Towards Ontology-Based Program Analysis

Yue Zhao*, Chunhua “Leo” Liao, Xipeng Shen*

ECOOP’16, July 18

LLNL-PRES-697897

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

*North Carolina State University

Lawrence Livermore National Laboratory

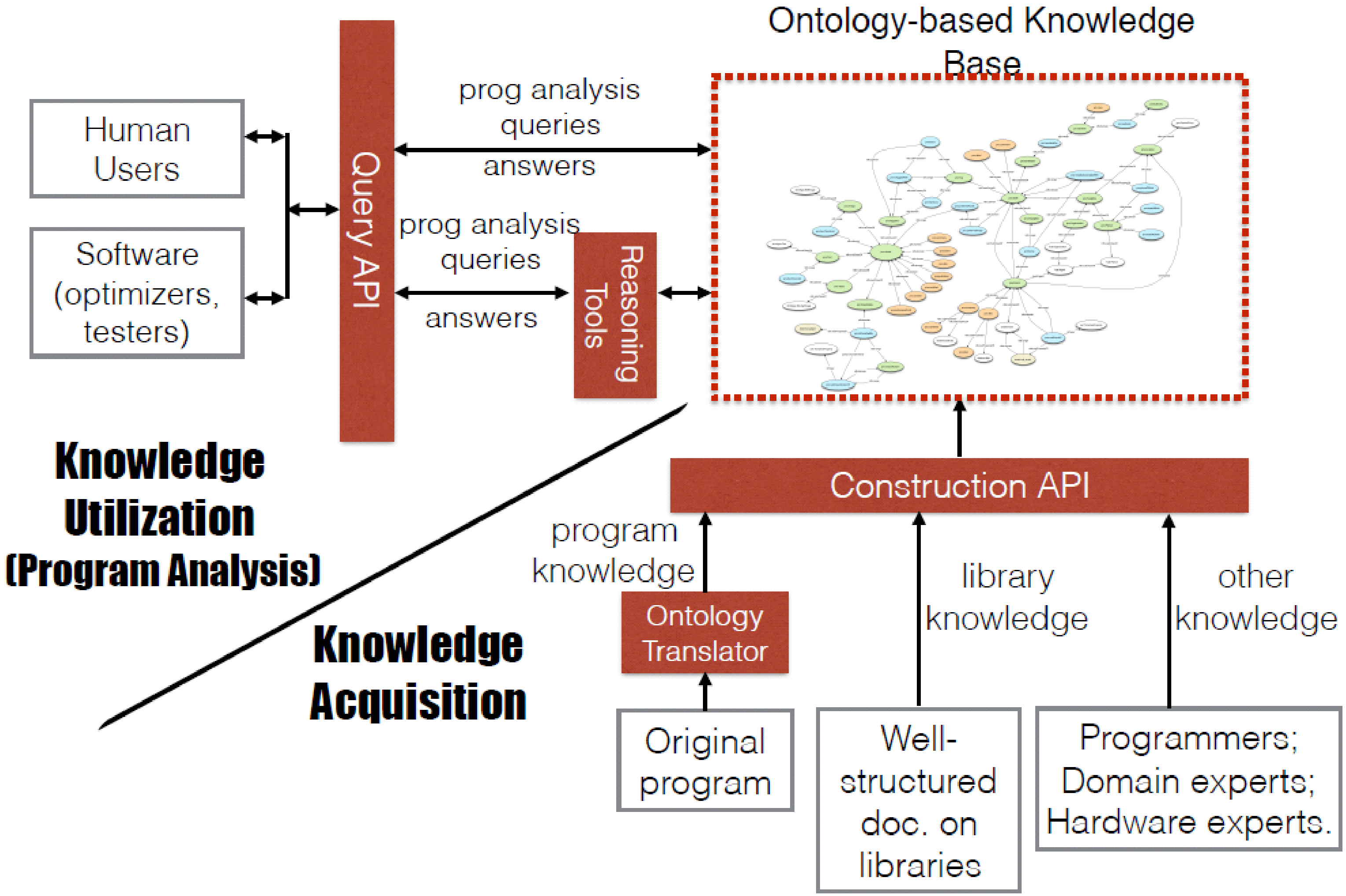
Agenda

- Motivation
- Basics of Ontology
- Design of PATO: Program Analysis Through Ontology
- Evaluation
- Conclusion

