Coordination and Concurrency in Multi-Engine Prolog

Paul Tarau

Department of Computer Science and Engineering
University of North Texas

Coordination'11: Reykjavik, Wednesday, June 8, 2011, 11:00am

Motivation

- Logic Programming languages make essential use of unification and backtracking $\Rightarrow$ concurrent programming models are by far more complex than in Functional Programming
- $\Rightarrow$ encapsulate unification, recursion and backtracking in independent computational units - logic engines
- $\Rightarrow$ interactors: an abstraction of answer generation and refinement in *logic engines*, supporting the agent-oriented view that *programming is a dialog between simple, self-contained, autonomous building blocks*
- resist temptation to map logic engines and threads directly $\Rightarrow$ encapsulate concurrency in higher order primitives, similar to existing sequential constructs
- $\Rightarrow$ *ability to separate concurrency for performance and concurrency for expressiveness*

Outline

- an overview of logic engines - our coroutining primitives
- "real concurrency": thread coordination with Hubs
- concurrency for performance: higher-order predicates encapsulating multi-threading
- concurrency for expressiveness: coroutining implementation of fundamental coordination patterns
  - Linda blackboards
  - publish/subscribe with associative search
- conclusion

First Class Logic Engines

- a *logic engine* is a Prolog language processor reflected through an API that allows its computations to be controlled interactively from another *engine*
- very much the same thing as a programmer controlling Prolog's interactive toplevel loop:
  - launch a new goal
  - ask for a new answer
  - interpret it
  - react to it
- logic engines can create other logic engines as well as external objects
- logic engines can be controlled cooperatively or preemptively

The “Lean Prolog” Engine API: new_engine/3

new_engine(AnswerPattern, Goal, Interactor):

- creates a new instance of the Prolog interpreter, uniquely identified by `Interactor`
- shares code with the currently running program
- initialized with `Goal` as a starting point
- `AnswerPattern` is a term returned by the engine will be instances

The Engine API: get/2, stop/1

get(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
- is used to retrieve successive answers generated by an Interactor, on demand
- it is responsible for actually triggering computations in the engine

stop(Interactor):

- stops the Interactor
- `no` is returned for new queries

The `return` operation: a key co-routining primitive

return(Term)

- will save 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 `get/2` operation
- an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available

Application: exceptions

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

Exchanging Data with an Interactor

to_engine(Engine,Term):

- used to send a client's data to an Engine

from_engine(Term):

- used by the engine to receive a client's Data

Typical use of the Interactor API

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*:
   - the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
   - the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   - the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

Hubs

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

```
hub(Hub)

ask_interactor(Hub, Term)

tell_interactor(Hub, Term)

stop_interactor(Hub)
```

Multiple threads are created around a hub with:

```
launch_logic_threads(AnswersAndGoalsPatterns, Hub)
```

Java side of a Hub

```
private Object port;

synchronized public Object ask_interactor() {
    while(null==port) {
      try {
        wait();
      } catch(InterruptedException e) {
        if(stopped)
          break;
      }
    }
    Object result=port;
    port=null;
    notifyAll();
    return result;
}
```

A multi-purpose higher order predicate: multi_fold

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 – a kind of “Map-Reduce” applied to answer streams.

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

A variation, multi_findall: "structured" OR-parallelism

`multi_findall(XGs, Xss)` marks answers and sorts by goal

- for each `(X:-G)` on the list `XGs` it starts a new thread
- then aggregates solutions as if `findall(X, G, Xs)` were called
- It collects all the answers `Xs` to a list of lists `Xss` in the order defined by the list of goals `XGs`.

```
multi_findall(XGs, Xss):-
  mark_answer_patterns(XGs, MarkedXGs, 0),
  multi_all(MarkedXGs, MXs),
  collect_marked_answers(MXs, Xss).
```

```
?-multi_findall([(I:-for(I, 1, 10)),
                 (J:-member(J, [a, b, c]))], Rss).
Rss = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [a, b, c]]
```

Stopping after the first K answers: multi_first

- `multi_first(K, XGs, Xs)` runs list of goals `XGs` of the form `Xs:-G` 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
- this happens when we detect that the first K answers have been computed or that there are no more answers

Cooperative Coordination - Concurrency without Threads

- `new_coordinator(Db)` uses a database parameter `Db` to store the state of the Linda blackboard
- The state of the blackboard is described by the dynamic predicates
  - `available/1` keeps track of terms posted by `out` operations
  - `waiting/2` collects pending `in` operations waiting for matching terms
  - `running/1` helps passing control from one engine to the next

```
new_coordinator(Db):-
  db_ensure_bound(Db),
  db_dynamic(Db, available/1),
  db_dynamic(Db, waiting/2),
  db_dynamic(Db, running/1).
```

Agents as cooperative Linda tasks

```
new_task(Db, G):-
  new_engine(nothing, (G, fail), E),
  db_assert(Db, running(E)).
```

Three cooperative Linda operations are available to an agent. They are all expressed by returning a specific pattern to the `Coordinator`.

```
coop_in(T):-return(in(T)), from_engine(X), T=X.

coop_out(T):-return(out(T)).

coop_all(T, Ts):-return(all(T, Ts)), from_engine(Ts).
```

