An s(CASP) In-Browser Playground based on Ciao Prolog*

Guillermo García-Pradales[1,2], José F. Morales[1,2], Manuel Hermenegildo[1,2], Joaquín Arias[3] and Manuel Carro[1,2]

[1] *Universidad Politécnica de Madrid, Spain*
[2] *IMDEA Software Institute, Madrid, Spain*
[3] *CETINIA, Universidad Rey Juan Carlos, Madrid*

Abstract

In recent years Web browsers are becoming closer and closer to full-fledged computing platforms. Ciao Prolog currently includes a browser-based playground which allows editing and running programs locally in the browser with no need for server-side interaction. The playground is built from reusable components, and allows easily embedding runnable Prolog programs in web pages and documents. These components can also be easily used for the development of specific, fully browser-based applications. The purpose of this paper is to present a browser-based environment for developing s(CASP) programs, based on the Ciao Prolog playground and its components. This s(CASP) playground thus runs locally on the browser with no need for interaction with a server beyond code download. After briefly introducing s(CASP) and Ciao Prolog, we provide an overview of the architecture of the Ciao playground, based on a compilation of the Ciao engine to the WebAssembly platform, describe its adaptation to create the s(CASP) playground, and present some of its capabilities. These include editing and running s(CASP) programs, sharing them, obtaining (sets of) answers, and visualizing and exploring explanations.

Keywords

s(CASP), Ciao Prolog, Web Playground, WebAssembly, Prolog, ASP, Browser-based applications.

1. Introduction

In recent years there has been an explosion of *online* tools and resources to create, compile, and execute programs in all major programming languages. Notable examples are https://tio.run/, which allows the execution of 681 different programming languages, dialects, and implementations; or other commercial sites like replit https://replit.com/, GitHub CodeSpaces https://github.com/features/codespaces, or GitPod https://www.gitpod.io/. Prolog is not an exception. An early example is represented by the Ciao Prolog JavaScript compiler back-end [1], that enabled the use of Prolog and, in general, (constraint) logic programming to develop not

*Partially funded by MICINN projects PID2019-108528RB-C21 *ProCode* and RTI2018-095390-B-C33 *InEDGEMobility* (MCIU/AEI/FEDER, UE), by the Comunidad de Madrid program P2018/TCS-4339 *BLOQUES-CM*, by the EIT Digital program *BDExp*, and by the Tezos foundation.

2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE'22), August 1, 2022

guillermo.gpradales@alumnos.upm.es (G. García-Pradales); josef.morales@imdea.org (J. F. Morales); manuel.hermenegildo@imdea.org (M. Hermenegildo); joaquin.arias@urjc.es (J. Arias); manuel.carro@imdea.org (M. Carro)
0000-0002-7804-4797 (G. García-Pradales); 0000-0001-9782-8135 (J. F. Morales); 0000-0002-7583-323X (M. Hermenegildo); 0000-0003-4148-311X (J. Arias); 0000-0001-5199-3135 (M. Carro)

just the server side, but also the client side of web applications, running fully on the browser. Tau Prolog [2] and the tuProlog playground (https://pika-lab.gitlab.io/tuprolog/2p-kt-web) are Prolog interpreters in JavaScript which also make it easy to run Prolog in a web page, serverless. SWISH (https://swish.swi-prolog.org), provides access via browser to (a subset of) SWI-Prolog running on a server, which makes it possible to edit logic programs and run queries, and also to create "notebooks" of code and queries.