Motivation

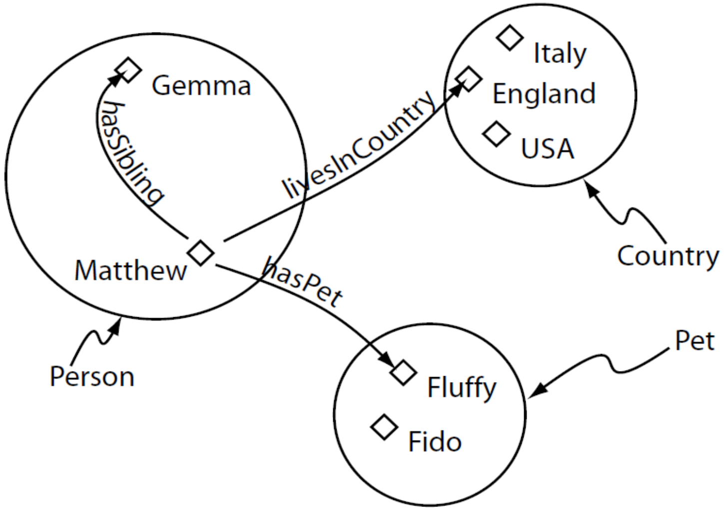
- Program analysis:
  - extract properties of programs: control flow, data flow, etc.
  - fundamental for optimization, correctness checking
- Two implementation approaches:
  - Imperative approach - traditional way
    - Tied to customized internals of a compiler
  - Declarative analysis - proposed to overcome the productivity issues.
    - Using logic programming (e.g., Datalog) to describe rules
- limitations
  - Ad-hoc representation of input and results: hard to reuse/merge intermediate or final results across implementations
  - Not extensible: focus on program behaviors only, not linked with knowledge from other domains (hardware, algorithm, etc)

Our approach: ontology based program analysis



Background – what is ontology

- Definitions:
  - a formal specification for explicitly representing knowledge of the entities in some domains
- Theory foundation:
  - description logics (DLs)
- Several dialects with different expressiveness and efficiencies (for reasoning)
  - Most popular: Web ontology Language (OWL)
  - Triples: (subject, property, object)
- Maturing ecosystems: GUI, parsers, reasoners, etc.



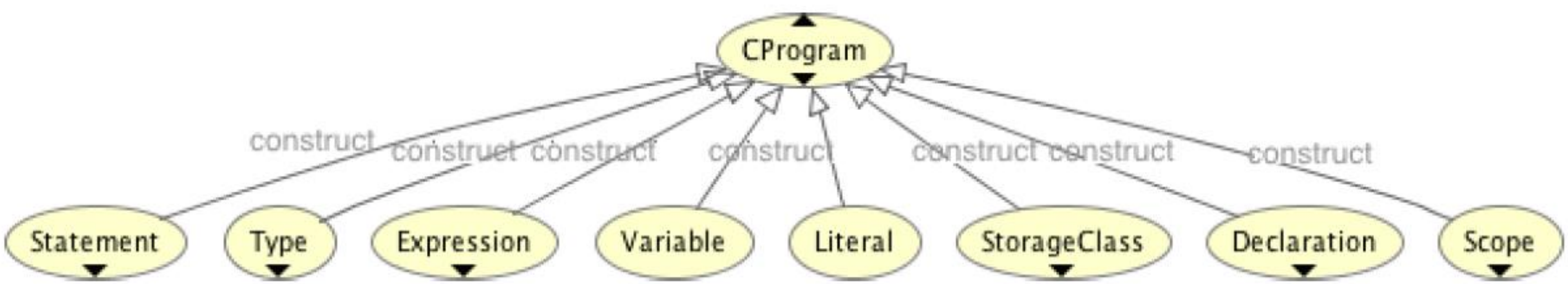
Ontological knowledge in a domain

- Concepts, Relations, Individuals

(Matthew, instanceOf, Person)
(Matthew, hasPet, Fluffy)
..

