Web development with Tau Prolog

José A. Riaza[a]

[a]*Department of Computing Systems, University of Castilla-La Mancha, Albacete, 02071, Spain*

Abstract

Tau Prolog is a client-side Prolog interpreter fully implemented in JavaScript, which aims at implementing ISO Prolog Standard. Tau Prolog has been developed to be used with either Node.js or a browser seamlessly, and therefore, it has been developed following a non-blocking, callback-based approach to avoid blocking web browsers. Taking the best from JavaScript and Prolog, Tau Prolog allows the programmer to handle browser events and manipulate the Document Object Model (DOM) of a web using Prolog predicates. In this paper we describe the main packages of Tau Prolog for interacting with the Web, and we present its programming environment.

Keywords: Prolog, Web, JavaScript

1. Introduction

HTML, CSS and JavaScript are the basic building blocks of the Web. JavaScript is a programming language that allows the programmer to implement complex things on web pages. The standard for JavaScript is ECMAScript [1, 2]. A web browser provides an ECMAScript host environment for client-side computation, which in turn provides a means to attach scripting code to events such as change of focus, form submission, and mouse actions. The scripting code is reactive to user interaction, and there is no need for a main program. JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. A JavaScript runtime is single threaded and uses a message queue, where each message is completely processed before any other message is treated. See [3] for more details about the event loop. A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions.

While most online Prolog systems, such as SWISH [4], are remote servers with an installed version of the Prolog system which receive code, execute it and return the results, Tau Prolog[1] [5] is fully implemented on JavaScript, and therefore the code can be analyzed and executed directly on the client-side. This allows Tau Prolog to easily create interfaces for the JavaScript Web APIs in order to interact with web pages using Prolog. There are other approaches, such as Prolog to JavaScript compilers [6], or systems like Yield Prolog [7] that allow to simulate the behavior of logic programming in JavaScript through generators. SWI-Prolog pengines [8] also provides an abstraction to create and query Prolog engines over HTTP from JavaScript running in a web client. However, Tau Prolog provides a full client-side Prolog implementation (including features like modules, meta-predicates mechanism, DCGs, term and goal expansion, and ISO Prolog predicates). Furthermore, Tau Prolog is based on a simple and extensible representation, in order to be easy to modify at runtime by users (something difficult to achieve with the previous approaches). With the aim of mitigating the negative effects of the JavaScript concurrency model, Tau Prolog has been developed following a non-blocking, callback-based approach, since looking for computed answers in Prolog can be a heavy task for certain programs.

The source code of Tau Prolog is available on GitHub[2], and it is freely distributed with npm.[3] Tau Prolog aims at implementing the ISO Prolog Standard [9], designed to promote the applicability and portability of Prolog text and data among several data processing systems. The main purpose of Tau Prolog is to provide a complete Prolog system that can be embedded in a web page, for use in web applications that can benefit from Prolog, such as in the validation and intelligent completion of web forms, and in tasks that require dynamic adaptions of the DOM itself based on logical rules. In last years, Tau Prolog has been gaining popularity, both in academia [10, 11, 12] and in real world applications [13, 14], showing some of the potential uses of Tau Prolog. Furthermore, the Tau Prolog sandbox provides an ideal programming environment for teaching, since it does not require any installation and allows derivations to be graphically visualized.

In this paper we describe the main packages of Tau Prolog for interacting

[1]`http://tau-prolog.org`

[2]`https://github.com/tau-prolog/tau-prolog`

[3]`https://www.npmjs.com/package/tau-prolog`

with the Web, and we present its programming environment. The structure of this paper is as follows. In Section 2 we show how to manipulate the DOM with Tau Prolog. Then Section 3 is devoted to describe the event handling. In Section 4 we describe the JavaScript foreign function interface of Tau Prolog. Finally, in Section 5 we show the programming environment for Tau Prolog.

2. DOM manipulation

The *Document Object Model* (DOM) [15] is an API for accessing and manipulating HTML and XML documents, which are presented as node trees. Tau Prolog's DOM package defines new term types and the `dom` module, which adds predicates that allow the user to access and modify the DOM.

