Message sending

Note that message sending is only the same as calling an object's predicate if the object does not inherit (or import) predicate definitions from other objects (or categories). Otherwise, the predicate definition that will be executed may depend on the relationships of the object with other Logtalk entities.

Operators used in message sending

Logtalk uses the following three operators for message sending:

```
:- op(600, xfx, ::).
:- op(600,  fx, ::).
:- op(600,  fx, ^^).
```

It is assumed that these operators remain active (once the Logtalk preprocessor and runtime files are loaded) until the end of the Prolog session (this is the usual behavior of most Prolog compilers). Note that these operator definitions are compatible with the pre-defined operators in the Prolog ISO standard.

Sending a message to an object

Sending a message to an object is accomplished by using the `::/2` infix operator:

```
| ?- Object::Message.
```

The message must match a public predicate declared for the receiving object or a Logtalk/Prolog built-in predicate, otherwise an exception will be thrown (see the Reference Manual for details).

Broadcasting

In the Logtalk context, broadcasting is interpreted as the sending of the same message to a group of objects or the sending of several messages to the same object. Both needs can be achieved by using the message sending method described above. However, for convenience, Logtalk implements an extended syntax for message sending that makes programming easier in these situations.

If we wish to send several messages to the same object, we can write:

```
| ?- Object::(Message1, Message2, ...).
```

This is semantically equivalent to:

```
| ?- Object::Message1, Object::Message2, ... .
```

We can also write:

```
| ?- Object::(Message1; Message2; ...).
```

This will be semantically equivalent to writing:

```
| ?- Object::Message1; Object::Message2; ... .
```

To send the same message to a set of objects we can write:

```
| ?- (Object1, Object2, ...)::Message.
```

This will have the same semantics as:

```
| ?- Object1::Message, Object2::Message, ... .
```

If we want to use backtracking to try the same message over a set of objects we can write:

```
| ?- (Object1; Object2, ...)::Message.
```

This will be equivalent to:

```
| ?- Object1::Message; Object2::Message; ... .
```

Sending a message to *self*

While defining a predicate, we sometimes need to send a message to *self*, i.e., to the same object that has received the original message. This is done in Logtalk through the `::/1` prefix operator:

```
::Message
```

We can also use the broadcasting constructs with this operator:

```
::(Message1, Message2, ...)
```

or:

```
::(Message1; Message2; ...)
```

The message must match a public or protected predicate declared for the receiving object, a private predicate within the scope of the *sender*, or a Logtalk/Prolog built-in predicate otherwise an error will be thrown (see the Reference Manual for details). If the message is sent from inside a category or if we are using private inheritance, then the message may also match a private predicate.

Calling an overridden predicate definition

When redefining a predicate, sometimes we need to call the inherited definition in the new code. This possibility, introduced by the Smalltalk language through the `super` primitive, is available in Logtalk through the `^^/1` prefix operator:

```
^^Predicate
```

Most of the time we will use this operator by instantiating the pattern:

```
Predicate :-
    ...,            % do something
    ^^Predicate,    % call inherited definition
    ... .           % do something more
```

Message sending and event generation

Every message sent using `::/2` operator generates two events, one before and one after the message execution. Messages that are sent using the `::/1` (message to *self*) operator or the `^^/1` super mechanism described above do not generate any events. The rational behind this distinction is that messages to *self* and *super* calls are only used indirectly in the definition of methods or to execute additional messages with the same target object (represented by *self*). In other words, events are only generated when using an object's public interface. They can not be used to break object encapsulation.

If we need to generate events for a public message sent to *self*, then we just need to write something like:

```
Predicate :-
    ...,
    self(Self),       % get self reference
    Self::Message,    % send a message to self using ::/2
    ... .
```

If we also need the sender of the message to be other than the object containing the predicate definition, we can write:

```
Predicate :-
    ...,
    self(Self),         % get self reference
    {Self::Message},    % send a message to self using ::/2
    ... .               % sender will be the pseudo-object user
```

See the session on Event-driven programming for more details.

Message sending performance

Logtalk implements dynamic binding, coupled with a cache mechanism that avoids repeated lookups of predicate declarations and predicate definitions for the same messages. This is a solution common to other programming languages supporting dynamic binding. Message lookups are automatically cached the first time a message is sent. Cache entries are automatically removed when loading entities or using Logtalk dynamic features which invalidate the cached lookups.