Challenge 1 – ontology design

- The vocabulary in an ontology must be:
  - Generic, intuitive, extensible
  - Cover both input programs and output analysis results
- Solution:
  - Language standard-oriented: e.g. C99 language standard
  - Iteratively add more concepts and relations of program analysis
- Our draft ontology for C programs
  - 178 concepts and 68 properties



Challenge 2 - knowledge generation

- Generation: manual design + automatic generation
  - Stanford Protégé GUI
  - Compiler-based ontology generator
  - Logic reasoning based generation

- Entities are named with IRI format to avoid conflicts
  - Qualified names: http://example.com/owl/CProgram#Type
  - Source location (begin to end): http://example.com/file1.c#3:1, 5:1
  - Scoped naming to reduce impact of code changing : http://example.com/file1.c#foo()#1:6,1:10

Example: generating ontology entities from a C code

- The ontology generator
  - Based on the ROSE source-to-source compiler framework developed at LLNL
  - Walk the AST of C code and translate program information in triple format

```
0 // s.c
1 int a = 0;
2 int foo() {
3  for (int i = 0; i < 10; i++) {
4   a = a + i;
5  }
6  return 0;
7 }
```

```
('3:1,5:1', rdf:type, 'ForStatement')
('3:6,3:14', rdf:type, 'VariableDecl')
('3:6,3:10', rdf:type, Variable')
('3:1,5:1', 'hasForInit', '3:6,3:14')
('3:1,5:1', 'hasForTest', '3:17,3:22')
('3:1,5:1', 'hasForIncr', '3:25,3:27')
('3:1,5:1', 'hasBody', '3:30,5:1')
```

*prefix of IRI is omitted to be brief
http://example.com/file1.c#3:1, 5:1

Challenge 3 – knowledge utilization

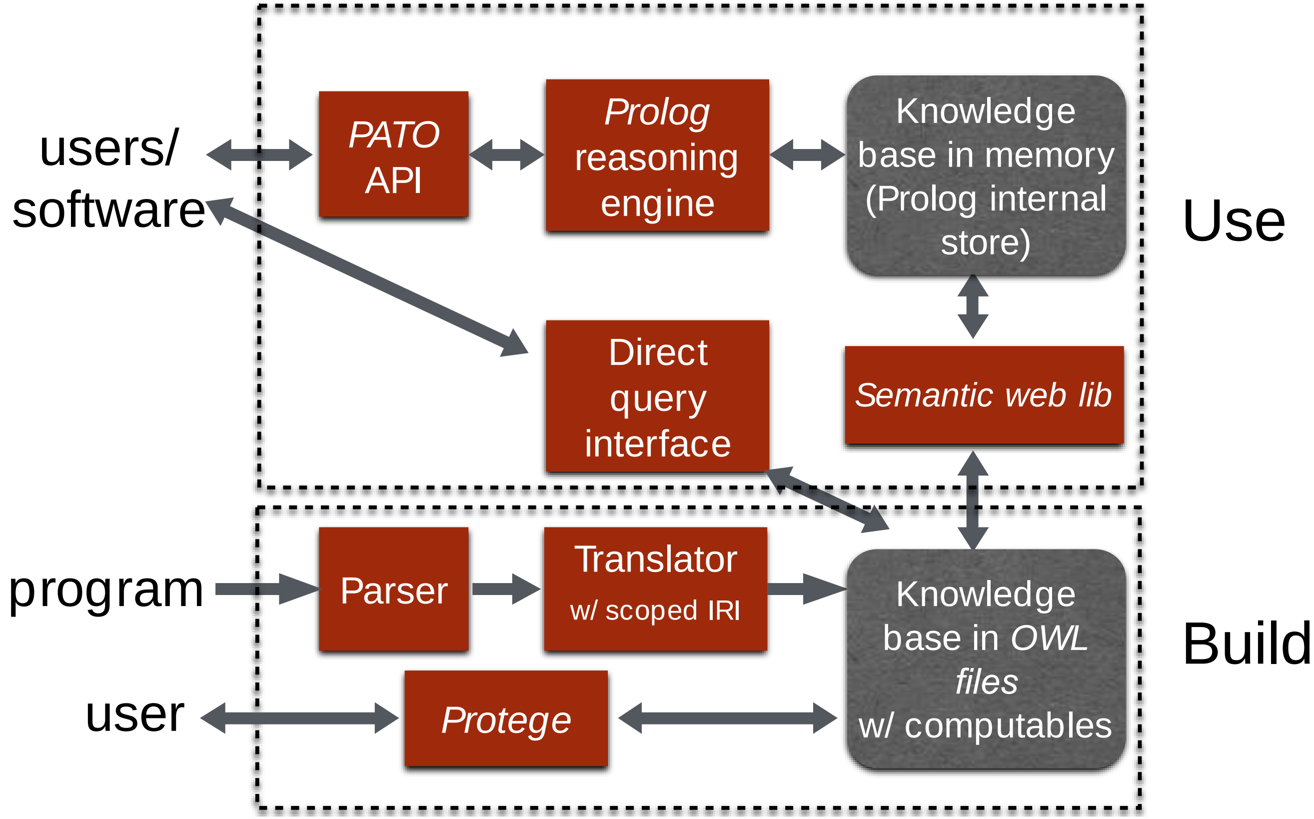
- Many logic programming tools to utilize ontology
  - Reasoners: HermiT, FaCT++, SWI-Prolog, …
  - We use SWI-Prolog for its versatility and interoperability with C/C++
  - Describe program analysis as Prolog rules

```
% Prolog rules:  FACTS :- Conditions
descendant(X,Y) :- child(X,Y). % a child is a descendant
descendant(X,Y) :- child(X,Z), descendant(Z,Y). % recursively check descendant
% Semantic web library query
?- rdf('machine1', rsdf:type, Y). % check an entity machine1 's type (NamedIndividual, Computer)
```