- Selector predicates provides methods that make it quick and easy to retrieve element nodes from the DOM by matching against properties. The `dom` module includes three non-deterministic predicates to look for DOM elements: `get_by_id/2`, `get_by_class/2` and `get_by_tag/2`. If there are no elements in the DOM with the specified identifier, class or tag, the predicates fail silently. If there is more than one, they find all of them on backtracking.

- The `create/2` predicate takes an atom standing for an HTML tag and creates a new HTML object. Newly created HTML objects can be inserted in the DOM using the following predicates: `append_child/2`, `insert_after/2` and `insert_before/2`.

- Starting at the HTML objects retrieved or created with the previous predicates, the `parent_of/2` and `sibling/2` predicates allow to go through the DOM.

- The following predicates allows the user to consult or modify the content, attributes and styles of HTML objects: `get_attr/3`, `set_attr/3`, `get_html/2`, `set_html/2`, `get_style/3`, `set_style/3`, `add_class/2`, `remove_class/2` and `has_class/2`.

- Lastly, this module also includes predicates to create animations, such as `hide/1`, `show/1` or `toggle/1`, that hides, shows, or toggles the visibility of an HTML object, respectively.

Example 1. *Suppose we want to create a web application to make shopping lists.*[4] *The interface contains a text input with id "`input_item`" to enter the name of the product, a text input with id "`input_count`" to enter the quantity, and a button with id "`button_add`" to add items.*

```
:- use_module(library(dom)).
:- use_module(library(format)).

get_item(Item) :-
    get_by_id(input_item, Input),
    get_attr(Input, value, Item).

get_count(Count) :-
    get_by_id(input_count, Input),
    get_attr(Input, value, Atom),
    atom_chars(Atom, Chars),
    number_chars(Count, Chars).

add_item(Item, Count) :-
    get_by_id(todo, ToDo),
    create(li, Li),
    append_child(ToDo, Li),
    open(Li, write, Stream),
    format(Stream, "~a (~d)", [Item, Count]),
    close(Stream).
```

The `add_item/2` predicate adds a new item to the list, and the `get_item/1` and `get_count/1` predicates retrieve the data from the inputs. Notice that we can open an HTML element as a regular Prolog stream, and read from it or write to it.

3. Event handling

The `dom` module also enables the dynamic assignation of events, in order to run a Prolog goal when some browser event is triggered. The `bind/4` method takes an HTML object, an atom representing an event type (`click`,

[4]`http://tau-prolog.org/examples/shopping-list`

`mouseover`, `mouseout`, etc.), an event and a goal, and binds the HTML object with the Prolog goal for that type of event. The third argument is bound to a new term that represents an HTML event, from which we can read information using the `event_property/3` predicate. Finally, the `unbind/2` and `unbind/3` predicates allow us to remove the events attached to an HTML object. The `prevent_default/1` predicate allows us to prevent the browser default behaviour regarding an event (for instance, to keep a form from being sent). A list with the events supported by the browser can be found in [16].

Example 2. *To finish the web application shown in Example 1, we need to be able to add a new item to the list every time the user clicks the button.*

```
main :-
    get_by_id(button_add, Button),
    bind(Button, click, _, (
        get_item(Item),
        get_count(Count),
        add_item(Item, Count)
    )).
```

Here, the `main/0` predicate attaches an event to the button in order to add a new item to the list (with the current values of the inputs) every time it is clicked. Therefore, all the application needs to run is to consult the program and execute the Prolog goal "`main`" when the DOM is ready.

It is even possible to deploy simple games or to make animations in the browser with Prolog, just by manipulating the DOM and handling events, as we show in the following example.

Example 3. *Suppose we want users to be able to drag and drop items in our web application, where draggable elements are identified by the class name "`draggable`".*[5]

```
:- use_module(library(dom)).
:- use_module(library(js)).

:- dynamic(draggable/1).
```

[5]http://tau-prolog.org/examples/draggable

```
main :-
    forall(get_by_class(draggable, Elem), (
        bind(Elem, mousedown, _, asserta(draggable(Elem))),
        bind(Elem, mouseup, _, retractall(draggable(Elem))),
        bind(Elem, mousemove, Event, (
            draggable(Elem),
            event_property(Event, pageX, X),
            event_property(Event, pageY, Y),
            move(Elem, X, Y)
        ))
    )).

move(Elem, X, Y) :-
    get_prop(Elem, offsetWidth, Width),
    get_prop(Elem, offsetHeight, Height),
    Top is Y - Height/2,
    Left is X - Width/2,
    set_style(Elem, top, px(Top)),
    set_style(Elem, left, px(Left)).
```

Here, the `main/0` predicate finds all draggable elements, and attaches to each of them a series of events. When the user clicks on a draggable item, Prolog stores a fact that indicates that the element can be moved. When the user releases the item, the fact is removed. When the user moves the cursor over an item, if that element can be moved, Prolog updates the element's position based on the event data.

