Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Matemáticas e Informática

Trabajo Fin de Grado

Análisis Automático de Ejemplos de Código

Autor: Daniela Ferreiro de Aguiar
Tutor: Manuel Hermenegildo Salinas

Madrid, Julio 2021

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Matemáticas e Informática

Título: Análisis Automático de Ejemplos de Código

Julio 2021

Autor: Daniela Ferreiro de Aguiar
Tutor: Manuel Hermenegildo Salinas
Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Los cuadernos computacionales son unas de las herramientas más utilizadas en el ámbito de la investigación para documentar resultados, hallazgos o software. Esto es debido a que admite, en un mismo documento, combinar texto, imágenes, código, etc., permitiendo así generar tutoriales u otros materiales didácticos con diferentes ejemplos y mantenerlos actualizados.

Siguiendo esta linea, en el presente documento presentamos un método con características similares a los cuadernos computacionales, con el propósito de mantener los tutoriales del lenguaje de programación Ciao Prolog actualizados y sin elementos obsoletos. Para ello se ha implementado la herramienta **Exfilter**. Esta herramienta es capaz de analizar un determinado código con CiaoPP, el prepocesador de Ciao, y aplicarle uno o varios filtros con el fin de obtener el resultado deseado. Además, para facilitar la inclusión de estos resultados en los tutoriales hemos implementado un plugin para LPdoc.

Para comprobar su funcionamiento y eficacia presentaremos resultados experimentales del uso de la herramienta en tutoriales reales, así como diferentes ejemplos para explicar su uso. Además, al final de este documento se adjuntará el manual de la herramienta.

Abstract

Computer notebooks are one of the most widely used tools in the field of research to document results, discoveries or software. This is because they can combine text, images, code, etc. in the same document. This allows users to generate tutorials or other didactic materials with different examples and to keep them updated.

Following this line, in the present document we present a method with similar features to computer notebooks, with the aim of keeping the tutorials of the Ciao Prolog programming language updated and without obsolete elements. For this purpose, the tool **Exfilter** has been implemented. This tool is able to analyse a given code with CiaoPP, the Ciao preprocessor, and apply one or more filters to it in order to obtain the desired result. In addition, to facilitate the inclusion of these results in the tutorials, we have implemented a plugin for LPdoc.

In order to demonstrate its effectiveness and efficiency, we will present experimental results of the use of the tool in real tutorials as well as different examples to explain its use. At the end of this document we will attach the manual of the tool.

Contents

Chapter 1

Introduction

It is vital that manuals and tutorials provide readers with sources and descriptions as well as feature examples, allowing users to have a better understanding of the concepts explained. However, it is often the case that these examples are not up to date. For this reason, tutorials and manuals need to be updated, incorporating the latest modifications, while eliminating obsolete items.

One way to deal with this is maintaining the documentation and the tutorials manually. However, the biggest issue with this is that every change in the system can end up with obsolete documentation due to the large number of updates. Moreover, it is a time-consuming process.

In this thesis we propose to automate the aforementioned process as much as possible. The main goal is to design a system capable of automating the different actions necessary for the maintenance of documents and tutorials of the Ciao [10] language analysis and, at the same time, design a method to incorporate the analized results (or projections of them) in documentation.

To achieve these goals, we have developed a tool that can be divided into the following three stages:

1. Collection of codes: The examples we want to show in the manuals and tutorials. The examples will be written in Ciao Prolog using its assertions library.

2. Analyze: run CiaoPP on the collection of codes and extract and filter the results. CiaoPP [13, 11, 14, 3] is the abstract interpretation-based preprocessor of the Ciao multi-paradigm program development environment. CiaoPP can perform a number of program debugging, analysis, and source-to-source transformation tasks on (Ciao) Prolog programs.

3. Generate manuals and tutorials with the extracted fragments of the output: the analysis results wil be added in documentation.

Additionally, we aim to test this tool on real tutorials and prove that it works properly.

The rest of the document is structured as follows: Section 2.1 presents some similar tools to undestand the purpose of our work and Section 2.3 introduces briefly the assertion language. Section 3 describes our tool, as well as the steps followed for its development and design. Section 4 evaluates the suitability by showing examples from real tutorials. Moreover, we will discuss and compare the results. Section 5 summarizes the conclusions and suggests some possible new lines of study. Finally, Section 6 analyses the impact of our work.

Chapter 2

Background and State of the Art

A few modifications or improvements in code can lead to vastly different results. For this reason, some tools are created to provide an oppotunity for researchers to generate and keep up to date tutorials, manuals or other teaching aids for their software. One of the most commonly used tools are computational notebooks. We have been inspired by them to develop a method with similiar features.

In this chapter, we will show some examples of computational notebooks, as well as the similarities they have with the tool we have developed. On the other hand, examples from the manuals will be written in Ciao Prolog using its assertions library. Section 2.3 introduces briefly the assertion language.

2.1 Computational notebook

A computational notebook or notebook interface is a computer file which combines executable code, images, and text in a single document, yielding what Stephen Wolfram has called a "computational essay" [23].

Notebooks are useful for the ease of including code, as well as generating and recording results. In addition, they are version control tools. Software bugs are an inevitable part of any programmer's development process. This situation enables computational notebooks to determinate when a bug was discovered or when it was fixed.

Nowadays there are a lot of different kinds of computational notebooks for many different programming languages with their own elements but with some common general features.

Although the idea of computational notebook is not new, MathCAD and Wolfram Mathematica were the first to introduce this concept in 1987 and 1988 respectively, it was popularized by other open sources varieties such as Jupyter Notebook or Emacs Org-Babel mode.

2.1.1 Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows creating and sharing documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more.

The Jupyter Notebook was heavily inspired by the Mathematica notebooks just like some other tools that followed. Despite its similarities, in the case of Mathematica [22], its main focus is computational mathematics. On the other hand, Jupyter is more flexible and it can be used for a wide variety of purposes as supports more than 40 programming languages.

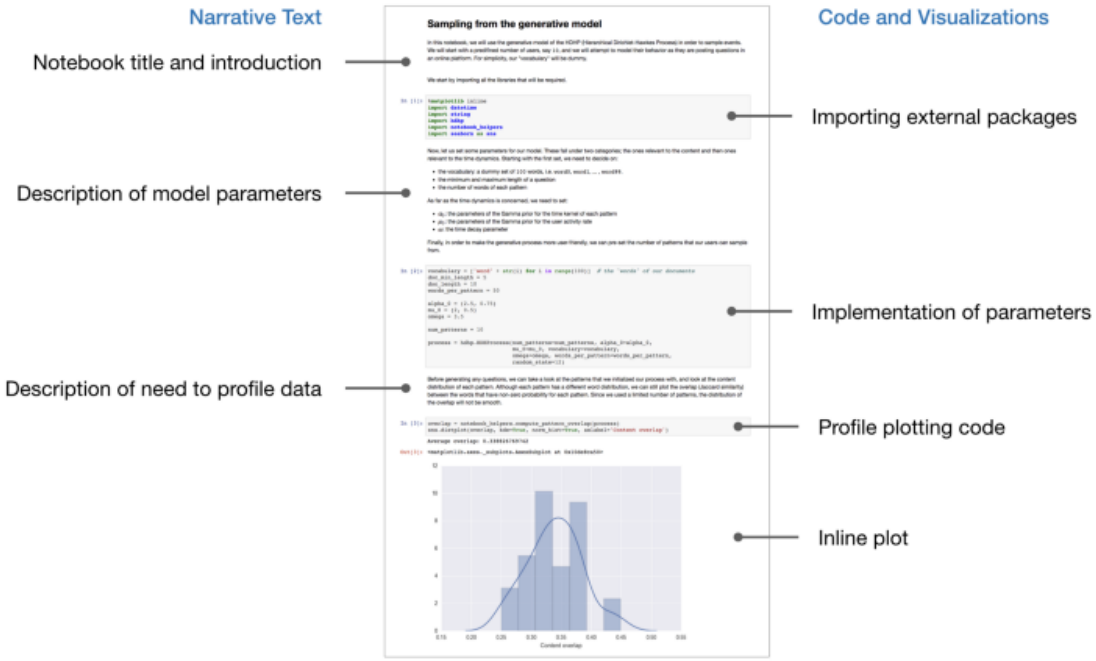


Figure 2.1: Example Jupyter Notebook

The Jupyter notebook [20] straddles these elements:

- A web application: a browser-based tool for interactive authoring of documents that allows creating and sharing documents that incluce code, visualizations and narrative text as seen in Figure 2.1, combining code cells and intermediate results.
- Notebook documents: a representation of all content visible in the web application.