Such approaches are attractive, but they also have their drawbacks. In the case of server-based solutions, they obviously introduce a dependency on the server. Maintaining a server-side infrastructure can represent a significant burden, and may also in some cases affect other aspects, such as scalability or privacy. Compilation to JavaScript was an attractive option at the time, since it was a client (i.e., browser)-based solution, but it is also not optimal, among other aspects because the resulting execution speed, while useful for many applications, does suffer with respect to native implementations (see [1]). This is even more pronounced in the case of Prolog interpreters written in JavaScript. It is precisely this performance impact that has led to the development of the WebAssembly virtual machine (https://webassembly.org/), which is currently supported by all major browsers. This has led in turn to the development of the Ciao Prolog playground (https://ciao-lang.org/playground) and its related components, which are based on a compilation of the Ciao engine to WebAssembly. As a result the playground allows editing and running programs locally in the browser with no need for server-side interaction, while offering performance that is competitive with native Prolog implementations. Other capabilities include:[1]

- Easy integration of editable, runnable Prolog code in documents, slides, notebooks, web pages, etc., under different formats, e.g., markup, html, pdf, among others.
- Built-in evaluation of unit tests, which can be used to include self-assessment in the notebook or manual.
- Also, built-in auto-documentation, static analysis, and other development tools.
- The modular architecture makes it possible to customize the playground for different applications.

Building on this last point, in this work we present the Ciao Playground for s(CASP) (https://ciao-lang.org/playground/scasp.html). s(CASP) is a novel non-monotonic reasoner, developed originally in Ciao Prolog and currently also available in SWI Prolog. A playground for s(CASP) was developed previously using SWISH [3]. Its execution is therefore server-based. The purpose of this work has been to achieve a *browser-local* development environment for s(CASP). To this end, the new s(CASP) playground uses the Ciao playground components and the Ciao Prolog engine compiled to WebAssembly. The built-in capabilities of this playground currently include editing and running s(CASP) programs, sharing them, obtaining (sets of) answers, and visualizing and exploring explanations.

In the following, after briefly introducing s(CASP) and Ciao Prolog, we provide an overview of the architecture of the Ciao playground and the compilation of the Ciao engine to the WebAssembly platform. We then describe its adaptation to create the s(CASP) playground and present some of its capabilities. Finally, we present some conclusions.

[1] See, e.g., this example: http://ciao-lang.org/ciao/build/doc/ciao_playground.html/factorial_peano_iso.html.

2. Background: Ciao Prolog and s(CASP)

Ciao Prolog [4] is a multi-paradigm programming system that, in addition to supporting logic programming (and, in particular, Prolog), provides the programmer with a large number of useful features from different programming paradigms and styles. Also, the use of each of these features (including those of Prolog) can be turned on and off at will for each program module. Thus, a given module may be using, e.g., higher order functions and constraints, while another module may be using assignment, predicates, Prolog meta-programming, and concurrency. Furthermore, the language is designed to be extensible in a simple and modular way using "packages" (syntactic and semantic extensions) and "bundles."

Another important aspect of Ciao is its programming environment, which provides a powerful preprocessor (with an associated assertion language) capable of statically finding non-trivial bugs, verifying that programs comply with specifications, and performing many types of optimizations (including automatic parallelization). Such optimizations produce code that is highly competitive with other dynamic languages or, with the (experimental) optimizing compiler, even that of static languages, all while retaining the flexibility and interactive development of a dynamic language. This compilation architecture supports modularity and separate compilation throughout. The environment also includes a powerful autodocumenter, lpdoc, and a unit testing framework, both closely integrated with the assertion system. More details can be found in [4, 5].

s(CASP) is a novel non-monotonic reasoner, that evaluates Constraint Answer Set Programs without a grounding phase, either before or during execution. s(CASP) supports predicates and thus retains logical variables (and constraints) both during the execution and in the answer sets. The operational semantics of s(CASP) relies on backward chaining, which is intuitive to follow and lends itself to generating explanations that can be translated into natural language [6]. The execution of an s(CASP) program returns partial stable models, that are the relevant subsets of the stable models, i.e., include only the (negated) literals needed to support the query. s(CASP) has been already applied in relevant fields mainly related to the representation of commonsense reasoning. More details can be found for example in [7, 8].

3. The Ciao Prolog Playground

In this section, we explain details of the Ciao Playground architecture and how we implemented its main features. Fig. 1 shows a (simplified) flowchart that illustrates how the different parts of the playground are compiled and integrated. We explain next the Ciao Playground architecture, using the labels in the components.

