CiaoPP Tutorials

A set of CiaoPP Tutorials
The Ciao Documentation Series
`https://ciao-lang.org/`
Generated/Printed on: 22 July 2023

The Ciao Development Team
Edited by:
Manuel Hermenegildo
Pedro López
José Francisco Morales

The Computational logic, Languages, Implementation, and Parallelism (CLIP) Lab
`https://www.cliplab.org/`
School of CS, T. U. of Madrid (UPM)
IMDEA Software Institute

Table of Contents

1 Summary

This document/site groups a number of **interactive tutorials** meant to help when starting to use `CiaoPP`. It is a companion to the **reference** manual for `CiaoPP`, which gives a fuller description of the system.

2 Introduction

This document groups a number of **tutorials** meant to help when starting to use `CiaoPP`. It is a companion to the **reference** manual for `CiaoPP`, which gives a fuller description of the system. Some of these tutorials are *interactive*, using the Active Logic Documents facility of `LPdoc`.

The following tutorials are available:

- Chapter 3 [CiaoPP at a glance], page 5. This brief tutorial provides a quick overview suitable for understanding the some basics about how `CiaoPP` works. It is however is not intended to be a tutorial on how to use `CiaoPP`. The following tutorial is recommended for this.
- ⟨undefined⟩ [A Gentle Introduction to Static Verification and Bug Finding with CiaoPP], page ⟨undefined⟩. This tutorial illustrates step-by-step the development of a concrete program starting from a given problem statement and a specification, expressed using the `Ciao` assertion language, to describe predicates and their properties. Our aim is to show how to use `CiaoPP` to prove *statically* whether these assertions hold and/or detect bugs, while stepwise developing the program. The tutorial also provides an introduction to the dynamic checking aspects of `CiaoPP`. In particular, the document includes the following topics:
  - Defining modules and exports
  - Type and Mode Analysis
  - Assertions and properties
  - Non-failure and Determinacy Analysis
  - Dynamic Bug Finding with `CiaoPP`'s Testing Facilities
- ⟨undefined⟩ [Program Development using the CiaoPP Program Processor], page ⟨undefined⟩ is a more advanced tutorial. The tutorial includes these sections:
  - Static Analysis and Program Assertions
    - Static Analysis Basics
    - Type Analysis
    - Non-failure and Determinacy Analysis
    - Size, Resources, and Termination Analysis
    - Decidability, Approximations, and Safety
  - Program Debugging and Assertion Checking
    - Assertions and Properties
    - Using analysis information for debugging
    - Static Checking of Assertions in System Libraries
    - Static Checking of User Assertions and Program Validation
    - Performance Debugging and Validation

2.1 About CiaoPP

CiaoPP 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. These tasks include:

- Inference of properties of the predicates and literals of the program, including types, modes and other variable instantiation properties, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc.
- Certain kinds of static debugging and verification, finding errors before running the program. This includes checking how programs call system library predicates and also checking the assertions present in the program or in other modules used by the program. Such assertions represent essentially partial specifications of the program.

- Several kinds of source to source program transformations such as program specialization, slicing, partial evaluation of a program, program parallelization (taking granularity control into account), inclusion of run-time tests for assertions which cannot be checked completely at compile-time, etc.
- The abstract model of the program inferred by the analyzers can be used to certify that untrusted mobile code is safe w.r.t. a given policy (this implements the *abstraction-carrying code* approach to mobile code safety).

The information generated by analysis, program properties to be verified, descriptions of the system libraries, directives provided to guide analysis, etc. are all written in the same assertion language, which is in turn also processed by the Ciao system documentation generator, `lpdoc`.

CiaoPP is distributed under the GNU general public license.

3 CiaoPP at a glance

Author(s): Isabel Garcia-Contreras, Daniela Ferreiro.

The purpose of this document is to quickly illustrate some of the kinds of static processing that `CiaoPP` can perform. Note that this is not intended to be a tutorial, for that please follow ⟨undefined⟩ [A Gentle Introduction to Static Verification and Bug Finding with CiaoPP], page ⟨undefined⟩ or ⟨undefined⟩ [Program Development using the CiaoPP Program Processor], page ⟨undefined⟩.