Although the documents generated by Jupyter Notebook are easily accessible through a browser, it is sometimes useful to have other formats available. That is why it provides conversion to static documents, i.e. HTML, LaTeX or Markdown formats, etc.

2.1.2 Emacs Org-Babel mode

Babel [6] is Org-mode's ability to execute source code within Org-mode documents and allowing many different languages work together.

Babel augments Org-mode [19] with support for code blocks by providing:

- Interactive and on-export execution of code blocks.
- Code blocks as functions that can be parameterised, refer to other code blocks, and be called remotely.
- Export to files.

We can see these features in the following Figure 2.2, where a piece of code can pass from an external data source to a Python code block, and then move on to an R code block.

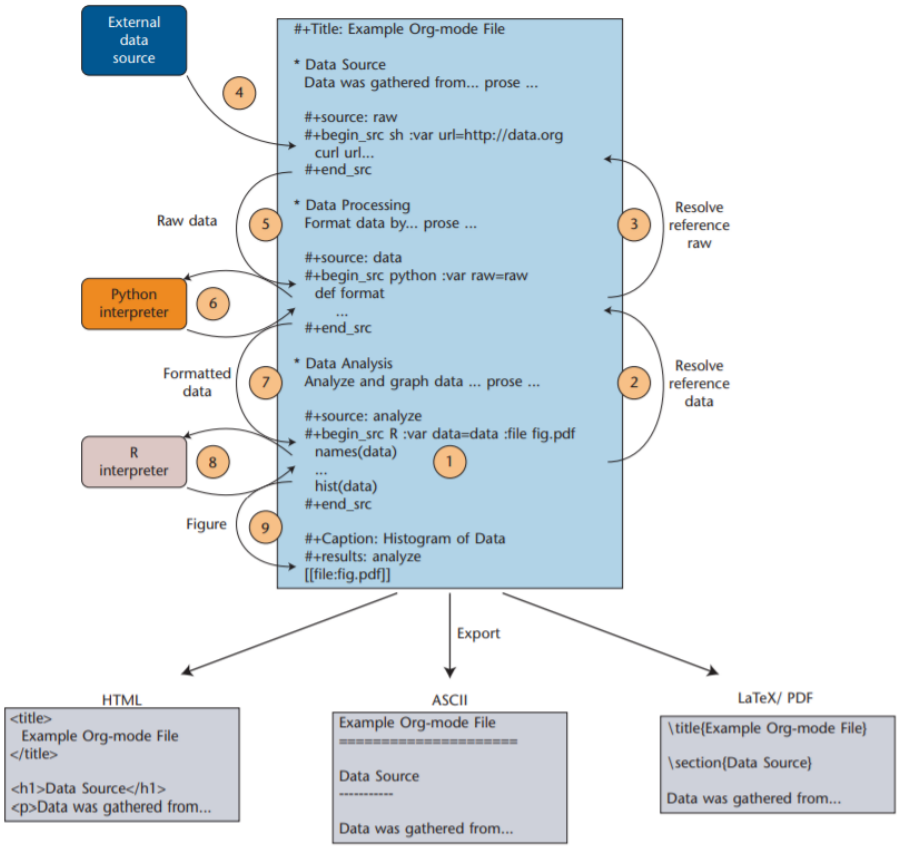


Figure 2.2: Active Org-mode document [18]

Finally, the document will be embedded into the exported document (HTML, ASCII, LaTeX, or another Org-mode supported format).

2.2 Exfilter Tool

Before Exfilter, tutorial examples, which were fragments of analysis results, were manually inserted into the document. Motivated by this problem, we have implemented a mechanism in order to insert the code fragments automatically generated by running our tool. Except for the lack of interactivity, because the example files are not automatically generated by LPdoc,[1] it is very similiar to notebook interfaces.

Figure 2.3 shows the scheme we have used to generate the tutorials (in our case, all tutorials are generated with LPdoc using Ciao). We will explain the process briefly below, as each part will be covereed in the following sections.

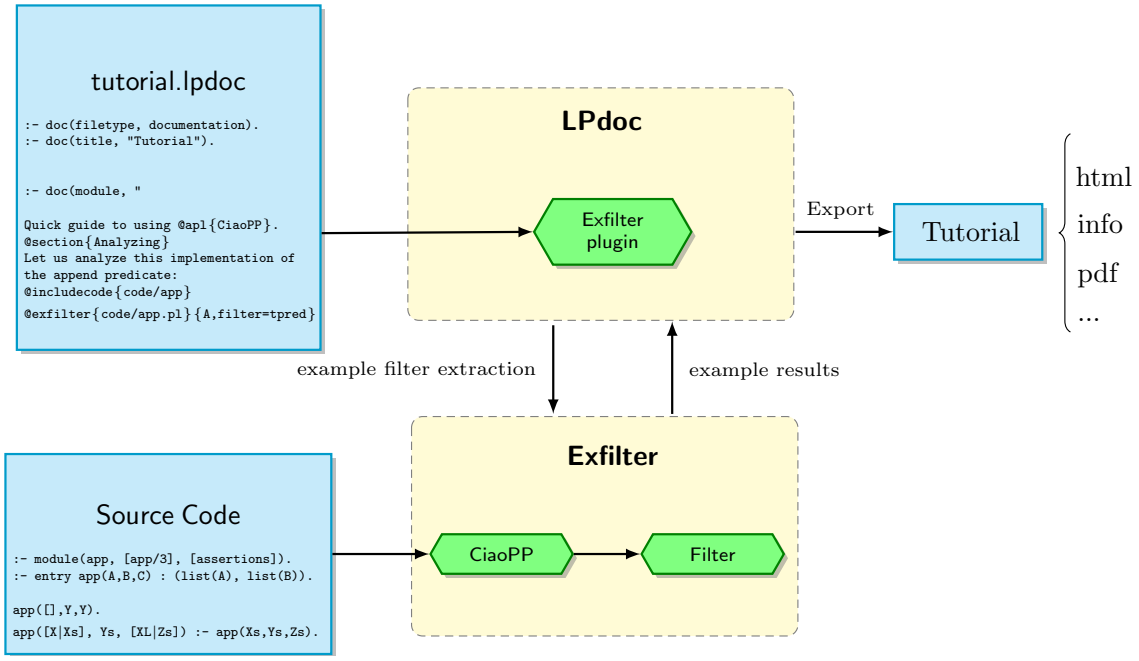


Figure 2.3: Exfilter process

The process works as follows: While we are writing the tutorial, if we want to show an example (e.g. the result of analysis of a given predicate), we will have to use the LPdoc plugin for exfilter, as shown in the example above. This is done by inserting the following command in the LPdoc document source:

```
@exfilter{code/app.pl}{A,filter=tpred}
```

This command is composed of, on the one hand, the predicate to which we want to apply Exfilter (`app/3`) and on the other hand, the analysis and the filter options that we want to apply (`-A,filter=tpred`). If this file has already been generated before, the contents of this file are simply included in the documentation. But if it does not exist, then LPdoc will create the file and we will have to run manually Exfilter to fill it in. Once Exfilter has done the analysis and applied the filter to the code (in the example above we want to analyse and filter the predicate

[1] Although this may change, it is currently a deliberate choice, aimed at allowing a clean separation of the generation of examples from the generation of manuals.

app/3), the result will be incorporated into the documentation. Finally, once we have finished the tutorial, using LPdoc we will be able to generate the output of the tutorial in HTML, PDF, etc.

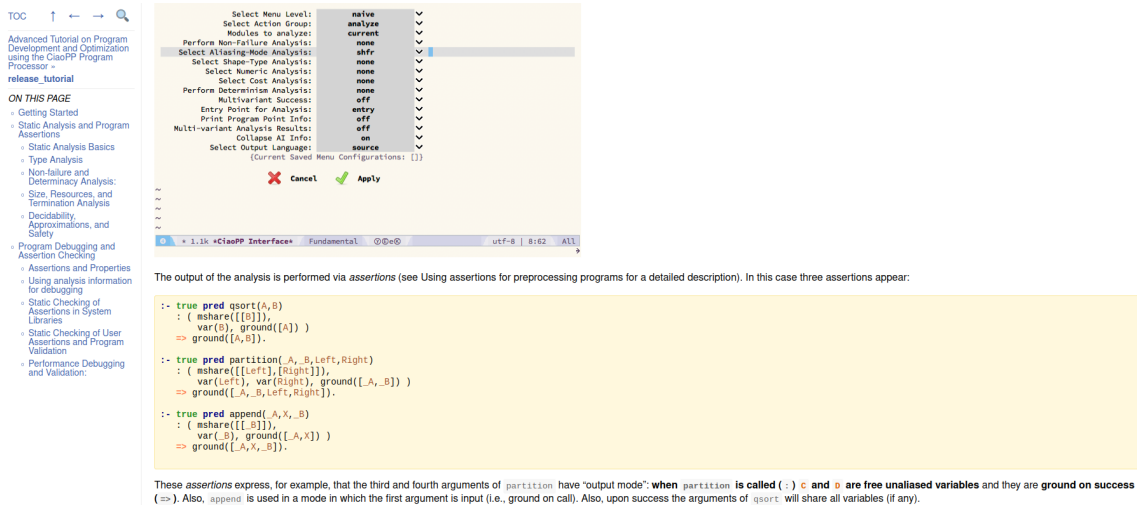


Figure 2.4: Tutorial exported to HTML

2.3 Ciao Assertions

Assertions [8] are linguistic constructs which allow expressing properties of programs. Syntactically they appear as an extended set of declarations, and semantically they allow talking about preconditions, (conditional-) postconditions, whole executions, program points, etc.

