Concurrent Programming Constructs and First-Class Logic Engines

Paul Tarau
University of North Texas
tarau@cs.unt.edu

Multi-threading has been adopted in today's Prolog implementations as it became widely available in implementation languages like C or Java.

An advantage of multi-threading over more declarative concurrency models like various AND-parallel and OR-parallel execution schemes, is that it maps to the underlying hardware directly: on typical multi-core machines threads and processes are mapped to distinct CPUs. Another advantage is that a procedural multi-threading API can tightly control thread creation and thread reuse.

On the other hand, the explicit use of a procedural multi-threading API breaks the declarative simplicity of the execution model of logic based languages. At the same time it opens a Pandora's box of timing and execution order dependencies, resulting in performance overheads for various runtime structures that need to be synchronized.

In this note we emphasize the *decoupling of the multi-threading API and the logic engine operations and encapsulation of multi-threading in a set of high-level primitives with a declarative flavor.*

Our guiding architectural principle can be stated concisely as follows: *separate concurrency for performance from concurrency for expressiveness.* Arguably, it is a good fit with the general idea behind declarative programming languages – delegate as much low level detail to underlying implementation as possible rather than burdening the programmer with complex control constructs.

We have implemented it in the context of a Java-based system called *Lean-Prolog*, centered around *first-class logic engines* providing an API that supports reentrant instances of the language processor and can express control, metaprogramming and interoperation with stateful objects and external services.

First-class logic engines as answer generators

Our language constructs have evolved progressively into a practical Prolog implementation framework starting with [1] and continued with [2], [3] and [4].

A *first-class logic engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it. Logic engines are seen, in an object oriented-style, as implementing the *interface* `Interactor`. This supports a *uniform* interaction mechanism with a variety of objects ranging from logic engines to file/socket streams and iterators over external data structures.

The `ask_interactor/2` operation is used to retrieve successive answers generated by an `Interactor`, on demand. It is also responsible for actually triggering computations in the engine. The query

```
ask_interactor(Interactor, AnswerInstance)
```

tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned. Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `ask_interactor/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with `Interactor`'s iteration over answers. Backtracking over `Interactor`'s creation point, as such, makes it unreachable and therefore subject to garbage collection.

With a semantics similar to the `yield return` construct of `C#` and the `yield operation` of Ruby

```
return(Term)
```

saves the state of the engine and transfer *control* and a *result* `Term` to its client. The client will receive a copy of `Term` simply by using its `ask_interactor/2` operation.

By using a sequence of `return` and `ask_interactor` operations, an engine can provide a stream of *intermediate/final results* to its client. The mechanism is powerful enough to implement a complete exception handling mechanism simply by defining

```
throw(E) :- return(exception(E)).
```

When combined with a `catch(Goal, Exception, OnException)`, on the client side, the client can decide, upon reading the exception with `ask_interactor/2`, if it wants to handle it or to throw it to the next level.

Coroutining logic engines

Coroutining has been in use in Prolog systems mostly to implement constraint programming extensions. The typical mechanism involves *attributed variables* holding suspended goals that may be triggered by changes in the instantiation state of the variables. Our focus, however, is on a different form of coroutining, induced by the ability to switch back and forth between engines. The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable another engine, that uses an engine's services to *inject* new goals (executable data) to an arbitrary inner context of an engine [4]. This is achieved using two built-ins

```
to_engine(Engine, Data)
```

that is called by another engine communicating Data to this engine, and

```
from_engine(Data)
```

that is called by the engine itself to receive a client's Data.

Hubs and threads

As a key difference with typical multi-threaded Prolog implementations like Ciao-Prolog [5] and SWI-Prolog [6], our Interactor API is designed up front with a clear separation between *engines* and *threads* seen as *orthogonal* language constructs.

To ensure that communication between logic engines running concurrently is safe and synchronized, we hide the engine handle and provide a producer/consumer data exchanger object, called a `Hub`, when multi-threading.

A `Hub` can be seen as an interactor used to synchronize threads. On the Prolog side it is introduced with a constructor `hub/1` and works with the standard interactor API:

```
ask_interactor(Hub, Term)
tell_interactor(Hub, Term)
stop_interactor(Hub)
```

On the Java side, each instance of the `Hub` class provides a synchronizer between M producers and N consumers. A `Hub` supports data exchanges through a private object `port` and it implements the `Interactor` interface. Consumers issue `ask_interactor/2` operations that correspond to `tell_interactor/2` operations issued by producers.

A group of related threads are created around a `Hub` that provides both basic synchronization and data exchange services using the built-in:

```
new_logic_thread(Hub, X, G)
```

They share the code areas but duplicate symbol tables to allow independent symbol creation and symbol garbage collection without the need to synchronize or suspend thread execution.

High-level concurrency with higher-order constructs

Encapsulating concurrent execution patterns in high-level abstractions, when performance gains are the main reason for using multiple threads, avoids forcing a programmer to suddenly deal with complex procedural issues when working with (mostly) declarative constructs in a language like Prolog. It is also our experience that in an exclusively dynamically-typed language like Prolog this reduces software risks significantly.

One of the deficiencies of sequential or multi-threaded `findall`-like operations is that they might build large lists of answers unnecessarily. With inspiration drawn from combinators in functional languages, one can implement a more flexible multi-threaded `fold` operation instead.

