Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Inteligencia Artificial

CODE SEARCH: A SEMANTIC, ABSTRACT INTERPRETATION-BASED APPROACH

Author: Isabel Garcia-Contreras

Director: Manuel V. Hermenegildo

Co-director: José F. Morales

Madrid, April 2, 2017

CODE SEARCH: A SEMANTIC, ABSTRACT INTERPRETATION-BASED APPROACH

Author: Isabel Garcia-Contreras
Director: Manuel V. Hermenegildo
Co-director: José F. Morales

Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

April 2, 2017

Thanks

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.

Resumen

Los programadores tienen acceso actualmente a un gran número de repositorios de código que aumenta cada día. Sin embargo, el gran potencial en cuanto a reutilización que ofrecen estos servicios se ve limitado por el hecho de que encontrar el código apropiado es una tarea muy difícil. La mayor parte de los buscadores de código actualmente están basados en técnicas sintácticas, como *signature matching*, o extracción de palabras clave. Estas técnicas son imprecisas (porque dependen básicamente de la documentación) y a la vez no ofrecen unos lenguajes de búsqueda muy expresivos. En esta tesis de máster se propone un nuevo enfoque basado en preguntar a los buscadores por características *semánticas* del código, obtenidas automáticamente del mismo. Los programas se *pre-procesan* utilizanto técnicas de análisis estático, basadas en la interpretación abstracta, lo que permite obtener aproximaciones semánticas seguras. Se presenta un nuevo lenguaje de búsqueda usado para expresar las características semánticas deseadas como especificaciones parciales. Se encuentra código relevante comparando estas especificaciones parciales con la semántica inferida de cada elemento de los programas. Nuestro enfoque es totalmente automático y no depende de anotaciones de los programadores o documentación. Es más potente y flexible que *signature matching* porque es dependiente del dominio abstracto y las propiedades (en vez de sólo de los tipos). También es capaz de razonar con relaciones entre propiedades, como la implicación o la abstracción, en lugar de sólo la igualdad. Es más robusto frente a diferencias sintácticas en el código. En este documento se describe este enfoque y se evalúa la implementación de un prototipo en el sistema Ciao.

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 at the same 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 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. 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.

Contents

List of Figures

List of Tables

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.

1.1 Related work

Most code search engines have so far addressed this problem through syntactic techniques such as keyword extraction and signature matching.

1.1.1 Keyword extraction

[14] 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 it is classes, methods, or interfaces). Other recent work has used a similar approach combined with ranking

techniques. For example, [15] 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. Also, the code query languages offered are limited in the sense that they do not allow expressing semantic characteristics of the programs to look for.

1.1.2 Signature matching

An alternative to keyword search is to query instead the signatures present in the code, an approach already proposed in [24] for finding code written in a functional language. In this work, the solver within λProlog was used to *match* the signatures present in the code against some pre- and post-condition specifications used as search keys. The Haskell code browser Hoogle [16] combines this type of signature matching with keyword matching. In the same line [23] 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.

1.2 Our approach

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 meaning and behavior 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.

1.3 Structure of the document

In the rest of the thesis, we start by describing in Chapter 2 the Ciao system, the framework that we have used to develop our prototype and within we will be looking for code. In Chapter 3 we present some keyword matching techiques and how they are included in the Ciao System. Chapter 4 contains our theoretical basis: preliminaries and notation, a review of program semantics approximation by Abstract Interpretation, and a description of our code query language and code search procedure. In Chapter 5 we report on a prototype implementation of our approach that uses the abstract analysis engines of the within the Ciao system and provide some performance results. Chapter 6 provides the User Manual and documentation for the prototype, including the module structure. Finally, Chapter 7 contains our conclusions and some proposals for future work.

2

The Ciao System

Ciao [11] is a modern, multiparadigm programming language with an advanced programming environment. The main motivation behind the system is to develop a combination of programming language and development tools that together help programmers produce better code in less time and with less effort.

A high-level view of the Ciao System is shown in Fig. 2.1. Blue-coloured boxes represent user-written code; green boxes represent different tools within the system: the compiler, LPdoc (detailed in Sect. 2.3) and the Ciao Preprocessor; and the red box represents the interpreter of the system. In this thesis, only some of them will be detailed, as not all of them are used.

In the following sections we explain how source files are organized in Ciao (the Ciao Module System), the Ciao asserions, and how documentation is generated (with LPdoc), based on such assertions.

2.1 The Ciao Module System

In this section we detail the main points of the Ciao Module System [3]. Modularity is a basic notion in modern computer languages. Modules allow dividing programs into several parts, which have their own independent name spaces and a clear interface with the rest of the program.

This isolated way of seeing the code has two main advantages. It allows a divide-

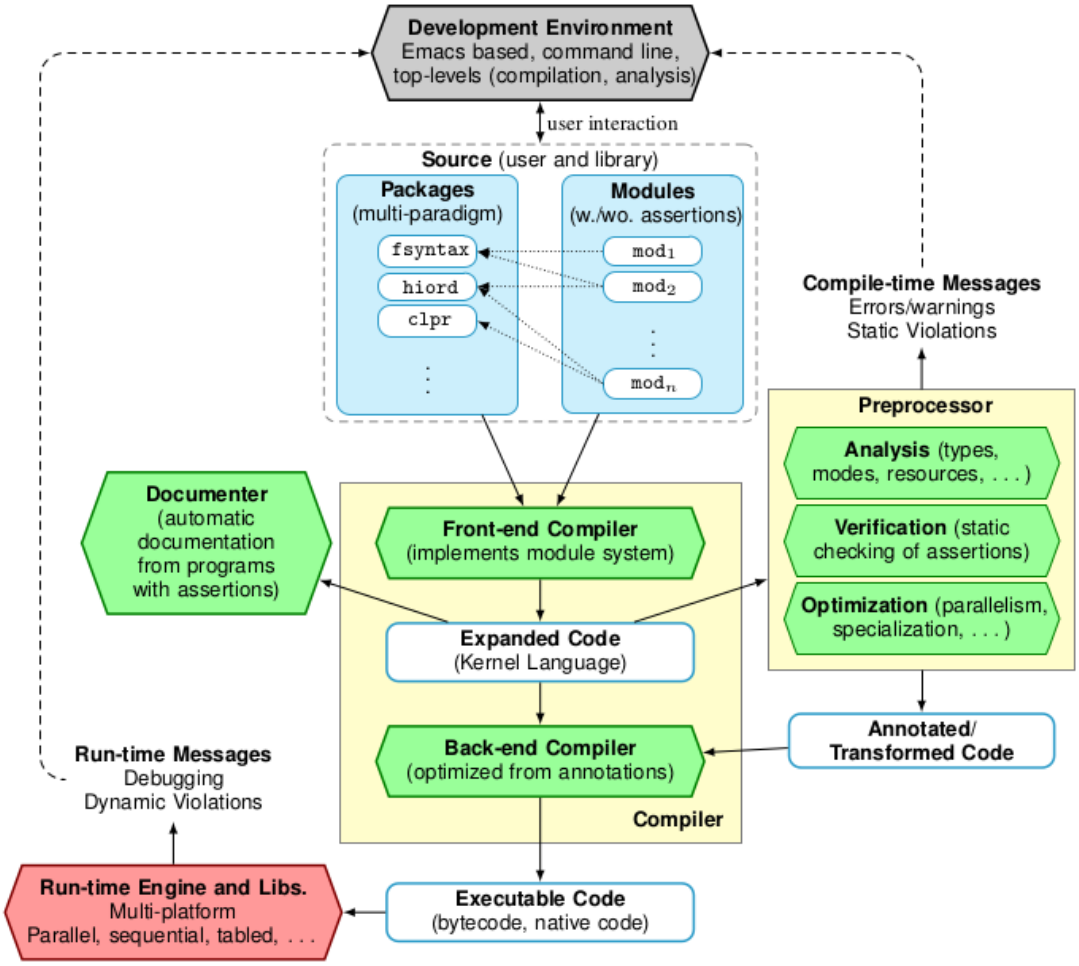


Figure 2.1: A high-level view of the Ciao system [11].

and-conquer approach to program development and maintenance and, in terms of efficiency, tools which work with programs can be more efficient if they can process a single program at a time.

Defining Modules: The source of a Ciao module is typically contained in a single file. The fact that a file contains a module is flagged by the presence of a "`:- module(...`" declaration at the beginning of the file. The predicates defined within a module will be visible only if they are exported.

Imports and Exports: A number of predicates in a module can be *exported*, i.e., made available outside the module, via explicit `:- export` declarations or in an export list in the `:- module(...` declaration.

Also, it is possible to import a number of individual predicates or also all pred-

icates from another module, by using `:- use_module` and `:- import` declarations. Those predicate must be previously exported by the concrete module.

Visibility Rules: The set of predicates which are visible in a module are predicates defined in that module plus the predicates imported from other modules. It is possible to refer to predicates with or without a *module qualification*. A module-qualified predicate name has the form *module:predicate*. An example of this form is the call `lists:append(A,B,C)`.

