Meta-predicate Semantics

Paulo Moura

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

Abstract. We describe and compare design choices for meta-predicate semantics, as found in representative Prolog predicate-based module systems and in Logtalk. We look at the consequences of these design choices from a pragmatic perspective, discussing explicit qualification semantics, computational reflection support, expressiveness of meta-predicate directives, meta-predicate definitions safety, portability of meta-predicate definitions, and meta-predicate performance. We also describe how to extend the usefulness of meta-predicate definitions. Our aim is to provide useful insights to discuss meta-predicate semantics and portability issues based on actual implementations and common usage patterns.

Keywords: meta-predicates, predicate-based module systems, objects.

1 Introduction

Prolog and Logtalk [1,2] meta-predicates are predicates with one or more arguments that are either goals or closures[1] used for constructing goals, which are called in the body of a predicate clause. Common examples are all-solutions meta-predicates such as `setof/3`[2] and list mapping predicates. Prolog implementations may also classify predicates as meta-predicates whenever the predicate arguments need to be module-aware. Examples include built-in database predicates, such as `assertz/1` and `retract/1`, and built-in reflection predicates, such as `current_predicate/1` and `predicate_property/2`.

Meta-predicates provide a mechanism for reusing programming patterns. By encapsulating meta-predicate definitions in library modules or library objects, exported and public meta-predicates allow client modules or client objects to reuse these patterns, customized by calls to local predicates.

In order to compare meta-predicate semantics, as found in representative Prolog predicate-based module systems and in Logtalk, a number of design choices can be considered. These include explicit qualification semantics, computational reflection support, expressiveness of meta-predicate directives, safety of meta-predicate definitions, portability of meta-predicate definitions, and meta-predicate performance.

[1] In Prolog and Logtalk, a closure is defined as a callable term used to construct a goal by appending one or more additional arguments.

[2] Following common practice, predicates are referenced by their *predicate indicators*, i.e. by compound terms with the format *Functor/Arity*.

G. Vidal (Ed.): LOPSTR 2011, LNCS 7225, pp. 155–172, 2012.

When discussing meta-predicate semantics, it is useful to define the contexts where a meta-predicate is *defined*, *called*, and *executed*. The following definitions extend those found on [3] and will be used in this paper:

Definition Context. This is the object or module containing the meta-predicate definition.

Calling Context. This is the object or module from which a meta-predicate is called. This can be the object or module where the meta-predicate is defined in the case of a local call or another object or module assuming that the meta-predicate is within scope.

Execution Context. This includes both the calling context and the definition context. It is comprised by all the information required by the language runtime to correctly execute a meta-predicate call.

In this paper, we make use of an additional definition:

Lookup Context. This is the object or module where we start looking for the meta-predicate definition (note that the definition can always be reexported from another module or inherited from another object).

This paper is organized as follows. Section 2 describes meta-predicate directives. Section 3 discusses the consequences of using explicit qualified meta-predicate calls and the transparency of control constructs when using explicit qualification. Section 4 describes computational reflection support for meta-predicates. Section 5 describes a set of compilation rules aimed to prevent the use of meta-predicates to break module or object encapsulation. Section 6 discusses the portability of meta-predicate directives and meta-predicate definitions. Section 7 describes how lambda expressions can be used to extend the usefulness of meta-predicate definitions. Section 8 presents some remarks on meta-predicate performance. Section 9 summarizes our conclusions and discusses future work.

2 Meta-predicate Directives

Meta-predicate directives are required for proper compilation of meta-predicates in both Logtalk and Prolog predicate-based module systems in order to avoid forcing the programmer to explicitly qualify all meta-arguments. Meta-predicate directives are also useful for compilers to optimize meta-predicate calls (e.g. when using lambda expressions as meta-arguments) and to be able to check meta-predicate calls for errors (e.g. using a non-callable term in place of a meta-argument) and potential errors (e.g. arity mismatches when working with closures). The design choices behind the current variations of meta-predicate directives translate to different trade-offs between simplicity and expressiveness. The meta-predicate template information declared via meta-predicate directives can usually be programmatically retrieved using built-in reflection predicates such as `predicate_property/2`, as we will discuss in Section 4.

2.1 The ISO Prolog Standard `metapredicate/1` Directive

The ISO Prolog standard for Modules [4] specifies a `metapredicate/1` directive that allows us to describe which meta-predicate arguments are normal arguments and which are meta-arguments using a predicate template. In this template, the atom `*` represents a normal argument while the atom `:` represents a meta-argument. We are not aware of any Prolog module system implementing this directive. The standard does allow for alternative meta-predicate directives, providing solely as an example a `meta/1` directive that takes a predicate indicator as argument. This alternative directive is similar to the `tool/2` and `module_transparent/1` directives discussed below. However, from the point-of-view of standardization and code portability, allowing for alternative directives is harmful, not helpful.