Notice that the `event_property/3` predicate and the event terms only make sense inside an event's goal, since they don't hold any information until the event is triggered. Any time an event is triggered, Tau Prolog forks the session that assigned the event, and it runs the goal in the new thread (just for the first answer).

4. Foreign function interface

A foreign function interface is a mechanism allowing a program written in a programming language to call routines written in another. The Tau Prolog's JavaScript package defines the `js` module, whose predicates allows the user to invoke JavaScript functions from Prolog programs, send ajax requests, and manipulate JSON data. The `js` module exports a few main predicates to invoke JavaScript functions:

- `apply/4`: invokes a JavaScript function with a list of arguments. The goal `apply(Context, Method, Arguments, Value)` is true if `Value` unifies with the result of calling the method `Method` of the JavaScript object `Context` with arguments `Arguments`. If `Method` is a JavaScript function, `Value` is the result of calling the JavaScript function in the context of the JavaScript object `Context`.

- `get_prop/[2-3]`: reads the value of a property of a JavaScript object. The goal `get_prop(Context, Property, Value)` is true if `Value` unifies with the value of the property `Property` of the JavaScript object `Context`.

- `global/1`: gets the global JavaScript object. The goal `gobal(Context)` is true if and only if `Context` is the global JavaScript object (`window` in browser or `global` in Node.js).

Example 4. *The* ***`Array.prototype.concat`*** *method is used to merge two or more arrays, so it can be applied in the context of an array to get the concatenation of a list of lists. Similarly, the* ***`String.prototype.concat`*** *method can be applied in the context of a string to get the concatenation of a list of atoms:*

```
?- apply([], concat, [[1,2],[3,4,5],[6]], Xs).
Xs = [1,2,3,4,5,6].

?- apply('', concat, [hello, ', ', world, !], Str).
Str = 'hello, world!'.
```

Here, the first and third arguments are translated to JavaScript objects. Lists are converted into arrays, and atoms into strings. Then, the method is applied in the context of the first object, using the objects of the third argument as parameters. Finally, the result is translated back from JavaScript to Prolog.

When a value can not be directly converted from JavaScript to Prolog, such as an object or a function, it is returned wrapped in a new type of term, pl.type.JSValue. JavaScript's objects can be explicitly converted from and to Prolog terms using the json_prolog/2 predicate.

Although the js module exports other predicates such as set_prop/[2-3], which sets the value of a property of a JavaScript object, and new/3, which creates an instance of an object type that has a constructor function, these predicates can be implemented by apply/4 and get_prop/3.

Example 5. *We can manually perform the actions of the new operator, and make a work-around with get_prop/[2-3] and apply/4:*

```
:- use_module(library(js)).

new(Constructor, Args, Instance) :-
    get_prop('Object', Object),
    get_prop(Constructor, prototype, Prototype),
    apply(Object, create, [Prototype], Result),
    apply(Constructor, call, [Result|Args], Instance).
```

Now, we can, for instance, use the function's constructor to create JavaScript functions from Prolog on the fly. This is one of the most convoluted ways to multiply a number by 2:

```
double_me(X, Y) :-
    get_prop('Function', Function),
    new(Function, [x, 'return x*2;'], DoubleMe),
    apply(DoubleMe, [X], Y).
```

The JavaScript foreign function interface of Tau Prolog is really simple but very effective. In fact, everything can be built on these three predicates. As we will see in the following example, these predicates allow to create interfaces with complex JavaScript APIs, such as drawing on <canvas> HTML elements.