3.1. Architecture based on WebAssembly

The Ciao Playground is a web-based Prolog development tool that allows the execution of programs and queries through the browser. In order to execute Prolog code on the browser, an off-the-shelf Ciao engine (see Section 3.2) has been compiled to WebAssembly using the Emscripten C compiler:

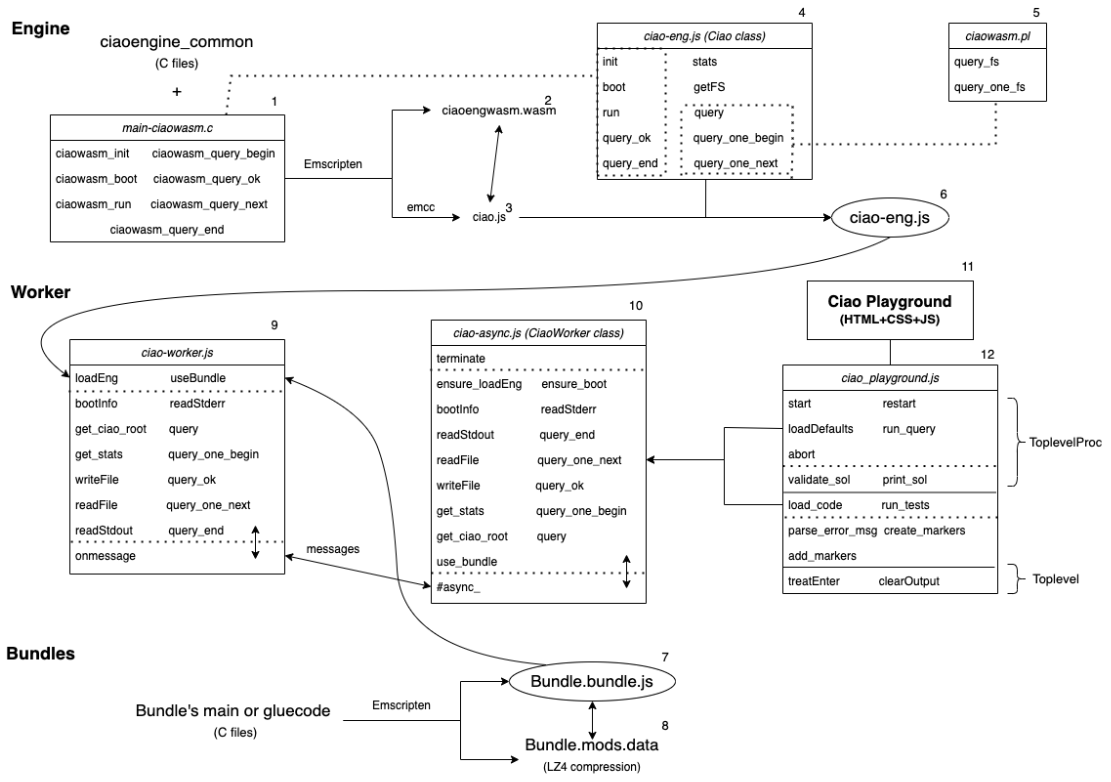


Figure 1: CiaoWasm and Ciao Playground architecture.

- WebAssembly[2] is a portable binary-code format to run high-performance applications on the web at near-native speeds.
- Emscripten[3] is an LLVM-based compiler that can generate WebAssembly binaries. It also has support for packing data files, which we use to deliver Ciao bundles as compact LZ4-compressed `data` files.

Thus, using the Ciao library for C, which offers functions for term creation, type conversions, term checks, queries and calls, a reduced version of the engine was compiled to WebAssembly with Emscripten. Additionally, thanks to the modular environment comprised of several independent bundles, they can be compiled independently and loaded from WebAssembly when required.