2.2 The Prolog `meta_predicate/1` Directive

The ISO Prolog specification of a meta-predicate directive suffers from one major shortcoming [5]: it doesn't distinguish between goals and closures. The de facto standard solution for specifying closures is to use a non-negative integer representing the required number of additional arguments.[3] By interpreting a goal as a closure requiring zero additional arguments, we can reserve the atom `:` to represent arguments that need to be module-aware without necessarily referring to a predicate. This convention is found in recent B-Prolog, GNU Prolog, Qu-Prolog, SICStus Prolog, SWI-Prolog, and YAP versions and is being adopted by XSB and other Prolog compilers. In Prolog compilers without a module system, or with a module system where module expansion only needs to distinguish between normal arguments and meta-arguments, using an integer for representing closures can be useful for cross-reference tools and allows portable modularization extensions (such as Logtalk) to properly parse calls to proprietary built-in meta-predicates.

Despite being able to specify closure meta-arguments, there is still a known representation shortcoming. Some predicates accept a list of options where one or more options are module-aware. For example, the third argument of the predicate `thread_create/3` [7] is a list of options that can include an `at_exit/1` option. This option specifies a goal to be executed when a thread terminates. In this case, the argument is not a meta-argument but may *contain* a sub-term that will be used as a meta-argument. Although we could devise (a most likely cumbersome) syntax for these cases, the elegant solution for this representation problem is provided by the `tool/2` and `module_transparent/1` directives discussed below.

A minor limitation with the ISO Prolog `metapredicate/1` directive, which is solved by the `meta_predicate/1` directive, is the representation of the instantiation mode of the normal arguments. For representing the instantiation mode of

[3] This notation was first introduced on Quintus Prolog [6] in order to support meta-qualification and cross-referencing tools.

normal arguments, the atoms `+`, `?`, `@`, and `-` are commonly used,[4] as specified in the ISO Prolog standard [8]. However, using mode indicators in `meta_predicate/1` directives is no replacement for a *mode* directive. Consider the following two `meta_predicate/1` directives for the standard `once/1` meta-predicate and the de facto standard `forall/2` meta-predicate:

```
:- meta_predicate(forall(0, 0)).
:- meta_predicate(once(0)).
```

For `forall/2`, `0` means `@`.[5] For `once/1`, `0` means `+`.[6] Thus, using mode indicators in meta-predicate directives is inherently ambiguous (but still common practice).

2.3 The Logtalk `meta_predicate/1` Directive

Logtalk uses a `meta_predicate/1` directive, based on the Prolog `meta_predicate/1` directive described above, extended with meta-predicate mode indicators for representing a predicate indicator, `(/)`, a list of predicate indicators, `[/]`, a list of goals, `[0]`, and an existentially qualified goal, `^`.[7] In addition, the atom `:` is replaced by `::` for consistency with the message sending operator. Logtalk uses this information to verify meta-predicate definitions, as discussed in [3]. As Logtalk supports a `mode/2` predicate directive[8] for specifying the instantiation mode and the type of predicate arguments (plus the predicate determinism), the atom `*` is used to represent normal arguments in `meta_predicate/1` directives.

The extended set of meta-predicate mode indicators allows Logtalk to specify accurate meta-predicate templates for virtually all proprietary built-in meta-predicates found on all compatible Prolog compilers. This allows Logtalk to cope with the absence, limitations, differences, and sometimes ambiguity of meta-predicate templates in those Prolog compilers. Unfortunately, some Prolog compilers still don't implement the `meta_predicate/1` predicate property, while some other Prolog compilers return ambiguous meta-predicate templates due to the use of the `:` meta-predicate mode indicator for any kind of meta-argument.

2.4 The Ciao Prolog `meta_predicate/1` Directive

Ciao Prolog uses a `meta_predicate/1` directive that supports an extensive set of meta-predicate mode indicators [9] that, although apparently not adopted

[4] The meaning of these mode indicators atoms is as follows: `+` – argument must be instantiated; `-` – argument must be a variable; `?` – argument can be either instantiated or a variable; `@` – argument will not be (further) instantiated.

[5] The `forall/2` meta-predicate implements a generate and test loop using negation. Thus, no variable bindings are returned when calling it.

[6] The `once/1` meta-predicate proves its argument, possibly further instantiating it, and committing to the first solution found.

[7] This meta-predicate mode indicator was originally suggested by Jan Wielemaker and first implemented on SWI-Prolog 5.11.25. It is useful when defining wrappers for the `bagof/3` and `setof/3` built-in meta-predicates whenever the goal argument may use the `^/2` existential quantifier.

[8] `http://logtalk.org/manuals/refman/directives/mode2.html`

elsewhere, subsumes in expressive power the sets of meta-predicate mode indicators found on other Prolog compilers and in Logtalk. For example, it is possible to specify that a meta-argument should be a clause or, more specifically, a fact, using the mode indicators `clause` and `fact`. Moreover, a `list(Meta)` mode indicator, where `Meta` is itself a mode indicator, allows easy specification of lists of e.g. goals, predicate-indicators, or clauses.