The Coordinator's Handler

```
handle_in(Db, T, E):-
  db_retract1(Db, available(T)),
  !,
  to_engine(E, T),
  db_assert(Db, running(E)).
handle_in(Db, T, E):-
  db_assert(Db, waiting(T, E)).

handle_out(Db, T):-
  db_retract(Db, waiting(T, InE)),
  !,
  to_engine(InE, T),
  db_assert(Db, running(InE)).
handle_out(Db,T):-
  db_assert(Db, available(T)).
```

The Coordinator Dispatch Loop

```
coordinate(Db):-
  repeat,
    ( db_retract1(Db, running(E))->
        ask_interactor(E, the(A)),
        dispatch(A, Db, E),
        fail
     ; !
    ).
```

Its `dispatch/3` predicate calls the handlers as appropriate.

```
dispatch(in(X), Db, E):-handle_in(Db, X, E).
dispatch(out(X), Db, E):-handle_out(Db, X),
  db_assertz(Db, running(E)).
dispatch(all(T, Ts), Db, E):-handle_all(Db, T, Ts, E).
dispatch(exception(Ex), _, _):-throw(Ex).
```

Coordination Example

```
test_coordinator:-
  new_coordinator(C),
  new_task(C,
    foreach(member(I, [0, 2]),
      ( coop_in(a(I, X)),println(coop_in=X) )
    )
  ),
  new_task(C,
    foreach(member(I, [3, 2, 0, 1]),
      ( println(coop_out=f(I)),coop_out(a(I, f(I))) )
    )
  ),
  ...
  coordinate(C),
  stop_coordinator(C).
```

Running the Coordination Example

```
?- test_coordinator.
coop_out = f(3)
coop_out = f(2)
coop_out = f(0)
coop_in = f(0)
coop_out = f(1)
coop_in = f(2)
coop_in = f(1)
coop_in = f(3)
```

- “concurrency for expressiveness" in terms of the logic-engines-as-interactors API provides flexible building blocks for the encapsulation of high-level concurrency patterns

Coordinating publishers and subscribers

- a cooperative publish/subscribe mechanism that uses multiple dynamic databases and provides, as an interesting feature, associative search through the set of subscribed events
- the predicate `publish(Channel,Content)` initializes a `Channel`, implemented as a dynamic database together with a time stamping mechanism
- `Content` is a fact to be added to the database, for which the user can (and should) provide indexing declarations to support scalable large volume associative search.

Consuming and searching associatively

- `consume_new(Subscriber,Channel,Content)` reads the next message on Channel
- it ensures, by checking and updating `Channel` and subscriber-specific time stamps, that on each call it gets one *new* event, if available
- `peek_at_published(ContentPattern, Matches)` supports associative search for published content, independently of the fact that it has already been read
- it supports indexing on `ContentPattern`
- it provides to an agent the set of subscribed events matching `ContentPattern`

Initiating Publishing

- `init_publishing(ContentIndexings)` sets up indexing using list of `ContentIndexings` of the form `pred(I1,I2,...In)` where `I1,I2...In` can be `1` (indicating that an argument should be indexed) or `0`.
- indexing is important: it can provide constant time associative retrieval

The time-stamping mechanism

- a few predicates manage the time stamping mechanism, needed to ensure that subscribers get all the events in the order they have been published:

```
set_time_of(Role,Key,T)

get_time_of(Role,Key,R)

increment_time_of(Role,Key,T1)

remove_time_of(Role,K)
```

- a few other predicates provide clean-up operations, like removing from all the channels the published content as well as the tracking of subscribers

A publish/consume example

?− pubtest.

```
publish(sports, wins(rangers)) publish(politics, loses(meg))
publish(sports, loses(bills)) publish(sports, wins(cowboys))
publish(politics, wins(rand))

consume_new(joe, sports, wins(rangers))
consume_new(mary, sports, wins(rangers))
consume_new(joe, sports, loses(bills))

consume_new(joe, politics, loses(meg))
consume_new(joe, politics, wins(rand))
consume_new(mary, sports, loses(bills))
```

After running the example

the final state of various databases is:

```
time_of($publishing, sports, 2).
time_of($publishing, politics, 1).
time_of(sports, joe, 2).
time_of(politics, joe, 2).
time_of(sports, mary, 2).

wins(rangers) :- published_at(0).
wins(cowboys) :- published_at(2).

wins(rand) :- published_at(1).

loses(meg) :- published_at(0).
loses(bills) :- published_at(1).
```

Conclusion

- by decoupling logic engines and threads, programming language constructs for coordination can be kept simple when their purpose is clear
- *multi-threading for performance* is separated from *concurrency for expressiveness*
- we keep language interpreters independent of multi-threading
- our language constructs are particularly well-suited to take advantage of today's multi-core architectures where keeping busy a relatively small number of parallel execution units is all it takes to get predictable performance gains, while reducing the software risks coming from more complex concurrent execution mechanisms designed with massively parallel execution in mind

Questions?

Lean Prolog and a few related papers are at:

- http://logic.cse.unt.edu/research/LeanProlog