3.2. Implementation based on CiaoWasm

CiaoWasm is distributed as a Ciao bundle that provides a variant of the Ciao engine and a set of rules for the Ciao builder, allowing the packing of collections of Ciao bytecode as `.data` files. The `.data` file (label 8) contains all the binary files needed to fully load the bundle, which

[2]https://webassembly.org/
[3]https://emscripten.org/

are loaded by the JavaScript file `.bundle.js` (label 7), including the instructions to import the main compiled code. On the other hand, the `ciaoengine_common` C files (the canonical engine) are compiled along with `main-ciaowasm.c` (label 1). The last one defines some basic functions for the initialization and boot in addition to the queries and calls. It includes the functions `ciaowasm_init`, to initialize the engine, `ciaowasm_boot`, for the engine boot, and `ciaowasm_run`, to run a query.

Emscripten generates the WebAssembly module (label 2) and the auto-generated JavaScript file that makes use of it (label 3). These files are complemented with `ciao-eng.js` to create the JavaScript interface for the playground. `ciao-eng.js` (label 4) includes functions to initialize the Emscripten simulated file system and the functions defined in `main-cioawasm.c`. Besides, it also defines the functions `query`, `query_one_begin` and `query_one_next`, which execute the Prolog predicates `query_fs` and `query_one_fs` in the Emscripten file system. These predicates are defined in the `ciaowasm.pl` file (label 5) and run a query through the simulated file system — they read a query from a `ciaowasm-in.pl` file, execute it obtaining all or one of its solutions respectively and then write them in `ciaowasm-out.pl`. The output of this process is the final `ciao-eng.js` file (label 6) containing the JavaScript Ciao functions and importing the main files for the engine.

The engine of Ciao is controlled by the class `CiaoWorker` (label 10). This class acts as a wrapper of the main functions provided by the CiaoWasm bundle and communicates with the main thread via a message system using `postMessage()` to send data and the `onmessage` event to receive it. This worker is defined in the `ciao-worker.js` file (label 9), which includes functions like `loadEng` to import the engine's `ciao-eng.js` file and `useBundle` to load the `.bundle.js` file. In addition, the `query`, `query_one_begin`, `query_ok`, ... functions call the Ciao functions defined in the engine. This file is complemented with `ciao-async.js` (label 10), which defines the `CiaoWorker` class that calls the `ciao-worker.js` (label 9) functions asynchronously. This is performed through the `#async_` function, which communicates with the other file via messages, as any worker. The asynchronous technique in JavaScript enables web programs to run long tasks without blocking the page to wait for the processes to end. Promises are returned by asynchronous functions and are executed when the function finishes. The purpose of introducing asynchrony in this worker is to allow Ciao to perform interactive and long-running tasks without blocking the page and to be able to control the execution of Ciao.

Furthermore, as explained before, Ciao runs natively on the browser and thanks to the lack of a server and the high performance of WebAssembly, the process is fast and smooth, once data has been loaded (for the first time).

Example 1 (Ciao-core). Consider the `ciao-core` bundle of size 7.5 MB:

- The first time the browser imports this bundle it takes about 200-250 ms to download.
- Then, since the bundle remains in the browser's cache, to import the core the download time decreases to 30-40 ms.

Additionally, thanks to the WebAssembly's near-native speeds, the execution of queries is also fast. While, the current version of WASM is limited to a 32-bit architecture, a 64-bit version (which is in the works) would improve its performance.

As stated above, the Ciao system is modular and composed of several independent bundles, which are loaded when starting the playground. These bundles also enable changing the language and s(CASP) is its own language inside the Ciao environment. Moreover, since the playground executes the queries by calling a custom-made Prolog predicate — in charge of performing the task —, it can be modified to run s(CASP) actions by default (see Section 4).

3.3. Interface using Monaco

The playground interface (label 11) consists of several editor components placed side by side and is written using JavaScript and the Monaco editor API.[4] The two main components are:

- A main editor where the user types the programs.
- A top-level to perform queries, which are executed through a web worker providing asynchronous functions to communicate with Ciao.