- Efficiency concerns
  - Significant performance improvement in logic tools: SWI-Prolog
  - Well designed reasoning schema, e.g., avoid cycles, repeat reasoning, etc
  - Interfacing with high performance language to offload some work

The Program Analysis through Ontology(PATO) framework

- Put various components together into a unified framework for program analysis



Experience

- Canonical loop analysis – to recognize the loop in the canonical form as defined in the OpenMP specification

```
for (init-expr; test-expr; incr-expr)
    structured-block
```

```
// init-expr
var = lb
integer-type var = lb
random-access-iterator-type
   var = lb
pointer-type var = lb
```

```
// test-expr
var relational-op b
b relational-op var

// relational-op:
< <= > >=
```

```
// incr-expr
++var
var++
--var
var–
var += incr
var -= incr
var = var + incr
var = incr + var
var = var - incr
```

Canonical loop analysis

- The specification written as declarative Prolog rules (partial)

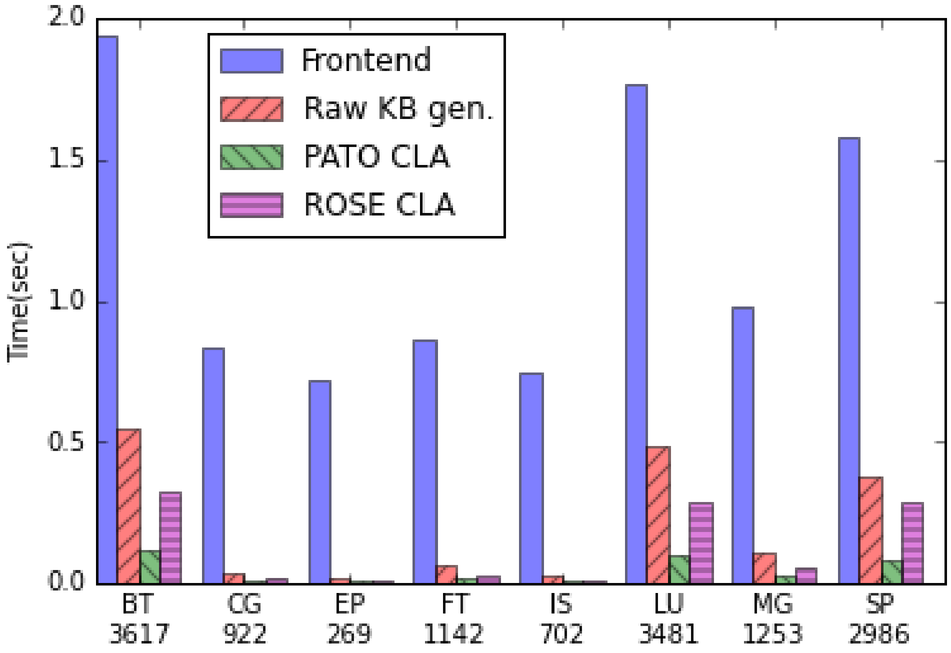
```
% top level rule to find canonical loop
canonicalLoop(Loop) :-   top level matching
 isForStatement(Loop), !, %'!' prevents backtracking
 hasForInit(Loop, InitExpr), %',' means logic AND
 canonicalInit(InitExpr, LoopVar),
 hasForTest(Loop, TestExpr),
 canonicalTest(TestExpr, LoopVar),
 hasForIncr(Loop, IncrExpr),
 canonicalIncr(IncrExpr, LoopVar),
 (
  hasType(LoopVar, 'IntType'); %';' means logic OR
  hasType(LoopVar, 'PointerType')
 )
 hasBody(Loop, ForBody),
                              matching for loop init-expression
% supportive rules to find canonical init-exp
canonicalInit(Init, LoopVar) :-
 hasOperator(Init, AssignOperator), !,
 hasLeftOperand(AssignOperator, VarRef),
 referTo(VarRef, LoopVar),
 hasRightOperand(AssignOperator, LB).
```

%Prolog rules syntax:
