Towards Ontology-Based Program Analysis*

Yue Zhao[1], Guoyang Chen[2], Chunhua Liao[3], and Xipeng Shen[4]

1 **Department of Computer Science, North Carolina State University, USA**
yzhao30@ncsu.edu
2 **Department of Computer Science, North Carolina State University, USA**
gychen1991@gmail.com
3 **Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, USA**
liao6@llnl.gov
4 **Department of Computer Science, North Carolina State University, USA**
xshen5@ncsu.edu

Abstract

Program analysis is fundamental for program optimizations, debugging, and many other tasks. But developing program analyses has been a challenging and error-prone process for general users. Declarative program analysis has shown the promise to dramatically improve the productivity in the development of program analyses. Current declarative program analysis is however subject to some major limitations in supporting cooperations among analysis tools, guiding program optimizations, and often requires much effort for repeated program preprocessing.

In this work, we advocate the integration of ontology into declarative program analysis. As a way to standardize the definitions of concepts in a domain and the representation of the knowledge in the domain, ontology offers a promising way to address the limitations of current declarative program analysis. We develop a prototype framework named PATO for conducting program analysis upon ontology-based program representation. Experiments on six program analyses confirm the potential of ontology for complementing existing declarative program analysis. It supports multiple analyses without separate program preprocessing, promotes cooperative Liveness analysis between two compilers, and effectively guides a data placement optimization for Graphic Processing Units (GPU).

1998 ACM Subject Classification D.3.4 [Programming Languages] Processors—Compilers

Keywords and phrases ontology, compiler, program analysis

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.26

1 Introduction

Program analysis [45] is a common way for deriving various properties of a program from its code. It is fundamental for many aspects of modern computing, including program optimizations, vectorization and parallelization, performance or correctness bug identification, task scheduling, and so on.

* This material is based upon work supported by DOE Early Career Award (DE-SC0013700), and the National Science Foundation (NSF) under Grants No. 1455404, 1455733 (CAREER), and 1525609. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE and NSF.

There are mainly two ways to implement a program analysis. A traditional way is imperative, in which, thousands of lines of code (often in some imperative programming languages) is developed based on some compiler framework for analyzing program constructs, types, control or data flows to infer certain properties of the target program. Programmers typically need to go through some steep learning curve about the structure and internal details of a complex compiler, while the results are often unsatisfactory: The code is often difficult to maintain, and bugs are common [63]. The analysis, being specific to a particular compiler, is hard to extend, to compose, or to reuse for other compilers.

The second approach, *declarative program analysis*, has been proposed to overcome the productivity issues [56, 32, 17]. With it, the developers just need to define some abstract domains and then use some logic programming language (e.g., Datalog [60]) to describe the analysis rules that govern the relations or properties of interest. Some automatic tools can then automatically do the inferences over a certain representation of some relations in the target program to find out the wanted relations or properties of the program. Experiments have shown that, with this approach, the code size of a program analysis often reduces by orders of magnitude compared to the imperative approach, and the analyses become easier to maintain and extend [13]. Moreover, with the substantial improvement in optimizations of the logic processing engines (e.g., bddbddb [59] and numerous optimizations to inference engines [62]), the performance and scalability concerns of declarative program analyses have been largely resolved.

In this work, we aim to further improve this promising paradigm of program analysis, particularly, to investigate solutions to three most important limitations of the current declarative program analysis:

- *Cooperations.* By expressing the analysis at a high level, different analysis tools could potentially reuse an analysis, and different analyses could get composed together into a more sophisticated analysis. However, in practice, these benefits have been difficult to achieve in general, due to the differences in the analysis-specific representations of programs and relations. In one analysis, the domain may be variable names and heap addresses, and the relation may be "assigning one address to a variable"; in another analysis, the domain may be expressions and the relation may be "calculated before a program point". To compose the two analyses, the variable names and heap addresses in the first analysis may have to be mapped to the expressions in the second domain, which would require much code development (likely entwined with the code in the compilers), especially if the two analyses were developed by different users based on different compilers that use different intermediate representations (IRs).
- *Optimizations.* So far, explorations of declarative program analysis have been focused on understanding program behaviors (largely for the purpose of debugging), for which, program-level knowledge has been enough. It is, however, insufficient for another important purpose of program analysis, guiding program optimizations. For the multi-facet dependence of performance, program optimizations often need knowledge from various sources: the program itself, the hardware, the algorithms, the program input datasets, and various domain-specific or problem-specific knowledge. Consequently, to provide useful optimization guidance, declarative program analysis must support the representations of the various kinds of knowledge, and allow easy linkage among them, even if the various kinds of knowledge may come from different sources. The analysis-specific nature of the current declarative analysis designs offers poor support to these needs.
- *Preprocessing.* A declarative program analysis typically requires some preprocessing to extract useful relations from the target programs to build up a relational database. This

step hurts the productivity benefits of this approach: As it is usually tightly coupled with some compiler framework, it is tedious and error-prone to develop. What makes this especially problematic is that different program analyses often *use different relations or ways to define the same or similar relations*. As a result, preprocessing needs to be developed for almost every newly developed program analysis, seriously throttling the productivity benefits of declarative program analysis.

In this work, we advocate the integration of ontology into declarative program analysis to address the three limitations all together. Our proposal comes from the observation that all the three major limitations essentially stem from a single fundamental shortcoming in current declarative program analysis: the lack of a systematic conceptual framework to govern the definition, representation, and organization of the various kinds of knowledge (relations in a program, rules, domains, hardware configurations, etc.) related with program analysis. The ad-hoc analysis-specific approach used in today's designs of declarative program analysis is the fundamental reason for the much effort required for preprocessing, and the barriers for supporting cooperations and optimizations.

Ontology, a concept originated from Philosophy, refers to the study of the nature of being, as well as the basic categories of being and their relations [47]. In recent decades, it has become a branch in Information Science, serving as the primary way to standardize the concept definitions and knowledge representations for a domain. It includes three concrete components: A standard vocabulary and definitions of some common concepts and their relations in a given domain, a standard format (e.g., the Web ontology Language (OWL)) for representing the various instances and concepts in any concrete problems in the domain and their relations, and a whole set of *tools* that have been developed in the last several decades for the development of an ontology and automatic logic inference upon it.

The key idea in our proposal is to leverage ontology to help standardize the definitions of domains, relations, and other concepts in program analysis, and to establish a single flexible representation of program constructs as well as other kinds of knowledge related with program analysis and its usage. With that, knowledge from various sources may be linked seamlessly as long as they follow the standardized representation. Sharing the same conceptual framework and set of terminology, different program analyses will be easy to compose and interoperate together. The standardization will also make it possible to develop a single comprehensive database of the relations of a program to serve for various program analyses, removing the needs for the separate development of preprocessing for each analysis.

Ultimately, it would be desirable to establish a standard Program Analysis ontology to describe programs, analysis, and related concepts—liken how the Semantic Sensor Network ontology by W3C [15] facilitates the work in the sensor network domains. Reaching that goal would require the coordinated effort from the community and goes beyond the scope of this paper.

This work instead focuses on the following four-fold objectives:

- To introduce the idea of integrating ontology into program analysis, and explain the concept of *ontology-based program analysis* and its potential benefits.
- To investigate the feasibility of having a single representation of a program in ontology to facilitate various program analyses and hence reduce the much effort for developing program preprocessing as required in current declarative program analysis.
- To validate the promise of ontology as the representation of various kinds of knowledge related to program optimizations, and hence extend existing declarative program analysis to guide program optimizations.

- To confirm the benefits of ontology for facilitating easy cooperations of different analysis tools.

To reach the four objectives, we have developed a prototype framework named PATO (which stands for Program Analysis Through ontology). In this prototype, we explore the use of several principles to define a concept-proof ontology for C program representations. Based on it, we have developed five program analyses: canonical loop analysis, pointer analysis, control flow graph construction, data access pattern analysis, and GPU data placement guidance. These analyses differ in domains, relations, scopes, and intended usage. Our experiments show that a single ontology-based representation can successfully support all these analyses (without separate preprocessing per analysis). The analyses inherit the productivity benefits of declarative program analysis, reducing the lines of code by tens of times compared to imperative implementations. Using the liveness analysis on two compilers (ROSE [4] and LLVM [39]), we confirm the benefits of ontology for promoting cooperations among different analysis tools. And using GPU data placement optimization, we demonstrate the seamless linkage of various sources of knowledge (programs, domain experts, and hardware) enabled by ontology, and reveal the potential of ontology-based program analysis for guiding program optimizations.