Example 6. *Suppose we want to create a snake game in our web application, where the snake is controlled with keys w (up), a (left), s (down) and d (right), and the game is drawn on a canvas with id "snake".*[6] *The entry point is the predicate snake/0, which attaches an event to set the direction of the snake, and starts the logic of the game.*

[6]http://tau-prolog.org/examples/snake

```
snake :-
    once(get_by_tag(body, Body)),
    bind(Body, keydown, Event, (
        event_property(Event, key, Key),
        key_direction(Key, Direction),
        prevent_default(Event),
        retractall(direction(_)),
        asserta(direction(Direction))
    )),
    get_by_id(snake, Canvas),
    apply(Canvas, getContext, ['2d'], Ctx),
    random_between(1, 20, X),
    random_between(1, 20, Y),
    snake(Ctx, [(1,1)], (X,Y)).
```

The direction is stored by a fact of the dynamic predicate ***direction/1****. The* ***key_direction/1*** *predicate takes the key pressed and returns the new direction of the snake. The initial state of the game is a snake with an only block located in the square* $(1, 1)$*.*

```
snake(Ctx, Snake, Point) :-
    get_time(T0),
    draw(Ctx, Snake, Point),
    direction(Direction),
    update(Snake, Point, Direction, Snake1, Point1),
    get_time(T1),
    Timeout is max(0, 60-round(T1-T0)),
    set_timeout(Timeout, snake(Ctx, Snake1, Point1), _).
```

The ***snake/3*** *predicate draws the current state of the game and updates it, over and over. To control the time in which the game goes from one state to another, the time it takes to process a state is measured and subtracted from the time it takes to call the next state with* ***set_timeout/3****.*

```
draw(Ctx, Snake, Point) :-
    apply(Ctx, clearRect, [0,0,200,200], _),
    draw_point(Ctx, Point, red),
    forall(
        member(Body, Snake),
        draw_point(Ctx, Body, black)
    ).

draw_point(Ctx, (X,Y), Color) :-
    PointX is (X-1)*10,
    PointY is (Y-1)*10,
    set_prop(Ctx, fillStyle, Color),
    apply(Ctx, beginPath, [], _),
    apply(Ctx, rect, [PointX,PointY,10,10], _),
    apply(Ctx, fill, [], _).
```

The ***`draw/3`*** *predicate draws the point and the snake through the* ***`draw_point`*** *predicate, that uses the canvas API by calling the* ***`apply/4`*** *predicate. Figure 1 shows the snake game running at different states.*

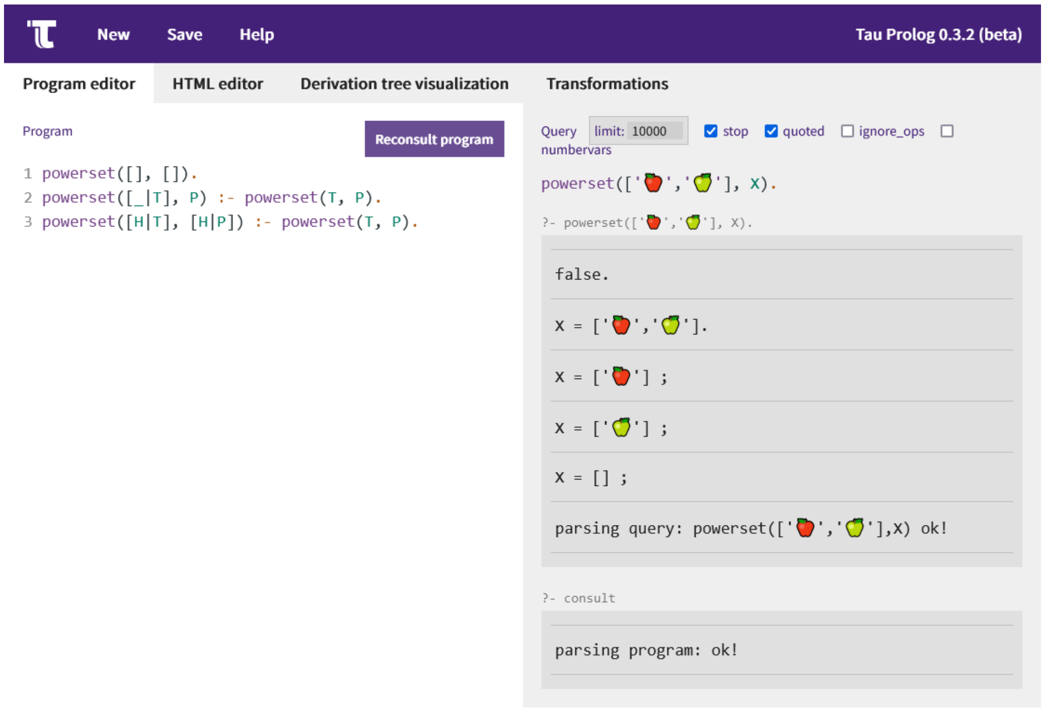
(a) Initial state. (b) Advanced state.