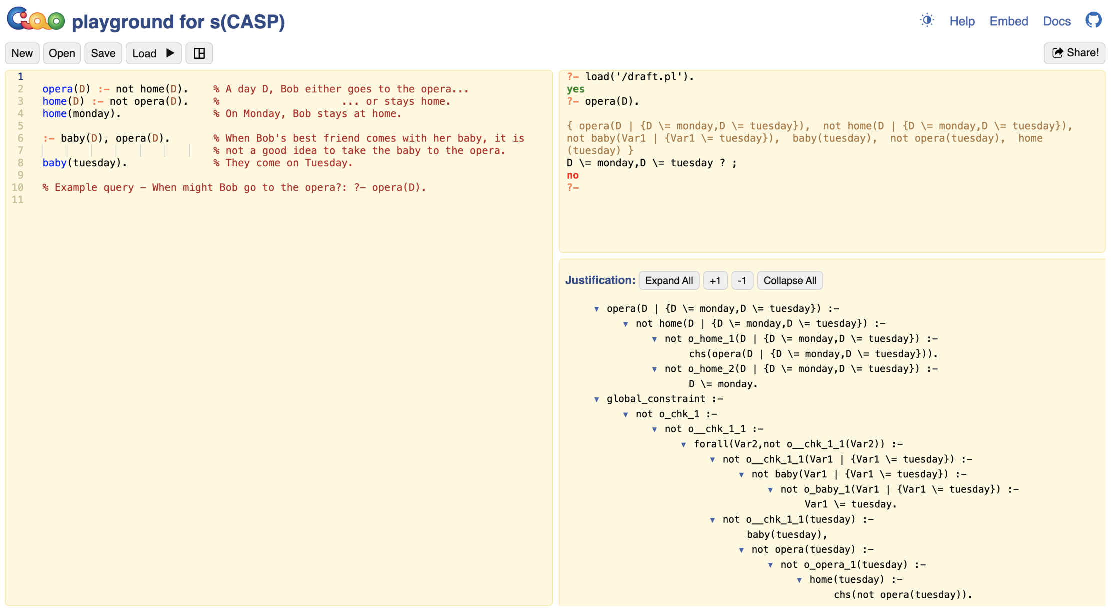
When the playground is opened, the page creates the `CiaoWorker` (label 10) to import the main bundles for the top-level — `core`, `builder` and `ciaodbg`. Besides, using the Monaco Editor API, the top-level (level 12) has been equipped with specific functions to trigger in case of pressing keys like `Enter` or the arrows. Thus, every time the user types a query and presses `Enter`, these functions call the `CiaoWorker` to perform the Ciao-related actions. In order to correctly print the output on the top-level, the solution of these queries is parsed and validated before finally showing them in the playground. For some of the most used actions, like loading the code in the top-level or running tests, the `load_code` and `run_tests` functions perform these tasks directly. Besides, the errors/warnings introduced by the user in their code are also displayed right where they were generated with a description message, similar to the Ciao Emacs *VeryFly*[5], using the functions `parse_error_msg`, `create_markers` and `add_markers` to parse the messages and mark them using Monaco's markers.

4. Ciao Playground for s(CASP)

The Ciao Playground for s(CASP) (see Fig 2) is a specialized playground specifically for running s(CASP) programs. While an implementation using Ciao Prolog's module interface can be used, similar to how it is implemented in SWI-Prolog, i.e., defining the predicate `?/1` to evaluate a given query using the s(CASP) module, the s(CASP) playground pre-processes the queries avoiding the use of `?/1`. I addition, the s(CASP) playground makes a specific use of the a third component in the Ciao playground interface, the *preview panel*, to display an expandable HTML justification tree (bottom right of Fig 2).

The source code of the Ciao Prolog Playground, including the playground for s(CASP), is available at https://github.com/ciao-lang/ciao.