We acknowledge that this work is just the first step towards ontology-based program analysis. The ontology implemented in this work is for only the representations of C programs (and GPU data placement), and the analysis has not exploited the potential of ontology much beyond logic programming upon ontology-based program representations. A full development of ontology-based program analysis would require also the ontology and formal definitions of common concepts in program analysis (e.g., dominator, post order, alias), their relations, and some deep investigations of the opportunities that ontology may bring to program analysis. Through this first step, by demonstrating the promise of ontology-based program analysis, we hope that this work will stimulate further studies by the community into this promising direction.

2 Background

Ontology is a concept originating in Philosophy, referring to the study of the nature of being, as well as the basic categories of being and their relations [47]. In recent decades, it has become a branch in Information Science for representing the knowledge in a particular domain. In this context, an ontology is a formal explicit description of a domain's knowledge, including concepts (or classes), properties of each concept (or relations) and individuals (or instances of classes) [53]. An ontology is often visualized as a graph in which nodes indicate concepts and edges indicate relations between concepts. It is often represented in some standard triple format as we will describe later in this section. Developing an ontology for a domain has many benefits, including 1) making domain knowledge explicit to expose what is known and what is unknown, 2) enabling knowledge interoperability by providing a common taxonomy and vocabulary, 3) providing knowledge reuse since the ontology is a persistent knowledge base, and 4) facilitating knowledge validation and reasoning using existing inference engines (or reasoners).

Figure 1 shows an example borrowed from an introductory article of ontology [47]. A class (e.g., "Winery") describes concepts in a domain. A class can have subclasses, representing the "kind of" relations among concepts (e.g., "red wine" may be a subclass of "wine"). A class can have many individual instances (e.g., "Chateau Lafite Rothschild Pauillac" is an instance of the class "Pauillac"). A property describes the attribute of a class or

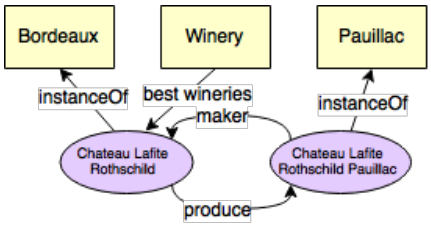

Figure 1 An example ontology for the domain of winery.

instance. It is usually represented as an edge between classes or instances. For instance, the "maker" property in Figure 1 shows that the maker of "Chateau Lafite Rothschild Pauillac" is "Chateau Lafite Rothschild". A property may have some constraints, which describe the value type, allowed values, the number of the values (cardinality), and other features of the values which the property can take.

The theory foundation of ontology is Description Logic (DL) [36], a family of formal knowledge representation languages for formal reasoning on the concepts of a domain. DL is expressive enough to build sophisticated knowledge bases while still supporting efficient inference. It has a popular standardized dialect, Web ontology Language or OWL [44]. DL allows the use of *axioms* to describe a knowledge base. For instance, an axiom $\mathsf{Person(Alex)}$ describes that "Alex" is an instance of class "Person", and $\mathsf{Person} \sqsupseteq \mathsf{Human}$ describes the subsumption relationship between concept Person and concept Human. DL languages could use different grammars. Conventionally, a simple yet uniform format to represent axioms is $(\mathsf{subject}, \mathsf{property}, \mathsf{object})$ triple. Axioms in DL languages can all be mapped to such a format. For instance, in $\mathsf{Person(Alex)}$, "Alex" is the subject, "instance of" is the property, and "Person" is the object. Visually, it is an "instance of" edge flowing from the "Alex" node to the "Person" node in an ontology graph.

Through decades of development, a large body of tools (e.g., Stanford Protégé [25], SWI-Prolog Semantic Web Library [62], etc.) have been developed for creating ontologies and automatic reasoning upon an ontology-based knowledge base, which enables automatic questions-and-answers, consistency check of the knowledge base, derivation of new knowledge, and so on.

Like declarative program analysis, most of these tools leverage logic programming languages (e.g., Prolog) for inferences. In logical programming, a program is composed of a list of rules written in the form of *clauses*:

$$\mathsf{H} \text{:-} \mathsf{B}_1, \ldots, \mathsf{B}_n.$$

which reads as

$$\mathsf{H\ is\ true\ if\ B}_1\ \mathsf{is\ true\ and\ ...\ B}_n\ \mathsf{is\ true}.$$

H is called the *head* of the rule and $\mathsf{B}_1, \ldots, \mathsf{B}_n$ is called the body. When the body is empty, the rule becomes *fact*; for instance, $\mathsf{variable(a)}$ states the fact that a is a variable.

3 Overview

By offering a generic way to represent the knowledge in a domain, ontology simplifies the accumulation and share of various kinds of knowledge among people or software agents. At the same time, it allows automatic analysis and utilization of knowledge through high-level

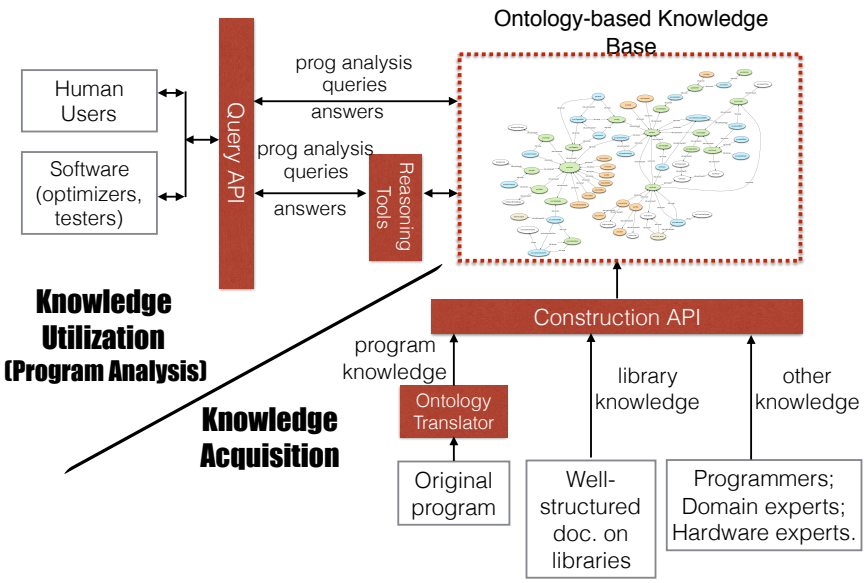

Figure 2 Overview of using ontology for program analysis.

declarative logic programming, thanks to its description logic foundation. These properties make it potentially valuable for facilitating program analysis which is essentially about reasoning about the knowledge related to a program.

Figure 2 illustrates the basic idea of ontology-based program analysis. It centers around a knowledge base built upon ontology. The knowledge base may consist of the basic knowledge about the code of the target program, as well as other knowledge (e.g., architecture attributes) relevant to the program analysis. An *ontology converter*, equipped with a parser, derives the basic program knowledge from the code of the program, expresses the knowledge in a standard format, and puts it into the knowledge base. This basic knowledge may include the structures and components (control blocks, data structures, etc.) of the program. In addition, the knowledge base may include some knowledge that could be imported about some libraries, or directly input by a domain expert about some properties of the program or the hardware it executes on (for optimizations).

Built upon description logic, ontology-based program analysis keeps the conveniences of declarative program analysis. Rather than writing thousands of lines of code inside a complex compiler, users can simply write some logic queries about the kind of properties (e.g., which loops are canonical loops) of the program that they want to know. These queries should follow some ontology query APIs. The APIs will then return the answers that are automatically obtained from the ontology-based knowledge base. For complex queries, it can leverage many existing ontology reasoning tools [62, 55]. Users of the ontology-based knowledge base can be humans or software agents (e.g., tools for program optimizations or testing.)