Assertions are used for multiple purposes, including writing specifications, reporting static analysis and verification results to the programmer, describing unknown code, generating test cases automatically, or producing documentation [14, 13, 17].

We will explain one of the most commonly-used components of the Ciao assertion language: `Pred` assertions. A detailed description of the full language can be found in [16, 1]. `Pred` assertions are of the form:

:- [*Status*] **pred** *Pred* [: *Precond*] [=> *Postcond*] [+ *CompProps*].

where *Status* is a qualifier of the meaning of the assertion, marked by prefixing the assertion itself with the keywords: `check`, `trust`, `true`, `checked`, and `false`. This specifies respectively whether the assertion is provided by the programmer and is to be checked or to be trusted, or is the output of static analysis and thus correct information, or the result of processing an input assertion and proving it correct or false. The `check` status is assumed by default when no explicit status keyword is present. *Pred* is a predicate descriptor that denotes the predicate that the assertion applies to. There can be multiple `pred` assertions for a predicate. *Precond* is the precondition under which the `pred` assertion is applicable. *Postcond* states a conditional postcondition. It expresses that in any

call to *Pred* if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* should also succeed in the success state. Both *PreCond* and *PostCond* can be empty conjunctions (meaning true), and in that case they can be omitted. For example, if the *Precond* is omitted, then the assertion is expressed as:

:- [*Status*] **pred** *Pred* [: *true*] [=> *Postcond*] [+ *CompProps*].

This means that for any call to *Pred* that succeeds, then *Postcond* should also succeed in the success state.
Finally, *CompProps* refers to a sequence of states and we refer to them as properties of computations.

For example, the assertion:

```
1 :- pred nrev (A,B) : list(A) = > list(B).
```

expresses that calls to predicate `nrev/2` with the first argument bound to a list are admissible, and that if such calls succeed then the second argument should also be bound to a list.

In addition, `entry` assertions are identical to `pred` assertions, except that they refer to external calls to the module (or predicate).

```
1 :- entry nrev/2 : {list,ground} * var.
```

This assertion is an example of an `entry` assertion: a `pred` assertion addressing calls from outside the module. It informs the CiaoPP analyzers that in all external calls to `nrev/2`, the first argument will be a ground list and the second one a free variable.

The `pred` assertion schema is in fact syntactic sugar for combinations of atomic assertions of the following three types:

:- **calls** *Pred* [: *Precond*].
:- **success** *Pred* [: *Precond*] [=> *Postcond*].
:- **comp** *Pred* [: *Precond*] [+ *CompProps]*.

which describe all the admissible call states, the success states, and computational properties for each set of admissible call states (in this order).

For example, we may want to require that if `qsort` is called with a list in the first argument position and the call succeeds, then on success the second argument position should also be a list. This is declared as follows:

```
1 :- success qsort(A,B) : list(A) => list(B).
```

In addition, we may also require that in all calls to predicate `qsort` the first argument should be a list. The following assertion will do:

```
1 :- calls qsort(A,B) : list(A).
```

We can also require for example that all calls to this predicate that have a list in the first argument and a variable in the second argument do not fail, as follows:

```
:- comp qsort(A,B) : (list(A) , var(B)) + not_fails.
```

Chapter 3

The Exfilter tool and plugin

As mentioned several times throughout the document, our main goal is to generate manuals and tutorials automatically in order to keep Ciao's documentation synchronized with the system and this avoid having to keep track of what documentation needs to be changed when an update is made. In order to accomplish this goal we have created Exfilter.

This chapter covers the implementation process of Exfilter. We will see how the tool works and what modifications have been made during the course of the project, because as we started testing it with the tutorials, we realised the need to add new features.

In addition, while implementing Exfilter, we started to develop a plugin for LPdoc for the purpose of including the fragments analyzed in the tutorials and manuals. We will explain this idea in more detail in Section 3.2.

3.1 Exfilter

Exfilter first runs CiaoPP on a collection of codes, i.e., it starts a CiaoPP preprocessing session for each piece of code, annotated with assertions. These assertions describe some properties which the programmer requires to hold of the program. In addition, Exfilter then applies one or more filters to the analysis results: the aim of this process is to extract fragments of the output (either messages or the final output) as desired for their inclusion in manuals and tutorials.

The command to launch Exfilter has the following format: `ciao-exfilter [PATH]`. The tool accepts a result path file name whose name encodes the input file plus a list of actions, options, and filters. For the given parameters it runs CiaoPP and extracts and filters the results into the given result path.

For example, the following call:

```
ciao-exfilter results/bugqsort-A-ftypes=eterms-fmodes=none-filter=tpred.txt
```

will execute CiaoPP with options `'-A'`, `'-ftypes=eterms'`, `'-fmodes=none'`, gather

the output, take the true assertions, and leave the result in the specified result path. This process will be described in greater detail in the next section.

3.1.1 All module results

The first version of the source was too large to handle in one piece, so we have separated it into the following core parts:

- **Decoding parameters:** Firstly, we extract *Name* without the extension from *ResultPath*. Secondly, we split the different elements separated by '- -'. Once the elements have been extracted, we can proceed to identify the source code's file name, the filter, and the rest of the options. This will give us the different parameters.

```
decode_params(ResultPath, File, Opts, Filter) :-
    path_split(ResultPath, _, Name),
    path_splitext(Name, NameNoext, '.txt'),
    atom_codes(NameNoext, Cs),
    split_items(Cs, Items),
    Items = [File0|Opts0],
    atom_codes(File, File0),
    ( select("filter="||Filter0, Opts0, Opts1) ->
        true
    ; Filter0 = "tpred", % Default filter
      Opts1 = Opts0
    ),
    as_options(Opts1, Opts),
    atom_codes(Filter, Filter0).
```

- **Source code:** Once the parameters have been decoded and we have obtained the source code, we proceed to run CiaoPP.

```
run1(ResultPath) :-
    decode_params(ResultPath, File, Opts, Filter),
    path_concat('code', File, SrcFile0),
    atom_concat(SrcFile0,'.pl',SrcFile),
    run_and_filter(SrcFile,Opts,Filter,ResultPath).
```

- **Run Ciaopp:** CiaoPP [13, 11, 14, 3] allows analyzing the code, verifying that programs comply with specifications, and performing many types of program optimizations.

  In this part, we run CiaoPP on the source code with the corresponding action and options we obtained in the previous step and output the result into a file. There are three different types of action:

  1. **'-A':** Call the preprocessor to analyze your program, in order to infer properties of the predicates and literals in your program.
  2. **'-O':** Uses the preprocessor to perform optimizations (partial evaluation, abstract specialization, parallelization, ...)

