Scalable Web Reasoning using Logic Programming Techniques

Gergely Lukácsy[1] Péter Szeredi[2]

[1]Digital Enterprise Research Institution (DERI)
Galway, Ireland

[2]Budapest University of Technology and Economics (BUTE)
Budapest, Hungary

Introduction

Goals

- an expressive DL reasoning framework that solves instance retrieval problems when large amounts of underlying data are expected
- data in external database/triple store
- distributed and scalable execution

In this presentation

- we provide extensions of the DL reasoning system DLog that transforms the DL reasoning task into the execution of a Logic Program
- main result: initial design of DLog Abstract Machine (DAM) - a virtual machine for the execution of DLog programs
- secondary result: an outline of a new parallel architecture for the DLog system that is built around the DAM idea

Part I: the DLog framework

The DLog framework

The DLog system in a nutshell

DLog is a resolution based Description Logic $\mathcal{SHIQ}$ ABox reasoning system implemented in Prolog/C++

- DLog creates a Prolog program from a DL knowledge base
- the queries in DLog are focused, reasoning consists of two phases
- DLog is remarkably faster than its competitors, for a lot of benchmarks
- DLog is available to download (`http://www.dlog-reasoner.org`)

The generic transformation scheme

- Input: arbitrary set of DL clauses (TBox $\rightarrow$ FOL clauses $\subseteq$ DL-clauses)
- Output: a Prolog program equivalent with the input wrt. instance retrieval
- Idea: (1) two-fold specialisation of Prolog Technology Theorem Proving (PTTP) - an approach to build a FOL theorem prover on top of Prolog; (2) applying prolog-level optimisations on the output

Code generated from: $\exists$hasSpouse.Man $\sqsubseteq$ Woman

```
woman(X, L0)  :- member(A, L0),
                 A == woman(X), !, fail.        %loop elim.
woman(X, L0)  :- member(not_woman(X), L0), !.  %ancestor res.
woman(X, L0)  :- L1 = [woman(X)|L0],            %new anc.list
                 hasSpouse(X, Y), man(Y, L1).  %original clause
woman(X, _)   :- abox:woman(X).                 %ABox facts

not_man(Y, L0) :- L1 = [not_man(Y)|L0],         %contrapositive
                  hasSpouse(X, Y), not_woman(X, L1).
```

Optimizations - decomposition

Basic idea

- split a body into independent components
- make sure that the truth value of each component is only calculated once

Example

„someone is happy if she has a child having both a clever and a pretty child”

```
Happy(A) :-
     hasChild(A, B),
     (   hasChild(B, C),
         Clever(C) -> true            ⟶ first component
     ),
     (   hasChild(B, D),
         Pretty(D) -> true            ⟶ second component
     ).
```

Optimizations - superset

Basic idea

- determine for each predicate P a set of instances S for which $I(P) \subseteq S$ holds ($I(P)$ denotes the set of solutions of P)
- reduce the initial instance retrieval problem to a finite number of deterministic instance checks

The generic superset schema

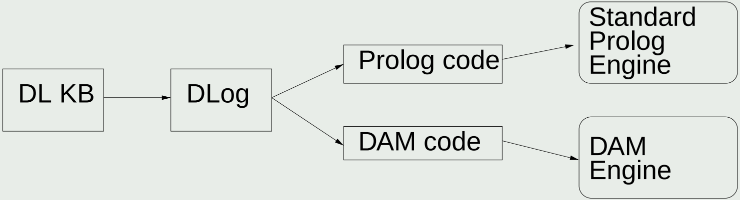
```
choice_Concept(A, AL) :-
        (   nonvar(A) -> deterministic_Concept(A, AL)
        ;   member_of_superset_Concept(A),
            deterministic_Concept(A, AL)
        ).

% A is a specific instance
deterministic_Concept(A, AL) :- ..., !.
...
```

Part II: the DLog Abstract Machine

The DLog Abstract Machine (DAM)

