Abstract Code Browsing *

ISABEL GARCÍA-CONTRERAS[1] JOSÉ F. MORALES[1] MANUEL V. HERMENEGILDO[1,2]

[1] *IMDEA Software Institute* (*e-mail:* {isabel.garcia, josef.morales, manuel.hermenegildo}@imdea.org)
[2] *School of Computer Science, Technical University of Madrid (UPM)* (*e-mail:* manuel.hermenegildo@upm.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Programmers currently enjoy access to a very high number of code repositories and libraries of ever increasing size. The ensuing potential for reuse is however hampered by the fact that searching within all this code becomes an immensely difficult task. Most code search engines are based on syntactic techniques such as signature matching or keyword extraction. However, these techniques are inaccurate (because they basically rely on documentation) and at the same time do not offer very expressive code query languages. We propose a novel approach that focuses on querying for *semantic* characteristics of code obtained automatically from the code itself. Program units are preprocessed using static analysis techniques, based on abstract interpretation, obtaining safe semantic approximations. A novel, assertion-based code query language is used to express desired semantic characteristics of the code as partial specifications. Relevant code is found by comparing such partial specifications with the inferred semantics for each program element. Our approach is fully automatic and does not rely on user annotations or documentation. It is more powerful and flexible than signature matching because it is parametric on the abstract domain and properties, and does not require type definitions. Also, it reasons with relations between properties, such as implication and abstraction, rather than just equality. It is also more resilient to syntactic code differences. We describe the approach and report on a prototype implementation within the Ciao system.

KEYWORDS: Semantic Code Search, Abstract Interpretation, Assertions.

1 Introduction

The code sizes of current software systems and libraries grow continuously. The open-source revolution implies that programmers currently enjoy access to many repositories which are very often large. While this code abundance brings great potential for code reuse, with the ensuing coding time savings, it also brings about a new problem: searching within these code bases is becoming an immensely difficult task.

Most code search engines have so far addressed this problem through syntactic techniques such as keyword extraction and signature matching. (Maarek et al. 1991) is an early example of the work based on information retrieval techniques. It used keywords extracted from man pages described in natural language. More recent code search engines like Black Duck Open Hub (`http://code.openhub.net`) use the same techniques but including also keyword extraction from variable names in the code itself. They combine those keywords with very simple specifications of the kind of code the user is looking for (e.g., whether

* This research has received funding from the EU FP7 agreement no 318337, ENTRA, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2015-67522-C3-1-R *TRACES* projects, and the Madrid M141047003 *N-GREENS* program.

it is classes, methods, or interfaces). Other recent work has used a similar approach combined with ranking techniques. For example, (McMillan et al. 2012) uses annotations in code instead of man pages in order to cluster features from Java packages. The idea is that multiple users will rank over time how the packages match the search. Google code search (`https://github.com/google/codesearch`) is based on regular expressions. While keyword and regular expression search is obviously useful, the fact that these techniques rely on documentation (including the names of identifiers in the code) means that they also have shortcomings. They are clearly of limited use if the code has no comments, existing comments are wrong, or other elements like variable, module, or procedure names are not representative and/or not easy to match against. In general, searching for keywords can miss a lot of matches (so that code will be overlooked) because things can be expressed in different ways or even in a different (natural) language.

An alternative to keyword search is to query instead the signatures present in the code, an approach already proposed in (Rollins and Wing 1991) for finding code written in a functional language. The solver within λProlog was used to *match* the signatures present in code against some pre- and post-condition specifications used as search keys. The Haskell code browser, Hoogle (Mitchell 2008), combines this type of signature matching with keyword matching. In the same line (Reiss 2009) combines these two techniques with test cases as a means for specification. Signature matching is a more formal approach than keyword matching, but it is still essentially syntactic, relies on the presence of signatures in the program, and is limited to the properties of the language of the signatures, i.e., generally types.

We propose a new approach that focuses on querying for *semantic* characteristics of code that are inferred automatically from the code itself. Instead of relying on user-provided signatures, comments, or identifier names, the code bases are pre-analyzed using static analysis techniques based on abstract interpretation, obtaining safe approximations of the semantics of the program. The use of different abstract domains allows generating a wide (and user extensible) variety of properties (generalized types, instantiation modes, variable sharing, constraints on values, etc.) that can be queried. To this end we also propose a flexible code query language based on assertions that expresses specifications composed of these very general properties. These abstract query specifications are used to reason against the abstract semantics inferred for the code, in order to select code elements that comply with the queries.

Our approach is fully automatic and does not rely on user annotations or documentation. Although assertions in the code can also help the analysis, they are not needed, i.e., the approach works even if the code contains no assertions or signatures, since the program semantics is inferred by the abstract interpreter. It is thus more powerful than signature matching methods (which it subsumes), which require such signatures and/or type definitions. The proposed approach also reasons with relations between properties, such as implication and abstraction, rather than just matching, which allows much more expressive search and more accurate results. Our approach is also much more flexible, since it is parametric on the abstract domain and properties, i.e., the inference and the search can be based on any property for which an abstract domain is available and not just syntactic match of the properties in the signature language (generally types). It can also be tailored through new abstract domains to fit particular applications. Our approach can be more powerful than (and in any case is complementary to) keyword-based information-retrieval systems because its is based on a semantic analysis of the code, and is thus independent of documentation. It is also more resilient to syntactic differences (including code obfuscation techniques) such as, e.g., non descriptive names of functions/variables. Combining these two approaches (actually done in our implementation) is straightforward and is thus not addressed in this paper.

2 Preliminaries, Abstract Interpretation, and Assertions

We denote by VS, FS, and PS the set of variable, function, and predicate symbols, respectively. Variables start with a capital letter. Each $p \in \mathsf{PS}$ is associated with a natural number called its *arity*, written $\mathsf{ar}(p)$ or $\mathsf{ar}(f)$. The set of terms TS is inductively defined as follows:[1] $\mathsf{VS} \subset \mathsf{TS}$, if $f \in \mathsf{FS}$ and $t_1, \ldots, t_n \in \mathsf{TS}$ then $f(t_1, \ldots, t_n) \in \mathsf{TS}$ where $\mathsf{ar}(f) = n$. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and t_i are terms. A *predicate descriptor* is an atom $p(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are distinct variables. A *clause* is of the form H :- $B_1, \ldots, B_n$ where H, the *head*, is an atom and $B_1, \ldots, B_n$, the *body*, is a possibly empty finite conjunction of atoms. We assume that all clause heads are normalized, i.e., H is of the form of a predicate descriptor. Furthermore, we require that each clause head of a predicate p have identical sequence of variables $X_{p_1}, ..., X_{p_n}$. We call this the *base form* of p. This is not restrictive since programs can always be put in this form, and it simplifies the presentation. However, in the examples and in the implementation we handle non-normalized programs. A *definite (constraint) logic program*, or *program*, is a finite sequence of clauses. The concrete semantics used for reasoning about goal-dependent compile-time semantics of logic programs will use the notion of generalized AND trees (Bruynooghe 1991). A generalized AND tree represents the execution of a query to a Prolog predicate. Basically, every node of a generalized AND tree contains a call to a predicate, adorned on the left with the call substitution to that predicate, and on the right with the corresponding success substitution. The concrete semantics of a program P for a given set of queries Q, $[\![P]\!]_Q$, is the set of generalized AND trees that represent the execution of the queries in Q for the program P. We will denote a node in a generalized AND tree with $\langle L, \theta_c, \theta_s \rangle$, where L is the call to a predicate p in P, and θ_c, θ_s are the call and success substitutions over $vars(L)$ adorning the node, respectively. The $calling\_context(L, P, Q)$ of a predicate given by the predicate descriptor L defined in P for a set of queries Q is the set $\{\theta_c | \exists T \in [\![P]\!]_Q \; s.t. \; \exists \langle L', \theta_c, \theta_s \rangle \; in \; T \wedge \exists \sigma \in ren \; L\sigma = L'\}$, where ren is a set of renaming substitutions over variables in the program at hand. We denote by $answers(P, Q)$ the set of answers (success substitutions) computed by P for query Q.

Inferring the Program Semantics by Abstract interpretation: As mentioned in the introduction, our approach for finding predicates semantically is based on pre-processing program units using static analysis techniques, in order to obtain safe approximations of the semantics of the predicates in these units. Our basic technique for this purpose is *abstract interpretation* (Cousot and Cousot 1977), an approach for static program analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). Although not strictly required, we assume D_α has a lattice structure with meet ($\sqcap$), join ($\sqcup$), and less than ($\sqsubseteq$) operators. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to D$. Concrete operations on D values are approximated by corresponding abstract operations on D_α values. The key result for abstract interpretation is that it guarantees that the analysis terminates, provided that D_α meets some conditions (such as finite ascending chains) and that the results are safe approximations of the concrete semantics (provided D_α safely approximates the concrete values and operations).