2.5 The `tool/2` and `module_transparent/1` Directives

An alternative to the `meta_predicate/1` directive, found in ECLiPSe [10] and SWI-Prolog [11], is to declare meta-predicates as *module transparent*, forgoing the specification of which arguments are normal arguments and which arguments are meta-arguments. For this purpose, ECLiPSe provides a `tool/2` directive while SWI-Prolog provides a (apparently deprecated) `module_transparent/1` directive. These directives take predicate indicators as arguments and thus support a simpler, user-friendlier, solution when compared with the `meta_predicate/1` directive. More important, these directives allow the definition and encapsulation of meta-predicate definitions that cannot be (unambiguously) expressed using the alternative `meta_predicate/1` directive. Consider the following example, adapted from AutoBayes[9], an open-source NASA application:

```
cases(Pattern, [(Pattern -> Action)| _]) :-
    !,
    once(Action).
cases(Pattern, [_ | Cases]) :-
    cases(Pattern, Cases).
```

The `cases/2` meta-predicate is described as implementing a C-style pattern matching switch. The AutoBayes application is coded in plain Prolog. Thus, in the absence of an encapsulation mechanism, no meta-predicate directive is required. But if we attempt to modularize this code, using either a Prolog predicate-based module system or Logtalk, the `meta_predicate/1` directive can only express that somewhere in the second argument of the `cases/2` predicate there is a sub-term that is a meta-argument:

```
:- meta_predicate(cases(*, ::)).    % using Logtalk syntax
```

The inherent ambiguity is that, in general, the body of a library meta-predicate definition can contain both meta-calls to local predicates and meta-calls to client predicates. In the absence of explicit qualification, a system needs to know, for each meta-call, if it is going to executed in the context of the *caller* or in the context of the *callee*. The ECLiPSe `tool/2` directive and the SWI-Prolog `module_transparent/1` directive can be interpreted as assuming that all meta-calls are to client predicates.[10] This is a strong but fair assumption, which allows

[9] http://ti.arc.nasa.gov/tech/rse/synthesis-projects-applications/autobayes/

[10] Given that meta-calls and meta-arguments can always be explicitly qualified, this assumption does not prevent the definition of meta-predicates that perform local meta-calls.

e.g. the encapsulation and reuse of the `cases/2` meta-predicate above. In fact, this assumption is true for most common meta-predicate definitions. However, we have shown in [3] that distinguishing between goals and closures and specifying the exact number of closure additional arguments is necessary to avoid misusing meta-predicate definitions to break module and object encapsulation.

3 Explicit Qualification Semantics

The semantics of explicit qualification is perhaps the most significant design decision on meta-predicate semantics. This section compares two different semantics, found on actual implementations, for the explicit qualification of meta-predicates and control constructs.

3.1 Explicit Qualification of Meta-predicate Calls

Given an explicit qualified meta-predicate call, we have two choices for the corresponding semantics:

1. The explicit qualification sets only the initial lookup context for the meta-predicate definition. Therefore, all meta-arguments that are not explicitly-qualified are called in the meta-predicate calling context.
2. The explicit qualification sets both the initial lookup context for the meta-predicate definition and the meta-predicate calling context. Therefore, all meta-arguments that are not explicitly-qualified are called in the meta-predicate lookup context (usually the same as the meta-predicate definition context).

These two choices for explicit qualification semantics are also described in the ISO Prolog standard for modules. This standard specifies a read-only flag, `colon_sets_calling_context`, which would allow a programmer to query the semantics of a particular module implementation.[11]

Logtalk and the ECLiPSe module system implement the first choice. Prolog module systems derived from the Quintus Prolog module system [6], including those found on SICStus Prolog, SWI-Prolog, and YAP, implement the second choice (the native XSB module system is atom-based, not predicate-based).

In order to illustrate the differences between the two choices above, consider the following example, running on Prolog module systems implementing the second choice. First, we define a meta-predicate library module:

```
:- module(library, [my_call/1]).      % library exports my_call/1

:- meta_predicate(my_call(0)).        % my_call/1 takes a goal
```

[11] The `colon_sets_calling_context` read-only flag also means that two Prolog implementations could be fully compliant with ISO Prolog modules standard and still meta-predicate definitions written for one implementation would not be usable in the other implementation.

```
my_call(Goal) :-                        % as meta-argument
    write('Calling: '), writeq(Goal), nl,
    call(Goal).

me(library).                            % me/1 is a local predicate
```

The `my_call/1` meta-predicate simply prints a message before calling its argument (which is a goal, as declared in its meta-predicate directive). Second, we define a simple client module that imports and calls our meta-predicate using a local predicate, `me/1`, as its argument:

```
:- module(client, [test/1]).            % client exports test/1

:- use_module(library, [my_call/1]).    % import the meta-predicate

test(Me) :-                             % call the meta-predicate
    my_call(me(Me)).                    % using implicit qualification

me(client).                             % me/1 is a local predicate
```

To test our code, we use the following query:

```
| ?- client:test(Me).
Calling: client:me(_)
Me = client
yes
```

This query provides the expected result: the meta-predicate argument is called in the context of the client, not in the context of the meta-predicate definition. But consider the following seemingly innocuous changes to the client module:

```
:- module(client, [test/1]).

test(Me) :-                             % call the meta-predicate
    library:my_call(me(Me)).            % using explicit qualification

me(client).
```

In this second version, instead of importing the `my_goal/1` meta-predicate, we use explicit qualification in order to call it. Repeating our test query now gives:[12]

```
| ?- client:test(Me).
Calling: library:me(_)
Me = library
yes
```

[12] The test could not be performed using Ciao Prolog, which reports a bad module qualification error in the explicit qualified call, complaining that the meta-predicate is not imported, despite the library module being loaded. Importing the predicate eliminates the error but also makes the interpretation of the test result ambiguous.