FACTS :- Conditions

some tricks are used for efficiency, e.g., cut (!)

Results: NAS Parallel Benchmark (NPB) Suite

- Productivity
  - 190 lines Prolog rules
  - 380 lines for C++ imperative implementation in ROSE
- Efficiency
  - Comparable to imperative one
- Extensibility
  - C++ canonical loops using random access iterators, like *std::vector<T>::iterator*
  - Just adding the concept *RandomAccessIterator*



Pointer Analysis

- More complex analysis like the (Andersen's) pointer analysis
  - Points-to set for a pointer x: pts(x)
  - Relevant operations and pts(x) propagation rules

Constraint type	Statement	Propagation rule
Base	$p = \&b$	$loc(b) \in pts(p)$
Simple	$p = q$	$pts(p) \supseteq pts(q)$
Complex	$p = *pp$	$\forall v \in pts(pp) \cdot pts(p) \supseteq pts(v)$
Complex	$*pp = q$	$\forall v \in pts(pp) \cdot pts(v) \supseteq pts(q)$

- PATO Step 1. generate relations related to pointer analysis, e.g.:
  - p = &b : (p, isStackLocOf, b)
  - p=malloc(): (p, heapLoc, …)
  - q = *p : (q, load, p)
  - *p = q : (p, load, q)

Pointer Analysis (cont.)

- Step 2. Propagating the points-to relations on the constraints
  - Two approaches:
    - Using pure logic specification, e.g.,

```
% p = q
pointsTo(P, X) :-  copy(P, Q), pointsTo(Q, X).
% p = *pp
pointsTo(P, X) :-  load(P, PP), pointsTo(PP, V), pointsTo(V, X).
```

      – efficiency depends on the reasoning system. E.g., Datalog, bottom-up, OK; Prolog, top-down, expensive redundancy.
    - Instead, solving the graph transitive closure problem by an iterative worklist algorithm in Prolog – explicit bottom-up algorithm until converge.

```
andersenPtr :-
 select(WorkList, V),
 propLoad(V); propStore(V);
 propFieldLoad(V); propFieldStore(V);
 propEdge(V),
 andersenPtr. % itratively execute the analysis
```