The predicate `multi_fold(F, XGs, Xs)` runs a list of goals `XGs` of the form `Xs :- G` and combines, with `F`, their answers, to accumulate them into a single final result, without building intermediate lists.

```
multi_fold(F, XGs, Final) :- hub(Hub),
  length(XGs,ThreadCount),
  launch_logic_threads(XGs, Hub),
  ask_interactor(Hub, Answer),
  (Answer = the(Init) -> fold_thread_results(ThreadCount, Hub, F, Init, Final)
  ; true
  ),
  stop_interactor(Hub),Answer=the(_).
```

The predicate `multi_fold` relies on the predicate `launch_logic_threads` to run threads initiated by the goal list `XGs`. When launching the threads, we ensure that they share the same `Hub` for communication and synchronization.

```
launch_logic_threads([], _Hub).
launch_logic_threads([(X :- G)|Gs], Hub) :-
  new_logic_thread(Hub, X, G),
  launch_logic_threads(Gs, Hub).
```

Once all threads are launched, we use the predicate `fold_thread_results` to collect results computed by various threads from `Hub`, and to combine them into a single result, while keeping track of the number of threads that have finished their work.

```
fold_thread_results(0, _Hub, _F, Best, Best).
fold_thread_results(ThreadCount, Hub, F, SoFar, Best) :- ThreadCount > 0,
  ask_interactor(Hub, Answer),
  count_thread_answer(Answer, ThreadCount, ThreadsLeft, F, SoFar, Better),
  fold_thread_results(ThreadsLeft, Hub, F, Better, Best).

count_thread_answer(no, ThreadCount, ThreadsLeft, _F, SoFar, SoFar) :-
  ThreadsLeft is ThreadCount-1.
count_thread_answer(the(X), ThreadCount, ThreadCount, F, SoFar, Better) :-
  call(F, X, SoFar, Better).
```

A typical application is the predicate `multi_best(F, XGs, M)`, which runs a list of goals `XGs` of the form `N :- G` where `N` is instantiated to a numeric value. By using `max/3` to combine the current best answers with a candidate one it extracts at the the maximum `M` of all answers computed (in an arbitrary order) by all threads.

```
multi_best(XGs,R) :- multi_fold(max,XGs,R).
```

Note that, as in the case of its `fold` cousins in functional languages, `multi_fold` can be used to emulate various other higher order predicates. For instance, a concurrent `findall`-like predicate is emulated as `multi_all(XGs,Xs)` which runs a list of goals `XGs` of the form `Xs :- G` and combines all answers to a list using `list_cons`.

```
multi_all(XGs, Rs) :- multi_fold(list_cons,[([] :- true)|XGs],Rs).
```

```
list_cons(X, Xs, [X|Xs]).
```

A different pattern arises from combinatorial search algorithms where one wants to *stop multiple threads* as soon as a first solution is found.

The predicate `multi_first(K, XGs, Xs)` runs each goal of the form `Xs :- G` on the list `XGs`, until the first `K` answers `Xs` are found (or fewer, if less then `K` answers exist). It uses a very simple mechanism built into Lean Prolog's multi-threading API: when a `Hub` interactor is stopped, all threads associated to it are notified to terminate.

```
multi_first(K, XGs, Xs) :- hub(Hub),
  length(XGs, ThreadCount),
  launch_logic_threads(XGs, Hub),
  % code similar to fold_thread_results
  collect_first_results(K, ThreadCount, Hub, Xs),
  stop_interactor(Hub).
```

In particular, searching for at most one solution is possible:

```
multi_first(XGs,X) :- multi_first(1,XGs,[X]).
```

Note also that `multi_first` provides an alternative to using `CUT` in Prolog as a means to limit search, while supporting a scalable mechanism for concurrent execution.

Conclusion

By decoupling logic engines and threads, programming language constructs can be kept simple when their purpose is clear – *multi-threading for performance* is separated from *concurrency for expressiveness.* Our language constructs are particularly well-suited to take advantage of today's multi-core architectures where keeping busy the actual parallel execution units results in predictable performance gains, while reducing the software risks coming from more complex concurrent execution mechanisms.

The current version of LeanProlog containing the implementation of the constructs discussed in this note, and a few related papers are available at *http://logic.cse.unt.edu/tarau/research/LeanProlog* .

Acknowledgment

We thank NSF (research grant 1018172) for support.

References

[1] Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In Lloyd, J., ed.: Computational Logic–CL 2000: First International Conference, London, UK (July 2000) LNCS 1861, Springer-Verlag.

[2] Tarau, P.: Logic Engines as Interactors. In Garcia de la Banda, M., Pontelli, E., eds.: Logic Programming, 24-th International Conference, ICLP, Udine, Italy, Springer, LNCS (December 2008) 703–707

[3] Tarau, P., Majumdar, A.: Interoperating Logic Engines. In: Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, Georgia, Springer, LNCS 5418 (January 2009) 137–151

[4] Tarau, P.: Concurrent programming constructs in multi-engine prolog. In: Proceedings of DAMP'11: ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, New York, NY, USA, ACM (2011)

[5] Carro, M., Hermenegildo, M.V.: Concurrency in Prolog Using Threads and a Shared Database. In: ICLP. (1999) 320–334

[6] Wielemaker, J.: Native Preemptive Threads in SWI-Prolog. In Palamidessi, C., ed.: ICLP. Volume 2916 of Lecture Notes in Computer Science., Springer (2003) 331–345