TPLP **24** (1): 1–21, 2024. © The Author(s), 2023. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068423000352 First published online 18 September 2023

*Tau Prolog: A Prolog Interpreter for the Web**

JOSÉ A. RIAZA
Department of Computing Systems, University of Castilla-La Mancha, 02071 Albacete, Spain
(*e-mail:* JoseAntonio.Riaza@uclm.es)

submitted 22 July 2021; revised 30 January 2023; accepted 28 August 2023

Abstract

Tau Prolog is a client-side Prolog interpreter fully implemented in JavaScript, which aims at implementing the 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 architecture of Tau Prolog and its main packages for interacting with the Web, and we present its programming environment.

KEYWORDS: Tau Prolog, logic programming, Prolog interpreter, JavaScript

1 Introduction

JavaScript, along with HTML and CSS, constitute the basic building blocks of the Web. Standardized by the ECMAScript specification (Ecma International 2021a;b), JavaScript allows the programmer to implement complex behaviors on web pages. In fact, browsers offer an ECMAScript host environment for client-side computation, providing a means to attach scripting code to events such as change of focus, form submission and mouse actions. Hence, the 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, 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.[1] A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions.

* We thank Miguel Riaza for implementing the parser of Tau Prolog, and for his help in the core development. We thank Dr. Jose Maria Garcia-Garcia for documenting most of the predicates available on Tau Prolog until the present. We thank Dr. Paulo Moura for his help testing Tau Prolog and fixing many bugs of built-in predicates, and for his work on integrating Tau Prolog with Logtalk. We thank Dr. Markus Triska for providing his module for formatting text, originally written for Scryer Prolog. We thank Jan Burse for his help in reporting bugs. We thank Dr. Ginés Moreno and Dr. Pascual Julián-Iranzo for comments on the manuscript. This work has been partially supported by the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2019-104735RB-C42 (SAFER).

[1] See Mozilla Contributors (2021a) for more details about the event loop.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

While most online Prolog systems, such as SWISH introduced by Wielemaker *et al.* (2015), are remote servers with an installed version of the Prolog system which receive code, execute it and return the results, Tau Prolog (Riaza 2022) 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 (Morales *et al.* 2012), or systems like Yield Prolog (Thompson 2021) that allow to simulate the behavior of logic programming in JavaScript through generators. SWI-Prolog pengines (Lager and Wielemaker 2014) also provides an abstraction to create and query Prolog engines over HTTP from JavaScript running in a web client. In the last years, some Prolog systems such as Ciao Prolog,[2] SWI-Prolog[3] and Trealla Prolog[4] added support for WebAssembly compilation (García-Pradales *et al.* 2022) which allows them to run in most major browsers. In addition, we found a few JavaScript Prolog implementations like Tau Prolog really capable of parsing and executing Prolog code in a browser, such as jsProlog,[5] Dogelog[6] and hitchhiker Prolog[7] – an implementation of a Prolog virtual machine proposed by Tarau (2018). 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, that are not present in these JavaScript implementations. 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 approaches such as pengines or WebAssembly).

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. Web workers (Mozilla Contributors 2021d) provide another means for web to execute scripts in background threads without interfering with the user interface, but they don't have direct access to the DOM, it is necessary to execute an entire script, and only serialized objects can be exchanged. For these reasons, the callback-based approach was chosen. Furthermore, users can embed a Tau Prolog computation into a web worker using the current interface.

