Towards a Study of Meta-Predicate Semantics

Paulo Moura
Dep. of Computer Science, Univ. of Beira Interior, Portugal
Center for Research in Advanced Computing Systems
INESC Porto, Portugal

http://logtalk.org/ pmoura@logtalk.org CICLOPS 2010

Motivation

Motivation

- Logtalk needs to correctly compile calls to built-in meta-predicates and module meta-predicates.

Motivation

- Logtalk needs to correctly compile calls to built-in meta-predicates and module meta-predicates.
- Lack of Prolog standard meta-predicate directives and meta-predicate semantics is a portability nightmare.

Meta-predicates

Meta-predicates

- Meta-predicates are predicates with one or more arguments that are called as goals on the body of a predicate clause.

Meta-predicates

- Meta-predicates are predicates with one or more arguments that are called as goals on the body of a predicate clause.
- Meta-arguments may also be *closures*.

Meta-predicates

- Meta-predicates are predicates with one or more arguments that are called as goals on the body of a predicate clause.
- Meta-arguments may also be *closures*.
- A closure is a callable term used to construct a goal by appending one or more arguments.

Meta-predicates

Meta-predicates

- Predicates that modify other predicates or provide access to predicate information are also often classified as meta-predicates.

Meta-predicates

- Predicates that modify other predicates or provide access to predicate information are also often classified as meta-predicates.

- Examples include database predicates (e.g. `assertz/1`, `retract/1`, `clause/2`) and reflection predicates (e.g. `current_predicate/1`, `predicate_property/2`).

Meta-predicates

Meta-predicates

- Meta-predicates allows reuse of programming patterns.

Meta-predicates

- Meta-predicates allows reuse of programming patterns.
- Meta-predicates are particularly useful in the presence of encapsulation mechanisms such as modules or objects.

Meta-predicates

- Meta-predicates allows reuse of programming patterns.
- Meta-predicates are particularly useful in the presence of encapsulation mechanisms such as modules or objects.
- Defining an exported or public meta-predicate within a module or an object allows client modules and objects to reuse predicates customized by calls to local predicates.

Some examples

Some examples

```
foreach([], _, _).
foreach([Element| List], Count, Goal) :-
   \+ \+ (Count = Element, call(Goal)),
   foreach(List, Count, Goal).
```

Some examples

```
foreach([], _, _).
foreach([Element| List], Count, Goal) :-
   \+ \+ (Count = Element, call(Goal)),
   foreach(List, Count, Goal).


fold_left(_, Result, [], Result).
fold_left(Closure, Acc, [Arg| Args], Result) :-
   call(Closure, Acc, Arg, Acc2),
   fold_left(Closure, Acc2, Args, Result).
```

Meta-predicate semantics topics

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives
- Explicit qualification semantics

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives
- Explicit qualification semantics
- Transparency of control constructs

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives
- Explicit qualification semantics
- Transparency of control constructs
- Computational reflection support

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives
- Explicit qualification semantics
- Transparency of control constructs
- Computational reflection support
- Safety of meta-predicate definitions

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives
- Explicit qualification semantics
- Transparency of control constructs
- Computational reflection support
- Safety of meta-predicate definitions
- Portability of meta-predicate directives and definitions

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives
- Explicit qualification semantics
- Transparency of control constructs
- Computational reflection support
- Safety of meta-predicate definitions
- Portability of meta-predicate directives and definitions
- Meta-predicate performance

Meta-predicate semantics topics

- Expressiveness of meta-predicate directives
- Explicit qualification semantics
- Transparency of control constructs
- Computational reflection support
- Safety of meta-predicate definitions
- Portability of meta-predicate directives and definitions
- Meta-predicate performance
- Extending meta-predicate usefulness

Prolog meta-predicate directives

Prolog meta-predicate directives

- ISO Prolog standard `metapredicate/1`

Prolog meta-predicate directives

- ISO Prolog standard `metapredicate/1`
- Quintus `meta_predicate/1`

Prolog meta-predicate directives

- ISO Prolog standard `metapredicate/1`
- Quintus `meta_predicate/1`
- ECLiPSe `tool/2`