2.2 Ciao Assertions

Assertions are linguistic constructions for expressing abstractions of the meaning and behavior of programs. Herein, we will use the `pred` assertions of [19]. 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:

$$\texttt{:- pred } \mathit{Head} \texttt{ : } \mathit{Pre} \texttt{ => } \mathit{Post}\texttt{.}$$

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* [19, 21]. 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.
6 list([]).
7 list([_|T]) :- list(T).
```

Note also the definition of the `list/1` property (in this case a regular type) in line 7. 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:

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

2.3 The Documentation Generator

`LPdoc` is a tool within the Ciao system [9] that automatically generates program documentation for (C)LP systems. Its main functionality is to generate a reference manual automatically from one or more source files of (constraint) logic programing systems. The operation of LPdoc is illustrated in Fig. 2.2.

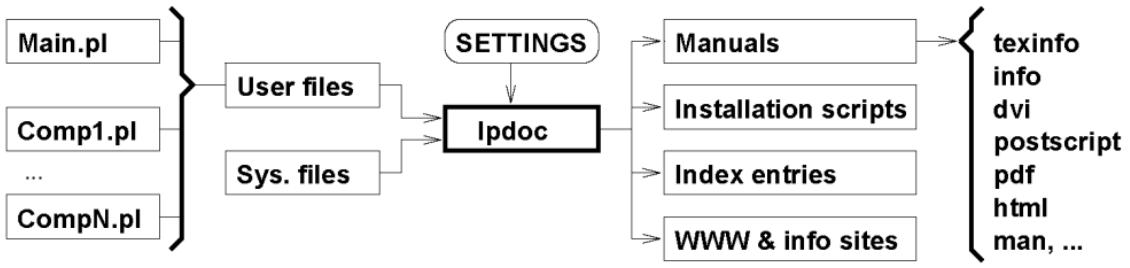


Figure 2.2: Overall operation of LPdoc.

It combines the information from a number of user and system files (as specified in an user-provided configuration file –SETTINGS in Fig. 2.2) an produces manuals in a number of formats which can include bibliographic citations and images.

The quality of the generated documentation can be greatly enhanced by including within the program text *assertions* for the predicates in the program. The assertions and comments included in the source file need to be written using the Ciao *assertion language* [20].

The manual describing the implementation of the prototype included within this thesis was generated from the implementation itself using `LPdoc` (see DeepFind Manual).

3

Traditional search

Searching with keywords is sometimes really useful but it can be limited if the user does not know the exact words used within the code that he/she is looking for. In this section we explore two different options, that can be combined, to easily improve the exact word search.

3.1 Regular expression search

The first improvement that we will explore are regular expressions. Regular expressions are sequences of characters that define a pattern. They can be useful when the user does not know the exact name of the predicate or wants to find code that contains a concrete word or set of characters. Including this kind of expressions for specifying keyword requirements improves the expressivity of the search.

The Ciao System already contains and implementation of regular expression-based search in the librowser library: `apropos/1`. It uses the POSIX regular expression language. The usage of this library is detailed in the User's Manual. In the following example we want to find the predicates with a name beginning with "write" and of arity 2:

```
?- apropos('write.*'/2).
dht_misc:write_pr/2
profiler_auto_conf:write_cc_assertions/2
mtree:write_mforest/2
```

```
transaction_concurrency:write_lock/2
transaction_logging:write/2
provrml_io:write_vrml_file/2
provrml_io:write_terms_file/2
...

yes
?-
```

3.2 Fuzzy string search

Although searching with regular expressions can be very useful, if the user makes a spelling mistake, writes a small letter instead of a capital letter, misses a dash, etc, many results can be overlooked.

To deal with these mistakes, and to improve the amount of found predicates, we have implemented a basic fuzzy search algorithm (also known as a similarity search algorithm). Fuzzy search algorithms are characterized by a *metric*, i.e., a function of distance between two words, which provides a measure of their similarity.

3.2.1 Metric

Formally, a metric on a set X is a function

$$d : X \times X \to [0, \infty)$$

where $[0, \infty)$ is the set of non-negative real numbers, and for all $x, y, z \in X$, the following conditions are satisfied:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \iff x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$

We have explored a number of metrics, such as *Hamming*, *Levenshtein* and *Damerau-Levenshtein* distances (see [6]). *Hamming* distance [8] is a metric for sets of words of equal lenght, which is not really useful in practice because it does not cover all the types of mistakes that we exposed earlier, like forgetting a letter.

Levenshtein distance

Informally, the *Levenshtein* distance [13] between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.

Mathematically, to express the *Levenshtein* distance between two strings a and b a function $lev_{a,b}(i,j)$ is defined, whose value is a distance between an i-symbol prefix (initial substring) of string a and a j-symbol prefix of b. The *Levenshtein* distance between two strings a, b (of lenght $|a|$ and $|b|$ respectively) is given by $lev_{a,b}(|a|,|b|)$ where:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & otherwise \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

Some examples of this function:

- Dist_{Lev}(append, apend) = 1
- Dist_{Lev}(append, appennd) = 1
- Dist_{Lev}(append, appemd) = 1
- Dist_{Lev}(append, appedn) = 2

As the examples show, this function is very useful for measuring edits, insertions and deletions. Transpositions, on the other hand, are not counted as one mistake (last example). This function can be easily extended in order to "correctly" count those errors.

Damerau-Levenshtein distance

The *Damerau-Levenshtein* distance [5] between two words is the minimum number of operations needed to transform one string into the other. Operations are defined as an insertion, deletion, or substitution of a single character, or a transposition of two adjacent characters. In [5], it is stated that this set of errors correspond to more than 80% of all human misspellings and that they should be corrected with at most one edit operation.

Formally, to express the *Damerau-Levenshtein* distance between two strings a and b a function $dam_{a,b}(i,j)$ is defined, whose value is a distance between an i-symbol prefix (initial substring) of string a and a j-symbol prefix of b. The function

is defined recursively as follows:

$$dam_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} dam_{a,b}(i-1,j)+1 \\ dam_{a,b}(i,j-1)+1 \\ dam_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ dam_{a,b}(i-2,j-2)+1 \end{cases} & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \\ & \text{and } a_{i-1} = b_j, \\ min \begin{cases} dam_{a,b}(i-1,j)+1 \\ dam_{a,b}(i,j-1)+1 \\ dam_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & otherwise. \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

With this distance, the same examples as above:

- Dist_{Dam}(append, apend) = 1
- Dist_{Dam}(append, appennd) = 1
- Dist_{Dam}(append, appemd) = 1
- Dist_{Dam}(append, appedn) = 1

3.2.2 Fuzzy predicate search

We have determined that the most practical distance is the *Damerau-Levenshtein* distance because it counts transposition as one mistake.

We have implemented a simple linear search algorithm that checks all exported predicates in a given set of Ciao modules. Linear search performs well: checking the specifications against the set of exported predicates of 1300 modules takes from $1.5s$ to $4s$ approximately. Here is an example of usage, we try to find predicates with the name "write" but we misspell it:

```
?- apropos('wirte').
Predicate wirte not found. Similar predicates:

transaction_logging:write/2
write:write/1
write:write/2

yes
?-
```

4

Abstract Code Search

In this chapter we detail our new approach to code searching: searching with semantic characteristics. After introducing some preliminaries and notation in Section 4.1 and reviewing program semantics approximation by Abstract Interpretation in Section 4.2, in Section 4.3 we present our code query language, in Section 4.4 and Section 4.5 we describe our code search procedure and give some examples. In Section 4.7 we present the pseudocode of the algorithms needed for the implementation of this approach.

4.1 Preliminaries and Concrete Semantics

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. In the following, we assume that all clause heads are normalized, i.e., H is of the

[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 work we will refer interchangeably to substitutions or constraints, and to the current substitution or the constraint store.

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. Let ren denote a set of renaming substitutions over variables in the program at hand. The concrete semantics used for reasoning about goal-dependent compile-time semantics of logic programs will use the notion of generalized AND trees, described in [1]. 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 adorned 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.

4.2 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* [4], 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 [18], available within the Ciao/CiaoPP system [10, 12]. 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 in abstract interpretation, $\bot$ 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 work we use analyses that are multivariant on calls.

The analysis is said to be *multivariant on successes* if more than one triple $\langle P, \lambda^c, \lambda_1^s\rangle, \ldots, \langle P, \lambda^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^s \neq \lambda_j^s$ for some i, j may be computed for the same predicate p and call substitution λ^c. Different analyses may be defined with different levels of multivariance. Our analysis allows both types of multivariance, but multivariance on calls is switched on by default while multivariance on success is off by default, for efficiency reasons.

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

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. Code will be asumed to be written within the Ciao module system, specified in Chapter 2. 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 [2]. 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.

4.3 Abstract Code Search

We now 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 [25]. 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 Heads. 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. It will be* false *if they were proven not to hold in that domain. Lastly, it will be* 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). },
2              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 [22, 21]) 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:*

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

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 separately.

4.4 'Calls' Condition Matching

The idea of 'calls' conditions is to restrict the search with properties that must hold before the code is executed. This properties are usually defined by the author of the code via regular Ciao assertions.

Intuitively, a 'calls' condition $C = \texttt{calls}(X(V_1, \ldots, V_n), Pre)$ is checked for a predicate if the admissible calls of the predicate are within the set of Pre conditions.

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.* $\exists \sigma \in ren,\ L = p(V'_1, \ldots, V'_n) = X(V_1, \ldots, V_n)\sigma, \lambda^c \sqsubseteq \lambda^-_{TS(Pre\ \sigma, P)}$.

Intuitively, a condition $C = \texttt{calls}(X(V_1, \ldots, V_n), Pre)$ is false for a predicate if the admissible calls of the predicate and the set of Pre conditions are disjoint.

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.