Goal-dependent abstract interpretation: We will be using goal-dependent abstract interpretation, concretely the PLAI algorithm (Muthukumar and Hermenegildo 1992), available within

[1] We limit for simplicity the presentation to the Herbrand domain, but the approach and results apply to constraint domains as well. In the rest of the paper we will refer interchangeably to substitutions or constraints, and to the current substitution or the constraint store.

the Ciao/CiaoPP system (Hermenegildo et al. 2005; Hermenegildo et al. 2012). PLAI takes as input a program P, an abstract domain D_α, and an abstract initial call pattern[2] $\mathcal{Q}_\alpha = L{:}\lambda$, where L is an atom, and λ is a restriction of the run-time bindings of L expressed as an abstract substitution $\lambda \in D_\alpha$. The algorithm computes a set of triples $analysis(P, L{:}\lambda, D_\alpha) = \{\langle L_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle L_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each $\langle L_i, \lambda_i^c, \lambda_i^s\rangle$ triple, L_i is an atom, and λ_i^c and λ_i^s are, respectively, the abstract call and success substitutions, elements of D_α. Let Q be the set of concrete queries described by $L{:}\lambda$, i.e., $Q = \{L\theta \mid \ \theta \in \gamma(\lambda)\}$. In addition to termination, correctness of abstract interpretation provides the following guarantees:

- The abstract call substitutions cover all the concrete calls which appear during execution of initial queries in Q. Formally, $\forall p'$ in P $\forall \theta_c \in calling\_context(p', P, Q)$ $\exists \langle L', \lambda^c, \lambda^s\rangle \in analysis(P, L{:}\lambda)$ s.t. $\theta_c \in \gamma(\lambda^c)$, where L' is a base form of p'.
- The abstract success substitutions cover all the concrete success substitutions which appear during execution, i.e., $\forall i = 1 \ldots n$ $\forall \theta_c \in \gamma(\lambda_i^c)$ (which, as we saw before, cover all the calling contexts) if $L_i\theta_c$ succeeds in P with computed answer θ_s then $\theta_s \in \gamma(\lambda_i^s)$.

The abstract interpretation process is monotonic, in the sense that more specific initial call patterns yield more precise analysis results. As usual, $\bot$ denotes the abstract substitution such that $\gamma(\bot) = \emptyset$. A tuple $\langle P_j, \lambda_j^c, \bot\rangle$ indicates that all calls to predicate p_j with substitution $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success substitutions.

Multivariance: The analysis (as well as the assertion language presented later) is designed to discern among the various usages of a predicate. Thus, multiple usages of a procedure can result in multiple descriptions in the analysis output, i.e., for a given predicate P multiple $\langle P, \lambda^c, \lambda^s\rangle$ triples may be inferred and queried. This will allow finding code more accurately. More precisely, the analysis is said to be *multivariant on calls* if more than one triple $\langle P, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle P, \lambda_n^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate. In this paper we use analyses that are multivariant on calls.

Analysis target: We will look for predicates in a predefined set of programs or modules. Each of them will be analyzed independently and we will denote with $analysis(m, D_\alpha, \mathcal{Q}_\alpha)$ the analysis of a module m with respect to the set of call patterns $\mathcal{Q}_\alpha$ in domain D_α. The reason for this kind of analysis is that normally users are looking for independent libraries to reuse. We assume for concreteness the Ciao module system (Cabeza and Hermenegildo 2000). It is a strict module system, i.e., a system in which modules can only communicate via their interface. The interface of a module contains the names of the exported predicates and the names of the imported modules. When performing the analysis, only the exported predicates will be considered for the initial calls. We will use $exported(m)$ to express the set of predicate names exported by module m.

An issue in the computation performed by $analysis(m, D_\alpha, \mathcal{Q}_\alpha)$ is that, from the point of view of analysis, the code of the module m to be analyzed taken in isolation is *incomplete*, in the sense that the code for procedures imported from other modules is not available to analysis. The direct consequence is that, during the analysis of a module m, there may be calls $P : CP$ such that the procedure P is not defined in m but instead it is imported from another module m'. A number of alternatives are available (and implemented in the system in which we conduct our experiments, Ciao) in order to deal with these inter-modular connections (Bueno et al. 2001). We assume, without loss of generality, that for these external calls, we will trust the assertions present in the imported modules for the predicates they export, and use their information in the individual module analysis.

[2] We use sets of calls patterns in subsequent sections –the extension is straightforward.

Traditional Assertions: Assertions are linguistic constructions for expressing abstractions of the meaning and behavior of programs. Herein, we will use the `pred` assertions of (Puebla et al. 2000a). Such `pred` assertions allow specifying certain conditions on the state (current substitution or constraint store) that must hold at certain points of program execution. They are very useful for detecting deviations of behavior (symptoms) with respect to such assertions, or to ensure that no such deviations exist (correctness). In particular, they allow stating sets of *preconditions* and *conditional postconditions* for a given predicate. Such `pred` assertions take the form: `:- pred` $Head$ `:` Pre `=>` $Post$`.`

where $Head$ is a normalized atom that denotes the predicate that the assertion applies to, and the Pre and $Post$ are conjunctions of "`prop`" atoms, i.e., of atoms whose corresponding predicates are declared to be *properties* (Puebla et al. 2000a; Puebla et al. 2000b). Both Pre and $Post$ can be empty conjunctions (meaning true), in that case they can be omitted. The following example illustrates the basic concepts involved:

Example 1
These assertions describe different modes for calling a `length` predicate: either for `(1)` generating a list of length `N`, `(2)` to obtain the length of a list `L`, or `(3)` to check the length of a list:

```
1 :- pred length(L,N) : (var(L), int(N))  => list(L). %(1)
2 :- pred length(L,N) : (var(N), list(L)) => int(N).  %(2)
3 :- pred length(L,N) : (list(L), int(N)).            %(3)
4
5 :- prop list/1.    list([]).     list([_|T]) :- list(T).
```

Note also the definition of the `list/1` property (in this case a regular type) in line 5. Other properties (`int/1` –a base regular type, and `var/1` –a mode) are assumed to be loaded from the libraries (`native_props` in Ciao for these properties). □

The following definition relates a set of assertions for a predicate to the nodes which correspond to that predicate in the generalized AND tree for the current program P and initial set of queries $\mathcal{Q}$:

Definition 1 (*The Set of Assertion Conditions for a Predicate*)
Given a predicate represented by a normalized atom $Head$, and a corresponding set of assertions $\mathcal{A} = \{A_1 \ldots A_n\}$, with $A_i =$ "`:- pred` $Head$ `:` Pre_i `=>` $Post_i$`.`" the set of *assertion conditions* for $Head$ determined by $\mathcal{A}$ is $\{C_0, C_1, \ldots, C_n\}$, with:

$$C_i = \begin{cases} \texttt{calls}(Head, \bigvee_{j=1}^{n} Pre_j) & i = 0 \\ \texttt{success}(Head, Pre_i, Post_i) & i = 1..n \end{cases}$$

where `calls` (Head,Pre) states conditions on θ_c in all nodes $\langle L, \theta_c, \theta_s \rangle$ where $L \wedge Head$ holds, and `success` (Head,Pre,Post) refers to conditions on θ_s in all nodes $\langle L, \theta_c, \theta_s \rangle$where $L \wedge Head$ and $Pre \wedge \theta_c$ hold.

The assertion conditions for the assertions in the example above are:

$$\left\{ \begin{array}{llll} calls(& length(L,N), & ((var(L) \wedge int(N)) \vee (var(N) \wedge list(L)) \vee (list(L) \wedge int(N))), \\ success(& length(L,N), & (var(L) \wedge int(N)), & list(L)), \\ success(& length(L,N), & (var(N) \wedge int(L)), & int(N)), \end{array} \right\}$$

3 Abstract Code Search

In this section we propose the mechanism for defining abstract searches for predicates. Our objective now is not describing concrete predicates as before, but rather to state some desired semantic characteristics and perform a search over the set of predicates in some code P (our set of modules) looking for a subset of predicates meeting those characteristics. To this end we define the concept of *query assertions*, inspired by the *anonymous assertions* of (Stulova

et al. 2014). This requires extending our syntax so that in the normalized atoms that appear in the $Head$ positions of these assertions, the predicate symbol can be a variable from VS.

Definition 2 (*Query assertion*)
A query assertion is an expression of the form: `:- pred` L `:` Pre `=>` $Post$`.` where L is of the form $X(V_1, ..., V_n)$ and Pre and $Post$ are (optional) DNF formulas of prop literals.

We will use this concept to express conditions on the search. The intuition is that a query assertion is an assertion where the variable $X \in VS$ in the predicate symbol location of L will be instantiated during the search for code to predicate symbols from PS that comply with some query assertions. The following predicate defines the search:

Definition 3 (*Predicate query*)
A predicate query is of the form: `?- findp({` As `}, M:Pred/A, Residue, Status).` where:

- ***As*** is a set of query assertions, with the same arity and the same variable `Pred` as main functor of the different assertion $Head$s. This set can also include definitions of properties (e.g., regtypes or other `prop`s) used in the query assertions.
- **M:Pred/A** is a predicate descriptor, referring to a predicate `Pred` with arity `A` and defined in module **M** that corresponds to the information in the other arguments.
- **Residue** is a set of pairs of type $(condition, list(domain, status))$ which expresses the result of the proof of each condition in each domain. The status will be *checked* for those conditions that were proved to hold, in the given domain, *false* if they were proved not to hold, and *check* for conditions for which nothing could be proved.
- **Status** is the overall result of the proof for the whole set of conditions in the query assertion. It will be *checked* if all conditions are proved to be checked. It will be *false* if one condition is false. It will be *check* if neither *checked* nor *false* can be proved.

Predicate queries are our main means for conducting the semantic search for predicates. The query assertions and property definitions in As induce a series of *calls* and *success* assertion conditions (as per Def. 1) which are used to perform the filtering of candidate predicates. I.e., the `calls` conditions encode that the admissible calls of the matching predicates should be within the set of Pre conditions. The `success` conditions encode that, if Pre holds at the time of calling the matching predicate, and the execution succeeds, then the $Post$ conditions hold.