Figure 1: Snake game implemented with Tau Prolog.

5. Programming environment

The *Tau Prolog Sandbox*[7] (see Figure 2) is an online interactive interpreter of Prolog which runs the latest version of all Tau Prolog's packages availables so far. Programs can be freely saved and shared via URL. The browser version of Tau Prolog has a virtual file system, so even programs including operating system interactions –such as manipulate files and directories– can be executed in the sandbox without any problem (of course, interactions with the operating system in Node.js are real).

[7]http://tau-prolog.org/sandbox



```
powerset([], []).
powerset([_|T], P) :- powerset(T, P).
powerset([H|T], [H|P]) :- powerset(T, P).
```

Query limit: 10000 [✓ stop] [✓ quoted] ☐ ignore_ops ☐ numbervars

```
powerset(['🍎','🍏'], X).

?- powerset(['🍎','🍏'], X).

false.

X = ['🍎','🍏'].

X = ['🍎'] ;

X = ['🍏'] ;

X = [] ;

parsing query: powerset(['🍎','🍏'],X) ok!

?- consult

parsing program: ok!
```

Figure 2: Screenshot of the Tau Prolog Sandbox online tool running a Prolog program.

Below the Program editor tab, users can consult Prolog programs and query goals. In the query area, the user can set the limit of inferences in a derivation and the writing options defined by the ISO Prolog Standard [9]. After consulting a Prolog program, under the HTML editor tab, users can insert HTML code and run programs related to it. The Transformations tab lists all the clauses loaded into the session, and allows the user to apply some automatic transformations to the code, such as the unfolding transformation defined in [17]. This tab is especially convenient for viewing code transformations that a Prolog interpreter applies (body conversion, term and goal expansion, Definite Clause Grammar notation, etc). Finally, the Derivation tree visualization tab draws the derivation tree for Prolog goals. Figure 3 shows the graphical derivation tree generated by Tau Prolog for a goal. Each node displays the current goal and substitution in the inference process, where answers are leaves with empty goals "$\square$". Note that only variables in the original goal are shown in substitutions. Nodes representing answers and errors are highlighted with different colors.