3. **'-V':** To verify assertions (types, modes, determinacy, nonfailure, cost, ...) or others specifications.

```
run_and_filter(SrcFile,[Action|Opts],Filter,ResultPath) :-
    ( Action = '-A' -> OutputSource = out_file
    ; Action = '-O' -> OutputSource = out_file
    ; Action = '-V' -> OutputSource = out_std
    ; throw(error(unrecognized_action, run_and_filter/4))
    ),
    ( OutputSource = out_file -> RawExt = '.raw.pl'
    ; RawExt = '.raw.out'
    ),
    atom_concat(ResultPath, RawExt, ResultPathRaw),
    ciaopp_call(OutputSource, Action, SrcFile, Opts,
        ResultPathRaw),
    run_filter_on_file(Filter, ResultPathRaw, ResultPath).
```

- **Filter:** Once the output file produced by Ciao preprocessor is ready, we can proceed to apply the filter. There are seven different filters:

1. **Filter = all:** this filter keeps all the data.

```
run_filter(all, InStr, OutStr) :- !, OutStr = InStr.
```

2. **Filter = tpred:** filter all "true pred" assertions.

```
run_filter(tpred, InStr, OutStr) :- !, tpred(OutStr,
    InStr, []).

tpred([]) --> \+ [_], !.
tpred(Zs) --> truepred(Ys),!, tpred(Xs),
    {append(Ys,Xs,Zs)}.
tpred(Xs) --> [_], tpred(Xs).

truepred(Ys) --> ":- true pred ", tpkeep(Xs),
    {append(":- true pred ",Xs,XXs),
    append(XXs,"\n\n",Ys)}.

tpkeep(".") --> ".", !.
tpkeep([X|Xs]) --> [X], {X \= 0'. }, tpkeep(Xs).
```

3. **Filter = tpred_plus:** filter all "true pred" assertions including comp properties.

```
run_filter(tpred_plus, InStr, OutStr) :- !,
    tpredp(OutStr, InStr, []).

tpredp([]) --> \+ [_], !.
tpredp(Zs) --> truepredp(Ys),!, tpredp(Xs),
    {append(Ys,Xs,Zs)}.
tpredp(Xs) --> [_], tpredp(Xs).
```

```
6
7 truepredp(As) --> ":- true pred ", tpkeepp(Xs),
     tpkeep(Ys),
8     { append(":- true pred ",Xs,XXs), append(XXs,Ys,Zs),
        append(Zs,"\n\n",As) }.
9
10 tpkeepp("+") --> "+", !.
11 tpkeepp([X|Xs]) --> [X], {X \= 0'+}, {X \= 0'. },
     tpkeepp(Xs).
```

4. **Filter = tpred_regtype:** filter all "true pred" assertions and all regtypes.

```
1 run_filter(tpred_regtype, InStr, OutStr) :- !,
     tpredreg(OutStr, InStr, []).
2
3 tpredreg([]) --> \+ [_], !.
4 tpredreg(Zs) --> truepred(Ys),!, tpredreg(Xs),
     {append(Ys,Xs,Zs)}.
5 tpredreg(Zs) --> regpred(Ys),!, tpredreg(Xs),
     {append(Ys,Xs,Zs)}.
6 tpredreg(Xs) --> [_], tpredreg(Xs).
7
8 regpred(As) --> ":- regtype ", tpkeep(Xs), tpkeep(Ys),
9     { append(":- regtype ",Xs,XXs), append(XXs,Ys,Zs),
        append(Zs,"\n\n",As) }.
```

5. **Filter = warnings:** filter all WARNINGs.

```
1 run_filter(warnings, InStr, OutStr) :- !, warn(OutStr,
     InStr, []).
2
3 warn([]) --> \+ [_], !.
4 warn(Zs) --> warn_(Ys),!, warn(Xs), {append(Ys,Xs,Zs)}.
5 warn(Xs) --> [_], warn(Xs).
6
7 warn_(As) --> "WARNING", warnkeep(Xs),
8     { append("WARNING",Xs,XXs), append(XXs,"\n\n",As) }.
9
10 warnkeep([]) --> "}", !.
11 warnkeep([X|Xs]) --> [X], {X \= 0'} }, warnkeep(Xs).
```

6. **Filter = errors:** filter all ERRORs.

```
1 run_filter(errors, InStr, OutStr) :- !, err(OutStr,
     InStr, []).
2
3 err([]) --> \+ [_], !.
4 err(Zs) --> err_(Ys),!, err(Xs), {append(Ys,Xs,Zs)}.
5 err(Xs) --> [_], err(Xs).
```

```
6
7 err_(As) --> "ERROR", errkeep(Xs),
8     { append("ERROR",Xs,XXs), append(XXs,"\n\n",As) }.
9
10 errkeep([]) --> "}", !.
11 errkeep([X|Xs]) --> [X], {X \= 0'} }, errkeep(Xs).
```

7. **Filter = none:** no output.

```
1 run_filter(none, _, OutStr) :- !, OutStr = "".
```

- **Result in the result path:** Finally, the result obtained after applying the filter will be printed on the `ResultPath` file.

```
1 :- pred run_filter_on_file(File, NameFilter, InFile,
    OutFile)
2    #"Load @var(InFile), apply filter @var(Filter) and save
       @var(OutFile)".
3
4 run_filter_on_file(Filter, InFile, OutFile) :-
5     file_to_string(InFile, InStr),
6     run_filter(Filter, InStr, OutStr),
7     string_to_file(OutStr, OutFile).
```

To better understand this process, we will show the functioning of Exfilter using the following example. If we call:

```
ciao-exfilter results/qsort2--A--fana_nf=nf--fana_cost=resources--filter=tpred_plust.txt
```

- **Decoding parameters:** Exfilter extracts from

  `results/qsort2-A-fana_nf=nf-fana_cost=resources-filter=tpred_plus.txt`

  these elements:

  - The source file: `qsort2.pl`.
  - The filter: `tpred_plus`.
  - The options: `A`, `fana_nf=nf` and `fana_cost=resources`.

- **Source Code:** As we mentioned before, the source code in our example will be `qsort2.pl`

```
1 :- module(qsort2, [qsort/2],
    [assertions,predefres(res_steps)]).
2 :- entry qsort(A,B) : (list(num,A), var(B)).
3
4 qsort([X|L],R) :-
5     partition(L,X,L1,L2),
6     qsort(L2,R2), qsort(L1,R1),
7     append(R2,[X|R1],R).
8 qsort([],[]).
9
```

```
partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
    E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
    E >= C, partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

- **Run Ciaopp:** CiaoPP will be execute with the flags `A`, `fana_nf=nf` and `fana_cost=resources`.
- **Filter:** Once we have the analysis result, the filter will be applied. In our example `filter=tpred_plus`, therefore, we will keep the "`true pred`" assertions including `comp` properties.
- **Result in the result path:** The result will be written into a file which name is `qsort2-A-fana_nf=nf-fana_cost=resources-filter=tpred_plus`. The result is shown below:

```
:- true pred qsort(A,B)
   : ( mshare([[B]]),
       var(B), ground([A]), list(num,A), term(B) )
   => ( ground([A,B]), list(num,A), list(num,B) )
   + ( not_fails, covered ).

:- true pred qsort(A,B)
   : ( mshare([[B]]),
       var(B), ground([A]), list(num,A), term(B) )
   => ( ground([A,B]), list(num,A), list(num,B) )
   + ( not_fails, covered ).

:- true pred qsort(A,B)
   : ( list(num,A), var(B) )
   => ( list(num,A), list(num,B),
        size(lb,length,B,1) )
   + cost(lb,steps,length(A)+5).

:- true pred qsort(A,B)
   : ( list(num,A), var(B) )
   => ( list(num,A), list(num,B),
        size(ub,length,B,2**length(A)-1.0) )
   + cost(ub,steps,sum($(j),1,length(A),$(j)*2**(length(A)-
      $(j)))+2.0*2**length(A)+length(A)*2**(length(A)-1)-1.0).

:- true pred partition(_A,_B,Left,Right)
   : ( mshare([[Left],[Right]]),
       var(Left), var(Right), ground([_A,_B]),
          list(num,_A), num(_B), term(Left), term(Right) )
   => ( ground([_A,_B,Left,Right]), list(num,_A), num(_B),
```

```
        list(num,Left), list(num,Right) )
29    + ( not_fails, covered ).
30
31  :- true pred partition(_A,_B,Left,Right)
32    : ( list(num,_A), num(_B), var(Left), var(Right) )
33    => ( list(num,_A), num(_B), list(num,Left),
        list(num,Right),
34         size(lb,length,Left,0),
35         size(lb,length,Right,0) )
36    + cost(lb,steps,length(_A)+1).
37
38  :- true pred partition(_A,_B,Left,Right)
39    : ( list(num,_A), num(_B), var(Left), var(Right) )
40    => ( list(num,_A), num(_B), list(num,Left),
        list(num,Right),
41         size(ub,length,Left,length(_A)),
42         size(ub,length,Right,length(_A)) )
43    + cost(ub,steps,length(_A)+1).
44
45  :- true pred append(_A,X,_B)
46    : ( X=[_C|_D],
47        mshare([[_B]]),
48        var(_B), ground([_A,_C,_D]), list(num,_A), term(_B),
            num(_C), list(num,_D) )
49    => ( ground([_A,_B,_C,_D]), list(num,_A), list1(num,_B),
        num(_C), list(num,_D) )
50    + ( not_fails, covered ).
51
52  :- true pred append(_A,X,_B)
53    : ( mshare([[_B]]),
54        var(_B), ground([_A,X]), list(num,_A), list1(num,X),
            term(_B) )
55    => ( ground([_A,X,_B]), list(num,_A), list1(num,X),
        list1(num,_B) )
56    + ( not_fails, covered ).
57
58  :- true pred append(_A,X,_B)
59    : ( X=[_C|_D],
60        mshare([[_B]]),
61        var(_B), ground([_A,_C,_D]), list(unifier_elem,_A),
            term(_B), num(_C), list(unifier_elem,_D) )
62    => ( ground([_A,_B,_C,_D]), list(unifier_elem,_A),
        rt5(_B), num(_C), list(unifier_elem,_D) )
63    + ( not_fails, covered ).
64
65  :- true pred append(_A,X,_B)
66    : ( list(num,_A), list1(num,X), var(_B) )
```