In order for a programmer to understand this result, he/she needs to be aware that the `:/2` operator both calls a predicate in another module and changes the calling context of the predicate to that module. The first use is expected. The second use is not intuitive, is not useful, and often not properly documented. First, in other programming languages, the choice between implicitly-qualified calls and explicitly-qualified calls is one of typing convenience to the programmer, not one of semantics. Second, in the most common case where a client is reusing a library meta-predicate, the client wants to customize the meta-predicate call with its own local predicate. Different clients will customize the call to the library meta-predicate using different local predicates. In those cases where the meta-predicate is defined and used locally, explicit qualification is seldom necessary. We can, however, conclude that the meta-predicate definition still works as expected as the calling context is set to the library module. If we still want the `me/1` predicate to be called in the context of the client module instead, we need to explicitly qualify the meta-argument by writing:

```
test(Me) :-
    library:my_call(client:me(Me)).
```

This is an awkward solution but it works as expected in the cases where explicit qualification is required. It should be noted, however, that the idea of the `meta_predicate/1` directive is to avoid the need for explicit qualifications in the first place. But that requires using `use_module/1-2` directives for importing the meta-predicates and implicit qualification when calling them. This explicit qualification of meta-arguments is not necessary in Logtalk or in the ECLiPSe module system, where explicit qualification of a meta-predicate call sets where to start looking for the meta-predicate definition, not where to look for the meta-arguments definitions.

The semantics of the `:/2` operator in Prolog module systems (derived from the Quintus Prolog module system) is rooted in optimization goals.[13] When a directive `use_module/1` is used, most (if not all) Prolog compilers require the definition of the imported module to be available, thus resolving the call at compilation time. However, that does not seem to be required when compiling an explicitly qualified module call. For example, using recent versions of SICStus Prolog, SWI-Prolog, and YAP, the following code compiles without errors or warnings (despite the fact that the module `fictitious` does not exist):

```
:- module(client, [test/1]).

test(X) :-
    fictitious:predicate(X).
```

Thus, in this case, the `fictitious:predicate/1` call is resolved at runtime. In our example above with the explicit call to the `my_call/1` meta-predicate, the

[13] The goal of the original Quintus Prolog module system, according to former developers at Quintus, was to design a system with zero overhead over plain Prolog.

implementation of the `:/2` operator propagates the module prefix to the meta-arguments that are not explicitly qualified at runtime. This runtime propagation results in a performance penalty. Therefore, and not surprisingly, the use of explicit qualification is discouraged by the Prolog implementers. In fact, until recently, most Prolog implementations provided poor performance for `:/2` calls even when the necessary module information was available at compile time.

Logtalk and ECLiPSe illustrate the first choice for the semantics of explicitly-qualified meta-predicate calls. Consequently, both systems provide the same semantics for implicitly and explicitly qualified meta-predicate calls. Consider the following objects, corresponding to a Logtalk version[14] of the Prolog module example used in the previous section:

```
:- object(library).

    :- public(my_call/1).
    :- meta_predicate(my_call(0)).
    my_call(Goal) :-
        write('Calling: '), writeq(Goal), nl,
        call(Goal),
        sender(Sender), write('Sender:  '), writeq(Sender).

    me(library).

:- end_object.


:- object(client).

    :- public(test/1).
    test(Me) :-                         % call the meta-predicate
        library::my_call(me(Me)).       % using explicit qualification

    me(client).

:- end_object.
```

Our test query becomes:

```
| ?- client::test(Me).
Calling: me(_)
Sender:  client
Me = client.
yes
```

That is, meta-arguments are always called in the context of the meta-predicate call. Logtalk also implements common built-in meta-predicates such as `call/1-N`,

[14] We extend the definition of the `my_call/1` meta-predicate to also print the *sender* of the `my_call/1` *message* by using Logtalk's built-in predicate `sender/1`.

`\+/1`, `findall/3`, and `phrase/3` with the same semantics as user-defined meta-predicates. In order to avoid misinterpretations, these built-in meta-predicates are implemented as private predicates.[15] Thus, the following call is illegal and results in a permission error:

```
| ?- an_object::findall(T, g(T), L).
error(permission_error(access, private_predicate, findall(T,g(T),L)),
      an_object::findall(T, g(T), L),
      user)
```

The correct call would be:

```
| ?- findall(T, an_object::g(T), L).
```

3.2 Transparency of Control Constructs

One of the design choices regarding meta-predicate semantics is the transparency of control constructs to explicit qualification. The relevance of this topic is that most control constructs can also be regarded as meta-predicates. In fact, there is a lack of agreement in the Prolog community on which language elements are control constructs and which language elements are predicates. For the purposes of our discussion, we use the classification found on the ISO Prolog standard, which specifies the following control constructs: `call/1`, *conjunction*, *disjunction*, *if-then*, *if-then-else*, and `catch/3`. The standard also specifies `true/0`, `fail/0`, `!/0`, and `throw/1` as control constructs but none of these can be interpreted as a meta-predicate.