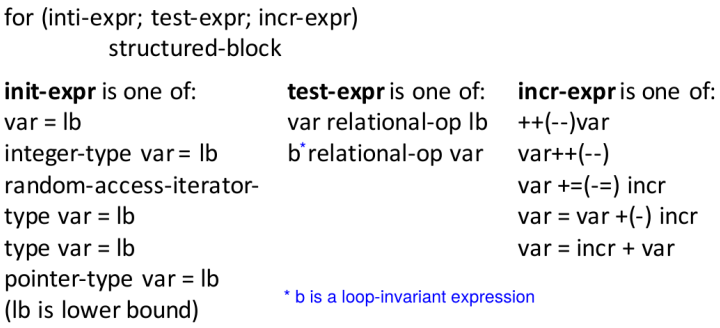

Figure 3 Specification of a canonical loop (derived from OpenMP manual [48]).

Example. We use canonical loop analysis to illustrate how the idea of ontology-based program analysis works. A canonical loop is a type of well-structured loop conforming to specifications as shown in Figure 3. Because of its regular structure, it has been the focus of many studies on parallelization and loop optimizations [35, 41, 16].

The goal of *canonical loop analysis* is to recognize whether a loop is in a canonical form. Traditional implementations of the analysis (e.g., the implementation in the ROSE compiler [4]) contains hundreds of lines of code for examining the IR of a loop. The code is tied to a particular internal data structure of the chosen compiler, hard to port to another compiler or maintain.

In a traditional program analysis development, the task requires an insertion of a separate pass over some intermediate representation of the whole program, which may need the development of thousands of lines of code. For example, in the ROSE compiler [4] (a source-to-source compiler broadly used in High Performance Computing), the pass works on an Abstract Syntax Tree (AST). To analyze the code at that level, a programmer needs to implement many lines of code written in procedural languages (C/C++). The canonical loop analysis in the ROSE compiler consists of 380 lines of source code for examining the representations of the structure of each loop and check them against the conditions in Figure 3.

In ontology-based program analysis, the process is simpler. The programmer needs to invoke some provided ontology converter on the code of the target program. An ontology-based knowledge base is then produced to capture the program constructs, components, and their relations. The programmer then just needs to use a declarative logic programming language to describe rules governing the forms that a canonical loop should conform. Treating those rules as queries on the ontology of the program, existing logic reasoners can then automatically find all the canonical loops in the target program. For example, a fragment of C code is shown in Listing 1. The code's corresponding ontology representation is shown in Listing 2. Each line is a triple: (subject, predicate, object). The numbers in the subjects or objects are the line and column numbers of the beginning and ending positions of a language construct in the source code. (Next section explains the triples in the example in details.) The different program constructs can be easily extracted by Prolog queries like those in Listing 3.

Allowing the use of logic programming, ontology-based program analysis inherits the productivity benefits of declarative program analysis. More importantly, it overcomes the three aforementioned shortcomings of existing declarative program analysis by leveraging ontology for standardizing the concept definitions in a domain and the flexible representation of various sources of knowledge.

Listing 1 Example C code fragment.

```
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 }
```

Listing 2 Sample ontology for C snippet ("x rdf:type y" indicates that y is the type of the code segment in the range x).

```
1 ('3:1,5:1', rdf:type, 'ForStatement')
2 ('3:6,3:14', rdf:type, 'VariableDecl')
3 ('3:6,3:10', rdf:type, Variable')
4 ('3:1,5:1', 'hasForInit', '3:6,3:14')
5 ('3:1,5:1', 'hasForTest', '3:17,3:22')
6 ('3:1,5:1', 'hasForIncr', '3:25,3:27')
7 ('3:1,5:1', 'hasBody', '3:30,5:1')
```

4 Challenges and Solutions

The challenges for integrating ontology into program analysis exist in each of the four main steps: the design of an ontology for the domain, the generation of the knowledge, the utilization of the knowledge base, and the design of the entire framework. In this section, we discuss each of the challenges and present some principles we use to address them. At the end, we describe PATO, the prototype framework we have developed to do program analysis upon ontology-based program representations.

4.1 Ontology Design

Challenges. To create an ontology for any domain, the first primary task is to define the vocabulary to be used in the domain. That includes the definition of the concepts, properties, and restrictions in the domain. These definitions establish the conceptual terms and their relations of the ontology-based knowledge base for the domain. Even though there are some de facto procedures on designing an ontology [47], program analysis has some special challenges. Program analysis has a large variety of tasks (e.g., loop analysis, data access pattern analysis, alias analysis, dependence analysis, liveness analysis, busy expression analysis, etc.) involving a huge set of diverse concepts and relations. Even a larger variety exists in the input programs. So the first question to use ontology for program analysis is how to design an intuitive, efficient and flexible ontology that can facilitate the various program analyses and input programs.

Solutions. In this work, we focus on the design of ontology for representing programs of a particular programming language (C). The design of the complete ontology of program analysis is left to future work.

Through our explorations, we have found the following three principles helpful.

Listing 3 Sample analysis rules.

```
1 isForStatement(Loop) :-
2  rdf(Loop, rdf:type, c:ForStatement).
3 hasForInit(Loop, InitExpr) :-
4  rdf(Loop, c:hasForInit, InitExpr).
```

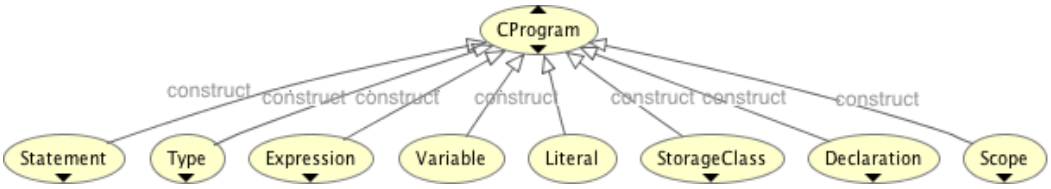

Figure 4 Top-level ontology for C program analysis.

Language standard-oriented design. When designing the vocabulary of an ontology, it helps if one starts with reusing language constructs and their categorizations defined in the standard of the programming language of the target programs. Despite the variety of the input programs, they are all artifacts following the particular programming language. With the constructs and their categorizations of the programming language covered, we can easily express programs using the language in the ontology-based knowledge base. This approach helps achieve a good coverage of the input programs with a vocabulary familiar to users. For example, to model C programs in an ontology, we followed the standard of C99 and enumerate all program constructs and concepts in a top-down fashion. Figure 4 shows a fraction of the top-level ontology for C programs. The main concepts in the domain include variables, expressions, statements, and so on. The "construct" on the edges indicate these vocabularies are the basic constructs in the C program domain.

Being generic in property designs. Besides classes of concepts, an ontology also contains a vocabulary for properties (or called relations). For example, an Expression instance may have some Type, an Identifier may refer to some Definition. Because there may be many properties to express, in our practice, we follow a principle trying to define properties in a generic way, and encode semantic meanings into concepts whenever possible. That allows the possible use of a small set of properties and their combinations to express a large number of possible properties in a program. For example, when describing *an identifier has static storage class*, one approach is to define a property hasStaticStorage and use it like (someVar hasStaticStorage true). Alternatively, we may define the concept Storage as a class and use a simpler property hasStorage as (someVar hasStorage static), where static is a member of Storage. There are several benefits for this second choice. First, using generic properties makes the set of property vocabularies small and thus easy to manage. For example, hasStorage is used to describe all storage classes instead of creating specific properties for each storage class. Second, stripping semantics from property make writing logic rules more flexible. For example, the knowledge of (someVar hasStorage static) can be queried by the keyword hasStorage. Otherwise, users may need to use many specific keywords as hasStaticStorage, hasExternStorage, and so on.

Continuous enrichment. In our exploration, we find that continuous enrichment of the ontology vocabulary can be helpful. For instance, in a canonical loop analysis, a user gives the description of the concept of a canonical loop. If that concept turns out to be needed frequently (by many users), the concept could then be integrated into the ontology

framework to save the need for repeated descriptions. Given that the program analysis ontology is intended to be used by a community, a complexity is that different users may use different names for a single user-defined relation (e.g., canonical loop), making the detection of the repeated use of a user-level concept difficult. Ontology-based logic reasoners come in handy. As the descriptions of user-defined relations all use the vocabulary in the same ontology, the reasoners can easily do a logic reasoning on the descriptions to decide whether two user-defined relations are equivalent. After recognizing the frequently needed user-level concepts, such concepts can be added into the standard ontology for future reuse.