```
67      => ( list(num,_A), list1(num,X), list1(num,_B),
68           size(lb,length,_B,length(X)+length(_A)) )
69      + cost(lb,steps,length(_A)+1).
70
71  :- true pred append(_A,X,_B)
72      : ( list(num,_A), list1(num,X), var(_B) )
73      => ( list(num,_A), list1(num,X), list1(num,_B),
74           size(ub,length,_B,length(X)+length(_A)) )
75      + cost(ub,steps,length(_A)+1).
```

The main problem in the first version of our implementation was that during the experimental evaluation, we observed that sometimes we only want a particular assertion instead of having all assertions. This will be explained in more detail in the next section.

3.1.2 Find module result

As we stated above, in order to obtain one or more specific "`true pred`" assertion, we had to add new options.

These options work as follows: Once we have already applied the filter, depending on what we want, another filter will be applied to achieve the desired result. So, if we only want the results of a specific predicate, we will add the option *name=Pred*, with *Pred* being the predicate. On the other hand, if we only want the assertions that contain a series of terms, we will add the option *assertion=[Terms]* where *Terms* is the list of terms to be matched. For example, if we need the assertions that contain the word cost and the word ub, then we need to add the option `assertion=[cost,ub]`.

Continuing with the example used before, if we now want to get the "true pred" assertions and comp properties but only for a given predicate, in this case the predicate partition, the command would be:

```
ciao-exfilter results/qsort2--A--fana_nf=nf--fana_cost=resources--name=partition--filter=tpred_plus.txt
```

The process is the same as the previous one, but once we have applied the `filter=tpred_plus` (the result shown above), another filter will be applied. In this case, only the assertions of the predicate partition will be kept.

The result will be written into a file whose name is

```
qsort2--A--fana_nf=nf--fana cost=resources--name=partition--filter=tpred_plus.txt
```

and the contents will be:

```
1  :- true pred partition(_A,_B,Left,Right)
2     : ( mshare([[Left],[Right]]),
3         var(Left), var(Right), ground([_A,_B]), list(num,_A),
             num(_B), term(Left), term(Right) )
4     => ( ground([_A,_B,Left,Right]), list(num,_A), num(_B),
          list(num,Left), list(num,Right) )
5     + ( not_fails, covered ).
```

```
6
7  :- true pred partition(_A,_B,Left,Right)
8     : ( list(num,_A), num(_B), var(Left), var(Right) )
9     => ( list(num,_A), num(_B), list(num,Left), list(num,Right),
10         size(lb,length,Left,0),
11         size(lb,length,Right,0) )
12    + cost(lb,steps,length(_A)+1).
13
14 :- true pred partition(_A,_B,Left,Right)
15    : ( list(num,_A), num(_B), var(Left), var(Right) )
16    => ( list(num,_A), num(_B), list(num,Left), list(num,Right),
17         size(ub,length,Left,length(_A)),
18         size(ub,length,Right,length(_A)) )
19    + cost(ub,steps,length(_A)+1).
```

If we now want only those assertions that contain the terms cost and ub, the command will be:

```
ciao-exfilter results/qsort2--A--fana_nf=nf--fana_cost=resources--name=partition--assertion=[cost,ub]--filter=tpred_plus.t
```

And the result will be:

```
1  :- true pred partition(_A,_B,Left,Right)
2     : ( list(num,_A), num(_B), var(Left), var(Right) )
3     => ( list(num,_A), num(_B), list(num,Left), list(num,Right),
4          size(ub,length,Left,length(_A)),
5          size(ub,length,Right,length(_A)) )
6     + cost(ub,steps,length(_A)+1).
```

As we can see, depending on what we want, several filters can be applied at the same time.

3.2 LPdoc plugin

Lpdoc [12, 9] is a tool which generates documentation manuals automatically from one or more logic program source files, written in ISO-Prolog, Ciao, and other (C)LP languages. A fundamental advantage of using lpdoc to document programs is that it is much easier to maintain a true correspondence between the program and its documentation, and to identify precisely to what version of the program a given printed manual corresponds.

In order to allow better formatting of on-line and printed manuals, in addition to normal text, certain formatting commands can be used within these strings. The syntax of all these commands is: `@command` or `@command{body}` where `command` is the command name and `body` is the (possibly empty) command body.

We have supplemented a plugin for lpdoc that recognises a command with syntax `@exfilter{File}{Options}` which concatenate `File` with `Options`, find the result file and ensure that this file exists, and incorporate it in the documentation. For example:

```
@exfilter{code/app.pl}{A,filter=tpred}
```

Moreover, LPdoc is specially relevant in our context because it includes a number of backends in order to generate the documentation in different formats such as texinfo, pdf, html, ascii, etc.

Chapter 4

Experimental evaluation

In this chapter, we describe an experimental evaluation of the tool Exfilter which we explained in the previous chapter. This experimental evaluation consists in:

- Recognizing the pieces of code that we want to analyze.
- Applying our tool.
- Comparing the outputs with the tutorial made manually.

We show some examples of the Advanced Tutorial on Program Development and Optimization using the Ciao Preprocessor [21] and the CiaoPP Quick Tutorial [7].

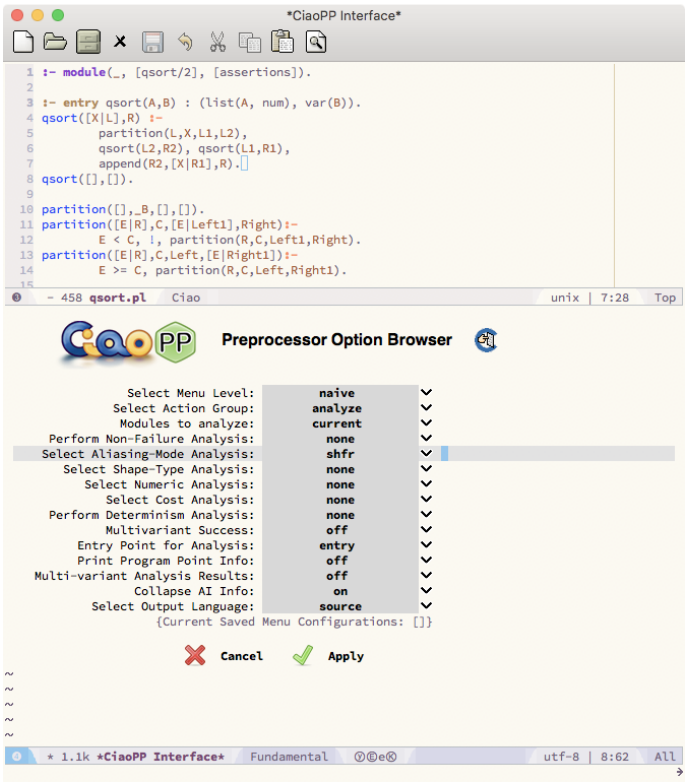
4.1 Static Analysis Basics

Consider the program defining a module which exports the `qsort` predicate:

```
:- module(qsort, [qsort/2], [assertions]).
:- entry qsort(A,B) : (list(num,A), var(B)).

qsort([X|L],R) :-
    partition(L,X,L1,L2),
    qsort(L2,R2), qsort(L1,R1),
    append(R2,[X|R1],R).
qsort([],[]).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
    E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
    E >= C, partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

The sharing and freenes analysis abstract domain computes freeness, independence, and grounding dependencies between program variables.

