove the dichotomy among nation and conduct: a predicate really states what's actual about an object. Predicates may be used to implement both variables and techniques, but the sort of difference is continually elective. It follows that we now not want separate definition, inheritance, and scope regulations for kingdom and conduct. Further, each nation and conduct can be effortlessly shared along inheritance hyperlinks or described regionally. This lets in us, as an instance, to outline methods in instances and to percentage object nation easily among descendant gadgets with out the want of first formalizing principles which includes shared example variables.

Static and dynamic language factors

Logtalk gadgets, protocols, classes, and predicates may be both dynamic or static. Predicates can be asserted into, and abolished from each static and dynamic gadgets at runtime. Gadgets, protocols, and categories can be described both in a supply file or created dynamically at runtime. If contained in a supply report, they may be defined as both static or dynamic entities. As such, we aren't restrained, for example, to define classes as static entities and times as runtime-best objects. We may also define an instance in a supply document within the identical way as a category may be defined. This is constant with Logtalk number one view of objects as encapsulation devices.

Support for each prototypes and classes

Logtalk is a neutral, impartial language, supporting each prototype-based totally and class- primarily based programming. We can use each kinds of objects at the equal time and freely trade messages among them. Logtalk instructions, times, and prototypes are sim- ply items — encapsulation entities — characterized through special rulesets for gaining access to their personal predicates and for reusing predicates inherited from different gadgets. Lessons and prototypes share the equal integrated predicates for developing, abolishing, and enumerating items. Similarly, they proportion the identical built-in techniques for dynamic object modification and the message sending mechanisms. Furthermore, protocols may be implemented, and categories can be imported, by using prototypes, lessons, and instances.

Event-pushed programming

Logtalk presents a conceptual integration of event-pushed programming into the object- oriented programming paradigm. The important thing for this integration is the translation of message sending because the best occasion that takes place in an item-oriented program. Therefore, Logtalk reinterprets the standards of occasion, reveal, event notification, and occasion handler in terms of objects, messages, and methods. This allows us to stay in the item- oriented programming paradigm whilst writing event-pushed programming code.

Crucial outcomes emerge from Logtalk programming exercise in using occasions to put into effect complicated dependency members of the family between items. These consequences aren't specific to Logtalk. As an alternative, they apply to most item-orientated programming languages. First, event-pushed programming is an important function of object-orientated languages for accomplishing a excessive level of item cohesion and fending off unnecessary object coupling on packages where object members of the family imply constraints at the state of taking part gadgets. Dependency mechanisms, as observed in Smalltalk and in different languages, provide only a partial answer, which can simplest be used whilst object methods include — or can be modified to contain — calls to the dependency mechanism techniques. 2nd, activities and monitors need to be supported as language primitives, incorporated with the message sending mechanisms. That is an vital requirement from a overall performance point of view, which precludes the implementation of event-driven programming at the application layer or through language libraries. Local language aid is fundamental in making occasion-pushed programming an powerful tool for trouble solving.

Category-based totally composition

Classes are the premise of issue-based programming in Logtalk. Classes seasoned-vide finer, functionally cohesive units of code that may be imported via any object. For this reason, categories play a role twin to that one performed via protocols for interface encapsulation. Similarly, classes provide several improvement benefits such as incremental compilation, updating an item — with out recompiling it — through updating its imported classes, and refactoring of complicated items into greater possible and reusable elements. Category-primarily based composition, inheritance, and instance variable-based composition offer complementary sorts of code reuse. Categories put into effect a composition mechanism where a category interface becomes a part of the interface of an object uploading it. This is in contrast to example variable-primarily based composition, however similar to what takes place with inheritance. Logtalk aid for public, protected, and private category importation affords similarly flexibility. By means of applying the concepts of separation of worries common to element-primarily based programming processes, classes provide opportunity answers to multi-inheritance designs that can be implemented inside the context of single inheritance languages.

Bibliography

- ✓ ACM/IEEE. Computer Curricula 2001: extent II — computer technology — Straw- man Draft. Http://www.Laptop.Org/schooling/cc2001/, March 2000.
- ✓ James Noble, Antero Taivalsaari, and Ivan Moore, editors. Prototype-based seasoned-gramming - principles, Languages and programs. Springer-Verlag, 1999.
- ✓ Mark J. Stefik, Daniel G. Bobrow, and Ken M. Kahn. Integrating get admission to-orientated programming into a multiparadigm environment. IEEE software, pages 10–18, January 1986.
- ✓ Adele Goldberg and David Robson. Smalltalk-80 — The language and its implementation. Series in laptop technological know-how. Addison-Wesley, 1983.
- ✓ Henry Lieberman. The usage of prototypical items to put in force shared behavior in object-orientated structures. In Meyrowitz [124], pages 189–214.

- ✓ Brad J. Cox and Andrew Novobilski. Object-oriented Programming: An Evolu- tionary approach. Addison-Wesley, 2nd version, June 1991.
- ✓ Swedish Institute for pc technology. Quintus Prolog domestic page. Http:// www.Sics.Se/isl/quintus/.
- ✓ Common sense Programming pals Ltd. LPA home web page. Http://www.Lpa.Co.United kingdom / machine. Http://www. Icparc.Ic
- ✓ The XSB research group. XSB home page. Http://xsb.Sourceforge.Net/college of Melbourne. The Mercury mission: introduction. Http://www.Cs. Mu.Ouncesau/studies/mercury/.
- ✓ Squeak.Org. Squeak domestic web page. Http://www.Squeak.Org/ solar Microsystems, Inc. Java soft internet website. Http://www.Javasoft.Com/.
- ✓ Apple laptop, Inc. Apple pc Technical Documentation: MacOS X Server — basis Framework lessons, 1999.
- ✓ Yukihiro Matsumoto. Ruby home page. Http://www.Ruby-lang.Org/en/.
- ✓ World extensive web Consortium. Cascading fashion Sheets (css). Http://www.W3. Org/style/CSS/.