When a control construct is transparent to explicit qualification, the qualification propagates to all the control constructs arguments that are not explicitly qualified. For example, the following equivalences hold for most Prolog module systems[16] (left column) and Logtalk (right column):[17]

```
M:(A, B)      ⇔ (M:A, M:B)        O::(A, B)      ⇔ (O::A, O::B)
M:(A; B)      ⇔ (M:A; M:B)        O::(A; B)      ⇔ (O::A; O::B)
M:(A -> B; C) ⇔ (M:A -> M:B; M:C) O::(A -> B; C) ⇔ (O::A -> O::B; O::C)
```

In Prolog module systems where the `:/1` operator sets both the meta-predicate lookup context and the meta-arguments calling context, the above equivalences are consistent with the explicit qualification semantics of meta-predicates described in the previous section. For example:

[15] Logtalk supports *private*, *protected*, and *public* predicates. A predicate may also be *local* if no scope directive is present, making the predicate invisible to the built-in reflection predicates (`current_predicate/1` and `predicate_property/2`).

[16] Note, however, that some Prolog compilers, such as Ciao and ECLiPSe, don't support explicit qualification of control constructs.

[17] Although both columns seem similar, the `::/2` Logtalk operator is a *message-sending* operator whose semantics differ from the module `:/2` *explicit-qualification* operator.

```
M:findall(T, G, L) ⇔ findall(T, M:G, L)
M:assertz(A)       ⇔ assertz(M:A)
```

This is also true for user-defined meta-predicates. For the example presented in the previous section, the following equivalence holds:

```
library:my_call(me(Me)) ⇔ my_call(library:me(Me))
```

Thus, the different semantics of implicitly and explicitly qualified meta-predicate calls allows the semantics of explicitly qualified control constructs to be consistent with the semantics of explicitly qualified meta-predicate calls.

In Logtalk, where explicit qualification of meta-predicates calls only sets the lookup context, the semantics of control constructs are different: the above equivalences are handy, supported, and can be interpreted as a shorthand notation for sending a set of messages to the same object. ECLiPSe implements a simpler design choice, disallowing the above shorthands, and thus treating control constructs and meta-predicates uniformly. We can conclude that ensuring the same semantics for implicitly and explicitly qualified meta-predicate calls requires either disallowing explicit qualification of control constructs (as found on e.g. Ciao and ECLiPSe) or different semantics for explicitly qualified control constructs, and thus a clear distinction between control constructs and predicates.

4 Computational Reflection Support

Computational reflection allows us to perform computations about the *structure* and the *behavior* of an application. For meta-predicates, structural reflection allows us to find where the meta-predicate is defined and about the meta-predicate template, while behavioral reflection allows us to access the meta-predicate execution context. As described in Section 1, a meta-predicate execution context includes information about from where the meta-predicate is called. This is only meaningful, however, in the presence of a predicate encapsulation mechanism such as modules or objects. Access to the execution-context is usually not required for common user-level meta-predicate definitions but can be necessary when meta-predicates are used to extend sytem meta-call features. In Logtalk, full access to predicate execution context is provided by the `sender/1`, `self/1`, `this/1`, and `parameter/2` built-in predicates. For Prolog compilers supporting predicate-based module systems, the following table provides an overview of the available reflection built-in predicates:

Prolog compiler	**Built-in reflection predicates**
Ciao 1.10	`predicate_property/2` (in library `prolog_sys`)
ECLiPSe 6.1	`get_flag/3`
SICStus Prolog 4.2	`predicate_property/2`
SWI-Prolog 5.10.4	`context_module/1`, `predicate_property/2`, `strip_module/3`
YAP 6.2	`context_module/1`, `predicate_property/2`, `strip_module/3`

From this table we conclude that the most common built-in predicate is `predicate_property/2`. Together with the ECLiPSe `get_flag/3` and the SWI-Prolog and YAP `context_module/1` predicates, these built-ins only provide *structural* reflection. Specifically, information about the meta-predicate template and the definition context of the meta-predicate. SWI-Prolog and YAP are the only systems that provide *built-in* access to the meta-predicate calling context using the predicate `strip_module/3`. As a simple example of using this predicate consider the following module:

```
:- module(m, [mp/2]).

:- meta_predicate(mp(0, -)).
mp(Goal, Caller) :-
    strip_module(Goal, Caller, _),
    call(Goal).
```

After compiling and loading this module, the following queries illustrate both the functionality of the `strip_module/3` predicate and the consequences of explicit qualification of the meta-predicate call:

```
| ?- mp(true, Caller).
Caller = user
yes

| ?- m:mp(true, Caller).
Caller = m
yes
```

For Prolog compiler module systems descending from the Quintus Prolog module system, it is possible to access the meta-predicate calling context by looking into the implicit qualification of a meta-argument:

```
:- module(m, [mp/2]).

:- meta_predicate(mp(0, -)).
mp(Goal, Caller) :-
    Goal = Caller:_,
    call(Goal).
```

After compiling and loading this module, we can reproduce the results illustrated by the queries above for the SWI-Prolog/YAP version of this module. One possible caveat would be if the Prolog compiler fails to ensure that there is always a single qualifier for a goal. That is, that terms such as `M1:(M2:(M3:G))` are never generated internally when propagating module qualifications.