3.1 Setup

In order to follow these examples you need to either:

- Install `Ciao` on your computer, including the development environment (see ⟨undefined⟩ [Installation], page ⟨undefined⟩ for more information). This includes `CiaoPP` (and `LPdoc`). You can then access `CiaoPP` from the Emacs interface or the command line.
- Run `CiaoPP` directly on your browser through the `Ciao` playground. To this end, load the examples into the playground by pressing the ◊ button ("Load in playground"), and then `CiaoPP` can be run clicking the ⟨More...⟩ button and selecting ⟨Analyze and check assertions⟩.

The instructions below use the Emacs interface but the steps can also be performed in the playground version.

3.2 Analyzing

Let us analyze this implementation of the append predicate:

```
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
```

Automatic analysis can be performed by using the direct access button in the Emacs interface .

By default, `CiaoPP` analyzes programs with a type domain (`eterms`) and a sharing/freeness (modes) domain (`shfr`). We can open the file with the inferred information by typing `C-c C-v`.

The inferred information is expressed with (`true`) assertions (for a more extensive tutorial of the assertion language see ⟨undefined⟩ [Using assertions for preprocessing programs], page ⟨undefined⟩). Assertion marked as `true` contain safe approximations (safe abstractions) of the behavior of the program that were inferred by the analyzer. In this case `CiaoPP` inferred:

```
:- true pred app(A,B,C)
   : mshare([[A],[A,B],[A,B,C],[A,C],[B],[B,C],[C]])
   => mshare([[A,B,C],[A,C],[B,C]]).

:- true pred app(A,B,C)
   : ( list(A), list(B), term(C) )
   => ( list(A), list(B), list(C) ).
```

The first assertion contains type information and expresses the fact that *predicate* `app/3` *is called (*`:` *field) with* `A` *and* `B` *bound to* `lists` *and, if it succeeds (*`=>` *field), then* `C` *will be a* `list`.

The second assertion contains information about how variables are shared. It was inferred that when `app(A,B,C)` is called arguments `A`, `B`, and `C` may be shared and if it succeeds they also will be shared, sice `C` is the concatenation of `A` and `B`.

3.3 Assertion Checking

In the example above, we have also added an assertion with properties that we want to prove about the execution of the program.

```
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
```

This assertion is stating that if predicate `append/3` is called with `A` and `B` bound to `lists`, then, if the call succeeds, `C` will be a free variable. This is not true in general of course. We can check this assertion by clicking the icon in the Emacs interface and we obtain:

```
ERROR (ctchecks_pred_messages): (lns 3-3) 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.
```

Since this assertion does not hold we get a message saying so.

Assertion checking can also be reported within the source code. If we open the output file the `CiaoPP` produces (`C-c C-v`), we see that the analyzer proves the assertion that we had included false:

```
:- checked calls app(A,B,C)
   : ( list(A), list(B) ).

:- false success app(A,B,C)
   : ( list(A), list(B) )
   => var(C).
```

So we correct it with:

```
:- pred app(A,B,C) : (list(A), list(B)) => list(C).
```

and run the analysis again. In this case we get no error messages and, as before, results can also be reported in the source file:

```
:- checked calls app(A,B,C)
   : ( list(A), list(B) ).

:- checked success app(A,B,C)
   : ( list(A), list(B) )
   => list(C).
```

3.4 Program Optimization

The following program naively implements a predicate that duplicates the first element of a list.

```
:- module(_, [dup_first/2], []).

dup_first([X|Xs], Zs) :-
    app([X], [X|Xs], Zs).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
```

`CiaoPP` can automatically optimize sources by using the button:

```
:- module(_1,[dup_first/2],[assertions]).

dup_first([A|B],[A,A|B]).
```

3.5 Advanced Preprocessing

`CiaoPP` has several parameters for each of the main actions. For more information, see the ⟨undefined⟩ [Program Development using the CiaoPP Program Processor], page ⟨undefined⟩.