Based on the three principles, we came up with an ontology for C program representations to facilitate C program analysis. It contains 178 concepts and 68 properties. It is not intended to be complete (e.g., many program analysis concepts are not yet defined), but is sufficient for examining the support of a single ontology to multiple different program analyses as we will show later in this paper.

4.2 Knowledge Generation

Challenges. Building up a knowledge base is essential for any application of ontology. For program analysis, the knowledge base shall include the important knowledge related with the to-be-analyzed program. There are three main questions to answer. *(1) ontology converter.* How to construct an ontology converter that can automatically convert a given program into the ontology representation needed by many common program analyses. *(2) Naming.* A program may contain functions, statements, expressions, variables, and so on. Any of them may appear many times in different locations. A challenge is what naming scheme the ontology should use to reference each reference to avoid ambiguity. *(3) Mapping.* One of the objectives of ontology-based program analysis is to facilitate the cooperations among different compilers and other program analysis tools. A difficulty is that they may have different internal representations of a program. To make them able to interact based on the program ontology, the instances of program constructs in the knowledge base should be possible to map to some common ground meaningful to the different tools. One intuitive choice of such common ground would be the source code of the program. That will also make it easy for human to collaborate with program analysis tools. A complexity in using source code for references is how to make the naming robust to code changes. *(4) Space.* The knowledge about a program can be tremendous. Besides the basic knowledge directly driven from the program, there could be many other kinds of higher-level knowledge such as canonical loops, data dependences among statements, and so on. The higher-level knowledge is derivable from the lower-level knowledge. The derivation through reasoning may take non-trivial time. But if all this knowledge is saved in the knowledge base, the space cost could be large. How to strike a good balance is important for practical usage of this new program analysis paradigm.

Solutions. We come up with the following solutions to these challenges.

Ontology converter. Our experience shows that an ontology converter can be easily created through a translator built on top of a source-to-source compiler such as ROSE [4]. A source-to-source compiler usually produces an abstract syntax tree (AST) representation which is close to the input code. The translator traverses the AST of the program to get structural and semantic information, which is then stored into the knowledge base as ontology. The program constructs are represented as individuals (i.e. instances) of some of the classes defined in the language ontology. Relations between them are represented by properties.

Naming and Mapping. In our naming scheme, we borrow the *internationalized resource identifiers (IRIs)* [28] that OWL uses. It helps avoid name conflicts. For the ontology of C program language, names of concepts are built directly from the corresponding terms used in the C language standard[1]. For example, the concept type is referenced as c:Type. An example IRI for the concept of types in a C program domain can look like "http://example.com/owl/CProgram:Type". Some aliases can be defined as short names for the prefix strings of a domain.

More care needs to be taken for designing the naming scheme for representing the instances of a program construct. Named constructs such as types, variables, functions can use the C++ qualified name concept to uniquely identify them. For an unnamed construct (e.g., an assignment statement) or a reference to named construct (e.g., variable reference), a common intuitive approach is to use its location in the source code of the program, such as file url, start location, end location where the location is a pair of (line number, column number). For instance, "http://my.com/file1.c, 3:1, 5:1" could refer to a loop that spans from the beginning of the third line to the beginning of the fifth line of file1.c. The problem with this scheme is that some minor changes to the original program may invalidate all the names of the constructs after the modification point.

We find *scoped IRI* useful to restrict the impact of a code change to the names. In *scoped IRI*, a name is composed of some qualified names and some relative locations in the source code. For constructs like functions, structures, global variables, we add their scopes before their names. Other constructs within these constructs are named by their locations, while the line numbers are relative to the start line number of their surrounding constructs rather than the beginning of the source-code file. Using this method, a global variable declared in the first line of a file "s.c" (from column 5 to 6) can be named as s.c::1:5,1:6, while a variable declared on the first line inside a function foo in file "s.c" (from column 6 to 10) can be named as s.c::foo()::1:6,1:10. Thus, if there is some change of the code, only the names in the same scope as the changing point is need to be updated.

Space. To address the space challenge of putting everything into a knowledge base, we split the knowledge base into a core knowledge base and multiple loadable supplemental knowledge bases. The core knowledge base is always loaded and others are loaded as needed. We also design a cache-like management mechanism to alleviate the problem. It maintains a buffer to store derived supplemental knowledge. When the upper limit of the buffer gets reached, it starts to evict some of the stored knowledge (which would need to be rederived when needed). For the eviction policy, there can be multiple choices: least-recently-used (LRU), least-frequently-used (LFU), and their variants.

4.3 Knowledge Utilization

Challenges. Some challenges also exist in the utilization of the knowledge base for program analysis. *(1) Efficiency.* In many cases (e.g., analyzing a large program), the runtime efficiency of conducting a program analysis could be important. A question to answer is whether the improved productivity of the new paradigm hurts the runtime efficiency and if so, how to improve the efficiency. *(2) Generality.* Using declarative programming languages could be awkward for some usage cases, especially when they involve some mathematical computations. Such cases however do exist in some program analysis and optimizations, for instance, when they relate with some performance models (an example is the data placement

[1] We follow the naming convention of UpperCamelCase for classes and lowerCamelCase for properties.

optimizations Section 5 will describe). Effectively overcoming such limitations is important for the general applicability of ontology-based program analysis.

Solutions. We address these issues by both creating some shortcuts and leveraging the features of existing ontology tools.

Efficiency. Recent years have seen some significant improvement of the performance of logic reasoners [55, 9]. Many optimizations have been developed. For instance, in SWI-Prolog, the ontology is stored as relation triples of (subject property object) with C extensions and some indices are built for each element in the triples. So a search of a particular element can be done in constant time. Additional optimizations can be applied to queries. For instance, Prolog provides the *cut* operator (i.e., the ! symbol) to avoid unwanted backtracking in search. We find that following some existing guidelines when writing queries [12] can be quite helpful for quickly narrowing down the reasoner's search space.

In scenarios where the relevant knowledge base is simple and consists of straightforward facts in triples (e.g., some memory configurations), one may construct a customized lightweight parser in high performance languages, which can further help achieve good performance than going through a heavy-weight logic reasoner.

Generality. Ontology-based logic reasoning meets the needs of many typical program analyses, but is not quite suitable for expressing analyses that involve a lot of mathematical computations (e.g., a regression-based performance modeling). We find that a mechanism called *computable* [54] originating in Robotics can help resolve the issue. Computable is some special ontology entity that can attach procedures to some classes or properties. When the deduction rule queries the individuals of one of the classes or the properties, the associated procedures are invoked to compute individuals for the target class or property. The procedure can be written in a wide range of programming languages (e.g., C, C++, Python).

Besides allowing the direct input of queries by users written in logic programming languages, ontology-based program analysis also allows queries coming from a third-party software (which could be written in even imperative languages like C, C++ or Java.) For such cases, the ontology can be processed with libraries for those languages (e.g., the OWL API [31] or Prolog interface to foreign languages[7]). That offers the conveniences for existing software tools (e.g., a compiler) to easily leverage ontology for program analysis.

4.4 Framework Design and PATO

The final challenge is how to organize the various components together into a unified framework for program analysis. It includes choosing the DL language and the reasoner that suit the needs of program analysis, designing and implementing the APIs for both knowledge base construction and queries, and integrating all the components together into a complete cohesive framework that offers effective support to various program analyses.

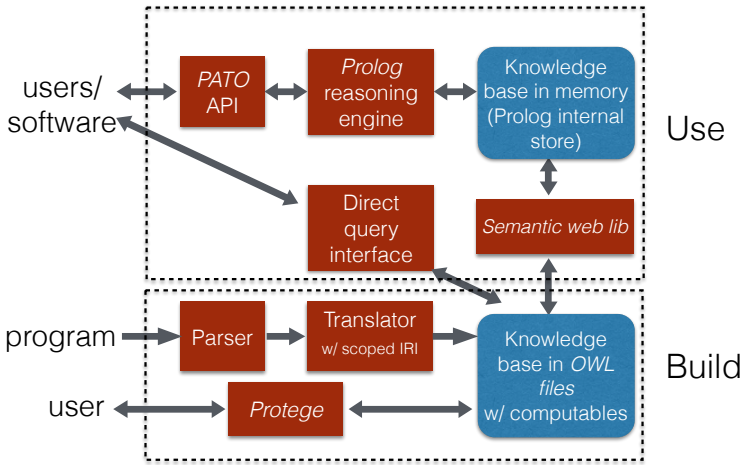

Figure 5 Structure of PATO.