Prolog meta-predicate directives

- ISO Prolog standard `metapredicate/1`
- Quintus `meta_predicate/1`
- ECLiPSe `tool/2`
- SWI-Prolog `module_transparent/1`

Extended `meta_predicate/1` directive

Extended `meta_predicate/1` directive

- Used on recent versions of Qu-Prolog, SICStus Prolog, SWI-Prolog, YAP (others to follow)

Extended `meta_predicate/1` directive

- Used on recent versions of Qu-Prolog, SICStus Prolog, SWI-Prolog, YAP (others to follow)
- Goals represented by the integer zero

Extended `meta_predicate/1` directive

- Used on recent versions of Qu-Prolog, SICStus Prolog, SWI-Prolog, YAP (others to follow)
- Goals represented by the integer zero
- Closures represented by non-negative integers

Extended `meta_predicate/1` directive

- Used on recent versions of Qu-Prolog, SICStus Prolog, SWI-Prolog, YAP (others to follow)
- Goals represented by the integer zero
- Closures represented by non-negative integers
- Other meta-arguments represented by "`:`" ("`::`" in the case of Logtalk)

Extended `meta_predicate/1` directive

- Used on recent versions of Qu-Prolog, SICStus Prolog, SWI-Prolog, YAP (others to follow)
- Goals represented by the integer zero
- Closures represented by non-negative integers
- Other meta-arguments represented by "`:`" ("`::`" in the case of Logtalk)
- Non meta-arguments represented by either "`*`" or an instantiation mode indicator ("`@`", "`+`", "`-`", or "`?`")

Limitations of the extended `meta_predicate/1` directive

- Using mode indicators in `meta_predicate/1` directives is no replacement for `mode/2` directives. An example:

```
:- meta_predicate(forall(0, 0)).
:- meta_predicate(setof(@, 0, -)).
```

For `forall/2`, "`0`" means "`@`" but for `setof/3` "`0`" means "+".

Limitations of the extended `meta_predicate/1` directive

Limitations of the extended `meta_predicate/1` directive

- No representation solution for meta-arguments that are sub-terms of a meta-predicate argument.
  An example:

  ```
  :- meta_predicate(thread_create(0, -, :)).
  ```

  The third argument of `thread_create/3` is a list of options that may contain an `at_exit/1` goal.

Limitations of the extended `meta_predicate/1` directive

- No representation solution for meta-arguments that are sub-terms of a meta-predicate argument.
  An example:

  ```
  :- meta_predicate(thread_create(0, -, :)).
  ```

  The third argument of `thread_create/3` is a list of options that may contain an `at_exit/1` goal.

- Only the meta-predicate implementor knows how to process the ":" meta-argument!

Some Definitions

Some Definitions

Definition context

Some Definitions

Definition context

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

Some Definitions

Definition context

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

Calling context

Some Definitions

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.

Some Definitions

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

Some Definitions

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 comprises both the calling context and the definition context. It includes all the information needed for the language runtime to execute a meta-predicate call.

Some Definitions

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 comprises both the calling context and the definition context. It includes all the information needed for the language runtime to execute a meta-predicate call.

Lookup context

Some Definitions

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 comprises both the calling context and the definition context. It includes all the information needed for the language runtime to execute a meta-predicate call.

Lookup context

This is the object or module where we start looking for the meta-predicate definition. The definition can always be reexported from another module or inherited from another object.

Explicit Qualification Semantics

Explicit Qualification Semantics

- The explicit qualification sets both the initial lookup context for the meta-predicate definition and the meta-predicate calling context. 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).

Explicit Qualification Semantics

- The explicit qualification sets both the initial lookup context for the meta-predicate definition and the meta-predicate calling context. 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).

- Semantics used e.g. on Ciao, Quintus Prolog, SICStus Prolog, SWI-Prolog, and YAP.

Explicit Qualification Semantics

- The explicit qualification sets both the initial lookup context for the meta-predicate definition and the meta-predicate calling context. 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).

- Semantics used e.g. on Ciao, Quintus Prolog, SICStus Prolog, SWI-Prolog, and YAP.

- Why? Explicit (module) qualification is seldom used. Without it, non-qualified meta-arguments are called in the meta-predicate calling context, matching user expectations.