Example 2
Given code P, the predicate query:

```
1 ?- findp({ :- pred X(A,B) : (list(A), var(B)) => int(B). }, M:X/2, Residue, Status)}).
```

indicates that the user is looking for predicates $p \in P$ with $\mathsf{ar}(p) = 2$, such that on calls the first argument is instantiated to a list and the second is a free variable, and that, when called in this way, if p succeeds, its second argument will be instantiated to an integer. A predicate that could match this query is, for example, the `length/2` predicate of arity 2 defined in module `lists`. The call to `findp` would unify `M:X` to `lists:length`. If the match is not complete for some reason, there may be some conditions "left to prove" in `Residue`. `Status` will summarize the result. □

We now address how a predicate matches the conditions in a predicate query in the form of Def. 3. To this end we provide some definitions (adapted from (Puebla and Hermenegildo 1999; Puebla et al. 2000b)) which will be instrumental in order to connect the literals in query assertions to the results of analysis.

Definition 4 (*Trivial Success Set of a Property Formula*)
Given a conjunction L of properties and the definitions for each of these properties in P, we define the *trivial success set* of L in P as:

$$TS(L,P) = \{\bar{\exists}_L\theta \mid \exists\theta' \in answers(P,(L,\theta)) \text{ s.t. } \theta \models \theta'\}.$$

where $\bar{\exists}_L\theta$ denotes the projection of θ onto the variables of L. Intuitively, it is the set of constraints θ for which the literal $L\theta$ succeeds without adding new "relevant" constraints to θ (i.e., without constraining it further).

For example, given the following program P:

```
1 list([]).
2 list([_|T]) :- list(T).
```

and $L =$ `list(X)`, both $\theta_1 = \{X = [1,2]\}$ and $\theta_2 = \{X = [1,A]\}$ are in the trivial success set of L in P, but $\theta = \{X = [1|\_]\}$ is not, since a call to `(X = [1|_], list(X))` will instantiate the second argument of $[1|\_]$. We now define abstract counterparts for Def. 4:

Definition 5 (*Abstract Trivial Success Subset of a Property Formula*)
Given a conjunction L of properties, the definitions for each of these properties in P, and an abstract domain D_α, an abstract constraint or substitution $\lambda^-_{TS(L,P)} \in D_\alpha$ is an *abstract trivial success subset* of L in P iff $\gamma(\lambda^-_{TS(L,P)}) \subseteq TS(L,P)$.

Definition 6 (*Abstract Trivial Success Superset of a Property Formula*)
Under the same conditions of Def. 5 above, an abstract constraint or substitution $\lambda^+_{TS(L,P)}$ is an *abstract trivial success superset* of L in P iff $\gamma(\lambda^+_{TS(L,P)}) \supseteq TS(L,P)$.

I.e., $\lambda^-_{TS(L,P)}$ and $\lambda^+_{TS(L,P)}$ are, respectively, a safe under-approximation and a safe over-approximation of the trivial success set for the property formula L with definitions P.

We assume that the code P under consideration has been analyzed for an abstract domain D_α, for a set of queries $\mathcal{Q}$. Let $\mathcal{Q}_\alpha$ be the representation of those queries, i.e., it is the minimal element of D_α so that $\gamma(\mathcal{Q}_\alpha) \supseteq \mathcal{Q}$. We derive $\mathcal{Q}_\alpha$ from the code by including in it queries for all exported predicates, affected by the calls conditions of any assertions that appear in the code itself affecting such predicates (this is safe because if analysis is not able to prove them, they will be checked in any case via run-time checks). If no assertions appear in the code for a given exported predicate, the analyzer will assume $\top$ for the corresponding query.

We will now relate, using the concepts above, the abstract semantics inferred by analysis for this set of queries with the search process. As stated in Def. 1, a set of assertions denotes different types of conditions (calls and success). We provide the definitions for each type.

Definition 7 (*Checked Predicate Matches for a 'calls' Condition*)
A calls condition $\texttt{calls}(X(V_1,\ldots,V_n), Pre)$ is abstractly 'checked' for a predicate $p \in P$ w.r.t. Q_α in D_α iff $\forall\langle L,\lambda^c,\lambda^s\rangle \in analysis(P,D_\alpha,\mathcal{Q}_\alpha)$ *s.t.* $\exists\sigma \in ren,\ L = p(V'_1,\ldots,V'_n) = X(V_1,\ldots,V_n)\sigma, \lambda^c \sqsubseteq \lambda^-_{TS(Pre\ \sigma,P)}$.

Definition 8 (*False Predicate Matches for a 'calls' Condition*)
A calls condition $\texttt{calls}(X(V_1,\ldots,V_n), Pre)$ is abstractly 'false' for a predicate $p \in P$ w.r.t. Q_α in D_α iff $\forall\langle L,\lambda^c,\lambda^s\rangle \in analysis(P,D_\alpha,\mathcal{Q}_\alpha)$ *s.t.* $\exists\sigma \in ren,\ L = p(V'_1,\ldots,V'_n) = X(V_1,\ldots,V_n)\sigma, \lambda^c \sqcap \lambda^+_{TS(Pre\ \sigma,P)} = \bot$.

Note that in these definitions we do not use directly the *Pre* and *Post* conditions, although they already are abstract substitutions. This is because the properties in the conditions stated by the user in assertions might not exist as such in D_α. However, it is possible to compute safe approximations ($\lambda^-_{TS(Pre,P)}$ and $\lambda^+_{TS(Pre,P)}$) by running the analysis on the code of the

```
:- module(_, [my_length/2, get_length/2, check_length/2, gen_list/2], [assertions]).

:- pred my_length(L,N) : (list(L), var(N))  => int(N).
:- pred my_length(L,N) : (list(L), int(N)).
% :- calls length(L,N) : ((mshare(L), ground(N)) ; (mshare([[L],[L,N],[N]]), var(N)).
my_length(L,N) :- length(L,N).

:- pred check_length(L,N) : (list(L), int(N)).
% :- calls check_length(L,N) : (mshare(L), ground([N])).
check_length(L,N) :- length(L,N).

:- pred get_length(L,N) : (list(L), var(N)).
% :- calls get_length(L,N) : (mshare([[L],[L,N],[N]]), var(N)).
get_length(L,N) :- length(L,N).

:- pred gen_list(L,N) : (var(L), var(N)) => (list(L), int(N))
# "Generates a list of random elements of random size".
%:- calls gen_list(L,N) : (mshare([[L],[L,N],[N]]), var(L), var(N)).
gen_list(L,N) :- length(L,N).

% Implementation of length/2 ...
```

Fig. 1: Program with assertions that define different calls.

property definitions using D_α (or using the available trust assertions, for built-ins). The fact that the resulting approximations are safe ensures correctness of the procedure both when checking calls and success conditions.

Example 3
Several checks against a 'calls' condition. Consider the program in Fig. 1 and the classic sharing and freeness (`shfr`) abstract domain (Muthukumar and Hermenegildo 1991). This analysis will infer the calls substitutions written in comments in Fig. 1, where `var/1` and `ground/1` have the usual meaning and `mshare/1` describes variable sharing (intuitively, two variables are in the same list if they may share, singletons mean that there may also be other non-shared variables). Note that, while the `var/1` property is understood natively by the `shfr` analyzer, other properties that appear in the assertions (`list/1`, `int/1`, etc.) are not. However, they also imply groundness and freeness information. The analysis approximates this information to the `shfr` domain. In the case of built-ins such as `int/1` this is done using the associated (trust) assertions in the libraries. Thus, if an argument is stated to have the property integer on calls (i.e., it is bound to an integer at call time, as in `length` and `check_length`) it is expressed as a ground term in the `shfr` domain. In the case of properties that are defined by programs, such as `list/1`, the property definition itself is analyzed with the target domain (`shfr`). However, `shfr` cannot infer too much about `list/1` since it does not have a representation for "definitely non-var." Other modes domains may be able to infer "non-var but not necessarily ground."

Assume now that we would like to find predicates that generate tuples of lists and their size, i.e., the predicate has to accept a usage in which both of the arguments are free variables. This search can be expressed with the following predicate query:

```
?- findp({:- pred P(L, Size) : (var(L), var(Size)).}, M:P/A, Residue, Status).
```

The corresponding calls condition is: `calls`$(X(L, Size), (var(L), var(Size)))$. We discuss some interesting aspects of the search results:

- `gen_list/2`: This is a predicate of interest in the context of the predicate query because it expects both of its arguments to be variables, and they will be bound during the execution to what we might want (a list and an integer). Formally, the conditions are proved to hold for this predicate, because:
$$(\lambda^{-}_{TS((var(L),var(Size)),P)} = \{var(L), var(Size)\}) \sqsupseteq (\lambda^{c} = var(L), var(Size)).$$

- `check_length/2`: This is not a predicate of interest because its calling modes require both arguments to be instantiated. Formally, the condition is abstractly false for `check_length` because:
  $(\lambda^{+}_{TS((var(L),var(Size)),P)} = \{var(L), var(Size)\}) \sqcap (\{mshare(L), ground(Size)\} = \bot)$.
- Both `my_length/2` and `get_length/2` are predicates which do not match what we are looking for, because they require at least one argument to be instantiated. However, using only the `shfr` domain this cannot be proved (it would if the domain could represent `nonvar/1`, which would then be incompatible with `var/1`). The status for this condition for these predicates will be *check*, meaning that the finder could not infer information regarding those conditions for the predicate, but still the user might be interested in it. □

The point of filtering by calling modes is to avoid mixing behaviors. This can be interesting for example with predicates that, depending on the call, on success return in an argument either a free variable or an instantiated term. Consider an (admittedly not very nice) predicate `read_line(Line, Size)` such that if a line is correctly read, its size will be `Size` and if not, `Size` will be a free variable. Assume that we would like instead an error to be displayed if the line is not correctly read. Then, we need a predicate that requires `Size` to be an integer. `check_length` is a relevant predicate then (and can be combined with `read_line/2` as: `read_line(Line, Size), check_length(Line, Size).`). In this case `length` is not useful, since it accepts the second argument as a free variable.