To answer these questions, we have developed a prototype framework named PATO, which integrates all the aforementioned solutions to the various challenges, and leverages the power of existing ontology tools.

Figure 5 outlines the main components of PATO. The knowledge base in PATO can be in two forms, represented by the two round-corner boxes on the right part of the figure. On the disk, the knowledge is stored as OWL files. Each entry in the files is in an OWL triple (subject, property, object). For instance, (var1, hasValue, 0) means a variable has a constant value of 0. The reason for selecting this form is that OWL triple is one of the standard (and space-efficient) formats for ontology representations and is accepted by many ontology tools. Another part of the knowledge base is the *computables*, which are attached to the concepts and properties in the triple collection. The cache-like buffering mechanism is used for the space efficiency of the knowledge base.

Two of the primary ways to add knowledge into the knowledge base are shown at the bottom of Figure 5. In the first way, there is a parser (based on ROSE [4]) for converting an input program code into an AST preserving source level information. We have also developed a translator that translates the basic program knowledge on the AST into OWL triples and stores them into the knowledge base. During the translation, the translator uses the scoped IRI as the naming scheme. In the second way, we adopt Stanford Protégé [25], an interactive tool for ontology creation and manipulation. Through its GUI, a user can intuitively add entries into the knowledge base. The plugins of Protégé also provide many other features to the users, such as visualization, validation, and querying.

There are two ways to use the knowledge base. The first is shown at the top of Figure 5. It is through the SWI-Prolog reasoning engine. To use the knowledge base in this way, at the beginning, the OWL files are loaded into memory; through it, the OWl files are converted into an internal knowledge base through the existing Semantic Web Library [62]. The in-memory organization of the knowledge base features some efficient indexing schemes. When seeing a query from a user or software agent, the Prolog engine would start working on the internal knowledge base to provide the answers. Prolog provides a general interface for input logic rules and queries. We have further developed a higher-level set of API tailored to represent some common terms used in program analysis tasks (e.g., loops, functions, etc.), through them, the users may write even more concise descriptions.

The second way to use the knowledge base is shown as the shortcut (the diagonal path) in Figure 5. Through a direct query interface we have developed in C++, the user or

software agent may directly work on the OWL file collection (and computables) without going through Prolog. This shortcut is useful when Prolog is not available to a user or too costly to use in some special scenarios. For tasks which only need some simple fact search (without the need for much reasoning), this approach is more efficient than the Prolog-based approach because it avoids the overhead in Prolog interpretation and other associated cost. Both the Prolog and the shortcut approach can insert new knowledge (e.g., derived in a previous program analysis) into the knowledge base.

It is worth noting that the rich set of available tools on ontology proves helpful in our development and usage of PATO. Besides the aforementioned usage of Protégé [25] for ontology development, we find the FaCT++ reasoner [55] helpful for checking the consistency of the ontologies, the Prology *semweb* library [61] useful for loading, parsing, and manipulating ontology-based knowledge bases, and the Prolog engine [62] a convenient tool for inferences upon the knowledge bases.

5 Experience

This section describes our experience of using PATO for program analysis. To examine the feasibility of using a single ontology to support multiple different program analyses efficiently, we implement five types of program analysis on PATO: canonical loop analysis, pointer analysis, control flow graph construction, data access pattern analysis, and GPU data placement guidance. They differ in domains, relations, scopes, and intended uses. Using GPU data placement optimization, we examine whether ontology can indeed enable seamless linkage of various sources of knowledge (programs, domain experts, and hardware) and whether ontology-based program analysis can actually help guide program optimizations. Using the liveness analysis on two compilers (ROSE [4] and LLVM Clang [39]), we examine the benefits of ontology for promoting cooperations among different analysis tools.

For the interest of space, our description concentrates on the canonical loop analysis, the pointer analysis, and the cooperative liveness analysis. We briefly cover the other analyses and the GPU data placement experiment at the end. Without noting otherwise, each reported timing result is the average of 10 times of repeated measurements collected on a machine equipped with Intel Core-i5 CPU of 3.2GHz (8GB DRAM, 500Gb HDD hard drive) running Ubuntu 14.04. For the results that show large variances, we also report the statistics on the variances. The Prolog used is SWI Prolog, and the primary compiler is the ROSE compiler (EDG 4x-Based versoin) [4].

5.1 Canonical Loop Analysis

We first explain the Prolog code for canonical loop analysis. As mentioned earlier, canonical loop analysis (CLA) checks whether a loop is in a predefined canonical form. Our experiment uses the *canonical loop form* defined in the OpenMP specification [48], shown in Figure 3. The specification of an OpenMP canonical loop can be written as declarative Prolog rules as shown in Listing 4. We use italic font to distinguish variable from normal symbols.

In the Prolog specification, the *head* cannonicalLoop(Loop) asks for individuals that satisfy all clauses in the *body*. The (,) plays the role of logic conjunction (AND operation). Every clause in the *body* is deducted by its own rule. In the end, the deduction is backed by queries on the existing knowledge base. The isForStatement(Loop) is a more readable wrapper of the ontology query Loop is-a ForStatement, where the variable Loop binds to individuals if ontology triples (some-loop is-a ForStatement) exist in the knowledge base. Once the Loop is bound to some individual, clauses like hasForInit(Loop, InitExpr) search the knowledge base

Listing 4 Prolog specification of an OpenMP canonical loop (italic upper-case for variables, lower-case for properties).

```
1  % top level rule to find canonical loop
2  canonicalLoop(Loop) :-
3   isForStatement(Loop), !, %'!' prevents backtracking
4   hasForInit(Loop, InitExpr), %',' means logic AND
5   canonicalInit(InitExpr, LoopVar),
6   hasForTest(Loop, TestExpr),
7   canonicalTest(TestExpr, LoopVar),
8   hasForIncr(Loop, IncrExpr),
9   canonicalIncr(IncrExpr, LoopVar),
10  (
11   hasType(LoopVar, 'IntType'); %';' means logic OR
12   hasType(LoopVar, 'PointerType')
13  )
14  hasBody(Loop, ForBody),
15
16 % supportive rules to find canonical init-exp
17 canonicalInit(Init, LoopVar) :-
18  hasOperator(Init, AssignOperator), !,
19  hasLeftOperand(AssignOperator, VarRef),
20  referTo(VarRef, LoopVar),
21  hasRightOperand(AssignOperator, LB).
22
23 % rules with same heading: combined using logic OR
24 canonicalInit(Init, LoopVar) :-
25  hasVarDecl(Init, LoopVar),
26  hasInitializer(Init, Initializer),
27  hasValue(Initializer, LB),
28 % the rest is omitted ...
```

for (some-loop hasForInit some-init-expr) triples. Then canonicalInit(InitExpr, LoopVar) checks if the found initial expression individuals conform to the language specification. The query also returns the loop variable LoopVar if it can find it.

The analysis further checks whether the loop's init construct conforms to the specification. The first rule handles the var = lb style while the second rule deals with other styles. Different rules with the same head name form the logic disjunction (OR operation). The *cut* operator (i.e., the ! symbol) is used to prevent unwanted backtracking. It means that, as long as the first rule matches the form hasOperator(Init, AssignOperator), there's no need to check the second rule of the variable declaration form. Line 11 to 12 are the rules to do the type checking of loop variables. The (;) means logic disjunction (equivalent to two separated rules).

Experimental Results. The ontology in PATO successfully supports the analysis. We compare it with the imperative implementation in the ROSE compiler. The algorithm in ROSE traverses the AST tree of a program to find *for* statement nodes and check whether their sub-tree are in the canonical form. The code is written in C++ and is specific to the ROSE AST's internal data structures. The time complexity is $\mathcal{O}(n)$, where n is the number of AST nodes.