Explicit Qualification Semantics

Explicit Qualification Semantics

- The explicit qualification sets only the initial lookup context for the meta-predicate definition. All meta-arguments that are not explicitly-qualified are called in the meta-predicate calling context.

Explicit Qualification Semantics

- The explicit qualification sets only the initial lookup context for the meta-predicate definition. All meta-arguments that are not explicitly-qualified are called in the meta-predicate calling context.

- Semantics used e.g. on ECLiPSe and Logtalk.

Explicit Qualification Semantics

- The explicit qualification sets only the initial lookup context for the meta-predicate definition. All meta-arguments that are not explicitly-qualified are called in the meta-predicate calling context.

- Semantics used e.g. on ECLiPSe and Logtalk.

- Why? Same semantics for explicit and implicit qualification, matching user expectations. Non-qualified meta-arguments are always called in the meta-predicate calling context.

Explicit qualification sets both lookup and calling contexts

Explicit qualification sets both lookup and calling contexts

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

:- meta_predicate(my_call(0)).

my_call(Goal) :-
    write('Calling: '), writeq(Goal), nl,
    call(Goal).

me(library).
```

Explicit qualification sets both lookup and calling contexts

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

:- meta_predicate(my_call(0)).

my_call(Goal) :-
  write('Calling: '), writeq(Goal), nl,
  call(Goal).

me(library).
```

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

:- use_module(library, [my_call/1]).

test(Me) :-
  my_call(me(Me)).

me(client).
```

Explicit qualification sets both lookup and calling contexts

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

:- meta_predicate(my_call(0)).

my_call(Goal) :-
    write('Calling: '), writeq(Goal), nl,
    call(Goal).

me(library).
```

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

:- use_module(library, [my_call/1]).

test(Me) :-
    my_call(me(Me)).

me(client).
```

```
?- client:test(Me).

Calling: client:me(_)
Me = client
yes
```

Explicit qualification sets both lookup and calling contexts

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

:- meta_predicate(my_call(0)).

my_call(Goal) :-
  write('Calling: '), writeq(Goal), nl,
  call(Goal).

me(library).
```

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

:- use_module(library, [my_call/1]).

test(Me) :-
  my_call(me(Me)).

me(client).
```

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


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

me(client).
```

```
?- client:test(Me).

Calling: client:me(_)
Me = client
yes
```

Explicit qualification sets both lookup and calling contexts

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

:- meta_predicate(my_call(0)).

my_call(Goal) :-
  write('Calling: '), writeq(Goal), nl,
  call(Goal).

me(library).
```

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

:- use_module(library, [my_call/1]).

test(Me) :-
  my_call(me(Me)).

me(client).


?- client:test(Me).

Calling: client:me(_)
Me = client
yes
```

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


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

me(client).


?- client:test(Me).

Calling: library:me(_)
Me = library
yes
```

Explicit qualification sets only the lookup context

Explicit qualification sets only the lookup context

```
:- object(library).

:- public(my_call/1).
:- meta_predicate(my_call(::)).

my_call(Goal) :-
  write('Calling: '), writeq(Goal),
  nl, call(Goal), sender(Sender),
  write('Sender: '), writeq(Sender).

me(library).

:- end_object.
```

Explicit qualification sets only the lookup context

```
:- object(library).

:- public(my_call/1).
:- meta_predicate(my_call(::)).

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

:- end_object.
```

Explicit qualification sets only the lookup context

```
:- object(library).

:- public(my_call/1).
:- meta_predicate(my_call(::)).

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

:- end_object.
```

```
?- client::test(Me).