[4]Monaco Editor, at https://microsoft.github.io/monaco-editor/, is a web-based editor provided by Microsoft.
[5]On-the-fly assertion checking, displaying the results graphically in the Emacs IDE using *flychecks* [5].



```

opera(D) :- not home(D).    % A day D, Bob either goes to the opera...
home(D) :- not opera(D).    %                          ... or stays home.
home(monday).               % On Monday, Bob stays at home.

:- baby(D), opera(D).       % When Bob's best friend comes with her baby, it is
                            % not a good idea to take the baby to the opera.
baby(tuesday).              % They come on Tuesday.

% Example query - When might Bob go to the opera?: ?- opera(D).

```

```
?- load('/draft.pl').
yes
?- opera(D).

{ opera(D | {D \= monday,D \= tuesday}),  not home(D | {D \= monday,D \= tuesday}),
not baby(Var1 | {Var1 \= tuesday}),  baby(tuesday),  not opera(tuesday),  home
(tuesday) }
D \= monday,D \= tuesday ? ;
no
?-
```

```
▾ opera(D | {D \= monday,D \= tuesday}) :-
    ▾ not home(D | {D \= monday,D \= tuesday}) :-
        ▾ not o_home_1(D | {D \= monday,D \= tuesday}) :-
              chs(opera(D | {D \= monday,D \= tuesday})).
        ▾ not o_home_2(D | {D \= monday,D \= tuesday}) :-
              D \= monday.
▾ global_constraint :-
    ▾ not o_chk_1 :-
        ▾ not o__chk_1_1 :-
            ▾ forall(Var2,not o__chk_1_1(Var2)) :-
                ▾ not o__chk_1_1(Var1 | {Var1 \= tuesday}) :-
                    ▾ not baby(Var1 | {Var1 \= tuesday}) :-
                        ▾ not o_baby_1(Var1 | {Var1 \= tuesday}) :-
                              Var1 \= tuesday.
                ▾ not o__chk_1_1(tuesday) :-
                      baby(tuesday),
                    ▾ not opera(tuesday) :-
                        ▾ not o_opera_1(tuesday) :-
                            ▾ home(tuesday) :-
                                  chs(not opera(tuesday)).
```

Figure 2: The Ciao Prolog Playground for s(CASP).

```
Opera(D) :- not home(D).    % A day D, Bob either goes to the opera...
home(D) :- not opera(D).    %                          ... or stays home.
home(monday).               % On Monday, Bob stays at home.

:- baby(D), opera(D).       % When Bob's best friend comes with her baby, it is
                            % not a good idea to take the baby to the opera.
baby(tuesday).              % They come on Tuesday.
```

Figure 3: The s(CASP) program `opera.pl`.

4.1. File and layout management

Above the editor component there are three buttons that allow writing *new* programs, *open*ing files, and/or *save*ing them. The files are loaded into the local storage of the browser, created in a cookie, called *code*, so the information is retained on the user's computer, i.e., no data is allocated on an external server.

A fourth button, called *load*, compiles and loads the program contained in the editor component into the top-level component.

Finally, the leftmost button above the editor component allows the user to change the layout. In particular, it allows different locations (and sizes) for the preview components.

4.2. Sharing code through the URL

A main feature of the Ciao Playground platform is its ability to share code and/or include code in notebooks written in different formats. In a similar way, the Ciao Playground for s(CASP) can be invoked, including a program, through a URL.

5. Using the s(CASP) playground

We now use the following example to explain the main features of the Ciao Playground for s(CASP).

Example 2 (Example 7 in [8]). Consider the program `opera.pl`, in Fig. 3, which models when Bob might go to the opera. The denial in line 5 expresses that `baby(D)` and `opera(D)` cannot happen simultaneously, i.e., when Bob's best friend comes with her baby, it is not a good idea to go to the opera. Thus, performing the query `?- opera(D)` results in the following partial model:

```
{ opera(D | {D \= monday,D \= tuesday}), not home(D | {D \= monday,D \= tuesday}),
  not baby(Var1 | {Var1 \= tuesday}), baby(tuesday), ... }
```

where the atom `opera(D | {D \= monday,D \= tuesday})` means that Bob can go to the opera any day except on Mondays and Tuesdays.

