Semantic 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)

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 increasingly 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 pre-processed 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 program elements. 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.

1 Introduction

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

* 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.

extraction from variable names in the code itself. They combine those keywords with relatively simple characteristics of the kind of code the user is looking for (e.g., whether it is classes, methods, or interfaces). Other recent work has used a similar approach combined with ranking techniques. For example, (McMillan et al. 2012) use annotations in code instead of man pages in order to cluster features from Java packages. They also incorporate the idea that multiple users will rank over time how packages match searches. 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, they are written in a different (natural) language, or other elements like variable, module, or procedure names are not representative and/or not easy to match against.

An alternative to keyword search is to query instead the signatures present in 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 in code against some pre- and post-condition specifications used as search keys. The Haskell code browser, Hoogle (Mitchell 2008), combines 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. Given their complementary nature, our implementation actually combines the two approaches of semantic and keyword-based search. Since the combination is straightforward, it is not described herein.

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 $H\,\text{:-}\,B_1, \ldots, B_n$ 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 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. 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 preprocessing 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

[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.

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: While our approach is valid for any analysis, we will be using for concreteness goal-dependent abstract interpretation, in particular the PLAI algorithm (Muthukumar and Hermenegildo 1992), available within the Ciao/-CiaoPP system (Hermenegildo et al. 2005; Hermenegildo et al. 2012). PLAI takes as input a program P, an abstract domain D_α,[2] and an abstract initial call pattern[3] $\mathcal{Q}_\alpha = L{:}\lambda$, 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 the 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, $\perp$ denotes the abstract substitution such that $\gamma(\perp) = \emptyset$. A tuple $\langle P_j, \lambda_j^c, \perp\rangle$ 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 (types of calls to) 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

[2] Also, a set of abstract domains.
[3] We use sets of calls patterns in subsequent sections –the extension is straightforward.

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 (Puebla et al. 2004). 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.

Traditional Assertions: Assertions are linguistic constructions for expressing abstractions of the meaning and behavior of programs. Herein, we will use for concreteness the `pred` assertions of (Puebla et al. 2000a) Such `pred` assertions allow stating sets of *preconditions* and *conditional postconditions* on the state (current substitution or constraint store) that hold or must hold for a given predicate. These assertions are instrumental for many purposes ranging from expressing the results of analysis to providing partial specifications which are then very useful for detecting deviations of behavior (symptoms) with respect to such assertions, or to ensure that no such deviations exist (correctness) (Puebla et al. 2000a). A `pred` assertion is of 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), and 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 $\mathcal{A} = \{A_1 \dots A_n\}$, with $A_i =$ "`:- pred` $Head$ `:` Pre_i `=>` $Post_i$`.`" the set of *assertion conditions* for $Head$ determined by $\mathcal{A}$ is $\{C_0, C_1, \dots, C_n\}$, 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: $\boxed{\texttt{:- pred } L \texttt{ : } Pre \texttt{ => } Post\texttt{.}}$ 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: $\boxed{\texttt{?- findp(\{ } As \texttt{ \}, M:Pred/A, Residue, Status).}}$
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 (Gallagher and de Waal 1994; Vaucheret and Bueno 2002) or other `properties`) 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 express 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 *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. *false* if one condition is false, and *check* if neither *checked* nor *false* can be proved. If **Status** is instantiated to e.g., *checked* in the query, only matching predicates are returned.

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:

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

indicates that the user is looking for predicates $p \in P$ with $\mathsf{ar}(p) = 2$, which allow calls in which 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, their second argument will be instantiated to an integer. A predicate that matches this query is, for example, the `length/2` predicate of Ex. 1, which we assume defined in module `lists`. The call to `findp` would then unify `M:X` to `lists:length`. `Residue` would contain the explanation of why the predicate matches (all conditions would be checked in this case; these conditions are illustrated later in other examples). Other possible matching predicates would be returned via backtracking. □

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:

```
list([]).
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 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 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). Concentrating for now on calls only, this analysis will infer the calls abstract states that are shown also in Fig. 1, as "true" `pred` assertions. There, `var/1` and `ground/1` have the usual meaning and `mshare/1` describes *variable sharing* (intuitively, two variables

```
:- 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)).
:- true pred my_length(L,N) : ( mshare([[L],[L,N],[N]]), var(N)).
:- true pred my_length(L,N) : ( mshare(L), ground(N) ).
my_length(L,N) :- length(L,N).

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

:- pred get_length(L,N) : (list(L), var(N)).
:- true pred 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".
:- true pred 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 stating different calls and (partial) analyzer output.