When discussing Logtalk message sending performance, two distinct cases should be considered: messages sent by the user from the top-level interpreter and messages sent from compiled objects. In addition, the message declaration and definition lookups may, or may not be already cached by the runtime engine. In what follows, we will assume that the message lookups are already cached.

Translating message processing to predicate calls

In order to better understand the performance tradeoffs of using Logtalk when compared to plain Prolog or to Prolog module systems, is useful to translate message processing in terms of predicate calls. However, in doing this, we should keep in mind that the number of predicate calls is not necessarily proportional to the time taken to execute them.

A message sent from a compiled object to another object translates to six predicate calls:

cache lookup
: one predicate call to a dynamic table

checking for *before* events
: two predicate calls assuming that no events are defined
(one of them to the built-in predicate `\+/1`)

method call
: one predicate call

checking for *after* events
: two predicate calls assuming that no events are defined
(one of them to the built-in predicate `\+/1`)

Given that events can be dynamically defined at runtime, there is no room for reducing the number of predicate calls without turning off support for event-driven programming. When events are defined, the number of predicate calls grows proportional to the number of events and event handlers (monitors). Event-driven programming support can be switched off for specific object using the compiler flag `events`. Doing so, reduces the number of predicate calls from six to just two.

Messages to *self* and *super* calls are transparent regarding events and, as such, imply only two predicate calls (the cache lookup and the method call).

When a message is sent by the user from the top-level interpreter, Logtalk needs to perform a runtime translation of the message in order to prove the corresponding goal. For user-defined messages/predicates, the runtime translation overhead corresponds to seventeen predicate calls. Thus, while sending a message from a compiled object corresponds to six predicate calls, the same message sent by the user from the top-level interpreter results in twenty-three predicate calls. Considering the time taken for the user to type the goal, this overhead is of no practical consequence.

When a message is not cached, the number of predicate calls depends on the number of steps needed for the Logtalk runtime engine to lookup the corresponding predicate scope declaration (to check if the message is valid) and then to lookup a predicate definition for answering the message.

Processing time

Not all predicate calls take the same time. Moreover, the time taken to process a specific predicate call depends on the Prolog compiler implementation details. As such, the only valid performance measure is the time taken for processing a message.

The usual way of measuring the time taken by a predicate call is to repeat the call a number of times and than to calculate the average time. A sufficient large number of repetitions would hopefully lead to an accurate measure. Care should be taken to subtract the time taken by the repetition code itself. In addition, we should be aware of any limitations of the predicates used to measure execution times. One way to make sense of numbers we get is to repeat the test with the same predicate using plain Prolog and with the predicate encapsulated in a module.

A simple predicate for helping benchmarking predicate calls could be:

```
benchmark(N, Goal) :-
    repeat(N),
        call(Goal),
    fail.

benchmark(_, _).
```

The rational of using a failure-driven loop is to try to avoid any interference on our timing measurements from garbage-collection or memory expansion mechanisms. Based on the predicate `benchmark/2`, we may define a more convenient predicate for performing our benchmarks. For example:

```
benchmark(Goal) :-
    N = 10000000,                   % some sufficiently large number of repetitions
    write('Number of repetitions: '), write(N), nl,
    get_cpu_time(Seconds1),         % replace by your Prolog-specific predicate
    benchmark(N, Goal),
    get_cpu_time(Seconds2),
    Average is (Seconds2 - Seconds1)/N,
    write('Average time per call: '), write(Average), write(' seconds'), nl,
    Speed is 1.0/Average,
    write('Number of calls per second: '), write(Speed), nl.
```

We can get a baseline for our timings by doing:

```
| ?- benchmark(true).
```

For comparing message sending performance across several Prolog compilers, we would call the `benchmark/1` predicate with a suitable argument. For example:

```
| ?- benchmark(list::length([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], _)).
```

For comparing message sending performance with predicate calls in plain Prolog and with calls to predicates encapsulated in modules, we should use exactly the same predicate definition in the three cases.

It should be stressed that message sending is only one of the factors affecting the performance of a Logtalk application. The strengths and limitations of the chosen Prolog compiler play a crucial role on all aspects of the development, reliability,

usability, and performance of a Logtalk application. It is advisable to take advantage of the Logtalk wide compatibility with most Prolog compilers to test for the best match for developing your Logtalk applications.

Last updated on: October 26, 2005