As we mentioned before, we can share code using URLs. As an example, by clicking https://ciao-lang.org/playground/scasp.html#[...], where `[...]` encodes the `opera.pl` program (Example 2) in plain text, the Ciao Playground for s(CASP) will be opened including that program in the editor component (see Fig. 2). Once the playground is running, the program is loaded automatically in the top level. If needed, we can modify the program, and the new version can be loaded into the top-level, clicking the *Load* button or typing the Emacs key binding `C-c l`.[6]

As also mentioned, the top-level component is used to make the queries and to display the partial models and the bindings resulting from the evaluation of the program by s(CASP). The top-level component in Fig. 2 shows the results for the evaluation of the query `?- opera(X)` for Example 2.

Additionally, the preview panel in Fig. 2 shows the expandable justification tree in HTML that we could generate by invoking `scasp --tree --large --html file` in a local installation of s(CASP). As expected, the user can navigate through the justification tree using the buttons (Expand All / +1 / -1 / Collapse All) to expand/collapse the decision branches.

6. Conclusions

In this paper we have presented a full operative environment where people can edit and consult s(CASP) programs, and "navigate" the results interactively.

[6]Note that this is the key binding defined by the Ciao mode for Emacs. Supporting these key bindings in addition to the buttons and mouse facilitates the use of the playground for Emacs users.

The environment is built on top of the Ciao Playground, which, thanks to WebAssembly, can run unmodified versions of Ciao on any device running a modern Web browser. WebAssembly is recent Web standard that defines a binary low-level code format. We use LLVM-based Emscripten to generate a WebAssembly version of the Ciao engine directly from its C sources. The core of the system is wrapped as JavaScript objects running in dedicated Web Workers. This offers a nice and extensible architecture that has a relative low maintenance cost: using unmodified engines and Prolog compilers takes advantages of decades of research in logic programming implementation, while the use of portable and standardized technology like WebAssembly, and even ISO C makes sure that the system will continue working. This is extremelly important to focus the development efforts.

One of the attractiveness of this approach is near-native speeds (fast enough for many applications) with instant delivery (no need to install anything). Combining a high-performing back-end with an easy and appealing user interface, we produced a powerful tool with several applications, such as: embedding s(CASP) programs in any kind of document via the URL, and interactive tutorials.

References

[1] J. F. Morales, R. Haemmerlé, M. Carro, M. V. Hermenegildo, Lightweight compilation of (C)LP to JavaScript, Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue 12 (2012) 755–773.

[2] τProlog Homepage, τProlog — an open source Prolog interpreter in javascript, http://tau-prolog.org, 2021. Last access: July 22, 2022.

[3] J. Wielemaker, J. Arias, G. Gupta, s(CASP) for SWI-Prolog, in: Proceedings of the 37th ICLP 2021 Workshops, volume 2970, CEUR-WS.org, 2021. URL: http://ceur-ws.org/Vol-2970/gdeinvited4.pdf.

[4] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, G. Puebla, An Overview of Ciao and its Design Philosophy, Theory and Practice of Logic Programming 12 (2012) 219–252. doi:10.1017/S1471068411000457.

[5] M. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, M. V. Hermenegildo, Verifly: On-the-fly Assertion Checking via Incrementality, Theory and Practice of Logic Programming 21 (2021) 768–784.

[6] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for goal-directed constraint answer set programming, in: Proceedings 36th International Conference on Logic Programming (Technical Communications), volume 325 of *EPTCS*, Open Publishing Association, 2020, pp. 59–72. doi:10.4204/EPTCS.325.12.

[7] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint Answer Set Programming without Grounding, Theory and Practice of Logic Programming 18 (2018) 337–354. doi:10.1017/S1471068418000285.

[8] J. Arias, G. Gupta, M. Carro, A Short Tutorial on s(CASP), a Goal-directed Execution of Constraint Answer Set Programs, in: Proceedings of the 37th ICLP 2021 Workshops, volume 2970, CEUR-WS.org, 2021. URL: http://ceur-ws.org/Vol-2970/gdepaper1.pdf.