In the case of ECLiPSe, a built-in predicate for accessing the meta-predicate calling context is not necessary. The `tool/2` directive works by connecting a meta-predicate interface with its implementation, which is extended with an extra argument that carries the meta-predicate calling context:

```
:- module(m).

:- export(mp/2).                % due to the tool/2 directive, the
:- tool(mp/2, mp/3).            % ECLiPSe runtime system passes the
mp(Goal, Caller, Caller) :-     % calling context of mp/2 in the
    call(Goal).                 % third argument of mp/3
```

After loading this module, repeating the above queries illustrates the difference in explicit qualification semantics between ECLiPSe and the other compilers:

```
[eclipse 16]: mp(true, Caller).
Caller = eclipse
Yes (0.00s cpu)

[eclipse 17]: m:mp(true, Caller).
Caller = eclipse
Yes (0.00s cpu)
```

Note that the module `eclipse` is the equivalent of the module `user` in other Prolog compilers.

5 Secure Meta-predicate Definitions

Meta-predicate definitions should not provide a mechanism for calling client predicates other than the ones intended by the meta-predicate calls. This, however, is mostly meaningful for languages such as Logtalk and for Prolog module systems, such as ECLiPSe and Ciao [12], that aim to enforce object and module predicate scope rules. The following set of compilation rules, discussed and illustrated in detail in [3], contribute to make meta-predicate definitions secure:

1. The meta-arguments of a meta-predicate clause head must be variables.
2. Meta-calls whose arguments are not variables appearing in meta-argument positions in the clause head must be compiled as calls to local predicates.
3. Meta-predicate closures must be used within a `call/2-N` built-in predicate call that complies with the corresponding meta-predicate directive.

These rules are implemented in Logtalk. For Prolog module systems whose design allows any module predicate to be called using explicit module qualification, these rules may be regarded as best practice for writing meta-predicates and thus useful for checking meta-predicate definitions for possible errors (e.g. as part of lint checkers). Note that the third compilation rule above requires a meta-predicate directive capable of representing the number of additional arguments taken by a closure. The reader is invited to consult [3] for full details.

6 Portability of Meta-predicate Definitions

The portability of meta-predicate definitions depends on three main factors: the use of implicit qualification when calling meta-predicates in order to avoid

the different semantics for explicitly qualified calls discussed in Section 3, the portability of the meta-predicate directives, and the portability of the meta-call primitives used when implementing the meta-predicates. Other factors that may impact portability are the preprocessing solutions for improving meta-predicate performance, described in Section 8, and the mechanisms for computational reflection about meta-predicate definition and execution, discussed in Section 4.

6.1 The call/1-N Control Constructs

The `call/1` control construct is specified in the ISO Prolog standard [8]. This control construct is implemented by virtually all Prolog compilers. The `call/2-N` control constructs[18], whose use is strongly recommended for meta-predicates working with closures [3], is included in the latest revision of the ISO Prolog Core standard. A growing number of Prolog compilers implement these control constructs but with different maximum values for `N`, which can raise some portability problems. Ideally, the `call/1-N` control constructs would support `N` up to the maximum predicate arity. That depends, however, on the design decisions of a Prolog compiler implementation. For the Prolog systems listed in the table below, only five out of twelve systems support a value of `N` up to the maximum predicate arity. From a pragmatic point-of-view, it is not common that user written code (but not necessarily user *generated* code) would require a large upper limit of `N`. Despite some lack of agreement, the only portability issue is Prolog compilers only supporting an arguably small value of `N`. The following table summarizes the implementations of the `call/2-N` control construct on selected Prolog compilers:

System	N	Notes
B-Prolog 7.4	10/65535	(interpreted/compiled i.e. maximum arity)
Ciao 1.10	255	(maximum arity using the `hiord` library)
CxProlog 0.95.0	9	—
ECLiPSe 6.1#68	255	(maximum arity)
GNU Prolog 1.3.1	11	—
JIProlog 3.0.2	5	—
K-Prolog 6.0.4	9	—
Qu-Prolog 8.12	9	—
SICStus Prolog 4.2	255	(maximum arity)
SWI-Prolog 5.10.4	8/1024	(interpreted/compiled i.e. maximum arity)
XSB 3.3	11	—
YAP 6.2	12	—

This table only lists *built-in* support for `call/2-N` control construct. While this control construct can be defined by the programmer using the built-in pred-

[18] A `call(Closure, Arg1, ...)` goal is true iff `call(Goal)` is true where `Goal` is constructed by appending `Arg1, ...` additional arguments to the arguments (if any) of the callable term `Closure`.

icate `=../2` and an `append/3` predicate, such definitions provide relative poor performance due to the construction and appending of temporary lists.