The code length of PATO-based analysis is about half of the imperative implementation in the ROSE compiler (190 versus 380 lines). We use the NAS Parallel Benchmarks (NPB) [8] for the measurement.

The results are shown in Figure 6. The frontend time corresponds to the parsing time of the *parser* in Figure 5. PATO uses the same frontend as the ROSE implementation. The

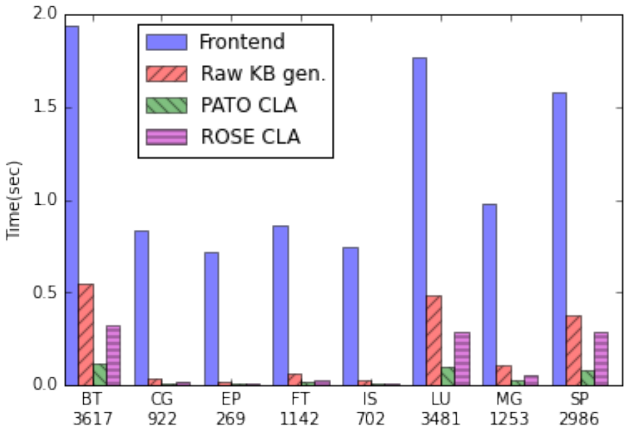

Figure 6 Time comparison of the canonical loop analysis. The X-axis shows the benchmark names and their numbers of source-code lines.

time of program knowledge generation (*KB gen*) represents the time taken by the ontology translator in the PATO framework. Finally, the other two bars are the time of the canonical loop analysis with PATO and ROSE respectively.

For the canonical loop analysis alone, we can see that the PATO declarative approach even beats the native implementation for most cases. It is slower only when the line number is small. One reason is that the ROSE's imperative implementation traverses the whole AST tree to find all for loops. For PATO's declarative implementation, the isForStatement relation information is stored explicitly and can be efficiently accessed by the hash-indexed keyword implementation of Prolog.

The ontology-based declarative analysis does have an obvious overhead: It needs an extra step to traverse the AST tree and build the initial knowledge base. However, as shown in the figure, the overhead is smaller than the frontend time. Besides, for the PATO system, the generated knowledge base can be reused for different program analyses (e.g., control flow graph analysis).

Extensibility. An appealing property of using ontology is the good extensibility. The previous discussion only covers canonical loops in C programs that use primitive loop variable types, such as pointer and integer types. OpenMP allows C++ canonical loops that use complex iterators as long as the iterator supports random access to the data elements. Examples of random access iterators include std::vector<T>::iterator and std::deque<T>::iterator. Also, programmers often define their own random access iterators. A conventional solution to extend an imperative CLA implementation is to store the known random access iterators, including the custom-defined ones, into a container for the compiler implementation to look up. The ad-hoc solution imposes barriers for exchanging the knowledge with other tools or developers. In PATO, adding such a support is simpler. One can easily add the concept RandomAccessIterator in the ontology and define it as is-a(Iterator)∧has(RandomAcess). The knowledge of is-a(Iterator) property can be gained by the knowledge builder while the knowledge has(RandomAcess) can be inserted into the knowledge base using the standard OWL API, either automatically by tools or manually by developers. The analysis rules can then reason about random access iterators. The resulting ontology can be then shared and reused by different analyses.

Table 1 Constraints in Andersen's pointer analysis.

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	$*pp = q$	$\forall v \in pts(pp) \cdot pts(v) \supseteq pts(q)$

Listing 5 The Prolog rule to match the address-taken instruction.

```
1 matchAddressTaken(RHS) :- hasOperator(RHS, AddressOp),
2  hasOperand(RHS, LocRef), referTo(LocRef, Var).
```

5.2 Pointer Analysis

This part describes our experience in implementing Andersen's pointer analysis in PATO. Andersen's pointer analysis is a well-known inclusion based analysis [5]. The analysis result is typically represented as the points-to set pts(x) for each pointer variable x. It is flow-insensitive and does not distinguish different program execution points but computes what the pointers may refer to at any time of program execution.

The analysis is commonly regarded as solving a set constraint problem. It classifies assignments involving pointers into several kinds: taking the address of a stack variable or a heap allocated space, copying a pointer from one variable to another, and assignments through dereferences or references to a multilevel pointer. It defines a propagation rule (or called constraint) for each of the cases as illustrated in Table 1 (for C programs).

The analysis consists of two steps. The first step preprocesses the target program to produce a simplified intermediate representation, which keeps only the statements that involve pointer manipulations. Each kind of the assignments involving pointers is represented with a special notation. For example, $p = \&b$ is represented as stackLoc(p, b), $p = malloc(...)$ is represented as $heapLoc(p, ...)$, $p = *pp$ becomes load(p, pp), and $*pp = q$ turns into load(pp, q).

The second step propagates the points-to relations based on those constraints and solves the problem by computing graph transitive closures through an iterative worklist algorithm [5].

In our implementation on PATO, the preprocessing step is done on the program ontology representation through some simple pattern matching rules. For example, the address taken pattern ($p = \&b$) is matched by the following: The implementation breaks complex statements into simple ones by introducing temporary variables. For example, $*p = *q$ becomes $tmp = *q; *p = tmp$. Structures are analyzed in a field-sensitive manner, while an array is regarded a single data object as done in most previous studies [5, 27, 13].

For the analysis step, the propagation rules in Table 1 can be directly mapped into logic rules in Prolog as shown in Listing 6.

However, our experiments show that Prolog inferences based on these rules are not efficient. It evaluates the rules in a top-down manner with deep recursions and costly search through a large space. Inspired by previous work [13, 59], we instead implement the classic worklist algorithm in Prolog as illustrated as follows:

Line 11 in the listing means that as soon as Prolog finds out that there is no edge from A to P, it stops search and instead add A into the worklist and continue with next A' in

Listing 6 The Prolog rules for the points-to computation.

```
% p = &a; p = malloc()
pointsTo(P, Loc) :- stackLoc(P, Loc); heapLoc(P, Loc).
% 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).
% *pp = q
pointsTo(V, X) :-  pointsTo(PP, V), store(PP, Q), pointsTo(Q, X).
```

Listing 7 The worklist algorithm for pointer analysis in Prolog.