```
1  :- module(_, [my_length/2, get_length/2, check_length/2,
       gen_list/2], [assertions]).
2
3  :- pred my_length(L,N) : (list(L), var(N))  => int(N).
4  :- pred my_length(L,N) : (list(L), int(N)).
5  % :- calls length(L,N) : ((mshare(L), ground(N)) ;
       (mshare([[L],[L,N],[N]]), var(N)).
6  my_length(L,N) :- length(L,N).
7
8  :- pred check_length(L,N) : (list(L), int(N)).
9  % :- calls check_length(L,N) : (mshare(L), ground([N])).
10 check_length(L,N) :- length(L,N).
11
12 :- pred get_length(L,N) : (list(L), var(N)).
13 % :- calls get_length(L,N) : (mshare([[L],[L,N],[N]]), var(N)).
14 get_length(L,N) :- length(L,N).
15
16 :- pred gen_list(L,N) : (var(L), var(N)) => (list(L), int(N))
17 # "Generates a list of random elements of random size".
18 %:- calls gen_list(L,N) : (mshare([[L],[L,N],[N]]), var(L),
       var(N)).
19 gen_list(L,N) :- length(L,N).
20
21 % Implementation of length/2 ...
```

Figure 4.1: Program with assertions that define different calls.

Example 3 ***Several checks against 'calls' conditions***

Consider the program in Figure 4.1 and the classic sharing and freeness (`shfr`) abstract domain [17]. This analysis will infer the calls substitutions written in comments in Figure 4.1, 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:

4.5 'Success' Condition Matching

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.

Intuitively, a success condition $C = \texttt{success}(X(V_1, \ldots, V_n), Pre, Post)$ is checked for a predicate if, if Pre holds at the time of calling the predicate and the execution succeeds then the $Post$ conditions hold. Formally,

Definition 9 (Checked Predicate Matches for a 'success' Condition) *A success condition* $C = \texttt{success}(X(V_1, \ldots, V_n), Pre, Post)$ *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.* $\exists \sigma \in ren, L = p(V'_1, \ldots, V'_n) = X(V_1, \ldots, V_n)\sigma$, $\lambda^c \sqsupseteq \lambda^+_{TS(Pre\ \sigma,P)} \rightarrow \lambda^s \sqsubseteq \lambda^-_{TS(Post\ \sigma,P)}$

Intuitively, a predicate does not hold the properties in a success assertion $C = \texttt{success}(X(V_1, \ldots, V_n), Pre, Post)$ if, given that Pre conditions hold and the predicate succeeds, the success conditions of the predicate and the $Post$ conditions are disjoint.

Definition 10 (False Predicate Matches for a 'success' Condition) *A success condition* $C = \texttt{success}(X(V_1, \ldots, V_n), Pre, Post)$ *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 Figure 4.2 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 Figure 4.3.

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).
```

Figure 4.2: A simple program analyzed.

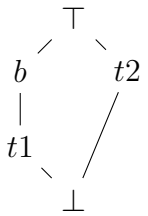


Figure 4.3: Inferred lattice of the module in Figure 4.2.

- `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., $t1 \implies b$, 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 $t2$. For-

mally, 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. Assume that we have the same analysis results as before. 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`. □

4.6 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 ***Checking with different domains***

Assume the program in Figure 4.1 and the analysis in Ex. 3, but that an analysis of regtypes is also performed (e.g., [7] or [26]):

Predicate	λ^c (regtypes)	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 predicate that checks the length of a list. The condition to be satisfied is `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. □

□

4.7 Algorithms

In this section we explain the pseudocode of the main algorithms used for the prototype implementation. The process of finding code has several phases: pre-analysis, module inspecting and predicate matching, which are detailed in the following sections.

4.7.1 Pre-analysis

The first step for finding code is to make sure that the code is analyzable, i.e., its semantic characteristics can be inferred in a finite (or in our case viable) amount of time and/or memory. To this end, we perform a generic pre-analysis with a timeout. We store which modules are analyzable and which are not in an auxiliary file. This way, we also store which have been already analyzed, allowing us to perform the pre-analysis incrementally.

The pre-analysis algorithm is detailed in Algorithm 1. It takes as an input a path in which modules or programs are located. In this path, source module files are extracted recursively. Each program unit will be loaded and analyzed independently and a report will be generated containing errors and/or timing statistics of the module. The information inferred is dumped to disk and can be used for matching later.

This algorithm is independent from the search. It has to be done only before the first search is executed. Once pre-analysis is done, search can be performed an arbitrary number of times.

Algorithm 1 Precompute Analysis

Input: Path of modules/programs
Output: Module analysis status, Analysis results, Timing report

```
1:  for all Module m ∈ Path do
2:      if load(m) has errors then store load errors
3:      else
4:          for all d ∈ Abstract Domains do
5:              if analyze(d) has errors then store analysis errors
6:              else if analyze(d) timeouts then store analysis errors and timeout
7:              else store analysis
8:              end if
9:          end for
10:     end if
11:     store module status
12: end for
```

4.7.2 Module inspecting

Each time the search is performed, all modules in the specified Path are inspected for matching predicates. Before restoring an analysis or re-analyzing a module we check whether it contains predicates with the appropriate arity and that meet the keyword specifications or not. Filtering is a much lighter task than analyzing, this is the reason why we to it at this point, to avoid spending time and resources analyzing modules that do not contain desired predicates.

If the module contains predicates that meet the specifications, its analysis is restored or it is reanalyzed. All conditions specified in the user's query are checked against each predicate, which will produce a residue. The residue is summarized in order to obtain an overall result of the conditions, as stated in Definition 3. The algorithm returns a list containing all filtered predicates, their residue and their summary. The pseudocode of this algorithm is shown in Algorithm 2.

Algorithm 2 Search Predicate

Input: As = Assertions (Arity), Keyword Specification, Modules
Output: list of (Predicate, Residue, Summary)

```
1: for all Module m ∈ Modules do
2:     Preds = {p ∈ exported(m) | ar(p) = Arity ∧ meets(p, Keyword_Spec)}
3:     if Preds ≠ ∅ then
4:         Analyze module/Restore analysis (depending on flag)
5:         for all Predicate p ∈ Preds do
6:             for all C ∈ As do
7:                 Obtain Matching Residue of C for p
8:             end for
9:             Summary = summarize_conditions(Residue)
10:            return p, Residue, Summary
11:        end for
12:    end if
13: end for
```

4.7.3 Predicate matching

Algorithms 3 and 4 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 3 Matching Status of a calls condition for a predicate p

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

1: **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**
2: Status = **Checked**
3: **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**
4: Status = **False**
5: **else**
6: Status = **Check**
7: **end if**

Algorithm 4 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: Residue

1: **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**
2: **if** $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$ **then**
3: Status = **Checked**
4: **else if** $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \bot$ **then**
5: Status = **False**
6: **else**
7: Status = **Check**, analysis accurate enough
8: **end if**
9: **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**
10: **if** $\lambda^s \sqsubseteq \lambda^-_{TS(Post,P)}$ **then**
11: Status = **Checked**
12: **else if** $\lambda^s \sqcap \lambda^+_{TS(Post,P)} = \bot$ **then**
13: Status = **False**
14: **else**
15: Status = **Check**, Refine analysis
16: **end if**
17: **else**
18: Status = **Check**, No information for that calls, Refine analysis
19: **end if**

5 Implementation

We have developed a prototype implementation on top of the Ciao/CiaoPP system. In this chapter, we explain shortly the features of the prototype, we show some interesting search examples. Finally, we evaluate the performance of the search.

5.1 Putting all together

In our prototype we have combined the semantic approach described in Chapter 4 with the traditional code search described in Chapter 3. Keyword specifications can be made with an optional directive, "apropos". Here there is a query to try to find predicates that write in streams, i.e., predicates that contain the word write and the first argument is of type stream:

```
?- findp({
      :- pred P(X, Y) => stream(Y).
      :- apropos('.*write.*').
      }, P, Residue, Status).

P = strings:write_string/2,
Residue =  ...,
Status = checked ? ;

P = fastrw:fast_write/2,
Residue =  ...,
```

```
Status = checked ?

yes
? -
```

We also have developed a module that outputs the results in `html` format to ease the search visualization. This view offers explanations for predicates for which properties could not be proved to hold. The module also generates links to the online Ciao documentation of the found predicates and their correspondent modules. An example of searching with this module is shown in Figure 5.1. Ideally, we want to use the documentation generator (LPdoc) described in section 2.3 in order to automatically generate information for any piece of code.