Role of the DAM



Properties of DLog programs

1. predicates can only be unary or binary → *single argument register*
2. there are no compound data structures → *unification is trivial*
3. concept predicate invocations are ground and deterministic → *no need for deep backtracking*
4. 2+3 → *no need for the heap and the trail stack*
5. arguments are always instance names → *no need for cell tagging*

Architecture of DAM

Data structures and registers

- Control stack: fixed sized frames for local environment/return address information; predicates receive arguments implicitly
- Choice point stack: deep backtracking for roles; communication with DB
- Bactrackable hash table (stack)
- Global registers: V (return value), PC (program counter), T (current control frame)

Control structures

- conjunction, disjunction and loops
- we assume that each predicate contains exactly one of these (can be achieved by introducing auxiliary predicates)

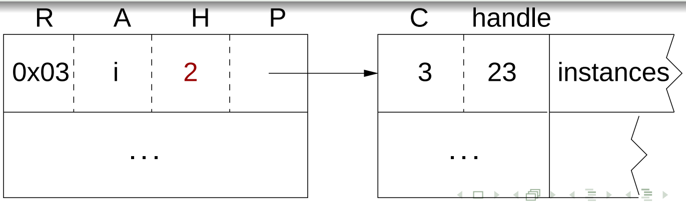
Data structure internals

Control stack

- the return address of the predicate (virtual register R);
- the actual instance (URI) being checked (virtual register A);
- the ancestor list, represented as an index (virtual register H);
- a pointer to the corresponding choice stack frame (virtual register P).

Choice point stack

- a counter used in implementing number restrictions (virtual register C);
- a handle used for interfacing with the triple store;
- a buffer for instances returned by the triple store.



Instruction set of DAM

Instruction	Arguments	Description
`put_ancestor`	N	extend the ancestor list in the local frame by the term with name N and argument A
`check_ancestor`	N	succeeds if the ancestor list contains a term with name N and argument A
`fail_on_loop`	N	fails if a loop occurred, i.e. the term with name N and argument A is present on the ancestor list
`call`	P	invokes procedure P in a new control frame
`execute`	P	invokes procedure P in the existing control frame
`exit_with`	S	returns from a procedure with status S, continues execution according to register R
`exit_on_failure`	–	returns from procedure if V = FAILURE
`exit_on_success`	–	returns from procedure if V = SUCCESS
`jump`	L	jumps to label L
`has_n_successors`	R, n	checks if instance A has at least n R successors; creates a choice point; loads the first choice to A
`count_and_exit`	–	decreases counter C if the previous instruction was successful; returns with success if C is 0
`next_choice`	–	loads the next solution from the choice stack to A
`abox_query`	Q	returns success if A is a solution of query Q

Translating DLog programs to DAM code

Conjunctions/Disjunctions

$g_1(X), \ldots, g_k(X)$

```
call g1
exit_on_failure
...
call g(k-1)
exit_on_failure
execute gk
```

$g_1(X) ; \ldots ; g_k(X)$

```
call g1
exit_on_success
...
call g(k-1)
exit_on_success
execute gk
```

Number restriction ($\geq nRC$)

```
  has_n_successors R n   ⟶ fails if A has not enough successors
label(1):
  call C                 ⟶ returns with success or failure
  count_and_exit         ⟶ if success : C--, returns success if C = 0
  next_choice            ⟶ set A to next successor, return fail if no more
  jump 1
```

Example translation

$\exists$hasSpouse.Man $\sqsubseteq$ Woman

```
predicate(woman):                  ⟶ A contains the instance to check
   fail_on_loop woman
   check_ancestor not_woman
   call aux_1                      ⟶ Original clause
   exit_on_success
   execute aux_2                   ⟶ Direct ABox call

predicate(aux_1):
   put_ancestor woman              ⟶ uses A, sets H
   has_n_successors hasSpouse 1
 label(1):
   call man,                       ⟶ Invokes another predicate
   count_and_exit
   next_choice
   jump 1
```