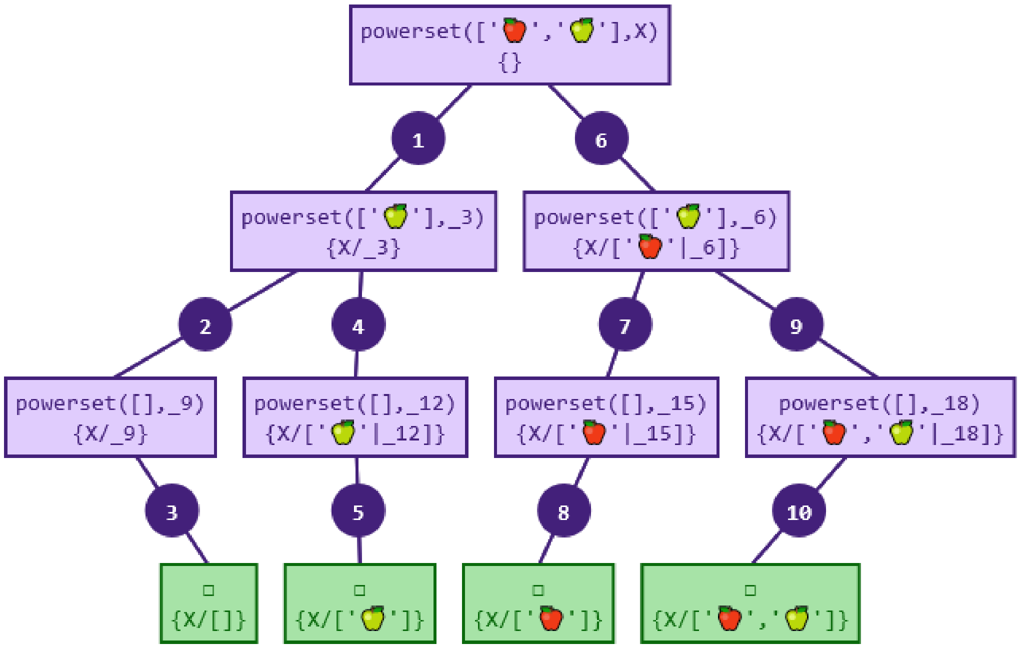


Figure 3: Graphical derivation tree generated by Tau Prolog.

6. Conclusions

In this paper we have described the features of Tau Prolog, a client-side Prolog interpreter for the Web fully implemented in JavaScript, ISO Prolog Standard compliant, which allows to execute –possibly asynchronous– Prolog programs in the browser overcoming the limitations of the JavaScript concurrency model. We have shown the main Tau Prolog's packages to interact with the Web, including a foreign function interface to invoke JavaScript code from Prolog, and the Tau Prolog's programming environment. These packages reflect the main advantages of our proposal compared to other server-side Prolog interpreters. As future work, we hope to incorporate features of Constraint Handling Rules [18] and Constraint Logic Programming [19] to Tau Prolog.

References

[1] Ecma International, ECMAScript 2022 internationalization api specification (2021).

URL `https://tc39.es/ecma402/`

[2] Ecma International, ECMAScript 2022 language specification (2021). URL `https://tc39.es/ecma262/`

[3] Mozilla Contributors, Concurrency model and the event loop (2021). URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`

[4] J. Wielemaker, T. Lager, F. Riguzzi, SWISH: swi-prolog for sharing, CoRR abs/1511.00915 (2015).

[5] J. A. Riaza, Tau Prolog: A prolog interpreter in javascript (2021). URL `http://tau-prolog.org/`

[6] J. F. Morales, R. Haemmerlé, M. Carro, M. V. Hermenegildo, Lightweight compilation of (c) lp to javascript, Theory and Practice of Logic Programming 12 (4-5) (2012) 755–773.

[7] Jeff Thompson, Yield prolog (2021). URL `http://yieldprolog.sourceforge.net`

[8] T. Lager, J. Wielemaker, Pengines: Web logic programming made easy, Theory and Practice of Logic Programming 14 (4-5) (2014) 539–552. doi:10.1017/S1471068414000192.

[9] Information technology – Programming languages – Prolog – Part 1: General core (1995).

[10] L. Brodo, R. Bruni, M. Falaschi, A logical and graphical framework for reaction systems, Theoretical Computer Science 875 (2021) 1–27.

[11] Y. Kirchev, K. Atkinson, T. Bench-Capon, Demonstrating the distinctions between persuasion and deliberation dialogues, in: International Conference on Innovative Techniques and Applications of Artificial Intelligence, Springer, 2019, pp. 93–106.

[12] V. Latypova, V. Martynov, A. Turganov, Decision support system in online training process management for implementing complex open ended assignments in engineering education, in: 2020 V International Conference on Information Technologies in Engineering Education (Inforino), IEEE, 2020, pp. 1–5.

[13] Yarnpkg, Constraints - Yarn Package Manager (2021).
URL `https://yarnpkg.com/features/constraints`

[14] P. Moura, The Logtalk Handbook, Release 3.54.0 Edition (Mar. 2022).

[15] World Wide Web Consortium, et al., Document object model (dom) level 3 core specification (2004).

[16] Mozilla Contributors, Event reference (2021).
URL `https://developer.mozilla.org/en-US/docs/Web/Events`

[17] H. Tamaki, Unfold/fold transformation of logic programs, Proc. of 2nd ILPC (1984) 127–138.

[18] T. Frühwirth, Theory and practice of constraint handling rules, The Journal of Logic Programming 37 (1-3) (1998) 95–138.

[19] J. Jaffar, J.-L. Lassez, Constraint logic programming, in: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1987, pp. 111–119.