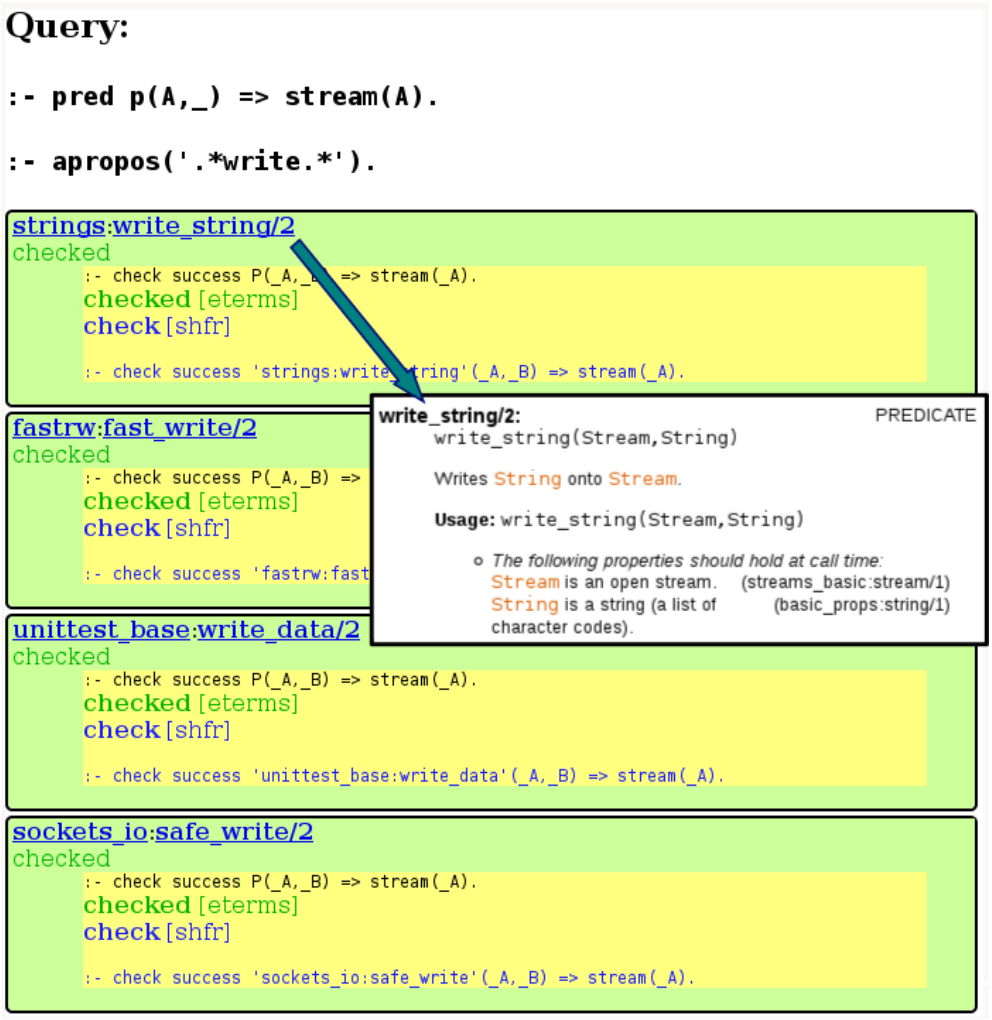


Figure 5.1: `pretty_finder` search example.

These and other features are fully detailed in the User's and Reference Manual in Chapter 6.

5.2 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).`[1]

The prototype finds `complete_graph/2` and `cycle_graph/2` in module `named_graphs.pl` (see Fig.A.2) 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

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

[1] 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.

`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. This way the prototype finds that `add_vertices/3`, `del_vertices/3`, `add_edges/3`, and `del_edges/3` (see Fig. A.1) hold the `success` condition.

5.3 Performance results

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 5.1: Assertion Checking times (μs).

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 **??**. 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.[2] 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 5.1. 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.

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

6

Prototype Manual

In this chapter we show the User's and Reference Manual from our prototype. This manual was generated from explanations written via assertions and computer readable coments with LPdoc.

Deepfind

A semantic code finder for Ciao Prolog

Isabel Garcia Contreras

Table of Contents

1 Summary

`Deepfind` is a bundle for the Ciao System which allows finding code in set of modules defined by the user by specifying abstract semantic characteristics of the arguments of the predicates.

2 Introduction

`Deepfind` is a bundle for the Ciao System which allows finding `Prolog` code in set of modules defined by the user by specifying abstract semantic characteristics of the arguments of the predicates.

2.1 Overview of this document

This document is divided in two parts:

- **Part I - User's Manual**. It explains the setup and usage of this bundle. DeepFind Usage details how to use this bundle and its options. Also, several execution examples are shown.
- **Part II - Reference Manual**. It details the Bundle structure and the specifications of all implemented modules.

3 PART I - User's Manual

Author(s): Isabel Garcia Contreras.
In this Manual we show how to install `Deepfind` and find code using it.

4 DeepFind installation

4.1 Quick installation

Requirements

1. Ciao and CiaoPP

4.2 Build and installation

You can automatically fetch, build, and install this bundle using:

```
ciao get ciao-lang.org/deepfind
```

This command stores the source and generates the binaries in the Ciao *workspace directory*. This directory is given by the value of the `CIAOPATH` environment variable (or `~/.ciao` if unspecified).

Binaries are placed in the `$CIAOPATH/build/bin` directory (or `~/.ciao/build/bin`). To call a binary without specifying its full path it is recommended to include this directory in your `PATH`:

```
export PATH=$CIAOPATH/build/bin:$PATH
# or export PATH=~/.ciao/build/bin:$PATH
```

4.3 Installation for developers

For installing this bundle it is recommended to define `CIAOPATH` (E.g., `~/ciao`) and clone this repository in your workspace.

```
git clone ssh://gitolite@ciao-lang.org/deepfind
```

Remember to update registered bundles after cloning

```
ciao rescan-bundles ~/ciao
```

5 DeepFind Usage

5.1 Load bundle modules

```
?- use_package(deepfind(find_syntax)).
?- use_module(deepfind(find)).
```

5.2 Set code location

By default, DeepFind will look in your ciao installation directory 'core/lib'. This can be changed with predicates for:

- Adding individual modules to search in:

```
?- find_add_mod('~/ciao-devel/core/lib/llists.pl').
```

- Setting a directory to search in recursively:

```
?- find_set_dir('~/ciao-devel/core/lib').
```

Once you have stablished in which directory you want to search, a pre-analysis has to be made filter modules which are not analyzable, i.e. it takes too much time or too much memory to get the results (otherwise the search could not finish).

This pre-analysis is made with an auxiliary bash script.

```
cd src/analysis
./batch_analyze [directory]
```

It can take a long time, depends of the quantity and complexity of the modules to be analyzed. A file will be created in data/ directory (mod_status.pl) with a list of modules in which the analysis is feasible.

5.3 Run a query

Find predicates that performs operations with lists

```
?- findp({ :- pred '_'(A,_,B) : list(A) => (list(A),list(B)). },P,R,S)
```

Some results:

```
P = idlists:substract/3,
Sum = checked,
Res = Explanation... ? ;

P = file_utils:stream_to_string/3,
Sum = false,
Res = Explanation... ? ;

P = set:ord_intersection/3,
Sum = check,
```

```
Res = Explanation... ? ;
```

If you want only the predicates that meet the conditions to be displayed, you can set the last argument to checked:

```
?- findp({ :- pred '_'(A,_,B) : list(A) => ((list(A),list(B)). },P,R,S),
    S = checked.
```

5.4 Combining with keyword search

Restrictions over predicate names can be made with `apropos` directive

```
?- findp({ As. :- apropos(Spec)}, P, Res, S).
```

Spec can be a regular expression or a keyword. Checking if the predicate name meets the requirements is made as in `apropos/1` in the Ciao System.

Example:

```
?- findp({ :- pred '_'/2 => term * string. :- apropos(write)}, P, Res, S).
```

5.5 Search options

There are two search options that can be chaged via flags:

- **allow_not_preanalyzed**
  - **on**: analyzes modules although they might have not been analyzed before
  - **off**: if a module wasn't marked as safe it is not analyzed

  This flag is useful if reanalysis is on, if not, preanalysis is needed for loading dumps.
- **reanalyze**
  - **on**: modules are analyzed before searching. SLOW but allows user-defined types.
  - **off**: information from dumps is loaded search in the module. QUICK but less flexible. Useful for roughly discard predicates.

5.6 Some examples

Examples can be found in `search_examples.pl`. This module can be loaded and used directly for using `DeepFind`.

For example:

```
?- search_demo(ID, _, Query, Comment, Explanation).

Comment = "Predicates that display strings",
Explanation = "We use a success assertion instead of a calls assertion
because code could lack of assertions and the condition would not be
checked. If the query requires this condition on exit, we avoid this
problem",
ID = strings,
Query = {:-pred P/1=>string} ?
```

Each example has:

- ID to differentiate it, all of them should be different.
- The arity of the query assertion.
- Query assertion.
- A short comment describing what kind of predicates the query is thought to gind.
- [Optional] An explanation of why this query is useful to find the code described in the comment.