Operational semantics of the instructions - 1

```
put_ancestor n:
    H = add_to_hash(A, n, H);
```
⟶ inserts term n(A) into the hash table

```
check_ancestor n:
    if (hash_search(A, n, H)) exit_with SUCCESS;
```
⟶ checks if term n(A) is in the hash table

```
fail_on_loop n:
    if (hash_search(A, n, H)) exit_with FAILURE;
```
⟶ checks if term n(A) is in the hash table

```
call p:
    T++; A = previous->A; H = previous->H; R = PC + 1;
    PC = &p;
```
⟶ invokes procedure in new frame

```
execute p:
    PC = &p;
```
⟶ invokes procedure in the current frame

Operational semantics of the instructions - 2

```
has_n_successors r n:
    if (!cardinality_check(A, r, n)) exit_with FAILURE;
    A = create_choice(A, r);
```
⟶ loads successors of A to the choice stack

```
count_and_exit:
    if (V == SUCCESS) P->C--;
    if (P->C == 0) exit_with SUCCESS
```
⟶ counts and exists if counter reaches zero

```
next_choice:
   if (!has_choice()) exit_with FAILURE;
   A = next_choice();
```
⟶ sets the next solution instance to A

```
abox_query q:
   V = abox_query(A, q);
```
⟶ executes a (complex) database query

Part III: parallel architecture for DLog

Parallelisation possibilities

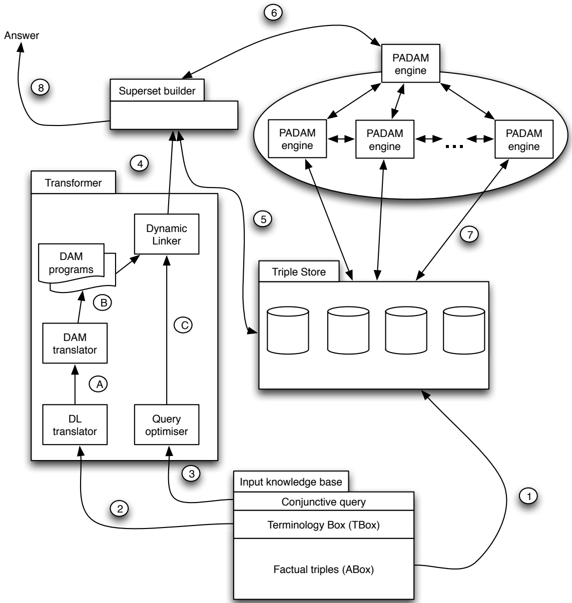
Fine-grained parallelism

- Idea: simplify LP parallelisation techniques to DLog programs → PADAM
- AND parallelism works well with decomposition
- OR paralellism involves speculative work

Coarse-grained parallelism

- Idea: introduce parallelism at the DLog architecture-level
- the superset expression is evaluated in parallel
- the instances in the superset are checked in parallel

The architecture of the Parallel DLog system.



Conclusion

Summary

- we introduced the Prolog based DLog reasoning system and provided two extensions to improve its scalability
- we presented the initial design of the DLog Abstract Machine, including its architecture, instruction set and operational semantics
- we outline a new parallel architecture for the DLog system that introduces parallelism at many levels of the execution

Future work

- implementation and performance evaluation
- refinement of the PADAM execution model
- designing the details of the communication between DLog and the underlying database/triple store

Recent DLog related publications

Gergely Lukácsy, Péter Szeredi.
Efficient description logic reasoning in Prolog: the DLog system.
Theory and Practice of Logic Programming (TPLP). 09(03):343-414, May, 2009. Cambridge University Press, UK.

Gergely Lukácsy, Péter Szeredi, and Balázs Kádár.
Prolog based description logic reasoning.
In Proceedings of the 24th International Conference on Logic Programming (ICLP 2008), pp. 455-469, Udine, Italy, December 2008.

Zsolt Zombori
Efficient Two-Phase Data Reasoning for Description Logics.
In Proceedings of the IFIP 20th World Computer Congress (IFIP AI 2008), pp. 393-402, Milano, Italy, September 2008.

Zsolt Zombori and Gergely Lukácsy.
A resolution based description logic calculus.
In Proceedings of the 22nd International Workshop on Description Logics (DL 2009), volume 477 of *CEUR*, Oxford, UK, July 2009.