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

propLoad(V) :-
 load(P, V),
 pts(V, A),  % for each A in pts(V)
 (\+ edge(A, P) -> assertz(edge(A, P)), add(WorkList, A)).
% others are omitted
```

pts(V, A'). The use of $\rightarrow$ enables an early stop of useless searches, bringing large performance benefits.

The entire implementation takes less than 500 lines of code, about 300 lines of which are for the preprocessing step. When being applied to the NPB benchmarks, the analysis takes no more than 2 seconds on a program. We have also applied the analysis to three programs [2] with more pointer operations: bzip2 (7K lines of source code), gzip (8.6K lines), and oggenc (58K lines). The analysis times are 2.1 sec, 3 sec, and 36 sec respectively.

5.3 Facilitating Cooperations

In the final experiment, we try to examine the potential benefits of leveraging the standard representation of ontology to promote synergy between different compilers. Particularly, we use Liveness analysis as an example.

As a *may-type* data flow analysis, Liveness analysis is conservative—that is, if a variable belongs to the Liveout set of a basic block, it means that the compiler is uncertain whether the variable is dead at the end of the basic block. So, for two different Liveness analyses (both are sound and conservative), if a variable belongs to the result from one analysis but not that from the other, we can conclude that that variable is not Live at the end of the said basic block. In another word, the intersection of the results of the two Liveness analyses gives a more precise result than either of the two analyses.

Both LLVM Clang and ROSE have their own Liveness analysis developed before. By using the ontology converters we have developed for the two compilers, we convert the Liveness analysis results from them into our ontology. By writing several lines of Prolog

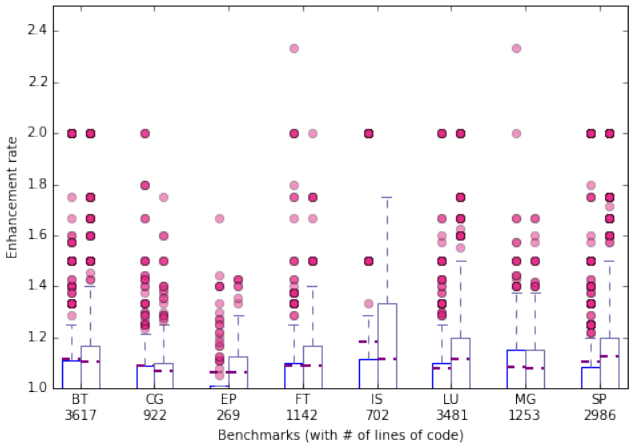

Figure 7 Enhancement of Liveness analysis by leveraging ontology to combine the results from the Liveness analysis in LLVM Clang and ROSE. (Dots show outliers). Left bars: the enhancement over LLVM Clang results; right bars: the enhancement over ROSE results.

code, the Prolog engine immediately extracts out the intersection of the sets of Live variables reported by the two compilers for each basic block of a given program.

We define a metric called *enhancement rate* to characterize the benefits of such a combination. Let A and B stand for two Liveness analyses and R_A and R_B be their Liveout set for a given basic block. The enhancement rate over A is defined as

$$enhancementRate(A) = \frac{|R_A|}{|R_A \cap R_B|}$$

The enhancement rate over B is defined similarly.

The results are shown in Figure 7. Box plots are used to show the distribution of the enhancement rates over all the basic blocks in the program. In the plot, the dots are outliers, and the intervals show the range of the lower 75 percentile of the enhancement in the observed result. The plot considers only the basic blocks whose Liveout sets are not empty in the result from at least one of the two compilers. The results indicate that the synergy improves the average precision of Liveness analysis by over 10% and 20% for LLVM Clang and ROSE respectively.

It is worth noting that the two compilers use different internal representations for programs and the Liveness analysis results. It is possible to write some special code to map their results to enable such a combination without using ontology. However, using the ontology designed in this work, the benefits come as simple side products of the ontology-based program analysis (by leveraging the converters and the standardized representation developed for many other program analyses). The productivity benefits would become even more prominent when many types of analyses cooperate across compilers.

5.4 Other Experience

For the interest of space, we briefly describe three other experiments.

One of them is an analysis to find out a program's array access patterns, including the access expressions to each array, lower and upper bounds of the loops surrounding an array reference, numbers of reads and writes to an array, and the ranges of its elements that are accessed. With PATO, each of the types of information can be easily extracted from the code

through just a few lines of Prolog statements. A previous imperative data access pattern analysis [14] has more than 2000 lines of source code. PATO, on the other hand, only needs 180 declarative rules to implement that analysis.

The second is to write code for control flow graph construction. It requires the examination of control flows of the whole program. On PATO, it is done through a set of simple rules based on the algorithm of inductive graph constructions [23]. We compare our PATO implementation with the implementations of ROSE and Clang [38], which also use the inductive construction algorithm but differ in the internal data structures and implementations. The PATO version uses only 400 lines of code, up to 8.75X shorter than the traditional compiler implementations (1200 and 3500 lines for ROSE and Clang respectively). The speed of the PATO analysis is similar to that of the Clang, and is 10-40% faster than that of the ROSE thanks to its efficient storage and query of the knowledge base.

Finally, we experiment with PATO for guiding data placement on Graphic Processing Units (GPU). A GPU has multiple types of memory with different performance characteristics. Prior studies have proposed some rule-based methods [34] and analytic model-based methods [14] for determine the placements of data that best suit the data access patterns of a GPU program and showed significant performance benefits. In this experiment, we build a data placement optimizer based on PATO, and find it much simpler to do than previous implementations. Because both the program knowledge and the hardware knowledge are represented in the standard triple format, they can be automatically linked into one single knowledge base, making automatic reasoning about data placement possible. It also simplifies the integration of heuristic rules and analytic models into a single analysis. Experiments on NVIDIA K20c GPU show that the PATO-based analysis gives similar placements as prior methods do [14]. Its analysis time is 5% longer, but it exhibits a better extensibility thanks to its ontology representation and declarative implementation. For example, in the previous work [14], to add the knowledge on some new GPUs memory features, one needs to extend a hardware specification language by adding some new constructs, and revise the code for parsing and performance modeling. In PATO, the needed changes are simpler: Through the Prolog GUI, users can easily add or remove a concept, and add queries on the new features into the analysis rules.

6 Related Work

Ontology has been used to build various knowledge bases in different domains, including Biology [6], Ambient Intelligence [19, 50, 51], Robotics [54], and others [43, 46, 49]. This work was enlightened by these studies, but concentrates on the special challenges facing program analysis.

In the software domain, ontology has been introduced, but mainly for software management and teaching of programming concepts, rather than program analysis. Specifically, Software ontology (SWO) [42] in the domain of software engineering focuses on the meta information of software (e.g., licenses, publishing processes, data formats). COPS [37] offers a sub-ontology for managing the knowledge related with image processing. Eden and others [21] have provide some theoretical discussions on the unique aspects in designing an ontology for programs, but without exploring the use of ontology for program analysis. There are several ontology designs for teaching some programming languages [52, 24]. This current work, to our best knowledge, is the first proposal on a systematic integration of ontology into program analysis.

There are some prior efforts trying to ease the difficulties in the development of program analysis. We discuss them in two aspects.

The first aspect is in the construction of a program analysis. Some prior studies have offered some interfaces for simplifying the construction of a program analysis. OpenAnalysis [3], for instance, introduces a set of analysis-specific interfaces (e.g., traverse all statements) as the building blocks for constructing a specific analysis. Other efforts in the same direction include GENOA [18] and StarTool [33]. These tools focus on imperative program analysis. There are some efforts that employ declarative program analysis to improve the productivity of program analysis development. JTransformer [1], for instance, is a tool integrated into Eclipes that allows the use of Prolog for analyzing and transforming Java source code. JunGL [57] introduces a scripting language for writing program analyses. It combines ML and Datalog. SemmleCode [58] is a tool that stores program-related data into a knowledge base, and allows the use of Datalog to write a program analysis that analyzes the program by querying the knowledge base. There are some other work [26, 13, 59] falling into the same category.

The second aspect is in the representation of a program. Some prior work has tried to develop a common software exchange format (SEF) for representing a program to facilitate the interoperation of software analysis and refactoring tools. Such a format needs both a schema (or called metamodel) that describes the objects and relationships, and a syntax that describes how model elements are to be stored and transmitted. The Dagstuhl Middle Metamodel (DMM) [40] is a representative of the former. DMM consists of a set of models that capture program elements and their relations. It follows some prior efforts such as Columbus [22] for C++ and the UML metamodel [11]. There are some other metamodels developed, such as Program Element Fact (PEF) developed in JTransformer [1], and DIMPLE [10]. Graphs are the most popular format for storing program elements in memory. TA [29] and TGraphs [20] are two examples, which are both based on typed graphs (i.e., directed graphs with attributes on both nodes and edges). GXL [30] was introduced as a generic way to use XML to represent such graphs. JunGL [57] uses some special graphs along with abstract syntax trees.