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 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 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 the second case of `my_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 obviously a predicate of interest in the context of the predicate query because it expects both of its arguments to be variables (plus, 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:
  $(\lambda^{+}_{TS((var(L),var(Size)),P)} = \{var(L), var(Size)\}) \sqcap (\{mshare(L), ground(Size)\} = \perp)$.

- 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 (using the `shfr` domain only) 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 `my_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 $\forall\langle L,\lambda^c,\lambda^s\rangle \in analysis(P,Q_\alpha)$ $s.t.$ $\exists\sigma \in ren,$ $L = p(V'_1,\ldots,V'_n) = X(V_1,\ldots,V_n)\sigma, \lambda^c \sqsupseteq \lambda^+_{TS(Pre\ \sigma,P)} \rightarrow \lambda^s \sqsubseteq \lambda^-_{TS(Post\ \sigma,P)}$.

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 $\forall\langle L,\lambda^c,\lambda^s\rangle \in analysis(P,Q_\alpha)$ $s.t.$ $\exists\sigma \in ren,$ $L = p(V'_1,\ldots,V'_n) = X(V_1,\ldots,V_n)\sigma, \lambda^c \sqsubseteq \lambda^-_{TS(Pre\ \sigma,P)} \wedge (\lambda^s \sqcap \lambda^+_{TS(Post\ \sigma,P)} = \bot)$.

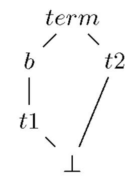
Example 4
Several checks against a 'success' condition. Assume that we analyze the module in Fig. 2 with a shape abstract domain D_α —in particular `eterms` (Vaucheret and Bueno 2002) (regular types). Originally, the code had no assertions, so the analysis was performed for any possible entry. As before, the inferred information is provided by the analyzer as "true" `pred` assertions (we omit the calls conditions for simplicity). The relation among these inferred abstract elements is shown in lattice form in Fig. 2a. The regular type b was included in the program and $t1$ 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, St).`
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:

- `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 $b \sqsubseteq b$.
- `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.

```
:- module(simple, _, [assertions, regtypes]).
:- true pred perfect(A) => b(A).        :- true pred mixed(X) => term(X).
perfect(b1).                             mixed(b0).
perfect(b0).                             mixed(b1).
                                         mixed(z).
:- true pred reduced(A) => t1(A).
reduced(b1).

:- true pred outb(A) => t2(A).          :- true pred hard(X) => term(X).
outb(z).                                 hard(X) :- functor(b1(_), X, _).

:- regtype b/1.   :- regtype t1/1       :- regtype t2/1.
b(b0).            t1(b1).               t2(z).
b(b1).
```



(a) Lattice of the inferred shapes ($term$ is $\top$).

Fig. 2: Another simple program with analysis information on success conditions.

- `outb/1`. This predicate is of no use, because its output (z) is completely different (disjoint) from that specified 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 $(\lambda^s = t2) \sqcap (\lambda^+_{TS(b,P)} = b) = \bot$.

Finally, we again have predicates (`mixed/1` and `hard/1`) that are not `checked` or `false`. As discussed before, this can 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. In the case of predicate `mixed/1`, it is the former: it is not what we are looking for because, although its possible outputs can be of type b, it can also produce 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: $(\lambda^s = \top) \sqcap (\lambda^+_{TS(b,P)} = b) = b \neq \bot$.

Predicate `hard/1` illustrates the latter case: that an abstract domain may not be precise enough to find all matching predicates. Intuitively, the success condition of the example should hold because its output shape is more restrictive than specified. However, the analyzer cannot infer that its output will be always `b1` because `functor/3` can produce any atom, and thus the inferred tuple will be $\langle hard(V), \top(V), \top(V)\rangle$. The reasoning 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. For example, in `:- pred X(A,B) : (list(A), var(B))`, the property `var(X)` cannot be represented in the (standard) regular types domain (`eterms`), 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 the `eterms` shape analysis is also performed:

Predicate	λ^c (eterms)	λ^c (shfr)
$gen\_list(L,N)$	$(term(L), term(N))$	$(mshare([[L],[L,N],[N]]), var(L), var(N))$
$get\_length(L,N)$	$(list(L), term(N))$	$(mshare([[L],[L,N],[N]]), var(N))$
$check\_length(L,N)$	$(list(L), int(N))$	$(mshare(L), ground([N]))$
$my\_length(L,N)$	$(list(L), term(N))$	$(mshare(L), ground(N))$
$my\_length(L,N)$	$(list(L), int(N))$	$(mshare([[L],[L,N],[N]]), var(N))$