Similarly to what we did for `calls` conditions, we provide definitions for stating whether a predicate matches for a given `success` condition and when it does not:

Definition 9 (*Checked Predicate Matches for a 'success' Condition*)
A success condition $\texttt{success}(X(V_1,\ldots,V_n), Pre, Post)$ is abstractly 'checked' for predicate $p \in P$ w.r.t. Q_α in D_α iff $\exists L = p(V'_1,...,V'_n)$ s.t. $\forall \langle L, \lambda^c, \lambda^s\rangle \in analysis(P, Q_\alpha)$ *s.t.* $\exists \sigma \in ren,\ L = p(V'_1,\ldots,V'_n) = X(V_1,\ldots,V_n)\sigma, \lambda^c \sqsupseteq \lambda^{+}_{TS(Pre\ \sigma,P)} \rightarrow \lambda^s \sqsubseteq \lambda^{-}_{TS(Post\ \sigma,P)}$

Definition 10 (*False Predicate Matches for a 'success' Condition*)
A success condition $\texttt{success}(X(V_1,\ldots,V_n), Pre, Post)$ is abstractly false for $p \in P$ w.r.t. Q_α in D_α iff $\exists L = p(V'_1,\ldots,V'_n)$ s.t. $\forall \langle L, \lambda^c, \lambda^s\rangle \in analysis(P, Q_\alpha)$ *s.t.* $\exists \sigma \in ren,\ L = p(V'_1,\ldots,V'_n) = X(V_1,\ldots,V_n)\sigma, \lambda^c \sqsubseteq \lambda^{-}_{TS(Pre\ \sigma,P)} \wedge (\lambda^s \sqcap \lambda^{+}_{TS(Post\ \sigma,P)} = \bot)$

Example 4
Several checks against a 'success' condition. Assume that we analyze the module in Fig. 2 with a shape abstract domain D_α. Originally, the code had no assertions, so the analysis was performed for any possible entry. The inferred information is shown in comments (omitting calls for simplicity). The generated lattice for the abstract elements is shown in Fig. 2a.

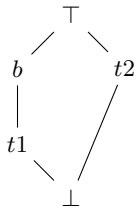
The regular type b was included in the program and types $t1$ and $t2$ were inferred by the analyzer. Suppose that we execute the query:

```
?- findp({:- pred P(V) : term(V) => b(V).}, M:P/A, Residue, Status).
```

The `success` condition of this query is $C = \texttt{success}(X(V), \text{term}(V), b(V))$. We discuss how the predicates match this condition:

- `simple:perfect/1`. This predicate behaves exactly as specified in the predicate query, because on success it produces an output of the same type as specified. Formally, the analysis infers $\langle perfect(V), \top(V), b(V)\rangle$ and $\lambda^{-}_{TS(b,P)} = b$ (trivially). Then, $\lambda^s \sqsubseteq \lambda^{-}_{TS(b,P)}$, because $b \sqsubseteq b$.

```
1  :- module(simple, _, [assertions, regtypes]).
2
3  %:- true pred perfect(A) => b(A).   :- true pred mixed(X) => top(X).
4  perfect(b1).                        mixed(b0).
5  perfect(b0).                        mixed(b1).
6                                      mixed(z).
7  %:- true pred reduced(A) => t1(A).
8  reduced(b1).                        %:- true pred hard(X) => top(X).
9                                      hard(X) :- functor(b1(_), X, _).
10 %:- true pred outb(A) => t2(A).
11 outb(z).
12
13 :- regtype b/1.   %:- regtype t1/1   :- regtype t2/1.
14 b(b0).           %t1(b1).           t2(z).
15 b(b1).
```



(a) Inferred lattice.

Fig. 2: A simple program analyzed.

- `simple:reduced/1`. Intuitively, this predicate does not match as well as `perfect` but all possible outputs are within $\gamma(b)$, therefore, it is a valid predicate. Formally, the analysis infers $\langle reduced(V), \top(V), t1(V)\rangle$, and $\lambda^{-}_{TS(b,P)} = b$ (trivially). As $t1 \sqsubseteq b$, i.e., $t1 \Rightarrow b$, this predicate meets the condition of Def. 9 to be checked.
- `simple:outb/1`. This predicate is of no use, because its output (z) is completely different from the one in the query (b). Formally, the analysis infers $\langle outb(V), \top(V), t2(V)\rangle$ and $\lambda^{-}_{TS(b,P)} = b$ so the conditions of the definition hold: $\lambda^{c} \sqsubseteq \lambda^{-}_{TS(Pre,P)}$ holds because $(\lambda^{c} = \top) \sqsubseteq (\lambda^{-}_{TS(term,P)} = \top)$ and $(\lambda^{s} \sqcap \lambda^{+}_{TS(Post,P)} = \bot)$ holds because $(\lambda^{s} = t2) \sqcap (\lambda^{+}_{TS(b,P)} = b) = \bot$.

Some predicates do not match any of the two statuses presented so far (checked or false conditions). This may be due to two reasons. The first is that the predicate may actually behave in such a way that the conditions in the query are really not checked or false. The second one is that the abstract domain may not provide accurate enough information to prove whether the conditions hold or not.

Intuitively, predicate `simple:mixed/1` is not what we are looking for because, although its possible outputs can be of type b, it can produce also type $t2$. Formally, the condition cannot be proved to hold or not, since the analysis inferred $\langle mixed(V), \top(V), \top(V)\rangle$:

- It cannot be checked, because the output type is more general than specified, and therefore it does not satisfy the condition in Def. 9: $(\lambda^{c} = \top) \sqsupseteq (\lambda^{+}_{TS(Pre,P)}) \rightarrow (\lambda^{s} = \top) \sqsubseteq (\lambda^{-}_{TS(b,P)} = b)$ (true $\rightarrow$ false).
- It is also not false because some of the outputs are the ones required in the specification. Formally, it does not satisfy the second condition of Def. 10: $(\lambda^{s} = \top) \sqcap (\lambda^{+}_{TS(b,P)} = b) = b \neq \bot$.

We now show how an abstract domain may not be precise enough to find all matching predicates. Intuitively, the success condition of the example should hold for `hard/1` because its output type is more restrictive than specified. However, the analyzer cannot infer that its output will be always `b0` because `functor/3` can produce any atom, and thus the inferred tuple will be $\langle hard(V), \top(V), \top(V)\rangle$. The reasoning process to set the status of proof of this condition as check is the same as with `mixed/1`. □

Combining information from different domains: Sometimes the information inferred using an abstract domain is not accurate enough to prove whether a condition holds or not but the information in another domain is. It depends on how the user expresses the query, and how accurately the abstract properties of the query can be approximated in each domain. An example is when properties of different domains are used in a query. For example: in

`:- pred X(A,B) : (list(A), var(B))`, the property `var(X)` cannot be represented in the regular types domain, so it will assume $\top$ for `B` which will lead to not being able to check it.

Combining domains is a useful technique to increase accuracy. An assertion condition is proved to hold (status checked) or not (status false) if the result can be proved in any analysis domain. The reason for this is the correctness of the analysis, which always computes safe approximations. This ensures that properties proved in each domain separately for the same set of queries cannot be contradictory. At most, if a property can be proved in a domain, other domains may not be accurate enough to decide that the property holds. Summarizing, the status of a condition given its proof status for a set of domains will be:

$$Status = \begin{cases} false & \text{if proved false in at least one domain} \\ checked & \text{if proved checked in at least one domain} \\ check & \text{otherwise} \end{cases}$$

Example 5

Assume the program in Fig. 1 and the analysis in Ex. 3, but that an analysis of regtypes is also performed (e.g., (Gallagher and de Waal 1994) or (Vaucheret and Bueno 2002)):

# Tuple	Predicate	λ^c (regtypes)	λ^c (shfr)
1	$gen\_list(L,N)$	$(term(L), term(N))$	$(mshare([[L],[L,N],[N]]), var(L), var(N))$
2	$get\_length(L,N)$	$(list(L), term(N))$	$(mshare([[L],[L,N],[N]]), var(N))$
3	$check\_length(L,N)$	$(list(L), int(N))$	$(mshare(L), ground([N]))$
4	$my\_length(L,N)$	$(list(L), term(N))$	$(mshare(L), ground(N))$
5	$my\_length(L,N)$	$(list(L), int(N))$	$(mshare([[L],[L,N],[N]]), var(N))$

The combination of both domains is really useful for proving certain conditions because they complement each other. Assume that we want to find predicate that checks the length of a list. The condition to be satisfied is $\texttt{calls}(X(L,Size), (list(L), num(Size)))$. According to the definitions of matching, the results in each domain will be:

PredName/A	regtypes proof	shfr proof	combined proof (Sum)
gen_list/2	check	false	false
get_length/2	check	false	false
check_length/2	checked	check	checked
my_length/2	check	check	check

The intuitive explanation of these results is:

- `gen_list`: In the `regtypes` domain this condition cannot be proved because the domain has no information about `var`. However, in the `shfr` domain it can be proved that the condition does not hold because it requires both arguments to be non-free variables, and the calling mode does the opposite. Then, that condition is false for this predicate.
- `get_length`: This case is similar to `gen_list`. It cannot be proved in the types domain because one argument was specified with instantiation information but it can be proved in the modes domain that is false.
- `check_length/2`: Matches the condition in the types domain, because the types are exactly the ones we were looking for. For this predicate, the mode domain is not useful because the information in the program assertions was specified with types only.
- `my_length/2`: At first sight this predicate matches the query because there is one calling mode that matches exactly as stated in the condition. However, according to the definition of calls condition, all admisible calling modes must be within the condition, and there is one calling mode that does not comply: the mode for calculating the length of the list. □

4 Prototype and evaluation

We have developed and evaluated a prototype implementation on top of the Ciao/CiaoPP system. The system implements both the pre-analysis of the code base and the user-level predicate matching search facilities, using the pre-computed analysis results. As mentioned in Section 2, by default modules are analyzed individually and the analysis trusts the assertions for imported predicates and the calls for exported predicates. However, modular analysis can also be used, as discussed later. The analysis results are cached on disk (as CiaoPP *dump* files) and reused while searching. Each time the search is performed in a module, its corresponding analysis dump is restored or it is reanalyzed with the abstractions of the constraints in the query, and conditions are checked. The algorithms that implement condition checking are described in Appendix B.

Searching with the prototype. To demonstrate some of the potential of our approach, let us consider looking for code that operates with graphs in the Ciao libraries. First we need to guess how graphs may be represented, i.e., their shape. Here there are two possible guesses:

```
% mathematical definition structure
:- regtype math_graph(Graph).
math_graph(graph(Vertices,Edges)):-
    list(Vertices),
    list(Edges, pair).

:- regtype pair/1.
pair((_,_)).
```

```
% adjacency list structure
:- regtype al_graph(_).
al_graph(A) :-
    list(A,al_graph_elem).

:- regtype al_graph_elem/1.
al_graph_elem(Vertex-Neighbors) :-
    list(Neighbors).
```

where `math_graph` is based on the mathematical definition: an ordered pair (V, E) comprising a set V of vertices, together with a set E of edges, which are 2-element subsets of V. The `al_graph` property captures an alternative representation for graphs as a list of vertices and their corresponding neighbors.

A query assertion for finding code using the first representation could be `:- pred P(X,Y) => math_graph(Y)`.[3] The prototype finds `complete_graph/2` and `cycle_graph/2` in module `named_graphs.pl` (see Fig.A 1) by matching this query against a generic analysis of the module. Note that this code is found although the module `named_graphs.pl` has no assertions nor shape definitions, i.e., it only contains plain Prolog code.

Let us search with the adjacency list representation. We want to find code to modify existing graphs, i.e., predicates that take as an input a graph and a list of elements and produce a new graph: `:- pred P(A,B,C) : (al_graph(A),list(B),var(C)) => al_graph(C)`. No code is found to hold these properties. To find out why, we extract the conditions from the query assertion:

$C_1 = \texttt{calls}(P(A,B,C), (al\_graph(A), list(B), var(C)))$ and

$C_2 = \texttt{success}(P(A,B,C), (al\_graph(A), list(B), var(C)), al\_graph(C))$,

`calls` conditions can not be checked if the code to be found lacks hand-written calls assertions. Therefore, we will focus on finding predicates that hold C_2. As the conditions of the calls substitution are very specific, a generic analysis may be not accurate enough to prove these conditions. We refine the predicate matching by executing predicates abstractly with the calls substitution in the success condition. To ensure greater precision, we perform also an inter-modular analysis, i.e., we analyze both modules main and imported, at the same time, so no information is lost. Thus, the prototype finds that in `add_vertices/3`, `del_vertices/3`, `add_edges/3`, and `del_edges/3` (see Fig. A 2) the `success` condition holds.

[3] As mentioned before, the user-defined shapes (or any other properties), in this case the regtypes above, must be included within queries. However, we just write the query assertion for brevity.

Ar\Cnds	1	1 (AVG)	2	2 (AVG)	3	3 (AVG)	4	4 (AVG)
1 (85 pr)	19,064	224	53,530	630	180,246	2,121	298,292	3,509
2 (74 pr)	110,092	1,488	207,871	2,809	221,061	2,987	477,440	6,452
3 (47 pr)	294,962	6,276	3,757,208	79,941	3,806,917	80,998	6,127,015	130,362
4 (12 pr)	5,116	426	12,939	1,078	22,508	1,876	30,300	2,525

Table 1: Assertion Checking times (μs).

Performance results. To measure the effectiveness and performance of the approach, we have set up an experiment that consists in analyzing a large part of the Ciao libraries and finding matching predicates of arity 1 to 4 and several assertion conditions. The experiments are run on a Linux server (Intel Xeon CPU E7450, 2.40GHz) with 16GB of RAM. We have selected 63 modules from the Ciao libraries to test the search, excluding those that could not be analyzed under a 1 minute timeout. The detailed analysis statistics are shown in Appendix C. The selection includes modules that are costly for the analyzer and others where the analysis is trivial (e.g., non-analyzable foreign code with trusted assertions) but useful for the search. Pre-analysis took $45s$ to analyze all of them ($660ms$ on average), and they required $3.5MB$ of disk space to be stored ($55.5KB$ on average). Restoring the analysis results each query (the 63 modules) takes $21.5s$ ($343ms$ on average). Note that the size of the cached analysis is small and could be kept in memory for subsequent queries.[4] The performance of checking, once the analysis results are available, depends on the arity, the number of predicates available with that arity, and the conditions specified in the query. The detailed timing results are shown in Table 1. Columns represent the number of assertion conditions of each query and rows their arity (in parenthesis the number of predicates present in the code with that arity). Cells represent the execution time needed to exhaustively check the predicates in the 63 modules. The *(AVG)* column represents the average time per predicate. It can take from $224\mu s$ (1 condition, 1 argument) to $130ms$ (4 conditions, 3 arguments). Summarizing, it takes, on average, $25s$ to execute a query, looking in all 63 modules, $21.5s$ of which are spent for restoring.

5 Conclusions

Motivated by the large size and number of the code bases available nowadays to programmers, we have proposed a novel approach to the code search problem based on querying for *semantic* characteristics of the programs. In order to obtain such characteristics automatically, program units are pre-processed using abstract interpretation-based static analysis techniques to obtaining safe semantic approximations. We have also proposed a novel, assertion-based code query language for expressing the desired semantic characteristics of the code as partial specifications. We have shown how relevant code can be found by comparing such partial specifications with the inferred semantics for each program element. Our approach is fully automatic and does not rely on user annotations or documentation. We have also reported on a prototype implementation that provides evidence that both the analysis and search are efficient despite the relatively naive implementation. These results are encouraging regarding the overall practicality of the approach. While we have prototyped our approach within the Ciao system, we argue that the techniques are quite general and they are easily extensible to other languages. However, the Ciao system does offer a number of advantages in this application, such as having a strict module system despite being a

[4] This feature was not implemented in the current version of the prototype.

"dynamic" language. This allows obtaining results that at the same time are more accurate and can be guaranteed to be correct. We believe that the proposed approach has a number of additional applications, such as, for example, detection of duplicated code.

References

Bruynooghe, M. 1991. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming 10*, 91–124.

Bueno, F., de la Banda, M. G., Hermenegildo, M., Marriott, K., Puebla, G., and Stuckey, P. 2001. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*. Number 2042 in LNCS. Springer-Verlag, 86–102.

Cabeza, D. and Hermenegildo, M. 2000. A New Module System for Prolog. In *International Conference CL 2000*. LNAI, vol. 1861. Springer-Verlag, 131–148.

Cousot, P. and Cousot, R. 1977. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*. ACM Press, 238–252.

Gallagher, J. and de Waal, D. 1994. Fast and Precise Regular Approximations of Logic Programs. In *Proc. of ICLP'94*. MIT Press, 599–613.

Hermenegildo, M., Puebla, G., Bueno, F., and Lopez-Garcia, P. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2 (October), 115–140.

Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., and Puebla, G. 2012. An Overview of Ciao and its Design Philosophy. *TPLP 12,* 1–2, 219–252. http://arxiv.org/abs/1102.5497.

Maarek, Y. S., Berry, D. M., and Kaiser, G. E. 1991. An information retrieval approach for automatically constructing software libraries. *Software Engineering, IEEE Transactions on 17,* 8, 800–813.

McMillan, C., Hariri, N., Poshyvanyk, D., Cleland-Huang, J., and Mobasher, B. 2012. Recommending source code for use in rapid software prototypes. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 848–858.

Mitchell, N. 2008. Hoogle overview. *The Monad.Reader* 12 (November), 27–35.

Muthukumar, K. and Hermenegildo, M. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*. MIT Press, 49–63.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming 13,* 2/3 (July), 315–347.

Puebla, G., Bueno, F., and Hermenegildo, M. 2000a. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer-Verlag, 23–61.

Puebla, G., Bueno, F., and Hermenegildo, M. 2000b. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*. Number 1817 in LNCS. Springer-Verlag, 273–292.

Puebla, G. and Hermenegildo, M. 1999. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs 41,* 2&3 (November), 279–316.

Reiss, S. P. 2009. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 243–253.