Here are some examples:

```
:- module(search_examples, [search_demo/5], [assertions]).

:- use_package(deepfind(find_syntax)).
:- use_package(regtypes).

search_demo(
    strings, 1,
    { :- pred '_'/1 => string. },
    "Predicates that display strings",
    "We use a success assertion instead of a calls assertion
     because code could lack of assertions and the condition would
     not be checked. If the query requires this condition on exit,
     we avoid this problem").

search_demo(
    transform_list, 2,
    { :- pred '_'(A,B) : list(A) => (list(A), list(B)). },
    "Predicates that transform lists", _).

search_demo(
    insert_list, 3,
    { :- pred '_'(A,_,B) : list(A) => (list(A), list(B)). },
    "Predicates that insert elements in lists", _).

search_demo(
    op_list_num, 2,
    { :- pred '_'(A,B) : list(A, num) => num(B). },
    "Predicates that gather a list of numbers: sum, etc ...",
    "This search can be more general if the calls condition is moved
     to the success conditions").

search_demo(sort, 2,
    { :- pred '_'(A,B) : list(A) => list(B).
      :- pred '_'(A,B) : list(B) => list(A). },
    "Sorting predicate", _).

search_demo(
    graph_structure, 1,
    {
      :- use_package(regtypes).
```

```
      :- regtype math_graph/1.

      :- pred 'P'/2 => math_graph * term.

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

      :- regtype pair/1.
      pair((_,_)).
    },
    "Looking for graph structures",
    "We look for success because there may not exist calls. We expect
     that a graph is a structure with a list of vertex (V) and a list
     of edges (E). We dont specify the shape of the edges or vertex
     because we do not know how they are represented in the code."
).

search_demo(
    op_graph, 3,
    {
      :- use_package(regtypes).

      :- pred 'P'(A,B,C) : (my_ugraph(A),list(B),var(C))
                        => my_ugraph(C).

      :- regtype my_graph_elem/1.
      my_graph_elem(_Vertex-_Neighbors) :- list(_Neighbors).

      :- regtype my_ugraph(_).
      my_ugraph(A) :-
          list(A,my_graph_elem).
    },
    "Looking for predicates for editing graphs",
    "Given the ugraph A, we want to add B (list of edges or vertex)
     and C will be the new constructed graph. We require therefore
     that C is a free variable.  This query does not mark the calls
     condition as checked because the code does not have any calls
     written. However, even the assertion were written, it would not
     be checked because it does not work with combined domains."
).

search_demo(
    op_graph2, 3,
    {
      :- use_package(regtypes).

      :- pred 'P'(A,B,C) :
```

```
          ((nonvar(A), nonvar(B), var(C)) ; (my_ugraph(A), list(B)))
          => my_ugraph(C).

        :- regtype my_graph_elem/1.

        my_graph_elem(Vertex-Neighbors) :- list(Neighbors).

        :- regtype my_ugraph(_).
        my_ugraph(A) :- list(A,my_graph_elem).
      },
      "Predicates for editing graphs",
      "This query is the same as in op_graph but with calls splited in
       domains so once calls assertions are written they can be checked"
).
```

6 The Ciao library browser

Author(s): Angel Fernandez Pineda, Isabel Garcia Contreras.

The `librowser` library provides a set of predicates wich enable the user to interactively find Ciao libraries and/or any predicate exported by them.

This is a simple example:

```
?- apropos(aggregates:'.*find.*').

aggregates:findnsols/5
aggregates:findnsols/4
aggregates:findall/4
aggregates:findall/3

yes
?-
```

`librowser` is specially useful when inside GNU Emacs: just place the cursor over a librowser response and press C-cTAB in order to get help on the related predicate. Refer to the "**Using Ciao inside GNU Emacs**" chapter for further information.

6.1 Usage and interface

- **Library usage:**

  It is not necesary to use this library at user programs. It is designed to be used at the Ciao *toplevel* shell: `ciaosh`. In order to do so, just make use of `use_module/1` as follows:

  ```
  use_module(library(librowser)).
  ```

  Then, the library interface must be read. This is automatically done when calling any predicate at librowser, and the entire process will take a little moment.So, you should want to perform such a process after loading the Ciao toplevel:

  ```
  Ciao 0.9 #75: Fri Apr 30 19:04:24 MEST 1999
  ?- use_module(library(librowser)).

  yes
  ?- update.
  ```

  Whether you want this process to be automatically performed when loading `ciaosh`, you may include those lines in your *.ciaorc* personal initialization file.
- **Exports:**
  - *Predicates:*

    `update/0`, `browse/2`, `where/1`, `describe/1`, `system_lib/1`, `apropos/1`.
- **Imports:**
  - *System library modules:*

    `regexp_code`, `read`, `fastrw`, `system`, `streams`, `lists`, `fuzzy_search`, `pathnames`, `paths_extra`, `write`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`, `internals`.
  - *Packages:*

    `prelude`, `nonpure`, `condcomp`, `assertions`, `regexp`.

6.2 Documentation on exports

update/0: PREDICATE

This predicate will scan the Ciao system libraries for predicate definitions. This may be done once time before calling any other predicate at this library.

update/0 will also be automatically called (once) when calling any other predicate at librowser.

Usage:

Creates an internal database of modules at Ciao system libraries.

browse/2: PREDICATE

This predicate is fully reversible, and is provided to inspect concrete predicate specifications. For example:

```
?- browse(M,findall/A).

A = 3,
M = conc_aggregates ? ;

A = 4,
M = aggregates ? ;

A = 3,
M = aggregates ? ;

no
?-
```

Usage: `browse(Module,Spec)`

Asocciates the given `Spec` predicate specification with the `Module` which exports it.

– *The following properties should hold at call time:*

`Module` is a module name (an atom) (`module_name/1`)

`Spec` is a **Functor/Arity** predicate specification (`pred_spec/1`)

where/1: PREDICATE

This predicate will print at the screen the module needed in order to import a given predicate specification. For example:

```
?- where(findall/A).
findall/3 exported at module conc_aggregates
findall/4 exported at module aggregates
findall/3 exported at module aggregates

yes
?-
```

Usage: `where(Spec)`

Display what module to load in order to import the given `Spec`.

– *The following properties should hold at call time:*

`Spec` is a **Functor/Arity** predicate specification (`pred_spec/1`)

describe/1: PREDICATE

This one is used to find out which predicates were exported by a given module. Very usefull when you know the library, but not the concrete predicate. For example:

```
?- describe(librowser).
Predicates at library librowser :

apropos/1
system_lib/1
describe/1
where/1
browse/2
update/0

yes
?-
```

Usage: `describe(Module)`

Display a list of exported predicates at the given `Module`

– *The following properties should hold at call time:*

`Module` is a module name (an atom) (`module_name/1`)

system_lib/1: PREDICATE

It retrieves on backtracking all Ciao system libraries stored in the internal database. Certainly, those which were scanned at `update/0` calling.

Usage: `system_lib(Module)`

`Module` variable will be successively instantiated to the system libaries stored in the internal database.

– *The following properties should hold at call time:*

`Module` is a module name (an atom) (`module_name/1`)

apropos/1: PREDICATE

This tool makes use of regular expresions in order to find predicate specifications. It is very usefull whether you can't remember the full name of a predicate. Regular expresions take the same format as described in library `patterns`. Example:

```
?- apropos('write.').

write:writeq/1
write:writeq/2

yes
?- apropos('write.*'/2).

dht_misc:write_pr/2
profiler_auto_conf:write_cc_assertions/2
mtree:write_mforest/2
```

```
transaction_concurrency:write_lock/2
transaction_logging:write/2
provrml_io:write_vrml_file/2
provrml_io:write_terms_file/2
unittest_base:write_data/2
write:write_canonical/2
write:writeq/2
write:write/2
write:write_term/2
strings:write_string/2
res_exectime_hlm_gen:write_hlm_indep_each/2
res_exectime_hlm_gen:write_hlm_indep_2/2
res_exectime_hlm_gen:write_hlm_dep/2
oracle_calibration:write_conf/2
bshare_utils:write_string/2
bshare_utils:write_string_list/2
bshare_utils:write_length/2
bshare_utils:write_neg_db_stream/2
bshare_utils:write_neg_db/2
bshare_utils:write_pos_db/2

yes
```

When no predicates are found with the exact search, this predicate will perform a fuzzy search which will find predicates at a distance of one edit, swap, deletion or insertion.

```
?- apropos('wirte').
Predicate wirte not found. Similar predicates:

transaction_logging:write/2
write:write/1
write:write/2

yes
?- apropos(apend).
Predicate apend not found. Similar predicates:

hprolog:append/2
lists:append/3
llists:append/2

yes
?-
```

Usage: `apropos(RegSpec)`

This will search any predicate specification `Spec` which matches the given `RegSpec` incomplete predicate specification.