It is performed by selecting the menu option Aliasing-Mode:



The output of the analysis is performed via assertions. In this case three assertions appear:

```
:- true pred qsort(A,B)
     : mshare([[A],[A,B],[B]])
    => mshare([[A,B]]).
:- true pred partition(A,B,C,D)
     : ( var(C), var(D), mshare([[A],[A,B],[B],[C],[D]]) )
    => ( ground(A), ground(C), ground(D), mshare([[B]]) ).
:- true pred append(A,B,C)
     : ( ground(A), mshare([[B],[B,C],[C]]) )
    => ( ground(A), mshare([[B,C]]) ).
```

(a) Tutorial

```
:- true pred qsort(A,B)
   : ( mshare([[B]]),
       var(B), ground([A]) )
   => ground([A,B]).

:- true pred partition(_A,_B,Left,Right)
   : ( mshare([[Left],[Right]]),
       var(Left), var(Right), ground([_A,_B]) )
   => ground([_A,_B,Left,Right]).

:- true pred append(_A,X,_B)
   : ( mshare([[_B]]),
       var(_B), ground([_A,X]) )
   => ground([_A,X,_B]).
```

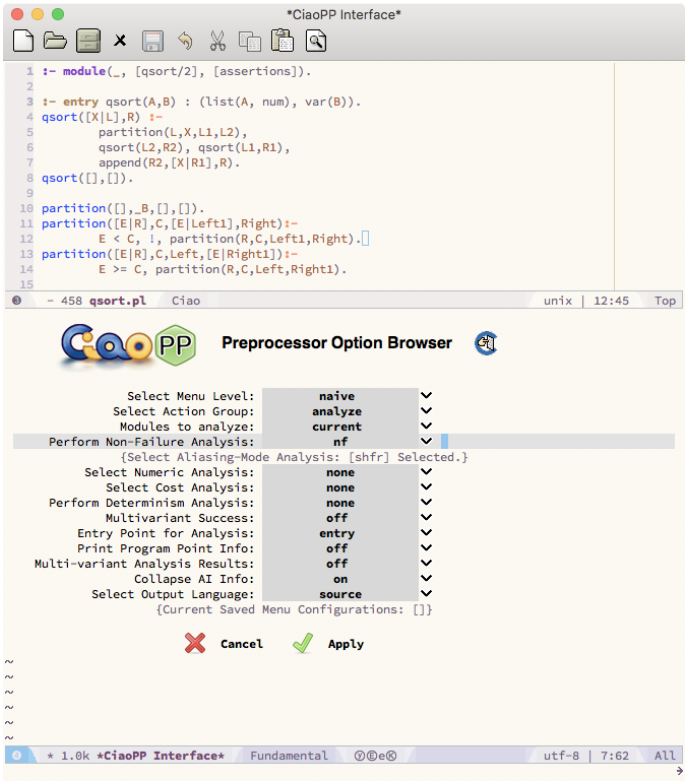
(b) Exfilter

Figure 4.1

4.2 Non-failure and Determinacy Analysis

CiaoPP includes a non-failure analysis, based on [4] and [2], which can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution or not terminate. It also can detect predicates that are “covered”,

i.e., such that for any input (included in the calling type of the predicate), there is at least one clause whose "test" (head unification and body builtins) succeeds. CiaoPP also includes a determinacy analysis based on [15], which can detect predicates which produce at most one solution, or predicates whose clause tests are mutually exclusive, even if they are not deterministic (because they call other predicates that can produce more than one solution). Programs can be analyzed with this kind of domains by selecting to perform *Non-Failure* Analysis with domain *nf*:



Analyzing `qsort` with the `nf` domain will produce (among others) the following assertion:

```
:- true pred qsort(A,B)
     : ( mshare([[B]]), var(B), ground([A]), list(num,A), term(B) )
    => ( ground([A,B]), list(num,A), list(num,B) )
     + ( not_fails, covered ).
```

(a) Tutorial

```
:- true pred qsort(A,B)
   : ( mshare([[B]]),
       var(B), ground([A]), list(num,A), term(B) )
  => ( ground([A,B]), list(num,A), list(num,B) )
  + ( not_fails, covered ).
```

(b) Exfilter

Figure 4.2

The + field in `pred` assertions can contain a conjunction of global properties of the computation of the predicate. `not_fails` states that if the precondition is met, the predicate will never fail.

4.3 Size, Resources, and Termination Analysis

CiaoPP can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates [5, 4]. The cost bounds are expressed as functions on the sizes of the input arguments and yield the number of resolution steps or other resources. Various measures are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. Note that obtaining a non-infinite upper bound on cost also implies proving termination of the predicate.

This resource analysis is parametric on the resources, therefore a package defining the resource to be used has to be imported in the module, in this case we use the default package that infers information about computational steps. This is done by replacing the first line by:

```
:- module(qsort, [qsort/2], [assertions,predefres(res_steps)]).
```

Also, to be able to infer lower bounds a non-failure and determinacy analysis has to be performed:

As an example, the following assertions are part of the output of the upper bounds analysis:

```
:- true pred qsort(A,B)
     : ( list(num,A), var(B) )
    => ( list(num,A), list(num,B), size(ub,A,length(A)), size(ub,B,exp(2,length(A))-1.0) )
     + cost(ub,steps,sum($(j),1,length(A),exp(2,length(A)- $(j))* $(j))+exp(2,length(A)-1)*length(A)+2.0*exp(2,length(A))-1.0).
:- true pred append(_A,X,_B)
     : ( list(_A,num), rt13(X), var(_B) )
    => ( list(_A,num), rt13(X), rt13(_B), size(ub,_A,length(_A)), size(ub,X,length(X)), size(ub,_B,length(X)+length(_A)) )
     + cost(ub,steps,length(_A)+1).
```

Figure 4.3: (a) Tutorial

```
:- true pred qsort(A,B)
   : ( list(num,A), var(B) )
   => ( list(num,A), list(num,B),
        size(ub,length,B,2**length(A)-1.0) )
   + cost(ub,steps,sum($(j),1,length(A),$(j)*2**(length(A)- $(j)))+2.0*2**length(A)+length(A)*2**(length(A)-1)-1.0).
```

```
:- true pred append(_A,X,_B)
   : ( list(num,_A), list1(num,X), var(_B) )
   => ( list(num,_A), list1(num,X), list1(num,_B),
        size(ub,length,_B,length(X)+length(_A)) )
   + cost(ub,steps,length(_A)+1).
```

Figure 4.4: (b) Exfilter

For example, the second assertion is inferring on success `size(ub,_B,length(X)+length(_A))`, which means that an (upper) bound on the size of the third argument of `append/3` is the sum of the sizes of the first and second arguments. The inferred upper bound on computational steps `(+ cost(ub,steps,length(_A)+1))` is the length of the first argument of `append/3`.

The following is the output of the lower-bounds analysis:

```
:- true pred qsort(A,B)
     : ( list(num,A), var(B) )
    => ( list(num,A), list(num,B), size(lb,A,length(A)), size(lb,B,1) )
     + cost(lb,steps,length(A)+3).
:- true pred append(_A,X,_B)
     : ( list(num,_A), rt13(X), var(_B) )
    => ( list(num,_A), rt13(X), rt13(_B), size(lb,_A,length(_A)), size(lb,X,length(X)), size(lb,_B,length(X)+length(_A)) )
     + cost(lb,steps,0).
```

Figure 4.5: (a) Tutorial

```
:- true pred qsort(A,B)
   : ( list(num,A), var(B) )
   => ( list(num,A), list(num,B),
        size(lb,length,B,1) )
   + cost(lb,steps,length(A)+5).
```

```
:- true pred append(_A,X,_B)
   : ( list(num,_A), list1(num,X), var(_B) )
   => ( list(num,_A), list1(num,X), list1(num,_B),
        size(lb,length,_B,length(X)+length(_A)) )
   + cost(lb,steps,length(_A)+1).
```

Figure 4.6: (b) Exfilter

The lower-bounds analysis uses information from the non-failure analysis, without which a trivial lower bound of 0 would be derived. In this case it is inferred that on success the lower bound of the third argument of append is

`size(lb,_B,length(X)+length(_A))` (the same as the upper bound!), and the upper bound on computational `steps + cost(lb,steps,0)`, which represents the case in which the first list to concatenate is empty.

4.4 Assertion Checking

Let us add an assertion with properties that we want to prove about the execution of the program.

```
1 :- module(_, [app/3], [assertions]).
2
3 :- entry app(A,B,C) : (list(A), list(B)).
4 :- pred app(A,B,C) : (list(A), list(B)) => var(C).
5 app([],Y,Y).
6 app([X|Xs], Ys, [X|Zs]) :-
7     app(Xs,Ys,Zs).
```

This assertion is stating that if the predicate is called with a `A` and `B` list, if it succeeds `C` will be a free variable.