Rollins, E. J. and Wing, J. M. 1991. Specifications as search keys for software libraries. In *ICLP*. Citeseer, 173–187.

Stulova, N., Morales, J. F., and Hermenegildo, M. V. 2014. Assertion-based Debugging of Higher-Order (C)LP Programs. In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM Press.

Vaucheret, C. and Bueno, F. 2002. More Precise yet Efficient Type Inference for Logic Programs. In *SAS'02*. Number 2477 in LNCS. Springer, 102–116.

These appendices are proposed as the supplementary, on-line material that would be published together with the final version if the paper is accepted. They are also intended for reference and to make the paper more self-sufficient.

Appendix A Example code

Sample code found with `math_graph` structure:

```
:- module(named_graphs, [complete_graph/2, cycle_graph/2], []).

:- use_module(library(lists), [append/3]).

complete_graph(N, graph(V,E)) :-
        count(N, V),
        generate_complete_edges(V, E).

generate_complete_edges(V, E) :-
        generate_complete_edges_(V, V, E).

generate_complete_edges_([], _, []).
generate_complete_edges_([V|Vs], AllV, E) :-
        generate_complete_edges_for_vertex(V, AllV, E1),
        append(E1, RestE, E),
        generate_complete_edges_(Vs, AllV, RestE).

generate_complete_edges_for_vertex(_, [], []) :- !.
generate_complete_edges_for_vertex(V, [V|Vs], E) :- !,
        generate_complete_edges_for_vertex(V, Vs, E).
generate_complete_edges_for_vertex(V, [V1|Vs], [(V, V1)|E]) :-
        generate_complete_edges_for_vertex(V, Vs, E).

cycle_graph(N, graph(V,E)) :-
        N = 2, !,
        V = [1,2],
        E = [(1,2),(2,1)].
cycle_graph(N, graph(V,E)) :-
        N > 1,
        count(N, V),
        generate_cycle_edges(V, E).

generate_cycle_edges([V1], [(V1, 1)]) :- !.
generate_cycle_edges([V1, V2|Vs], [(V1, V2)|Edges]) :-
        generate_cycle_edges([V2|Vs], Edges).

count(N, Lst) :-
        count_(1, N, Lst).
count_(I, N, []) :-
        I > N, !.
count_(I, N, [I|L]) :-
        I1 is I+1,
        count_(I1, N, L).
```

Fig. A 1: `named_graphs.pl` (Ciao library)

Sample code found with `al_graph` structure:

```
:- module(ugraphs, [add_vertices/3], [assertions,isomodes] ).

:- use_module(library(sets), [ord_union/3]).
:- use_module(library(sort), [sort/2]).

:- pred add_vertices(+Graph1, +Vertices, -Graph2)
#  "Is true if @var{Graph2} is @var{Graph1} with @var{Vertices} added to it.".
add_vertices(Graph0, Vs0, Graph) :-
        sort(Vs0, Vs),
        Vs = Vs0,
        vertex_units(Vs, Graph1),
        graph_union(Graph0, Graph1, Graph).
% ...
```

Fig. A 2: Fragment from `ugraphs.pl` (Ciao library).

Appendix B Algorithms for predicate matching

The algorithms presented in this section are used to decide whether a predicate is proven to match a condition (that condition is checked or false) or that it cannot say anything about that property holding (check).

Algorithm 1 Matching Status of a calls condition for a predicate p

Input: $Analysis(P, D_\alpha, \mathcal{Q}_\alpha)$, $p \in exported(P)$, $C = \texttt{calls}(H, (Pre_1; ...; Pre_n))$
Output: Status of proof

if $\forall \langle H, \lambda^c, \lambda^s \rangle \in Analysis$ *s.t.* $H = p(X_1, ..., X_n), \bigvee_i \lambda^-_{TS(Pre_i,P)} \sqsupseteq \lambda^c$ **then**
    Status = **Checked**
else if $\forall \langle H, \lambda^c, \lambda^s \rangle \in Analysis$ *s.t.* $H = p(X_1, ..., X_n), \bigvee_i \lambda^-_{TS(Pre_i,P)} \sqcap \lambda^c = \bot$ **then**
    Status = **False**
else
    Status = **Check**
end if

Algorithm 2 Matching Status of a success condition for a predicate p

Input: $Analysis(P, D_\alpha, \mathcal{Q}_\alpha)$, $p \in P$, $C = \texttt{success}(H, Pre, Post)$
Output: Status of proof

if $\exists\ \langle H, \lambda^c, \lambda^s \rangle \in Analysis$ s.t. $H = p(X_1, ..., X_n), \lambda^c = \lambda^+_{TS(Pre,P)}$ **then**
    if $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$ **then**
        Status = **Checked**
    else if $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \bot$ **then**
        Status = **False**
    else
        Status = **Check**, analysis accurate enough
    end if
else if $\exists\ \langle H, \lambda^c, \lambda^s \rangle \in Analysis$ s.t. $H = p(X_1, ..., X_n), \lambda^c \sqsupset \lambda^+_{TS(Pre,P)}$ **then**
    if $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$ **then**
        Status = **Checked**
    else if $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \bot$ **then**
        Status = **False**
    else
        Status = **Check**, Refine analysis
    end if
else
    Status = **Check**, No information for that calls, Refine analysis
end if

Appendix C Additional tables

Table C 1: Analysis statistics from core/lib modules: time(ms) and memory(B) consumption.

Module name	**load time**	**regtype ana time**	**regtype global stack mem**	**shfr ana time**	**shfr global stack mem**	**total analysis time**
dict	480	20	669,312	3,712	772,472	3,732
sets	548	116	1,462,696	1,512	1,923,720	1,628
assrt_write	760	172	1,404,136	1,240	420,392	1,412
sort	544	184	877,104	992	222,288	1,176
optparse_tr	744	32	814,168	1,068	950,272	1,100
translation	516	108	3,415,632	564	471,456	672
exsteps	664	28	990,872	296	1,186,552	324
assrt_write0	724	80	1,030,808	96	271,688	176
assrt_lib_extra	724	108	1,103,072	48	314,680	156
term_list	488	72	867,120	24	160,944	96
civil_registry	508	76	554,032	16	621,504	92
assertions_props	556	44	1,385,760	40	1,722,832	84
pl2wam_tables	484	40	2,887,440	32	3,086,248	72
embedded_tr	832	24	680,736	44	792,152	68
terms	528	44	571,912	12	653,248	56
ceval1	496	48	522,152	4	582,896	52
unittest_base	516	36	630,600	16	740,344	52
ceval2	528	44	522,320	4	583,064	48
errhandle	540	28	620,024	16	747,456	44
goal_trans	484	32	582,088	12	673,952	44
llists	480	24	526,384	12	592,568	36
file_utils	564	20	641,728	12	758,024	32
foreign_compilation	584	28	541,280	4	618,072	32
qsort	484	24	483,552	8	528,312	32
srcdbg	720	4	2,540,064	28	2,570,376	32
meta_props	500	24	496,800	4	535,128	28
strings	532	16	693,304	12	809,928	28
libpaths	552	16	439,304	8	475,040	24
metatypes_tr	468	24	439,216	0	477,136	24
attr_bench	796	16	2,641,584	4	2,737,680	20
between	472	16	438,144	4	467,216	20
iso_char	496	12	599,032	8	679,024	20
length	540	20	437,240	0	456,896	20
phrase_test	512	8	556,144	8	644,392	16
optparse_rt	488	4	456,144	8	501,440	12
relationships	532	8	479,552	4	519,952	12
res_exectime_rt	632	8	480,568	4	495,480	12
resources_tr	476	8	419,248	4	444,992	12
resources_types	484	8	483,864	4	534,696	12
streams	532	8	468,608	4	518,952	12
ttyout	500	8	450,688	4	498,456	12
bundle_params	484	4	2,476,504	4	2,499,576	8
ctrlcclean	524	8	396,168	0	428,456	8
miscprops	460	4	448,888	4	480,056	8
odd	488	4	407,320	4	421,968	8

(continued in next page)

Table C 1: Analysis statistics from core/lib modules: time(ms) and memory(B) consumption. *(continued).*

Module name	**load time**	**regtype ana time**	**regtype global stack mem**	**shfr ana time**	**shfr global stack mem**	**total analysis time**
old_database	492	4	494,040	4	529,896	8
pretty_names	488	4	416,456	4	432,328	8
dict_types	512	4	413,912	0	439,584	4
fastrw	512	0	447,968	4	471,416	4
prf_ticks_rt	636	0	506,008	4	520,416	4
res_nargs_res	524	0	393,080	4	407,560	4
test1	500	4	367,368	0	382,456	4
test4	520	4	372,968	0	384,120	4
assrt_synchk	496	0	375,848	0	384,968	0
c_itf_props	480	0	414,208	0	435,624	0
compressed_bytecode	500	0	367,288	0	376,488	0
doc_flags	512	0	426,312	0	455,768	0
doc_props	520	0	366,272	0	375,344	0
regtypes_tr	484	0	415,608	0	434,712	0
res_litinfo	528	0	498,792	0	526,104	0
runtime_ops_tr	460	0	375,136	0	389,048	0
test2	488	0	367,704	0	382,864	0
unittest_examples	472	0	384,632	0	396,216	0
TOTAL (63)	34,088	1,680	47,436,912	9,924	44,316,888	11,604
AVG	541.1	26.7	752,966.9	157.5	703,442.7	184.2

Table C 2: Analysis dump files statistics from core/lib modules.