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

PredName/A	eterms 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/2`: In the `eterms` 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/2`: This case is similar to `gen_list/2`: 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 it is false.
- `check_length/2`: Matches the condition in the `eterms` domain, because the shapes are exactly the ones we were looking for. For this predicate, the `shfr` domain is not necessary.
- `my_length/2`: At first sight this predicate matches the query because there is a calling mode that matches exactly as stated in the condition. However, according to the definition of calls condition, all admissible 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, against the 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, consider looking in the Ciao libraries for code that operates with graphs. First, we need to guess how graphs may be represented, i.e., their shape. Two possible guesses are:

```
:- regtype math_graph(Graph).
math_graph(graph(Vertices,Edges)):-
    list(Vertices), list(Edges, pair).

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

```
:- 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 adjacency list graph representation, as a list of vertices and their corresponding neighbors. A query assertion for finding code that uses the first representation could be `:- pred P(X,Y) => math_graph(Y).`[4] The prototype finds `complete_graph/2` and `cycle_graph/2` in module `named_graphs.pl` (see Fig. A 1) by matching this query against the analysis results for the module. Note that this code is found although this `named_graphs.pl` module has *no assertions or shape/regtype definitions*, i.e., it only contains plain Prolog code. Searching for the second representation, assume we look for code for modifying a graph, i.e., that takes as input a graph and a list of elements and produces a new graph:
`:- pred P(A,B,C) : (al_graph(A), list(B), var(C)) => al_graph(C).` I.e.:

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

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

No code is found for which both conditions hold, because `calls` can be checked only if the code has assertions (hand-written or inferred modularly). Therefore, we focus on finding predicates for which C_2 holds. Since the conditions on the calls substitution are very specific, we assume they were not considered by the default pre-analysis. We can refine the predicate matching by reanalyzing the predicates starting from the calls values in the success conditions. To ensure greater precision, we perform inter-modular analysis. Under these conditions the prototype finds that in `add_vertices/3`, `del_ vertices/3`, `add_edges/3`, and `del_edges/3` the `success` condition does hold (see Fig. A 2).

[4] As mentioned before, the user-defined shapes (or any other properties), in this case the regtypes above, must be included within the predicate queries. However, we just show 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 2: Predicate query matching times (μs).

Performance results. To measure the effectiveness and performance of the approach, we have set up an experiment that consists in analyzing part of the Ciao libraries and finding matching predicates of arity 1 to 4 for several assertion conditions. The experiments were run on a Linux server (Intel Xeon CPU E7450, 2.40GHz) with 16GB of RAM. As in the previous examples, we used the `shfr` and `eterms` domains (the Ciao system includes however a large number of other domains than can also be used in this application). We selected 63 modules from the Ciao libraries all of which can be analyzed within 1 minute for these abstract domains. The detailed analysis statistics can be found in Appendix C. The selection includes modules that are relatively costly for the analyses and others where analysis is trivial (e.g., non-analyzable foreign code with trusted assertions) but useful for the search. The pre-analysis of all the modules took $45s$ ($660ms$ on average), and the analysis dumps required $3.5MB$ of disk space ($55.5KB$ on average). Restoring the analysis results (for the 63 modules) takes $21.5s$ ($343ms$ on average). In the experiments this was done for each query, but note that since the size of the cached analysis is small it can be kept in memory for subsequent queries. The performance of matching, 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. Summarized timing results are shown in Table 2. Columns represent the number of assertion conditions in each predicate query and rows their arity (the parentheses show 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)* columns represent the average time per predicate: 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, most of which ($21.5s$) is spent loading the pre-analysis.

5 Conclusions

We have proposed a novel approach to the code search problem based on querying for *semantic* characteristics of the programs against a safe approximation of its semantics obtained via analysis. We have also discussed the advantages of our proposal over other approaches such as keyword search or signature matching. We have provided evidence that both the analysis and the search are sufficiently efficient, despite the relatively naive implementation, for practical use. Our implementation actually combines semantic code search with keyword-based and other types of search. A number of other extensions are also in progress, such as allowing permutations or extra arguments, and applying other program transformations. We believe the proposed approach has a number of additional applications, such as, for example, detection of duplicated code. While prototyped within the Ciao system, the techniques proposed, based on abstract interpretation theory, are general and directly applicable to other languages.

References

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

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., Correas, J., Hermenegildo, M., Bueno, F., García de la Banda, M., Marriott, K., and Stuckey, P. J. 2004. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In *Program Development in Computational Logic*. Number 3049 in LNCS. Springer-Verlag, 234–261.

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.

Appendices

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

(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**
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
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	26.7	752,967	157	703,443	184

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

(continued in next page)

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

Module name	**dump size(B)**	**restore time(s)**
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
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

(continued in next page)

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

Module name	**dump size(B)**	**restore time(s)**
assrt_synchk	58	20
doc_props	396	20
TOTAL (63)	3,512,057	21,596
AVG	55,747	343