– *The following properties should hold at call time:*

`RegSpec` is a predicate specification `Pattern`, `Pattern/Arity`, `Module:Pattern`, `Module:Pattern/Arity`. (`apropos_spec/1`)

6.3 Documentation on internals

apropos_spec/1: REGTYPE

Defined as:

```
apropos_spec(Pattern) :-
        atm(Pattern).
apropos_spec(Pattern/Arity) :-
        atm(Pattern),
        int(Arity).
apropos_spec(Module:Pattern/Arity) :-
        atm(Pattern),
        atm(Module),
        int(Arity).
apropos_spec(Module:Pattern) :-
        atm(Pattern),
        atm(Module).
```

Usage: `apropos_spec(S)`

`S` is a predicate specification `Pattern`, `Pattern/Arity`, `Module:Pattern`, `Module:Pattern/Arity`.

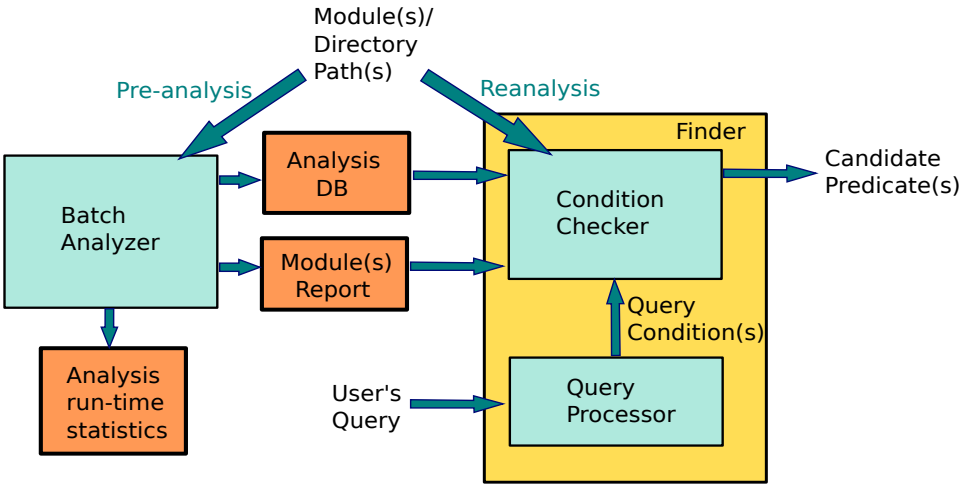
7 PART II - Reference Manual

Author(s): Isabel Garcia Contreras.

This manual details structure of the `Deepfind` bundle and the documentation of all modules involved.

8 Bundle structure

The essential modules for finding code are:



- **Batch Analyzer**: This module performs the static analysis of a list of modules or the modules located in a user-specified path. Modules are analyzed individually and the analysis trusts the assertions for exported and imported predicates. The analysis results are cached on disk (as CiaoPP `dump` files) and reused for each search. Also, a module report can be generated with analysis memory and time data, and timeout information.
- **Query Processor**: Preprocesses user's queries, i.e., normalizes the anonymous assertions, extracts the assertion conditions and stores them so they can be checked later. It extracts also predicate name requirements.
- **Condition Checker**: Its task is to prove that the conditions in the query hold or not for all predicates that potentially could meet the requirementes, i.e., those which have the same arity as specified in the query.

Other modules have been implemented that complement the previous ones:

- `DeepFind` **pretty printer**: Transform the prolog output of the query to html to improve results visualization.
- `DeepFind` **benchmark**: Generates statistics about time spent checking conditions.
- **Statistics Collecter**: gathers the output from the Batch Analyzer and summarizes it.

9 Finding code

Author(s): Isabel Garcia Contreras.

This module contains the main functionality of `DeepFind`.

`findp/4` is the predicate used for finding code in some determined modules. This modules are specified with `find_add_mod/1` and `find_set_dir/1`.

Phases of `findp/4`:

- **Process query**. The information in the query is extracted:
  - Query assertions.
  - Keyword requirements (specified via `:- apropos(String).` directives).
  - User-defined regular types.
- **Check conditions against code**, for each module:
  - Get predicates with the specified arity.
  - Filter predicates with keyword requirements.
  - Restore analysis/ make analysis.
  - Check conditions for the filtered predicates.

During this whole process, time and memory statistics are collected, and can be accessed with `dump_size/1`, `restore_time/1` and `check_time/1`.

9.1 Usage and interface

- **Library usage:**

  `:- use_module(deepfind(find)).`
- **Exports:**
  - *Predicates:*

    `findp/4`, `find_set_dir/1`, `find_add_mod/1`, `find_remove_mod/1`, `dump_size/1`, `restore_time/1`, `check_time/1`, `clean_search/0`, `clean_search_opts/0`, `display_search_stats/0`, `set_find_flag/2`, `set_pred_class/1`.
- **Imports:**
  - *Application modules:*

    `assertion_spec`, `file_collecter`, `find_aux`, `main_analysis`, `db_analysis`.
  - *System library modules:*

    `regexp_code`, `api_direc_assrt`, `driver`, `preprocess_flags`, `itf_db`, `aggregates`, `lists`, `p_dump`, `pathnames`, `prolog_sys`, `read`, `streams`, `fastrw`, `format`, `system`, `fuzzy_search`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*

    `prelude`, `nonpure`, `condcomp`, `assertions`, `argnames`, `regexp`, `api(api_internal_dec)`.

9.2 Documentation on exports

findp/4: PREDICATE

Usage: `findp(+Assertions,-Pred,-Residue,Status)`

Finds a predicate matching `Assertions`, `Pred` is the predicate that matches, `Residue` is how well it suits the assertions and `Status` is a summary of the residue

- *The following properties should hold upon exit:*

  `Status` is an atom. (`atm/1`)

find_set_dir/1: PREDICATE

Usage: `find_set_dir(Path)`

Sets a directory path within which code will be found.

- *The following properties should hold at call time:*

  `Path` is an atom. (`atm/1`)

find_add_mod/1: PREDICATE

Usage: `find_add_mod(pl(Module,ModulePath))`

Adds to the module search set within which code will be checked.

– *The following properties should hold at call time:*

`Module` is an atom. (`atm/1`)

`ModulePath` is an atom. (`atm/1`)

find_remove_mod/1: PREDICATE

Usage: `find_remove_mod(pl(Module,ModulePath))`

Removes a `Module` from the search.

– *The following properties should hold at call time:*

`Module` is an atom. (`atm/1`)

dump_size/1: PREDICATE

Usage: `dump_size(S)`

`S` is the amount of memory (in B) needed to store an analysis dump.

– *The following properties should hold upon exit:*

`S` is a number. (`num/1`)

The predicate is of type *data*.

restore_time/1: PREDICATE

Usage: `restore_time(T)`

`T` is the time spent restoring modules.

– *The following properties should hold upon exit:*

`T` is a number. (`num/1`)

The predicate is of type *data*.

check_time/1: PREDICATE

Usage: `check_time(T)`

`T` is the time spent checking conditions.

– *The following properties should hold upon exit:*

`T` is a number. (`num/1`)

The predicate is of type *data*.

clean_search/0: PREDICATE

Usage:

Removes data and assertions from the previous analysis

clean_search_opts/0: PREDICATE

Usage:

Reset search options

display_search_stats/0: PREDICATE

Usage:

Displays the list of individual modules within which code is to be found

set_find_flag/2: PREDICATE

Usage: `set_find_flag(Flag,Status)`

sets a search option: allow*not*preanalyzed or reanalyze. `Status` can be on/off

- *The following properties should hold at call time:*

  `Flag` is an atom. (`atm/1`)

  `Status` is an atom. (`atm/1`)

set_pred_class/1: PREDICATE

Usage: `set_pred_class(C)`

Select predicate class `C` for searching, i.e., exported, defined... (exported by default).

- *The following properties should hold at call time:*

  `C` is an atom. (`atm/1`)

10 Normalizing the query

Author(s): Isabel Garcia.

Save set of assertions (specifications) in a query as temporary modules.

Asserts data of predicate name requirements made by `:- apropos(Name).` directive.

Example:

```
?- use_package(deepfind(find_syntax)).
?- save_assertions({ :- pred ''(X) => int(X). :- apropos(length). }, M).

M = '/tmp/modROKkyL' ?
```

The generated file is `/tmp/modROKkyL.pl`:

```
:-module(_,['Σpvar'/1],[assertions]).
:- pred 'Σpvar'(_A)
        => int(_A).

:-impl_defined('Σpvar'/1).
```

10.1 Usage and interface

- **Library usage:**

  `:- use_module(deepfind(assertion_spec)).`
- **Exports:**
  - *Predicates:*

    `save_assertions/2`, `keyword_spec/1`.
- **Imports:**
  - *Application modules:*

    `tmp_mod`.
  - *System library modules:*

    `assrt_lib`, `write`, `dec10_io`, `assrt_write`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*

    `prelude`, `nonpure`, `condcomp`, `doccomments`, `assertions`, `isomodes`.

10.2 Documentation on exports

save_assertions/2: PREDICATE

Usage: `save_assertions(Curly,File)`

Saves a set of assertions (curly block) into a temporary `File`.