Module name	**dump size(B)**	**restore time(s)**
assrt_write	566,132	2,440
sort	524,490	1,772
translation	314,227	1,652
assrt_write0	142,058	1,228
assrt_lib_extra	138,689	1,132
assertions_props	142,057	1,084
sets	212,735	1,028
exsteps	257,632	916
term_list	103,583	780
errhandle	51,222	640
attr_bench	47,920	548
terms	59,107	536
phrase_test	37,034	516
file_utils	50,810	484
embedded_tr	80,129	440
strings	36,279	400
optparse_tr	136,929	384
unittest_base	46,705	356
civil_registry	30,588	328
dict	106,704	308

(continued in next page)

Table C 2: Analysis dump files statistics from core/lib modules.

Module name	**dump size(B)**	**restore time(s)**
iso_char	26,702	300
foreign_compilation	23,623	276
goal_trans	44,771	276
llists	22,500	260
ceval2	24,122	248
ceval1	21,995	232
qsort	17,867	200
pl2wam_tables	17,649	184
ttyout	9,631	164
streams	12,383	160
metatypes_tr	12,959	156
meta_props	19,571	148
libpaths	11,676	124
old_database	13,462	112
dict_types	6,807	108
relationships	7,951	108
between	11,756	100
fastrw	6,754	100
ctrlcclean	7,474	96
srcdbg	31,936	92
miscprops	6,056	88
doc_flags	4,678	84
optparse_rt	8,713	84
res_litinfo	6,662	80
resources_tr	8,358	80
bundle_params	6,528	72
c_itf_props	2,474	68
resources_types	3,864	64
length	4,503	60
test2	1,519	52
test1	1,517	48
odd	2,498	44
pretty_names	4,875	44
prf_ticks_rt	2,271	44
res_exectime_rt	2,676	44
runtime_ops_tr	3,204	40
res_nargs_res	2,646	36
compressed_bytecode	782	32
regtypes_tr	884	28
test4	218	24
unittest_examples	58	24
assrt_synchk	58	20
doc_props	396	20
TOTAL (63)	3,512,057	21,596
AVG	55,747	343

Responses on How the Reviewer Comments have been Addressed

This appendix includes our detailed answers to clarify how all the comments from the reviewers have been addressed in this revised version of the paper.

Reviewer 1

Comment: *The authors propose a novel method for searching in a code base a predicate (function/method ...) a desired functionality: the search is based on information about the predicate derived by abstract interpretation. The code base is pre-analysed and then as many queries as desired can be asked against the derived information. The query needs to be put in a format that reflects the abstract domains, so the programmer needs to be familiar with the relevant domains - which is a bit of drawback of the method, but not a complete disaster.*

Answer: This is indeed true to a large extent, but the situation can also be slightly better than it may appear: it is true that the algorithm depends on having domains that are relevant to the properties appearing in the query. And indeed the best results are obtained if the user is familiar with the available domains and uses "native" properties of these domains in the queries (fortunately these properties are documented in the system). However our approach (and in general CiaoPP) does not strictly require from the user this precise knowledge or even the use of the "native" domain properties. In fact, the programmer can write his/her own properties –as "pure" Prolog programs using a few native properties, like variable sharing or unification. These definitions are interpreted by the abstract domains by analyzing the code defining the properties: this process "translates" the properties to the abstract domain. The selection of the domains can be fully automatic, based on heuristics.

Note that the system still works with concrete properties that may not be represented precisely in the domain, thanks to the capacity of the system to perform and deal with over/under-approximations. Of course, in the worst case, the user-defined properties may abstracted as the "top" or "bottom" (if we are over or under-approximating) element in the abstract domain to ensure correctness.

We try to explain this in section 3 after the calls matching definitions:

"In these definitions we do not use directly the Pre and $Post$ conditions, although they already are abstract substitutions. This is because the properties in the conditions stated by the user in assertions might not exist as such in D_α. However, it is possible to compute safe approximations ($\lambda^-_{TS(Pre,P)}$ and $\lambda^+_{TS(Pre,P)}$) by running the analysis on the code of the property definitions using D_α (or using the available trust assertions, for built-ins)."

Comment: *The method is benchmarked for its efficiency. The effectiveness (whether indeed relevant information is obtained through this mechanism) is not discussed. That is a pity, but perhaps difficult to do. The method is obviously robust for name changes (predicates, variables ...) and might be better than other methods that are not based on this idea. It could be useful in other contexts (beyond Prolog, beyond a user searching a code base). On the whole, I like the idea, and lots of work must have been put in its implementation, as well as in the letting queries work with more than one domain.*

Answer: Thanks for your positive feedback. Indeed, our aim in this paper has been to present and develop the idea, implement it, and test its practicality from the analysis point of view, and on a relatively large code base (where efficiency was a priori one of our main concerns). In this sense we believe the results are quite encouraging.

With this enabling point settled, we have also been able to check that the system provides interesting results to our queries when working with it, finding code that would be very difficult to find with other approaches (e.g., when there are no comments, etc.). To address this better in the paper, we have added some of these examples and discussed them in some depth in this new version. This hopefully adds to the understanding of the precision/effectiveness of the approach.

A larger effectiveness study would indeed be interesting, but also quite complicated, as the reviewer realizes, and we feel the results provided are quite interesting by themselves and make for a significant contribution.

Comment: *The paper does not do a good job at reporting about the method or system, in the following sense: the initial idea is quite easy to grasp, but the paper then goes into too much detail about abstract interpretation: this cuts the appetite to read on; in fact, someone unfamiliar with AI will not learn anything, and the others will have seen most of it already.*

Answer: In general, we have tried to find a reasonable balance between a bit conflicting requirements from two reviewers: to provide more or less information about Abstract Interpretation. Note that most of the required abstract intrepretation concepts are in the "Inferring the Program Semantics by Abstract interpretation" paragraphs. It is a shallow introduction to abstract interpretaion and the actual algorithms implemented in the CiaoPP framework, but provides pointers for readers interested in the inner details. The rest of section 2 and section 3 deal with specification for Prolog programs (as Ciao assertions) and a precise description of how abstract interpretation results are used. We believe that this part is essential to make our results reproducible. Nevertheless, we have included in this version more examples and their explanation, in the hope that this material makes the paper more attractive to readers not familiar with abstract interpretation.

Comment: *The timings are relevant, but not the most important issue here: is this method better than other methods at providing programmers what they need ?*

Answer: Again, this is a very good question. We addressed above the issue of timing vs. effectiveness. Regarding the comparison with other methods, our first-level answer is that we feel that this technique is not necessarily a replacement for other methods, specially keywords-based search, but is rather complementary (or a generalization, if we consider that

the set of keywords appearing in the text of a program can be turned into an abstract domain). Regarding signature matching, our approach is also a generalization. In this regard, in this revised version we have hopefully succeeded with the new examples in demonstrating that we can handle realistic queries that are not possible to formulate in keyword-based or signature based searches (specially in the context of untyped languages). As also mentioned before, we agree that the final answer on whether this approach would be a critical tool for programmers would need a user study, but this is also quite complicated and requires significant infrastructure, as the reviewer realizes. As future work we have been thinking we could base that study on a public search service (at least for the Ciao libraries). But, again, we feel that the results provided are already quite interesting by themselves.

Comment: *Is the method robust for argument order change? Is the method robust for varying arity? (this might sound weird, but the required functioniality might be present in a subset of an existing predicate, or one predicate definition might have packed two arguments into one [difference list usage implemented with 2 arguments versus using -/2 for instance]).*

Answer: We believe there is nothing in the method that prevents supporting this and other extensions. In our view, they fall into the more general issue of considering matches for *transformed* versions of the predicates. This is not addressed directly in this work, but we feel it is straightforward to implement on top of our framework. That is, the search does not necessarily need to be against the exact versions of the predicates in the program, but matching can also be seen as being done against different transformations of these predicates that, e.g., swap arguments as needed (`p(X,Y,Z) :- q(Y,X,Z)`). This conceptual view would cover not only argument order changes but also partial evaluation/slicing (including argument subsets), changes in data representations, etc., etc. (i.e., "refactorings," in modern terms). The implementation itself does not need to really perform the transformations explicitly (i.e., this can be implemented directly in the matching), but we believe that viewing the overall process in this way (i.e., as partial evaluations / program transformations) provides a sound formal basis for the extensions. This extended search would obviously imply some extra cost. In any case, implementing these extensions and evaluating their cost is in our plans for the system.

Comment: *Remarks on page 13, table 1 and the benchmarks: you should really give more information on the kind of queries (give some examples) and more importantly, about the success/failure rates for the queries, first in terms of that the system has computed, and secondly, also in terms of how useful the result was;*

Answer: The purpose of that benchmark is to obtain some general performance numbers based on queries of different size. In this benchmark, a predicate being "checked," "false," or "check" for a query does not significantly affect the execution time because, either way, all conditions are matched in order to give the user as much information as possible (residue).

Comment: *please explain what e.g. (85 pr) means as a description of one row in the table: is it 85 calls to findp about 85 different predicates, with for each predicate 1 to 4 assertion conditions ? (or something else)*

"The (AVG) column represents the average per module" the first two figures are 19,064.0

*and 224.3; as it happens 224.3*85 = 19,065.5 and 85 is not the number of modules, so perhaps you meant "represents the average per predicate" (but it might depend on what (85 pr) means)*

Answer: "pr" means predicates. There are 85 predicates of arity 1 throughout the 63 modules, therefore, the conditions are checked for all those predicates. We have added a phrase clarifying this in the paper.