Performance

- Productivity
  - Totally less than 500 lines of Prolog
  - Vs. 2881 lines in LLVM's implementation
- Efficiency
  - Applied to NPB benchmark (700~3600 LOC), less than 2 sec. on single program
  - bzip2 (7K LOC), 2.1 sec
  - gzip (8.6K LOC), 3 sec
  - oggenc (58K LOC), 36 sec

Facilitating compiler cooperation

- The potential benefits of the standard representation of ontology to promote synergy between different compilers

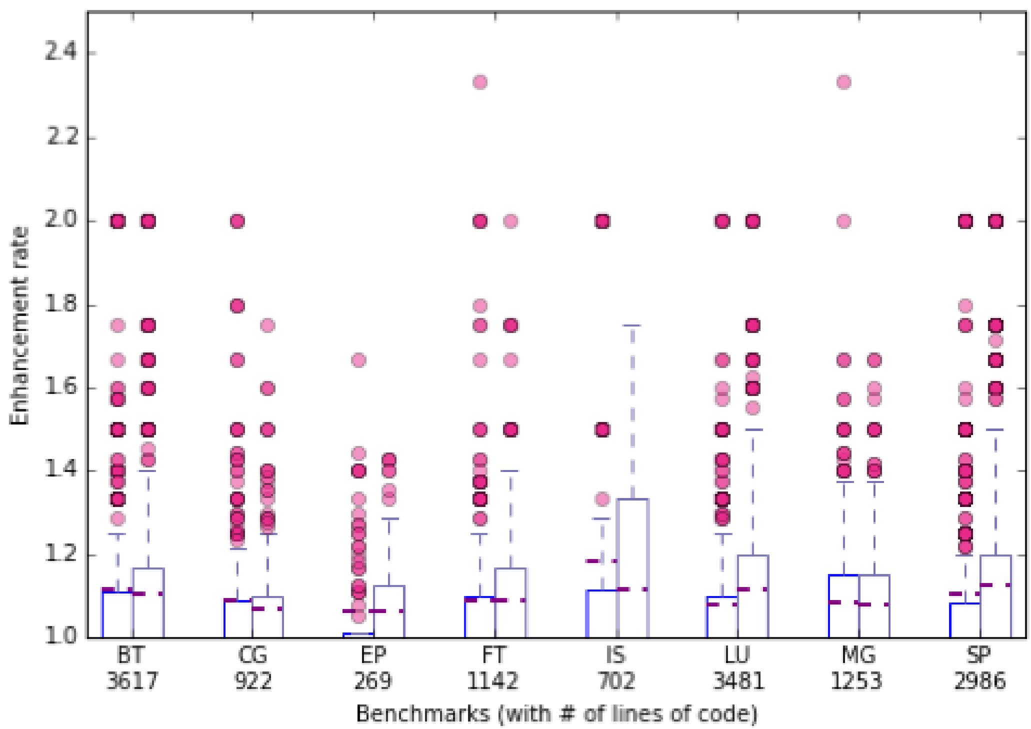
- Liveness analysis
  — a *may-type* data flow analysis
  — is conservative
  — Both LLVM/Clang and ROSE have own implementation
  — Can we synergy their result to improve the accuracy?

- Enhancement rate
  — Let A and B stand for two Liveness analyses and RA and RB be their Liveout set for a given basic block. The enhancement rate over A is defined as

$$enhancementRate(A) = \frac{|RA|}{|RA| \cap |RB|}$$

Liveness analysis – the enhancement rate

- Box plots show the distribution of the enhancement rates over all the basic blocks in the program
- Bars (left, right): average precision improvement for LLVM (10%) and ROSE (20%)



Conclusion

- Ontology-based program analysis has many benefits
  - Representation
    - Uniformly represent input programs and analysis results (both intermediate and final)
  - Implementation
    - Leverage logic reasoning of ontology to facilitate declarative program analysis
  - Cooperation
    - Reuse and merge of analysis results from different implementations

- Current source code releases
  - https://github.com/yzhao30/PATO-ROSE
  - https://github.com/yzhao30/PATO-Pointer-Analysis