- *The following properties should hold at call time:*

  `Curly` is currently a term which is not a free variable. (`nonvar/1`)

  `File` is a free variable. (`var/1`)

- *The following properties should hold upon exit:*

  `File` is an atom. (`atm/1`)

keyword_spec/1: PREDICATE

Usage: `keyword_spec(Expression)`

`Expression` is a regexp or a word that the found predicate name has to meet.

The predicate is of type *data*.

11 Temporary modules generation

Author(s): Isabel Garcia Contreras.

This module generates a module file in /tmp/ containing user-specified clauses.

When no module clause is specified, it is automatically generated exporting a given list of predicates and importing no packages.

The module name is randomly generated of the form modXXXXXX where each X is a random character or number verifying that there is no existing file with such name.

Example:

```
?- tmp_mod([(p(A) :- q(A)), (q(b))], [p], X).
X = '/tmp/modbv5UNt' ?
```

Module /tmp/modbv5UNt.pl:

```
:-module(_,[p],[]).
p(A) :-
        q(A).
q(b).
```

11.1 Usage and interface

- **Library usage:**

  `:- use_module(deepfind(tmp_mod)).`
- **Exports:**
  - *Predicates:*

    `tmp_mod/3`, `clean_tmp_mod/1`, `clean_all_tmp_mods/0`.
- **Imports:**
  - *System library modules:*

    `system`, `system_extra`, `write`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*

    `prelude`, `nonpure`, `condcomp`, `assertions`.

11.2 Documentation on exports

tmp_mod/3: PREDICATE

Usage: `tmp_mod(Clauses,ExportedPreds,FileBase)`

This predicate writes a list of clauses in a file with random name in /tmp.

– *The following properties should hold at call time:*

`Clauses` is a list. (`list/1`)

`ExportedPreds` is a list. (`list/1`)

`FileBase` is a free variable. (`var/1`)

– *The following properties should hold upon exit:*

`Clauses` is a list. (`list/1`)

`ExportedPreds` is a list. (`list/1`)

`FileBase` is an atom. (`atm/1`)

clean_tmp_mod/1: PREDICATE

Usage: `clean_tmp_mod(X)`

Removes a previously created tmp file and its associated files

– *The following properties should hold at call time:*

`X` is an atom. (`atm/1`)

clean_all_tmp_mods/0: PREDICATE

Usage:

Removes all the temporary files created by tmp_mod

12 Auxiliary predicates

Author(s): Isabel Garcia Contreras.

Auxiliary, generic predicates for DeepFind.

12.1 Usage and interface

- **Library usage:**
  `:- use_module(deepfind(find_aux)).`
- **Exports:**
  - *Predicates:*
    `dump_file/3`, `error_file/3`, `newer/2`, `string_contained/2`.
- **Imports:**
  - *System library modules:*
    `system`, `pathnames`, `lists`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `condcomp`, `assertions`.

12.2 Documentation on exports

dump_file/3: PREDICATE

Usage: `dump_file(FilePath,Module,DumpFile)`

Given a `FilePath` and `Module` name, generates its associated `DumpFile`

- *The following properties should hold at call time:*

  `FilePath` is an atom. (`atm/1`)

  `Module` is an atom. (`atm/1`)

  `DumpFile` is a free variable. (`var/1`)

error_file/3: PREDICATE

Usage: `error_file(FilePath,Module,ErrFile)`

Given a `FilePath` and `Module` name, generates its associated `ErrFile`

- *The following properties should hold at call time:*

  `FilePath` is an atom. (`atm/1`)

  `Module` is an atom. (`atm/1`)

  `ErrFile` is a free variable. (`var/1`)

newer/2: PREDICATE

Usage: `newer(FileA,FileB)`

`FileA` was modified later than `FileB`.

– *The following properties should hold at call time:*

`FileA` is currently instantiated to an atom. (`atom/1`)

`FileB` is currently instantiated to an atom. (`atom/1`)

string_contained/2: PREDICATE

Usage: `string_contained(L1,L2)`

`L1` is a sublist of `L2`.

– *The following properties should hold at call time:*

`L1` is a string (a list of character codes). (`string/1`)

`L2` is a string (a list of character codes). (`string/1`)

13 Analyzing code

Author(s): Isabel Garcia Contreras.

Ciaopp summarizer takes as an input a set of directories. It analyzes all prolog modules recursively contained in the directories and generates run time statistics.

13.1 Usage

This tool has to be used with an external timeout because the analysis of some modules could require too much memory or time to be performed.

After executing analyze_all/1, control.sh has to be started.

13.2 Generated files

- Analysis information: For each module abstract analysis information is stored in a `.dump` file generated in the same location. This information and can be restored later.
- Run-time statistics: Statistics of time and memory used are stored in as a term in a `.err` file in the same location as the original module. This file contains also the output of the analyzer.
- Status of the analysis: For each module, information of load and analysis success is stored in `data/mod_status.pl`.

  This allows also the script to be incremental, i.e., it does not repeat ciaopp analysis for a module if it has already been done.

  If the the user wants the tool to redo an analysis for all files, `mod_status.pl` has to be removed before starting.
- `last_analyzed_file.pl`: This file contains the file that is being analyzed.

13.3 Usage and interface

- **Library usage:**
  `:- use_module(deepfind(analysis/main_analysis)).`
- **Exports:**
  - *Predicates:*
    `analyze_all/1`, `safe/1`, `any_module/1`.
- **Imports:**
  - *Application modules:*
    `file_collecter`, `find_aux`, `db_analysis`.
  - *System library modules:*
    `lists`, `write`, `pathnames`, `aggregates`, `system`, `system_extra`, `port_reify`, `streams`, `io_alias_redirection`, `file_utils`, `driver`, `preprocess_flags`, `p_dump`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `condcomp`, `assertions`, `regtypes`.

13.4 Documentation on exports

analyze_all/1: PREDICATE

Usage: `analyze_all(Paths)`

Analyzes all prolog module files in a list of `Paths` if the have not been already analyzed.

- *The following properties should hold at call time:*
  `Paths` is a list of `atm`s. (`list/2`)

safe/1: PREDICATE

Usage: `safe(Module)`

A `Module` is safe if it could be loaded and analyzed by ciaopp.

- *The following properties should hold upon exit:*
  `Module` is an atom. (`atm/1`)

any_module/1: PREDICATE

Usage: `any_module(Module)`

`Module` is a prolog module.

– *The following properties should hold upon exit:*
`Module` is an atom. (`atm/1`)

14 Collecting prolog files

Author(s): Isabel Garcia Contreras.

`related_files_at` is a predicate originally located in library/librowser modified to get ciao modules, i.e., `.pl` files that have an associate `.itf` file

14.1 Usage and interface

- **Library usage:**
  `:- use_module(deepfind(analysis/file_collecter)).`
- **Exports:**
  - *Predicates:*
    `related_files_at/3`.
- **Imports:**
  - *Application modules:*
    `main_analysis`.
  - *System library modules:*
    `driver`, `lists`, `pathnames`, `system`, `dec10_io`, `write`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `condcomp`, `assertions`, `hiord`.

14.2 Documentation on exports

related_files_at/3: PREDICATE

Usage: `related_files_at(Dir,ModType,Files)`

`Files` is a list of Modules of type `ModType` and their path located recursively in `Dir`

- *The following properties should hold at call time:*
  `Dir` is an atom. (`atm/1`)
- *The following properties should hold upon exit:*
  `Files` is a list. (`list/1`)

Meta-predicate with arguments: `related_files_at(?,pred(1),?)`.

15 Collecting statistics

Author(s): Isabel Garcia Contreras.

Collects statistics from `.err` files created while analyzing and summarizes them to produce an output in html or tex format.

15.1 Usage and interface

- **Library usage:**
  `:- use_module(deepfind(statistics_collecter)).`
- **Exports:**
  - *Predicates:*
    `collect_all_stats/1`, `set_flag_restore_time/1`, `stat_collecter_set_flag/2`.
- **Imports:**
  - *Application modules:*
    `find_aux`.
  - *System library modules:*
    `lists`, `pathnames`, `system`, `format`, `streams`, `write`, `paths_extra`, `read`, `p_dump`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `condcomp`, `assertions`.

15.2 Documentation on exports

collect_all_stats/1: PREDICATE

Usage: `collect_all_stats(Paths)`

Collects the time and memory statistics from .err files in a list of `Paths`

- *The following properties should hold at call time:*
  `Paths` is a list. (`list/1`)

set_flag_restore_time/1: PREDICATE

Usage: `set_flag_restore_time(Status)`

Predicate for setting the flag to collect statistics of restoring time

- *The following properties should hold at call time:*
  `Status` is an atom. (`atm/1`)

stat_collecter_set_flag/2: PREDICATE

Usage: `stat_collecter_set_flag(Flag,Status)`

Predicate for changing flags of the output. The available flags are:

- output. Modify output format; html or tex.
- show_errors. on: display also modules that have errors while analyzing
- show_timeouts. on: display also modules that did not finish before a timeout.

– *The following properties should hold at call time:*

`Flag` is an atom. (`atm/1`)

`Status` is an atom. (`atm/1`)

16 DeepFind UI

Author(s): Isabel Garcia Contreras.