Calling: me(_G216)
Sender: client
Me = client.
yes
```

Transparency of control constructs

Transparency of control constructs

- Control constructs can be interpreted as meta-predicates.

Transparency of control constructs

- Control constructs can be interpreted as meta-predicates.
- Which semantics for explicitly qualified control constructs?

Transparency of control constructs

- Control constructs can be interpreted as meta-predicates.
- Which semantics for explicitly qualified control constructs?
- But should we really make a distinction between control constructs and meta-predicates?

Transparency of control constructs

- Control constructs can be interpreted as meta-predicates.
- Which semantics for explicitly qualified control constructs?
- But should we really make a distinction between control constructs and meta-predicates?
- Lack of agreement on the Prolog community!

Transparency of control constructs

Transparency of control constructs

- Control constructs in the ISO Prolog standard:
  - `call/1`, *conjunction*, *disjunction*, *if-then*, *if-then-else*, and `catch/3`
  - `true/0`, `fail/0`, `!/0`, and `throw/1`

Transparency of control constructs

- Control constructs in the ISO Prolog standard:
  - `call/1`, *conjunction*, *disjunction*, *if-then*, *if-then-else*, and `catch/3`
  - `true/0`, `fail/0`, `!/0`, and `throw/1`
- Control constructs in ISO Prolog standardization proposals:
  - `call/2-N`

Transparency of control constructs

Transparency of control constructs

- Prolog module systems:

Transparency of control constructs

- Prolog module systems:

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

Transparency of control constructs

- Prolog module systems:

```
M:(A, B) ⇔ (M:A, M:B)

M:(A; B) ⇔ (M:A; M:B)

M:(A -> B; C) ⇔ (M:A -> M:B; M:C)
```

- Logtalk:

Transparency of control constructs

- Prolog module systems:

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

- Logtalk:

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

Transparency of control constructs

Transparency of control constructs

- Consistent with systems where explicit qualification sets both the lookup and calling contexts:

Transparency of control constructs

- Consistent with systems where explicit qualification sets both the lookup and calling contexts:

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

Transparency of control constructs

- Consistent with systems where explicit qualification sets both the lookup and calling contexts:

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

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

Transparency of control constructs

- Consistent with systems where explicit qualification sets both the lookup and calling contexts:

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

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

- No distinction between control constructs and meta-predicates is necessary!

Transparency of control constructs

Transparency of control constructs

- What about systems where explicit qualification only sets the lookup and calling context?

Transparency of control constructs

- What about systems where explicit qualification only sets the lookup and calling context?
- ECLiPSe: (1) makes a distinction between control constructs and meta-predicates; (2) control constructs cannot be explicitly qualified.

Transparency of control constructs

- What about systems where explicit qualification only sets the lookup and calling context?
- ECLiPSe: (`1`) makes a distinction between control constructs and meta-predicates; (`2`) control constructs cannot be explicitly qualified.
- Logtalk: (`1`) makes a distinction between control constructs and meta-predicates; (`2`) built-in meta-predicates are private (e.g. `call/1`, `findall/3`).

Transparency of control constructs

- What about systems where explicit qualification only sets the lookup and calling context?
- ECLiPSe: (`1`) makes a distinction between control constructs and meta-predicates; (`2`) control constructs cannot be explicitly qualified.
- Logtalk: (`1`) makes a distinction between control constructs and meta-predicates; (`2`) built-in meta-predicates are private (e.g. `call/1`, `findall/3`).
- Both solutions ensure consistency with explicit qualification semantics.

Secure Meta-Predicate Definitions

Secure Meta-Predicate Definitions

- The meta-arguments of a meta-predicate clause head must be variables.

Secure Meta-Predicate Definitions

- The meta-arguments of a meta-predicate clause head must be variables.
- Meta-calls whose arguments are not variables appearing in meta-argument positions in the clause head must be compiled as calls to local predicates.

Secure Meta-Predicate Definitions

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

Computational reflection support

Prolog compiler	Structural reflection built-in predicates	Behavioral reflection built-in predicates
Ciao 1.10	`predicate_property/2` (in library `prolog_sys`)	(user-defined solution possible)
ECLiPSe 6.0	`get_flag/3`	(not necessary; see `tool/2` directive)
SICStus Prolog 4.1	`predicate_property/2`	(user-defined solution possible)
SWI-Prolog 5.9.10	`context_module/1` `predicate_property/2`	`strip_module/3`
XSB 3.2	`predicate_property/2`	?????
YAP 6.0	`context_module/1` `predicate_property/2`	(user-defined solution possible)

Behavioral reflection workarounds

Behavioral reflection workarounds

Quintus (*)

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

:- meta_predicate(mp(0, -)).

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

Behavioral reflection workarounds

Quintus (*)

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

:- meta_predicate(mp(0, -)).

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

(*) As long as no terms like `M1:(M2:(M3:G))` are never generated internally when propagating module qualifications!

Behavioral reflection workarounds

Quintus (*)

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

:- meta_predicate(mp(0, -)).

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

ECLiPSe

```
:- module(m).