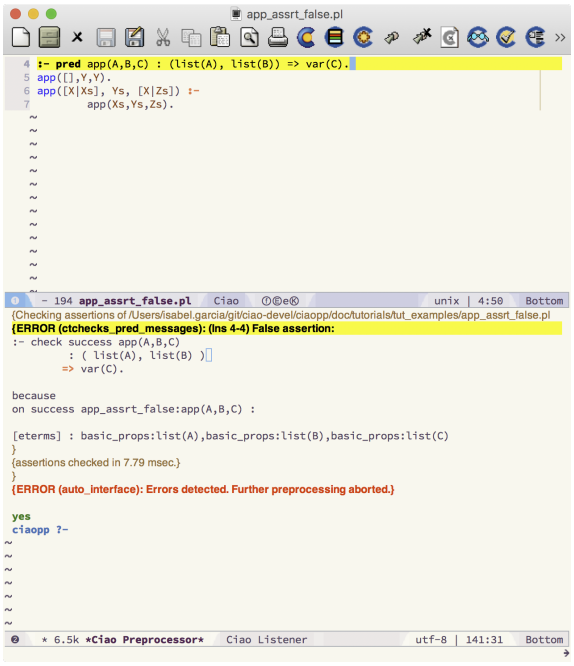


Figure 4.7: (a) Tutorial

```
ERROR (ctchecks_pred_messages): (lns 4-4) False assertion:
:- check success app(A,B,C)
   : ( list(A), list(B) )
   => var(C).
because the success field is incompatible with inferred success:
[eterms] basic_props:list(A),basic_props:list(B),basic_props:list(C)

ERROR (auto_interface): Errors detected. Further preprocessing aborted.
```

Figure 4.8: (b) Exfilter

Of course this assertion does not hold and we get a message saying so.

4.5 Execution times

In this section we will show some information in order to help us better understand the scheme we have chosen to implement Exfilter. We will use the same examples as in the previous section.

First of all, we will summarise the Exfilter scheme discussed at the beginning of the document: first, we indicate in LPdoc the examples we want to show. When we run LPdoc it creates the files and warns us that we need to run Exfilter to fill them. After the files are created, Exfilter is run on the files and once it has finished, LPdoc is run again, which will incorporate the content of the files into the documentation.

In the following table we have calculated the times for each part of the process. The second column shows the time LPdoc takes to generate the documentation before Exfilter is applied. The third column represents the time it takes for Exfilter to run the analysis and the last column indicates the time it takes LPdoc to generate the documentation after Exfilter has been applied.

Table of results of the analysis			
Example	LPdoc (s)	Analysis times (s)	LPdoc (s)
1	0.3225	0.951	0.337
2	0.3225	1.1775	0.337
3	0.3225	1.178+1.158	0.337
4	0.1345	0.9725	0.178

As we can see in this table, the time it takes to run Exfilter on the files is longer than the time it takes to generate the documentation with LPdoc. Therefore, we decided to divide this process into these phases. When there is an error in the analysis or if there is a complex example and it takes longer, it is easier to control and identify which part of the process is causing the error or is taking too long.

Chapter 5

Conclusions and Future Work

We have presented Exfilter, a tool which aims to keep the Ciao tutorials and manuals up to date. Although we have only applied it to two tutorials: *Advanced Tutorial on Program Development and Optimization using the CiaoPP Program Processor* and *CiaoPP Quick Tutorial*, the purpose of this tool is to apply it to all manuals and tutorials. We also implemented an LPdoc plugin in order to include Exfilter results in LPdoc.

After the experimental evaluation, we can observe the correct functioning of the tool and seeing how all the filters are working properly. Furthermore, by comparing the results in the tutorial with once that Exfilter extracted, we can see the improvements that have been made and how the tutorials were outdated. Moreover, during the experimental evaluation we could identified many inconsistencies that helped correct errors in the system. Therefore, we can see that exfilter fulfils its purpose: On the one hand, to automate the process of generating examples for tutorials and on the other hand, to find errors and inconsistencies in the examples in order to fix and improve the errors in the system.

Regarding the modifications or changes that can be made to the tool, it will depend on the needs that are created. For a specific example, there may be a need to create a filter or a new option.

As mentioned above, the main idea is to apply it to all Ciao tutorials. But for future use, this tool can be used for tutorials written in other languages, as CiaoPP does not only analyse the Ciao language.

Chapter 6

Impact Analysis

While there have been substantial improvements in resource consumption from the hardware point of view, significant progress can still be made in improving the energy efficiency of software. Energy is a critical resource in embedded software development and it is therefore necessary to be able to predict the energy consumption of a program before running it.

Software development in embedded systems is of special importance due to the large number of these devices involved in our daily life. Some of them may be as obvious as a bank terminal, a smartphone or a smartwatch, and others may go unnoticed like the router at home. Moreover, in a world dominated by technology, the growing need to connect everything, better known as the **Internet of Things**, opens up infinite opportunities to the emergence of new embedded systems that add to the list.

The consumption is directly related to the characteristics of the hardware that makes up a system, but it also depends on how and how much it is used, i.e., it depends on the software being efficient. CiaoPP helps in the the verification and optimization of the resource usage of programs, thus reducing the energy footprint of IT. Static analysis makes it possible to infer from the code information about the resource consumption of a program without exercising it. In this work, improving and facilitating the use of CiaoPP will lead to its more frequent use.

This will make it possible to verify or certify a program's energy specifications or to classify program's energy specifications or classify software based on energy consumption. In addition, the ability to statically infer the energy consumption of a program will allow the developer to optimize his code to make it more energy efficient. The objectives of the project are thus directly related to the reduction of the energy footprint of information technology, and can thus contribute significantly to the sustainable development objective.

Bibliography

[1] F. Bueno, M. Carro, M. V. Hermenegildo, P. Lopez-Garcia, and J.F. Morales (Eds.). The Ciao System. Ref. Manual (v1.20). Technical report, April 2021. Available at `http://ciao-lang.org`.

[2] F. Bueno, P. Lopez-Garcia, and M. V. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *FLOPS'04*, number 2998 in LNCS, pages 100–116. Springer-Verlag, 2004.

[3] F. Bueno, P. Lopez-Garcia, G. Puebla, and M. V. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP1/04, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2004.

[4] S. K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[5] S.K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[6] Tom Dye Eric Schulte, Dan Davison. Babel: Introduction.

[7] Isabel Garcia-Contreras. Ciaopp quick tutorial. Available at https://ciao-lang.org/ciao/build/doc/ciaopp.html/tut_quick_start.html.

[8] M. Hermenegildo and The Ciao Development Team. Why Ciao? –An Overview of the Ciao System's Design Philosophy. Technical Report CLIP7/2006.0, Technical University of Madrid (UPM), School of Computer Science, UPM, December 2006. Available from: http://cliplab.org/papers/ciao-philosophy-note-tr.pdf.

[9] M. V. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[10] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, R. Haemmerlé, E. Mera, J. F. Morales, and G. Puebla. An Overview of the Ciao System. In

N. Bassiliades et al., editor, *Proc. of RuleML-Europe 2011*, volume 6826 of *LNCS*, pages 2–3. Springer-Verlag, July 2011. (abstract of invited talk).

[11] M. V. Hermenegildo, F. Bueno, G. Puebla, and P. Lopez-Garcia. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[12] M. V. Hermenegildo and J.F. Morales. The LPdoc Documentation Generator. Ref. Manual (v3.0). Technical report, July 2011. Available at `http://ciao-lang.org`.

[13] M. V. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

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

[15] P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Pre-proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 19–39. Springer-Verlag, August 2004.

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

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

[18] Eric Schulte and Dan Davison. Active documents with org-mode. *Computing in Science and Engineering*, 13:66–73, 05 2011.

[19] Eric Schulte, Dan Davison, Tom Dye, and Carsten Dominik. A multi-language computiBng environment for literate programming and reproducible research. *Journal of Statistical Software*, 46:1–24, 01 2012.

[20] Jupyter Team. Jupyter notebook documentation (release 6.5.0.dev0), 2021. Available at https://buildmedia.readthedocs.org/media/pdf/jupyter-notebook/latest/jupyter-notebook.pdf.

[21] The Ciao Development Team. Advanced tutorial on program development and optimization using the ciaopp program processor. Available at https://ciao-lang.org/ciao/build/doc/ciaopp.html/release_tutorial.html.

[22] Wolfram. Wolfram notebooks: Environment for technical workflows, 2021. Available at https://www.wolfram.com/notebooks/.

[23] S. Wolfram. What is a computational essay?, 2017. Available at https://writings.stephenwolfram.com/2017/11/what-is-a-computational-essay/.