This module executes a `findp` query with some parameters already set, in order to make it simpler.

The result of the search is shown in a `html` file with some explanations and links to module/predicate documentation.

16.1 Usage and interface

- **Library usage:**
  `:- use_module(deepfind(pretty_finder)).`
- **Exports:**
  - *Predicates:*
    `search/1`, `checked_search/1`.
- **Imports:**
  - *Application modules:*
    `find`, `tmp_mod`.
  - *System library modules:*
    `http`, `html`, `streams`, `assrt_write`, `assrt_db`, `aggregates`, `system`, `system_extra`, `file_utils`, `write`, `paths_extra`, `lists`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `condcomp`, `assertions`, `argnames`, `regtypes`, `pillow`, `api(api_internal_dec)`.

16.2 Documentation on exports

search/1: PREDICATE

Usage: `search(As)`

Displays information about how predicates meet the conditions specified in the `As` query in a predefined path.

The results are shown in html format in `data/query_report.html`.

- *The following properties should hold at call time:*
  `As` is currently a term which is not a free variable. (`nonvar/1`)

checked_search/1: PREDICATE

Usage: `checked_search(As)`

Displays only predicates that meet the conditions specified in `As` query in a predefined path.

The results are shown in html format in `data/query_report.html`.

– *The following properties should hold at call time:*

`As` is currently a term which is not a free variable. (`nonvar/1`)

16.3 Documentation on internals

assertion_status/1: PREDICATE

Usage 1: `assertion_status(Status)`

This type represents all possible status of proof of an assertion

- `Status` = (true ; checked). The conditions could be proved to hold.
- `Status` = false. Conditions were proved to be false.
- `Status` = check. Neither false nor true could be proved.

17 Benchmarking DeepFind

Author(s): Isabel Garcia Contreras.

Module for testing time execution of queries with DeepFind

17.1 Usage and interface

- **Library usage:**

  `:- use_module(deepfind(find_test)).`
- **Exports:**
  - *Predicates:*

    `main_test/0`.
- **Imports:**
  - *Application modules:*

    `find`.
  - *System library modules:*

    `format`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*

    `prelude`, `nonpure`, `condcomp`, `assertions`, `deepfind(find_syntax)`.

17.2 Documentation on exports

main_test/0: PREDICATE

Usage:

Executes a set of queries defined in test_db.pl and generates a table in tex format with total and avg times of execution

17.3 Documentation on internals

summarize_search/1: PREDICATE

Usage: `summarize_search(As)`

Collects run-time data of the execution of the query given the assertion query `As`

- *The following properties should hold at call time:*

  `As` is currently a term which is not a free variable. (`nonvar/1`)

7

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 "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.

7.1 Future Work

Although our prototype is fully functional, there are many improvements that can be made, in terms of search accuracy, usability, and performance.

Using the generic pre-analysis information is useful for quickly checking conditions but, as we have seen, in some cases it is not precise enough. In the near future, we want to automate reanalysis for specific entries. Abstractly executing a predicate with the query calls substitution would highly improve the number of cases in which the system is able to prove whether a predicate meets or not some properties. On the other hand, this would also make the search process slower and this trade-off needs to be studied. This reanalysis can also be refined by performing an inter-modular analysis, i.e., to also execute abstractly for the specific calls substitutions the imported predicates.

An obvious area of future research is to explore the many other abstract domains and combinations available in the CiaoPP system and the literature. This will allow us to search based on shape isomorphism, determinism, code complexity, arithmetic relations among arguments, etc. Using these domains could improve the precision of the search also.

In the current version of the prototype, the order of the arguments affects the results of the query. Checking all possible combinations would presumably increase the cost of the checking algorithm. We would like also to develop our theory to consider partial evaluation of the code as a way to find the desired functionality.

Indexing the inferred properties could be a first solution to this problem (and would in fact be useful overall). However, there is no trivial general way to index these properties because they are domain dependent and we are not just looking for equality but also for subsumption.

In order to make this prototype more accessible, we want to improve our simple graphical interface to make it interactive. This includes for example connecting the UI module to LPdoc to offer information about the code in various formats as well as the source code (and its assertions), and also making the explanations provided by the system more detailed, so that the user can have a better idea of why the code meets or not his/her specifications.

In the medium term, we would like to enhance our prototype by extending the query language to allow the specification of test cases or execution examples; and by generalizing code analysis to other programming languages and combinations, taking advantages of the capacity of CiaoPP.

Finally, we would like to explore other applications of the notion of semantic search such as for example finding duplicated code and code synthesis.

Bibliography

[1] M. Bruynooghe. "A Practical Framework for the Abstract Interpretation of Logic Programs". In: *Journal of Logic Programming* 10 (1991), pp. 91–124.

[2] F. Bueno et al. "A Model for Inter-module Analysis and Optimizing Compilation". In: *Logic-based Program Synthesis and Transformation*. LNCS 2042. Springer-Verlag, 2001, pp. 86–102.

[3] D. Cabeza and M. Hermenegildo. *A New Module System for Prolog*. Technical Report CLIP8/99.0. Facultad de Informática, UPM, 1999.

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

[5] Fred J Damerau. "A technique for computer detection and correction of spelling errors". In: *Communications of the ACM* 7.3 (1964), pp. 171–176.

[6] *Fuzzy string search*. `http://ntz-develop.blogspot.com.es/2011/03/fuzzy-string-search.html`. Accessed: 2016-02-30.

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

[8] *Hamming distance*. `https://en.wikipedia.org/wiki/Hamming_distance`. Accessed: 2016-05-30.

[9] M. Hermenegildo. *A Documentation Generator for Logic Programming Systems*. Technical Report CLIP10/99.0. Facultad de Informática, UPM, 1999.

[10] M. Hermenegildo et al. "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)". In: *Science of Computer Programming* 58.1–2 (2005), pp. 115–140. ISSN: ISSN 0167-6423. DOI: `10.1016/j.scico.2005.02.006`. URL: `http://dx.doi.org/10.1016/j.scico.2005.02.006`.

[11] M. V. Hermenegildo et al. "An Overview of Ciao and its Design Philosophy". In: *Theory and Practice of Logic Programming* 12.1–2 (2012). http://arxiv.org/abs/1102.5497, pp. 219–252. DOI: `doi:10.1017/S1471068411000457`.

[12] M. V. Hermenegildo et al. "An Overview of Ciao and its Design Philosophy". In: *TPLP* 12.1–2 (2012). http://arxiv.org/abs/1102.5497, pp. 219–252.

[13] *Levenshtein distance*. `https://en.wikipedia.org/wiki/Levenshtein_distance`. Accessed: 2016-05-30.

[14] Yoëlle S Maarek, Daniel M Berry, and Gail E Kaiser. "An information retrieval approach for automatically constructing software libraries". In: *Software Engineering, IEEE Transactions on* 17.8 (1991), pp. 800–813.

[15] Collin McMillan et al. "Recommending source code for use in rapid software prototypes". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 848–858.

[16] Neil Mitchell. "Hoogle Overview". In: *The Monad.Reader* 12 (2008), pp. 27–35.

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

[18] K. Muthukumar and M. Hermenegildo. "Compile-time Derivation of Variable Dependency Using Abstract Interpretation". In: *Journal of Logic Programming* 13.2/3 (1992). Ed. by S. Debray, pp. 315–347.

[19] G. Puebla, F. Bueno, and M. Hermenegildo. "An Assertion Language for Constraint Logic Programs". In: *Analysis and Visualization Tools for Constraint Programming*. LNCS 1870. Springer-Verlag, 2000, pp. 23–61.

[20] G. Puebla, F. Bueno, and M. Hermenegildo. *An Assertion Language for Debugging of Constraint Logic Programs*. Technical Report CLIP2/97.1. Facultad de Informática, UPM, 1997. URL: `ftp://cliplab.org/pub/papers/assert\_lang\_tr\_discipldeliv.ps.gz`.

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

[22] G. Puebla and M. Hermenegildo. "Abstract Multiple Specialization and its Application to Program Parallelization". In: *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs* 41.2&3 (1999), pp. 279–316.

[23] Steven P Reiss. "Semantics-based code search". In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society. 2009, pp. 243–253.

[24] Eugene J Rollins and Jeannette M Wing. "Specifications as Search Keys for Software Libraries." In: *ICLP*. Citeseer. 1991, pp. 173–187.

[25] N. Stulova, J. F. Morales, and M. V. Hermenegildo. "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, 2014.

[26] C. Vaucheret and F. Bueno. "More Precise yet Efficient Type Inference for Logic Programs". In: *SAS'02*. LNCS 2477. Springer, 2002, pp. 102–116. ISBN: ISBN 3-540-44235-9.

Appendices

A

Example code

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).
% ...
```

Figure A.1: Fragment from `ugraphs.pl` (Ciao library).

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).
```

Figure A.2: `named_graphs.pl` (Ciao library)

B

Additional tables

Table B.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

(continued in next page)

Table B.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**
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
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

(continued in next page)

Table B.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**
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 B.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
iso_char	26,702	300
foreign_compilation	23,623	276

(continued in next page)

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

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

(continued in next page)

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

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