Comment: *The last sentence of section 4: "It took on average 25s to execute each query, looking in all 63 modules, 21.5s of which were used for restoring" what exactly constitues a query ? I had interpreted as a call to findp; and I thought that (85 pr) actually meant 85 calls to findp;*

Answer: A query is indeed one call to `findp`, 85 is the number of predicates of arity 1. We have also tired to clarify this.

Comment: *then I want to reconstruct either the number 25secs or 3.5secs (25-21.5) from the table, and I can't - it means I am missing something in your explanation*

Answer: The total running time of the query is 25 seconds. That includes:

- normalizing the query (negligible)
- loading modules (21.5s)
- check the predicates with the desired arity and find out if the conditions hold or not. (3.5s)

This is an example for one query (several conditions) that we measured in our implementation.

Comment: *Some issues of varying importance; abstract: "these techniques are at the same inaccurate" time missing*

Answer: Thanks, we have removed the first "at the same".

Comment: *p5 last line "(list(L) ∧ int(N)), int(N))" I didn't expect to see the last int(N)*

Answer: Thanks, done. This was a copy-paste error: the assertion does not have success part.

Comment: *p7 "(this will be explained later)" this seems already explained on p6; at the bottom of p7/start p8 it is explained again*

Answer: Thanks, done. We have removed the parenthesis and the second sentence: "The matching of a predicate against a predicate query will succeed (i.e, the result will be "checked") if all the conditions within the query are "checked" (proved) for the inferred abstract semantics. The matching will be false if one of the conditions is false."

Comment: *fig 1: get rid of the visible spaces in "Generates _ a _ list _ ofall _ random _ elements _ of _ random _ size"*

Answer: Done. We set option `showstringspaces=false` in `lstset`.

Comment: *p9 (and some later): the word "Intuitively," (and variants) is used very often; perhaps you mean something else ?*

Answer: Agreed, done. We have reworded these sentences.

Comment: *"we need a predicate that requires the Size to be an integer is needed" something wrong with the sentence*

Answer: Thanks, done.

Comment: *p11 "it can produce also type z" I saw no type z, but I see a type value z*

Answer: Yes, we have improved the explanation of this example with the analysis results within the code and a schema of the inferred lattice.

Comment: *p11 "not being able o check" to*

Answer: Thanks, done.

Comment: *Appendix B: this contributes nothing to the understanding or the appreciation of the basic idea in the paper*

Answer: We believe that these numbers provide insight for understanding the actual analysis cost (considering that the source code is available in the Ciao distribution). If needed we can summarize or shorten those tables but in any case the appendices should go in the on-line supplementary material.

Reviewer 2

Comment: *This paper presents a framework and a tool to browse for program code (here: predicates in logic programs) based on semantic criteria. For this purpose, the authors define their "semantic criteria" with abstractions of program behavior (contracts, assertions) w.r.t. some domains, like types or modes. Furthermore, they apply abstract interpretation techniques to analyze given libraries in order to match against some requirements. Although I agree with the authors that this approach is much more precise than code browsing based on keywords, the paper misses some convincing examples showing that interesting code could really be found by this method. The presented examples are quite small so that one has the impression that one can quickly write the code instead of browsing. It is not clear how more complex examples (e.g., sorting, algorithms for finite maps, computing strongly connected components,...) can be really specified in a way so that interesting code can be found. For more complex examples, I have the impression that keyword search, although imprecise, is more successful. Thus, I would like to see more convincing examples rather than a run-time evaluation.*

Nevertheless, I think that this is an interesting direction to go and a good application for logic programming techniques.

Answer: Thanks for your positive comments. We have included in the paper some more complex examples in the prototype evaluation section. In these examples we try to show how code for different representation of graphs is found. Those properties cannot be expressed in a keyword-based search. In our opinion, this shows the strengths of our method. It can find code features, independently of whether the programmer documented them.

Note that this method clearly depends on the power of the underlying domains. In this paper, we have used only a shapes domain (inferred regular types) and a sharing+freeness domain because in practice these are some of the most successful domains when analyzing the Prolog code in our libraries. As future work, we would certainly like to experiment with other domains (such as numerical domains like polyhedra, which are needed to reason, e.g., about sorting), and properties (like code complexity).

Finally, note that the method is not intended as a replacement for other methods, but is rather complementary (or a generalization, if we consider that the set of keywords appearing in the text of a program can be turned into an abstract domain).

Comment: *Minor comments:*

P. 2, last para: Signature matching methods do not require explicit signatures, since they can be automatically inferred, e.g., in strongly typed functional languages.

Answer: It is true that signatures can be inferred, e.g., in Hindley-Milner type systems, but note that in these type systems the types themselves have to be defined in the program or libraries. In our approach the types (shapes) are also inferred by the analysis. I.e., the program can be totally untyped Prolog code and the types (definitions) used by the program are inferred by the analysis. Also, we believe that the types signature approach is limited in, e.g., Hindley-Milner type systems by the expressiveness of this type system. For example, one could infer that a function returns some value of type Color (e.g., data Color = Red | Green | Blue), but not a particular subset of colors (Red | Blue). Type signature inference on more expressive type systems is known to be undecidable, in general, and thus the convenience of using abstract interpretation: it allows seeing both these more general type

systems (including regular types) and other abstract domains in a common setting.

Comment: *P. 3: I do not understand the requirement for identical sequences of variarbles in each clause. Does this mean that each clause has to use the same number of variables? This seems impossible or possible only with using superfluous variables.*

Answer: That sentence refers to the clauses within a predicate, and can be accomplished with a program transformation:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
===>
append(A,B,C) :- A=[],B=C.
append(A,B,C) :- A=[X|Xs],C=[X|Zs],append(Xs,B,Zs).
```

This is just a notational covenience that simplifies the presentation of the semantics and of abstract interpretation in the paper, but it is not necessary in general and it is certainly not required by our prototype, which works with standard programs. Internally, both the analyzer and compiler move head unifications around for different purposes independently of how programs are written.

Comment: *P. 9: The predicate query is not well-formed according to Defs. 2 + 3, since there is no postcondition.*

Answer: Thanks for spotting this: we have added to the assertion preliminaries the fact that either the *Pre* or *Post* conditions can be omitted from query assertions (this is the case in general in the other Ciao assertions and is detailed in the referenced paper about the assertion language).

Reviewer 3

Comment: *This paper presents a novel approach for code searching by semantic properties. A query language is used to specify the abstract property. Abstract interpretation is used to capture the semantic properties from the code. The paper is well written, contributions and clear and the topic is very interesting. Examples through the paper help to understand the introduced concepts.*

Since Definitions 7 and 8 make use of the lattice ordering and the meet operator, they should be formally introduced before (at least mention what these symbols are).

Answer: Agreed, thanks. We have added those operators to the abstract interpretation preliminaries.

Comment: *The authors assume (before example 3) correctness of the checking process as a consequence of the fact that approximations are safe. But the last has not been proven.*

Answer: As stated in section 2, the fundamental theorems of AI guarantee that the fixpoint computed is a (minimal) safe approximation of the concrete semantics.

This is of course provided the domains themselves safely approximate the concrete operations. This proof is necessary for each domain used in our approach. For the particular domains used in the paper this was done in the corresponding publications referenced. We have tried to clarify this in the paper. Overall, we have tried to find a reasonable balance between somewhat conflicting requirements from two reviewers, asking to provide respectively more or less information about Abstract Interpretation. Hopefully the results are acceptable for all.

Comment: *Regarding the defined abstraction, Definitions 9 and 10 seem to pose very demanding conditions for checking predicates for success conditions. My intuition says that you will be able to find predicates just if the property is a very simple one. Is that true?*

Answer: We have added an example of search with more complex properties in the prototype and evaluation section. We show how code can be found with graph shapes. We believe that those are not trivial properties (because they require nested structures, parameterized lists).

As mentioned in the answers to the other reviews, the power of our approach depends on the abstract domains used. We would like to experiment as future work with the combination of more domains and properties (numerical domains, code complexity, etc.).

In any case, note that the method is not intended as a replacement for other methods, but is rather complementary (or a generalization, if we consider that the set of keywords appearing in the text of a program can be turned into an abstract domain).

Comment: *Could you please add a discussion regarding precision of the approach? Benchmarks include data for time but not for solutions found.*

Answer: The table does show how many predicates the conditions have been matched against (in brackets). As mentioned previously, our aim in this paper has been to present and develop the idea, implement it, and test its *viability and practicality*, specially from the

analysis point of view, when working on a relatively large code base (where efficiency was a priori one of our main concerns). In this sense we believe the results are quite encouraging.

As stated above, with this enabling point settled, have also been able to check that the system provides interesting results to our queries when working with it, finding code that would be very difficult to find with other approaches (e.g., when there are no comments, etc.). As already mentioned, we have added some of these examples in the paper and discussed them in some depth in this new version. This hopefully adds to the understanding of the precision/effectiveness of the approach.

A larger effectiveness study would indeed be interesting, but also quite complicated, and we feel the results provided are quite interesting by themselves.

The experiments in Table 1 include a selection of queries of different sizes and the total query time. In this case the checking time was similar independently of the predicate status (checked, false, or check). We think that the number of predicates per status is irrelevant for the purposes of that table.

Comment: *Page 7, lines 17-19: the sentence is a bit complicated to read. Seems redundant.*

Answer: Thanks, fixed.

Comment: *Definition 6: there is a typo in the last sentence, the superscript of lambda should be + instead of -*

Answer: Fixed, thanks.