Appendix A

Example extraction, execution, and filter tool for Ciao/CiaoPP

Exfilter

Example extraction, execution, and filter tool for Ciao/CiaoPP

The Ciao Development Team

Table of Contents

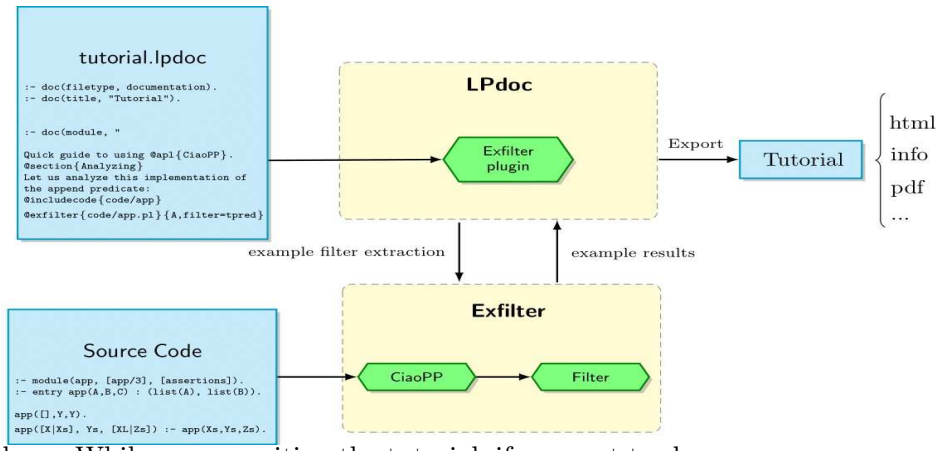
1 Summary

This manual describes **Exfilter**: A tool to run `Ciao`/`CiaoPP` on a collection of codes and extract fragments of the output (either messages or the final output) suitable for inclusion in manuals and tutorials. The tool can be divided into the following three stages:

1. Collection of codes: The examples we want to show in the manuals and tutorials. The examples will be written in Ciao Prolog using its assertions library.
2. Analyze: The tool accepts a result path file name whose name encodes an input file, a list of actions, options, and filters. For the given parameters it run CiaoPP and extract and filter the results into the given result path.
3. Generate manuals and tutorials with the extracted fragments of the output: the analysis results will be added in documentation.

exfilter

`Exfilter` is a tool for `Ciao`/`CiaoPP`. Its main goal is automating the different actions necessary for the maintenance of documents and tutorials of the `Ciao` language analysis. The following figure shows the scheme we have used to generate the tutorials:



The process works as follows: While we are writing the tutorial, if we want to show an example (e.g. the result analysis of a given predicate), we will have to use the LPdoc plugin for exfilter, in the example above you can see how it is used: `@exfilter{code/app.pl}{A,filter=tpred}`. This command is composed on the one hand, the predicate to which we want to apply Exfilter (`app/3`) and on the other hand, the analysis and the filter options we want to apply (`-A`,`filter=tpred`). If this file has already been generated before, then what it does is to include the contents of the file into the documentation. But if it does not exist, then LPdoc will create the file and we will have to run manually Exfilter to fill in it. Once Exfilter has done the analysis and applied the filter to the code (in the example above we want to analyse and filter the predicate `app/3`), the result will be incorporated into the documentation. Finally, once we have finished the tutorial, using LPdoc we will be able to generate the output of the tutorial in HTML, PDF, etc.

Usage and interface

- **Library usage:**

  ### Usage

  The command line to launch it has the following format: `ciao-exfilter [PATH]`. The tool accepts a result path file name whose name encodes an input file, a list of actions, options, and filters. For the given parameters it run CiaoPP and extract and filter the results into the given result path.

  - If we want to execute CiaoPP with options `-A`, `-ftypes=eterms`, `-fmodes=none`, filter true assertions, and generate the result in the specified result path the command line to use should be:

    ```
    ciao-exfilter
    results/bugqsort--A--ftypes=eterms--fmodes=none--filter=tpred.txt
    ```

  - But, if we only want the results of a specific predicate, the command line to use should be:

    ```
    ciao-exfilter results/bugqsort--A--name=qsort--filter=tpred.txt
    ```

  - If we need the assertions that contain the word cost and the word ub, then we need to add the option assertion=[cost,ub]:

    ```
    ciao-exfilter results/bugqsort--A--name=qsort--assertion=[cost,ub]--
    filter=tpred_plus.txt
    ```

  - The following call will look automatically in the results/ directory: `ciao-exfilter`

- **Exports:**
  - *Predicates:*

    `decode_params/6`, `run_and_filter/6`, `ciaopp_call/5`, `run_filter_on_file/5`, `run_filter/5`.

Documentation on exports

decode_params/6: PREDICATE

Firstly, we extract `Name` without the extension from `ResultPath`. Secondly, we split the different elements separated by '- -'. Once the elements have been extracted, we can proceed to identify the source code's file name, the filter and the rest of the options. This will give us the different parameters.

Usage: `decode_params(ResulthPath,File,Opts,Filter,NameFilter,Include)`

Extract the `File`, `Opts`, `Filter` and if there is a `NameFilter` or `Include` from the `ResultPath`

run_and_filter/6: PREDICATE

In this part, we run CiaoPP on the source code with the corresponding action and options we obtained in the previous step and output the result into a file.

Usage:
`run_and_filter(SrcFile,ListOpts,Filter,NameFilter,Include,ResultPath)`

Run CiaoPP with the flags `ListOpts`, apply the different filters `Filter`, `NameFilter`, `Include` and output the result into `ResultPath`

ciaopp_call/5: PREDICATE

Usage: `ciaopp_call(OutputSource,Action,SrcFile,Opts,OutFile)`

Call CiaoPP with the corresponding flags(`Action` and `Opts`) and output the result into `OutFile`

run_filter_on_file/5: PREDICATE

Once the output file produced by Ciao preprocessor is ready, we can proceed to apply the filter. Finally, the result obtained after applying the filter will be printed on the `ResultPath` file.

Usage: `run_filter_on_file(File,NameFilter,Include,InFile,OutFile)`

Load `InFile`, apply the filters and save `OutFile`

run_filter/5: PREDICATE

`run_filter(Filter, NameFilter, ListInclude, InStr, OutStr)`

`Filter`: The possible values are:

- **all**: keep all data
- **tpred**: all `true pred` assertions
- **tpred_plus**: all `true pred` assertions including comp properties
- **tpred_regtype**: all `true pred` assertions and all regtypes
- **warnings**: all WARNINGs
- **errors**: all ERRORs

`NameFilter`: Is a predicate. Exfilter filter the assertions of a specific predicate.

`ListInclude`: Is a list of words. Exfilter filter the assertions that include this words.

`OutStr`: Output after applying the filters in `InStr`

Documentation on imports

This module has the following direct dependencies:

– *Application modules:*

`stream_utils`, `streams`, `write`, `format`, `lists`, `process`, `process_channel`, `system`, `pathnames`, `read_from_string`, `llists`, `regexp_code`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `fsyntax`, `dcg`.

References

(this section is empty)

Appendix B

LPdoc plugin for exfilter

LPdoc plugin for exfilter

Table of Contents

1 Summary

This module defines the LPdoc `exfilter` command which incorporates the analized results (or projections of them) in documentation.

exfilter_lpdoc

LPdoc command `@exfilter{File}{Opts}` concatenates `File` with `Options`, find the result file and ensure that this file exists, and incorporate it in the documentation.

Usage and interface

- **Library usage:**

  `@exfilter{code/app.pl}{A,filter=tpred}`
  will include the contents of the file `results/app--A--filter=tpred.txt` in the documentation.
- **Exports:**
  - *Predicates:*
    `make_file_nofail/2`, `split_file/2`, `concatenate_items/3`, `concatenate_opts/2`, `split_opts/2`.
  - *Multifiles:*
    `doc_cmd_type/1`, `doc_cmd_rw/2`.

Documentation on exports

make_file_nofail/2: PREDICATE

Usage: `make_file_nofail(F,String)`

If `F` exists then reads all the characters from the `F` and returns them in `String`. If `F` does not exist then writes a Warning message to `F`

split_file/2: PREDICATE

Usage: `split_file(File,Name)`

`Name` is the `File` name

concatenate_items/3: PREDICATE

Usage: `concatenate_items(File,Opts,Result)`

`Result` is the concatenation of `File` and `Opts`

concatenate_opts/2: PREDICATE

Usage: `concatenate_opts(Item,[ConItems])`

Concatenate a list with '–'

split_opts/2: PREDICATE

Usage: `split_opts(ConcList,SplitList)`

Split a list separated by ','

Documentation on multifiles

doc_cmd_type/1: PREDICATE

Usage: `doc_cmd_type(exfilter(s,s))`

{file}{opts}

The predicate is *multifile*.

doc_cmd_rw/2: PREDICATE

Usage: `doc_cmd_rw(exfilter(File,Opts),R)`

Concatenates `File` with `Opts`, find the result file and ensure that this file exists.

The predicate is *multifile*.

Documentation on imports

This module has the following direct dependencies:

- *Application modules:*

  `lists`, `pathnames`, `system`, `stream_utils`, `system_extra`, `classic_predicates`, `process`.
- *Internal (engine) modules:*

  `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `io_basic`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `fsyntax`, `doc_module`.

References

(this section is empty)