6.2 Specification of Closures and Instantiation Modes in Meta-predicate Directives

The main portability issue of meta-predicate directives is the use of non-negative integers to specify closures and the atoms used to specify the instantiation mode of normal arguments. Although the use of non-negative integers comes from Quintus Prolog, it was historically regarded as a way to provide information to cross-reference and documentation tools, with Prolog compilers accepting this notation only for backward-compatibility with existing code. Other Prolog compilers such as Ciao define alternative but incompatible syntaxes for specifying closures. There is also some variation in the atoms used for representing the instantiation modes of normal arguments. Some Prolog compilers use an extended set of atoms for documenting argument instantiation modes compared to the basic set (`+`, `?`, `@`, and `-`) found in the ISO Prolog standard. It is therefore tempting to use these extended sets in meta-predicate directives, which will likely raise portability issues. Hopefully, recent Prolog standardization initiatives, specially the development of portable libraries, will lead to a de facto standard meta-predicate directive derived from the extended directive described in Section 2.2.

7 Extending Meta-predicate Definitions Usefulness

The usefulness of meta-predicate definitions can be extended by adding support for lambda expressions. In the same way meta-predicates avoid the repeated coding of common programming patterns, lambda expressions avoid the definition of auxiliary predicates whose sole purpose is to be used as arguments in meta-predicate calls. Consider the following example (using Logtalk lambda expression syntax[19]) where we compute the distance to the origin for each point in a list using a mapping meta-predicate:

```
| ?- meta::map([(X,Y),Z]>>(Z is sqrt(X*X+Y*Y)),[(1,4),(2,5),(8,3)],Ds).
Ds = [4.1231056256176606,5.3851648071345037,8.5440037453175304]
yes
```

Without lambda expressions, it would be necessary to define an auxiliary predicate to compute the distance from a point to the origin:

```
distance((X, Y), Distance) :-
    Distance is sqrt(X*X+Y*Y).

| ?- meta::map(distance, [(1,4),(2,5),(8,3)], Ds).
Ds = [4.1231056256176606,5.3851648071345037,8.5440037453175304]
yes
```

[19] `http://logtalk.org/manuals/refman/grammar.html#grammar_lambdas`

This example also illustrates an additional issue when using meta-predicates: the `map/3` list mapping meta-predicate accepts as first argument a closure that is extended by *appending* two arguments. Thus, an existing predicate for calculating the distance, e.g. `distance(X, Y, Distance)`, cannot not be used without writing an auxiliary predicate for the sole purpose of packing the first two arguments.

Native support for lambda expressions can be found in e.g. λProlog [13], Qu-Prolog, and Logtalk. For Prolog compilers supporting a module system, a library is available [14] that adds lambda expressions support. There is, however, a lack of community agreement on lambda expression syntax. But the main issue of taking advantage of lambda expressions is the performance penalty resulting from the runtime processing of lambda parameters. In the case of Logtalk, recent releases include a preprocessor for both meta-predicates and lambda expressions that eliminate the performance penalty when compared with hand-coded and optimized (non-meta-predicate) alternative solutions.

8 Meta-predicate Performance

Considering that meta-programming is often touted as a major feature of Prolog, the relative poor performance of meta-calls often drive programmers to avoid using meta-predicates in production code where performance is crucial. A common solution is to interpret meta-predicate definitions as high-level macros and to preprocess meta-predicate calls in order to replace them with calls to automatically generated auxiliary predicates whose definitions that do not contain meta-calls. This preprocessing is usually only performed on stable code as the auxiliary predicates often complicate debugging. The preprocessing code is often implemented in optional libraries, which can be found on Logtalk and several Prolog compilers such as ECLiPSe, SWI-Prolog, and YAP. Consider as an example adding `1` to every integer in the list `[1..100000]` using a simple recursive predicate, a mapping meta-predicate using a closure, and a mapping meta-predicate using a lambda expression, with and without preprocessing.[20] Using Logtalk 2.43.2 with YAP 6.3.0 we get (times in seconds):

	Non-optimized	Optimized
Recursive predicate	`0.002`	`0.002`
Mapping predicate with a closure	`0.072`	`0.008`
Mapping predicate with a lambda expression	`0.119`	`0.004`

Equivalent results are obtained using Prolog implementations of this example with meta-predicate and preprocessing libraries. A performance penalty of one order of magnitude is commonly observed when comparing hand-optimized code with meta-predicate alternatives without any preprocessing. The use of lambda

[20] The full source code of this example is available at `http://trac.logtalk.org/browser/trunk/examples/lambdas` (no preprocessing) and `http://trac.logtalk.org/browser/trunk/examples/lambdas_compiled` (using preprocessing).

expressions adds another source of performance penalty. As illustrated above, preprocessing both the meta-predicate calls and the lambda expressions closes the performance gap. But the preprocessing libraries require custom code for each meta-predicate. Thus, user-defined meta-predicates will fail to match the performance of library-supported meta-predicates unless the user also writes its own custom preprocessing code. A more generic solution for preprocessing meta-predicate definitions, based on more powerful compile time code analysis and partial evaluation techniques, is needed to make meta-predicate programming patterns more appealing for applications where performance is crucial.