This current work shares some similarities with these prior studies, such as the use of logical programming to simplify the development of program analysis, and the creation of a common program representation. However, this work differs from the prior studies in several major aspects. First, it is the first work that points out the potential of ontology for the program analysis community to standardize the conceptualization for program analysis, and to promote the reuse and interoperations of analysis tools. It demonstrates the potential benefits through the Liveness analysis by two full-fledged existing compilers. Second, this work is the first that points out the benefits of ontology as a unified representation for not only program elements and relations but also knowledge from other sources (e.g., hardware knowledge) that are essential for program optimizations. It demonstrates the promise through GPU data placement optimizations. Third, unlike many of the prior studies that attempt to create a standalone tool for program analysis (e.g., JTransformer [1], Semmlecode [58]), this work aims to proposing an approach or a paradigm, which could potentially be employed by many program analysis tools and compilers, as demonstrated by the expeirments described in the previous section.

7 Conclusion

This work demonstrates the promise of ontology for overcoming the three major limitations of today's declarative program analysis. The five types of data analyses on PATO show that a single ontology is able to support multiple different program analyses efficiently. The

GPU data placement experiments indicate the promise of ontology for seamlessly linking knowledge from different sources, extending declarative program analysis with the capability to effectively guide program optimizations. The cooperative Liveness analysis demonstrates that with ontology-based program analysis, cooperations among different compilers or other program analysis tools become simple, and the synergy turns out to be quite beneficial. Overall, the study shows some promise of the integration of ontology into program analysis. Establishing this new way of program analysis, however, requires the development of an ontology for Program Analysis and some deep investigation of the opportunities that ontology may bring to program analysis and optimizations. We hope that this work will prompt further investigations by the community into this promising direction[2].

Acknowledgments. We thank the ECOOP'16 reviewers for the helpful comments.

References

1 The JTransformer project. `http://sewiki.iai.uni-bonn.de/research/jtransformer/`.

2 Large single compilation-unit C programs. `http://people.csail.mit.edu/smcc/projects/single-file-programs/`.

3 OpenAnalysis at Rice University. `http://www.hipersoft.rice.edu/openanalysis/`.

4 ROSE compiler infrastructure. `http://www.rosecompiler.org/`.

5 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

6 Michael Ashburner, Catherine A Ball, Judith A Blake, David Botstein, Heather Butler, J Michael Cherry, Allan P Davis, Kara Dolinski, Selina S Dwight, Janan T Eppig, et al. Gene Ontology: Tool for the unification of biology. *Nature genetics*, 25(1):25–29, 2000.

7 Roberto Bagnara and Manuel Carro. Foreign language interfaces for Prolog: A terse survey. *ALP newsletter*, 15, 2002.

8 D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

9 Nick Bassiliades, Grigoris Antoniou, and Ioannis Vlahavas. A defeasible logic reasoner for the semantic web. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2(1):1–41, 2006.

10 William C Benton and Charles N Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 13–24. ACM, 2007.

11 Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) Addison-Wesley Object Technology Series*. Addison-Wesley Professional, 2005.

12 Ivan Bratko. *Prolog (3rd Ed.): Programming for Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

13 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM. `doi:10.1145/1640089.1640108`.

[2] The source code of PATO and the analyses are accessible through the following links: https://github.com/yzhao30/PATO-ROSE, https://github.com/yzhao30/PATO-Pointer-Analysis.

14 Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 88–100, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/MICRO.2014.20.

15 Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012. URL: http://www.w3.org/2005/Incubator/ssn/ssnx/ssn, doi:10.1016/j.websem.2012.05.003.

16 Leonardo Luiz Padovani Da Mata, Fernando Magno QuintãO Pereira, and Renato Ferreira. Automatic parallelization of canonical loops. *Sci. Comput. Program.*, 78(8):1193–1206, August 2013. doi:10.1016/j.scico.2012.09.006.

17 Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. *SIGPLAN Not.*, 31(5):117–126, May 1996. doi:10.1145/249069.231399.

18 Premkumar T Devanbu. GENOA: A customizable language-and front-end independent code analyzer. In *Proceedings of the 14th international conference on Software engineering*, pages 307–317. ACM, 1992.

19 Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. *Scenarios for ambient intelligence in 2010*. Office for official publications of the European Communities, 2001.

20 Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering–the TGraph approach. In *Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*. Citeseer, 2008.

21 Amnon H Eden and Raymond Turner. Problems in the ontology of computer programs. *Applied Ontology*, 2(1):13–36, 2007.

22 Rudolf Ferenc, Árpd Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus-reverse engineering tool and schema for C++. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 172–181. IEEE, 2002.

23 Charles N. Fischer, Ronald K. Cytron, and Richard J. LeBlanc. *Crafting A Compiler*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

24 Gopinath Ganapathi, Ravi Lourdusamy, and Veeraraghavan Rajaram. Towards ontology development for teaching programming language. In *World Congress on Engineering*, 2011.

25 John H Gennari, Mark A Musen, Ray W Fergerson, William E Grosso, Monica Crubézy, Henrik Eriksson, Natalya F Noy, and Samson W Tu. The evolution of protégé: an environment for knowledge-based systems development. *International Journal of Human-computer studies*, 58(1):89–123, 2003.

26 Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *ECOOP 2006–Object-Oriented Programming*, pages 2–27. Springer, 2006.

27 Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation - PLDI '01*, pages 254–263, New York, New York, USA, 2001. ACM Press. doi:10.1145/378795.378855.

28 Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, and Sebastian Rudolph. OWL 2 web ontology language primer. *W3C recommendation*, 27(1):123, 2009.

29 Richard C Holt. An introduction to TA: The tuple-attribute language. *University of Toronto, Toronto, Draft Mar*, 24, 1997.

30 Richard C Holt, Andreas Winter, and Andy Schürr. GXL: toward a standard exchange format. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 162–171. IEEE, 2000.

31 Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.

32 Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 20(4):104–115, October 1995. doi:10.1145/222132.222146.

33 Mark James and David Atkinson. STAR* TOOL- an environment and language for expert system implementation. *Jet Propulsion Laboratory Report NTR C*, 17536, 1988.

34 Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118, January 2011. doi:10.1109/TPDS.2010.107.

35 Mahmut Kandemir, J Ramanujam, and Alok Choudhary. Improving cache locality by a combination of loop and data transformations. *Computers, IEEE Transactions on*, 48(2):159–167, 1999.

36 Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A description logic primer. *arXiv preprint arXiv:1201.4089*, 2012.

37 Pascal Lando, Anne Lapujade, Gilles Kassel, and Frédéric Fürst. Towards a general ontology of computer programs. In *ICSOFT (PL/DPS/KE/MUSE)*, pages 163–170, 2007.

38 Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

39 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

40 Timothy C Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, 2004.

41 Chunhua Liao, DanielJ. Quinlan, JeremiahJ. Willcock, and Thomas Panas. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378, 2010. doi:10.1007/s10766-010-0139-0.

42 James Malone, Andy Brown, Allyson L Lister, Jon Ison, Duncan Hull, Helen Parkinson, and Robert Stevens. The software ontology (SWO): A resource for reproducibility in biomedical data analysis, curation and digital preservation. *Journal of Biomedical Semantics*, 5(1):25, 2014.

43 Cynthia Matuszek, John Cabral, Michael J Witbrock, and John DeOliveira. An introduction to the syntax and content of Cyc. In *AAAI Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 44–49. Citeseer, 2006.

44 Deborah L McGuinness and Frank Van Harmelen. OWL web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

45 Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2004.

46 Ian Niles and Adam Pease. Towards a standard upper ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*, pages 2–9. ACM, 2001.

47 Natalya F Noy and Deborah L McGuinness. Ontology development 101: A guide to creating your first ontology, 2001.

48 OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013. URL: `http://www.openmp.org/mp-documents/spec30.pdf`.

49 Adam Pease, Ian Niles, and John Li. The suggested upper merged ontology: A large ontology for the semantic web and its applications. In *Working notes of the AAAI-2002 workshop on ontologies and the semantic web*, volume 28, 2002.

50 Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an extensible context ontology for ambient intelligence. In *Ambient intelligence*, pages 148–159. Springer, 2004.

51 Natalia Díaz Rodríguez, Manuel P Cuéllar, Johan Lilius, and Miguel Delgado Calvo-Flores. A survey on ontologies for human behavior recognition. *ACM Computing Surveys (CSUR)*, 46(4):43, 2014.

52 Sergey Sosnovsky and Tatiana Gavrilova. Development of educational ontology for C-programming. 2006.

53 Steffen Staab and Rudi Studer. *Handbook on ontologies*. Springer Science & Business Media, 2013.

54 Moritz Tenorth and Michael Beetz. KnowRob—knowledge processing for autonomous personal robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4261–4266. IEEE, 2009.

55 Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: system description. In *Automated reasoning*, pages 292–297. Springer, 2006.

56 Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

57 Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: A scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, pages 172–181. ACM, 2006.

58 Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 880–881. ACM, 2007.

59 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag. `doi:10.1007/11575467_8`.

60 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM. `doi:10.1145/996841.996859`.

61 Jan Wielemaker. SWI-Prolog Semantic Web Library 3.0. URL: `http://www.swi-prolog.org/pldoc/package/semweb.html`.

62 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

63 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.