Questions and Answers

Lawrence Livermore National Laboratory

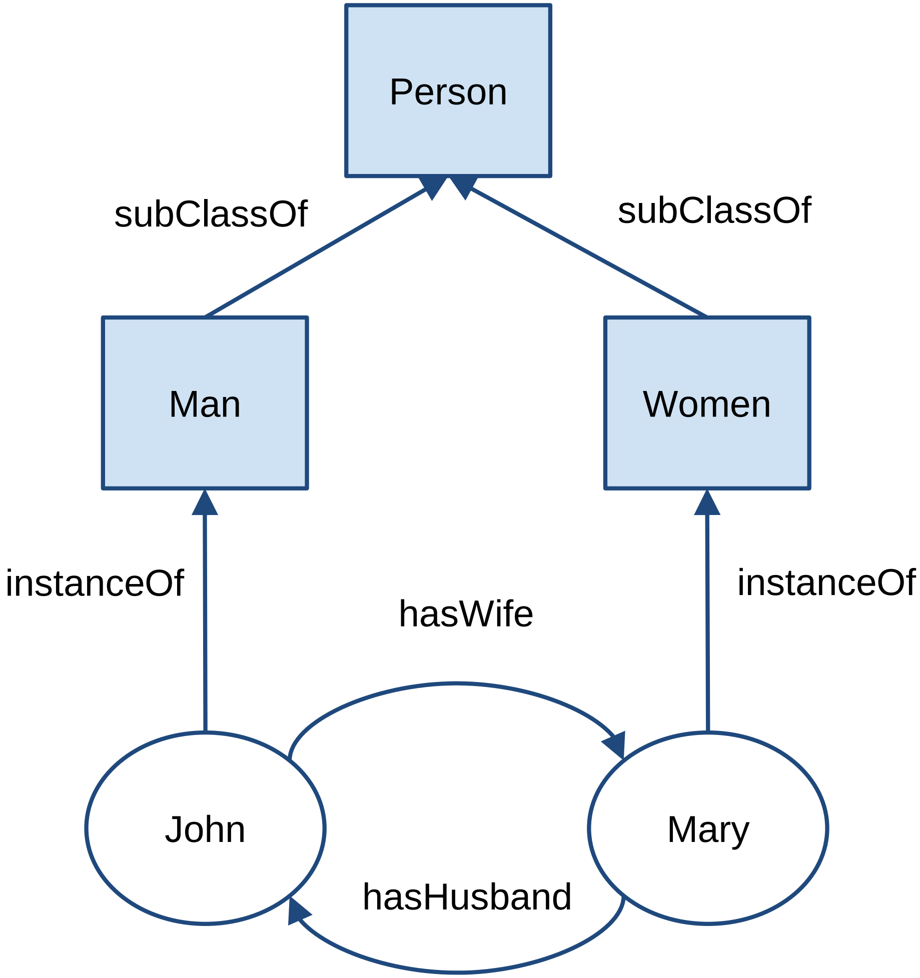
GPU data placement optimization

- We also experiment with PATO for guiding data placement on GPU, which requires
  - hardware memory characteristics
  - program knowledge for data access pattern matching
  - heuristic rules or analytic models to inference the decision
- Ontology is used
  - to link different knowledge into a unified knowledge base
  - to unify different components (hardware knowledge query, program pattern matching, logic reasoning rules)
- Benefits
  - Knowledge sharing: one of the major goals of ontology modeling
  - Extensibility: ontology is standard but schema-less, adding new concepts like new hardware features is easy

Overview

- Ontology-based program analysis
  - Centers around a knowledge base built upon ontology
  - Knowledge generation
    - E.g., an ontology converter, built on compiler parser, derives program knowledge from the source code, express it in standard format, put it in the knowledge base
  - Logic reasoning
    - Ontology-based program analysis keeps the convenience of declarative program analysis

Example family ontology using OWL



Family ontology in triplet format

(Man, subClassOf, Person)
(Woman, subclassOf, Person)

(John, instanceOf, Man)
(Mary, instanceOf, Women)

(John, hasWife, Mary)
(Mary, hasHusband, John)