9 Conclusions and Future Work

We presented and discussed a comprehensive set of meta-predicate design decisions based on current practice in Logtalk and in Prolog predicate-based module systems. An interesting result is that none of the two commonly implemented semantics for explicitly qualified calls provides an ideal solution that both meets user expectations and allows the distinction between meta-predicates and control constructs to be waived. By describing the consequences of these design decisions we provided useful insight to discuss meta-predicate semantics, often a difficult subject for inexperienced programmers and a source of misunderstandings when porting applications and discussing Prolog standardization. From the point-of-view of writing portable code (including portable libraries), the current state of meta-predicate syntax and semantics across Prolog compilers is still a challenge, despite recent community efforts. We hope that this paper contributes to a convergence of meta-predicate directive syntax, meta-predicate semantics, and meta-predicate related reflection built-in predicates among Prolog compilers. But the main obstacle to improving the de facto standardization of meta-predicate syntax and semantics is backwards compatibility. Understandably, most Prolog implementers are wary of making changes that would break compatibility with existing applications and upset long time users. Nevertheless, we recommend that Prolog module systems make the semantics of implicitly- and explicitly-qualified meta-predicate calls the same (as found in Logtalk and ECLiPSe) and forbid the explicit qualification of control constructs (as found in ECLiPSe and Ciao) and built-in meta-predicates (as found in Logtalk). These changes would contribute to simpler and more uniform semantics while avoid programming constructs with unclear meaning for novice programmers.

Future work will include comparing Logtalk and Prolog predicate-based module systems with Prolog atom-based module system and derived object-oriented extensions. The most prominent example of a Prolog compiler featuring an atom-based module system is XSB [15] (which is used in the implementation of the object-oriented extension Flora [16]). XSB does not support a meta-predicate directive. Explicit-qualification of meta-arguments is used whenever the atom-based semantics fail to provide the desired behavior for the implementation of a specific meta-predicate. Interestingly, although atom-based module systems appear to solve or avoid some the issues discussed along this paper, predicate-based

module systems are the most common implementation choice. This may be due to historical reasons but a deep understanding of the pros and cons of atom-based systems, at both the conceptual and implementation levels, will be required to perform a detailed comparison with the better known predicate-based systems.

Acknowledgements. We are grateful to Joachim Schimpf, Ulrich Neumerkel, Jan Wielemaker, and Richard O'Keefe for their feedback on explicitly-qualified meta-predicate call semantics in predicate-based module systems. We thank also the anonymous reviewers for their informative comments. This work was partially supported by the LEAP (PTDC/EIA-CCO/112158/2009) research project.

References

1. Moura, P.: Logtalk – Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
2. Moura, P.: Logtalk 2.43.2 User and Reference Manuals (October 2011)
3. Moura, P.: Secure Implementation of Meta-predicates. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 269–283. Springer, Heidelberg (2009)
4. ISO/IEC: International Standard ISO/IEC 13211-2 Information Technology — Programming Languages — Prolog — Part II: Modules. ISO/IEC (2000)
5. O'Keefe, R.: An Elementary Prolog Library, `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.html`
6. Swedish Institute for Computer Science: Quintus Prolog User's Manual (Release 3.5). Swedish Institute for Computer Science (December 2003)
7. Moura, P. (ed.): ISO/IEC DTR 13211–5:2007 Prolog Multi-threading predicates, `http://logtalk.org/plstd/threads.pdf`
8. ISO/IEC: International Standard ISO/IEC 13211-1 Information Technology — Programming Languages — Prolog — Part I: General core. ISO/IEC (1995)
9. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M.V., López, P., Puebla, G.: Ciao Prolog System Manual
10. Cheadle, A.M., Harvey, W., Sadler, A.J., Schimpf, J., Shen, K., Wallace, M.G.: ECLiPSe: A tutorial introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College, London (2003)
11. Wielemaker, J.: An overview of the SWI-Prolog programming environment. In: Mesnard, F., Serebenik, A. (eds.) Proceedings of the 13th International Workshop on Logic Programming Environments, Heverlee, Belgium, pp. 1–16. Katholieke Universiteit Leuven (December 2003); CW 371
12. Cabeza, D., Hermenegildo, M.V.: A New Module System for Prolog. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 131–148. Springer, Heidelberg (2000)
13. Nadathur, G., Miller, D.: An Overview of λProlog. In: Fifth International Logic Programming Conference, Seattle, pp. 810–827. MIT Press (August 1988)
14. Neumerkel, U.: Lambdas in ISO Prolog, `http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/ISO-Hiord`
15. Group, T.X.R.: The XSB Programmer's Manual: version 3.3 (April 2011)
16. Yang, G., Kifer, M.: Flora-2: User's manual (2001)