:- export(mp/2).
:- tool(mp/2, mp/3).

mp(Goal, Caller,
Caller) :-
  call(Goal).
```

(*) As long as no terms like `M1:(M2:(M3:G))` are never generated internally when propagating module qualifications!

`call/1-N` control constructs

Prolog compiler	N	Notes
B-Prolog 7.4	10/65535	(interpreter/compiler – maximum arity)
Ciao 1.10	255	(maximum arity using the `hiord` library)
CxProlog 0.94.0	9	—
ECLiPSe 6.0	1	—
GNU Prolog 1.3.1	11	—
JIProlog 3.0.2	5	—
K-Prolog 6.0.4	9	—
Qu-Prolog 8.10	1	(supports a `call_predicate/1-5` built-in predicate)
SICStus Prolog 4.1	255	(maximum arity)
SWI-Prolog 5.9.10	8	(`meta_predicate/1` directive limit)
XSB 3.2	11	—
YAP 6.0	12	—

Meta-predicate performance

Meta-predicate performance

- Relative poor performance of meta-predicates compared with straight predicate alternatives.

Meta-predicate performance

- Relative poor performance of meta-predicates compared with straight predicate alternatives.
- Performance of meta-calls (native `call/2-N` implementations; avoid building temporary lists).

Meta-predicate performance

- Relative poor performance of meta-predicates compared with straight predicate alternatives.
- Performance of meta-calls (native `call/2-N` implementations; avoid building temporary lists).
- Pre-processing of meta-predicate definitions (but usually only for system-provided meta-predicates).

Extending meta-predicate usefulness

Extending meta-predicate usefulness

- Qu-Prolog lambda expressions
  `http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html`

Extending meta-predicate usefulness

- Qu-Prolog lambda expressions
  `http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html`

- Ulrich Neumerkel's lambda library
  `http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/ISO-Hiord`

Extending meta-predicate usefulness

- Qu-Prolog lambda expressions
  `http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html`

- Ulrich Neumerkel's lambda library
  `http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/ISO-Hiord`

- Logtalk lambda expressions
  `http://logtalk.org/manuals/refman/grammar.html#grammar_lambdas`

Conclusions

Conclusions

- Still far from a de facto standard for declaring and defining meta-predicates.

Conclusions

- Still far from a de facto standard for declaring and defining meta-predicates.

- Most Prolog compilers without a module system don't define meta-predicate properties for built-in meta-predicates.

Conclusions

- Still far from a de facto standard for declaring and defining meta-predicates.
- Most Prolog compilers without a module system don't define meta-predicate properties for built-in meta-predicates.
- `call/2-N` still missing in some Prolog compilers.

Conclusions

- Still far from a de facto standard for declaring and defining meta-predicates.
- Most Prolog compilers without a module system don't define meta-predicate properties for built-in meta-predicates.
- `call/2-N` still missing in some Prolog compilers.
- The extended `meta_predicate/1` directive provides essential information for preventing misuse of closures.

Conclusions

Conclusions

- Explicit qualification sets both lookup and calling contexts:

Conclusions

- Explicit qualification sets both lookup and calling contexts:
    - Different semantics for explict and implicit meta-predicate calls: fails to meet user expectations.

Conclusions

- Explicit qualification sets both lookup and calling contexts:
  - Different semantics for explict and implicit meta-predicate calls: fails to meet user expectations.
  - Not necessary to distinguish between control constructs and meta-predicates.

Conclusions

Conclusions

- Explicit qualification sets only the lookup context:

Conclusions

- Explicit qualification sets only the lookup context:
  - Same semantics for explict and implicit meta-predicate calls: meets user expectations.

Conclusions

- Explicit qualification sets only the lookup context:
  - Same semantics for explict and implicit meta-predicate calls: meets user expectations.
  - Same semantics for built-in meta-predicates and user meta-predicates.

Conclusions

- Explicit qualification sets only the lookup context:
  - Same semantics for explict and implicit meta-predicate calls: meets user expectations.
  - Same semantics for built-in meta-predicates and user meta-predicates.
  - Necessary to distinguish between control constructs and meta-predicates.