The source code of Tau Prolog is available on GitHub,[8] and it is freely distributed with npm.[9] Tau Prolog aims at implementing the ISO Prolog Standard (ISO/IEC 13211-1 1995), 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 adaptations of the DOM itself based on logical rules. In last years, Tau Prolog has been gaining popularity, both in academia

[2] https://github.com/ciao-lang/ciaowasm.
[3] https://www.swi-prolog.org/build/WebAssembly.html.
[4] https://github.com/trealla-prolog/trealla.
[5] https://github.com/cubiwan/jsProlog.
[6] https://www.xlog.ch/izytab/doclet/en/docs/23_products/10_jekrun/package.html.
[7] https://github.com/CapelliC/hhprolog.
[8] https://github.com/tau-prolog/tau-prolog.
[9] https://www.npmjs.com/package/tau-prolog.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

(Brodo *et al*. 2021; Kirchev *et al*. 2019; Latypova *et al*. 2020) and in real world applications (Yarnpkg 2021; Moura 2022, Logtalk), showing some of the potential uses of Tau Prolog. Furthermore, the Tau Prolog sandbox provides an ideal environment for teaching, since it does not require any installation and allows derivations to be graphically visualized. Moreover, Tau Prolog is a good option for teaching thanks to its commitment to the Prolog ISO standard and its free accessibility. In addition, it keeps code and data (such as family relations of students, which are frequently used in Prolog exercises) confidential on the client.

In this paper we describe the architecture of Tau Prolog and its main packages for interacting with the Web. The structure of this paper is as follows. In Section 2 we describe the Tau Prolog architecture for loading programs and querying goals asynchronously. Then Section 3 is devoted to describe the main packages of Tau Prolog for the web interaction. In Section 4 we show the programming environment for Tau Prolog. Finally, in Section 5 we present some benchmarks measuring the performance of Tau Prolog.

2 Tau Prolog architecture

All the Tau Prolog functionality is embedded in a JavaScript object named `pl`, which is visible in the global scope after importing the library. In this section, we introduce the main interfaces to work asynchronously with Tau Prolog.

2.1 Prototypes and Prolog objects

Here, we use a ground representation in order to store and manipulate Prolog terms with JavaScript. JavaScript is a prototype-based language (Mozilla Contributors 2021c), where objects can have a `prototype` object, which acts as a template object that it inherits methods and properties from. Tau Prolog defines three basic prototypes to represent Prolog objects:

- The `pl.type.Var` prototype is used to represent logical variables in Prolog. The only argument that the constructor receives is the identifier of the variable as a string.
- The `pl.type.Num` prototype is used to represent numbers in Prolog. The constructor receives two arguments: the number representing the numeric value, and a boolean value which indicates if the value is a floating point number.
- The `pl.type.Term` prototype is used to represent atoms and compound terms in Prolog. The constructor receives a string identifying the term and, if the term is compound, an array of Prolog objects.

The `pl.type.Substitution` prototype is used to represent the substitutions in the resolution process and in the answers. The constructor receives, optionally, a JavaScript object binding variables with Prolog objects.

Example 1
The answer "`X=foo(Z), Y=3`" is represented by the following JavaScript object:

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
new pl.type.Substitution({
    "X": new pl.type.Term("foo", [new pl.type.Var("Z")]),
    "Y": new pl.type.Num(3, false)
});
```

The `pl.type.Session` and `pl.type.Thread` prototypes are used to represent sessions and threads of execution, respectively. The user must create a session in order to load programs. When a session is created, a new inner thread is associated to the session by default. The threads forked from the same session share some information, such as the knowledge base (facts and rules) and the operators table, but other elements such as the choice point stack are thread-dependent. Here, threads are simply objects that contain information about derivations, allowing to query several goals simultaneously in the same session. As mentioned above, JavaScript is single threaded, so Tau Prolog threads have nothing to do with parallel execution of multiple goals. Each thread contains its own stack. When a goal is queried, an entry is added to the thread's stack. An entry pushed on the stack in the resolution is called a *choice point*. A choice point is composed of a goal, a substitution, and a reference to its parent state, and it is represented in Tau Prolog by the prototype `pl.type.State`.

2.2 Modules and packages

A Tau Prolog's package is a JavaScript file that defines one or more modules. A module is characterized by a name, a set of predicates, and a set of visible predicates. Tau Prolog offers some modules to work with lists, manipulate the DOM, get statistics, interact with the operating system, format text, do random operations, etc., but users can build and distribute their own packages. All built-in predicates required by the ISO Prolog Standard (ISO/IEC 13211-1 1995) are defined in the `system` module, which is always visible from any module. There is another special module, the `user` module, which forms the initial working space of the user.

Modules can be defined as Prolog files and imported through the `use_module/1` directive, or they can be defined as Tau Prolog's packages, which offers the opportunity to define predicates in JavaScript to incorporate functionality not available in pure Prolog. The structure of a package is as follows:

```
var pl;
(function(pl) {
    var name = "";
    var predicates = function() { return {}; };
    var visible = [];
    if(typeof module !== "undefined") {
        module.exports = function(tau) {
            pl = tau;
            new pl.type.Module(name, predicates(), visible);
        };
    } else {
        new pl.type.Module(name, predicates(), visible);
    }
})(pl);
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

The set of predicates is an object indexed by predicate indicator (name/arity). The set of visible predicates is an array containing the predicate indicators of visible predicates. There are two ways to define predicates in a package: as a list of Prolog clauses (which is a high level description of programs as terms), and as a JavaScript function that directly manipulates the choice point stack. To define a predicate as a list of Prolog clauses, we must use the internal representation of the objects introduced in Section 2.1.

Example 2
The `append/3` predicate from the `lists` module of the Tau Prolog's lists package concatenates two lists (for readability, we omit the "`pl.type.`" text for constructors):

```
var predicates = function() {
   return {"append/3": [
      // append([], X, X).
      new Rule(
         new Term("append", [new Term("[]"),new Var("X"),new Var("X")]),
         null
      ),
      // append([H|T], X, [H|S]) :- append(T, X, S).
      new Rule(
         new Term("append", [
            new Term(".", [new Var("H"),new Var("T")]),
            new Var("X"),
            new Term(".", [new Var("H"),new Var("S")])]
         ),
         new Term("append", [new Var("T"),new Var("X"),new Var("S")])
      )
   ]};
};
```

Manually writing a Prolog predicate in this way can be tedious and prone to errors, so Tau Prolog objects include a `compile` method (which takes no arguments) to automatically generate this code. Therefore, to obtain this representation, the user only has to consult the Prolog program in a session and print out the compiled code.

Sometimes it is necessary to resort to JavaScript to implement some functionality, either because it is not possible to do it directly in Prolog, or for efficiency reasons. Therefore, a predicate can also be defined as a JavaScript function that directly manipulates the choice point stack of a thread. These functions take as parameters: *(i)* the thread of execution; *(ii)* the current choice point; and *(iii)* the selected atom of the current goal, that is, the leftmost atom of the current goal. A common scheme when implementing these functions is to get the arguments of the selected atom, check if there are errors (instantiation, type, domain, permission, etc.) and, if necessary, manipulate the choice point stack.

Example 3
The `random/3` predicate from the `random` module of the Tau Prolog's random package generates a random number between two given numbers:

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
function(thread, point, atom) {
    var lower = atom.args[0], upper = atom.args[1], rand = atom.args[2];
    if(pl.type.is_variable(lower) || pl.type.is_variable(upper)) {
        thread.throw_error(pl.error.instantiation(atom.indicator));
    } else if(!pl.type.is_number(lower)) {
        thread.throw_error(pl.error.type("number",lower,atom.indicator));
    } else if(!pl.type.is_number(upper)) {
        thread.throw_error(pl.error.type("number",upper,atom.indicator));
    } else if(!pl.type.is_variable(rand) && !pl.type.is_number(rand)) {
        thread.throw_error(pl.error.type("number",rand,atom.indicator));
    } else {
        if(lower.value < upper.value) {
            var float = lower.is_float || upper.is_float;
            var gen = lower.value + Math.random() *
                (upper.value - lower.value);
            if(!float) gen = Math.floor(gen);
            var unif = new pl.type.Term("=",
                [rand, new pl.type.Num(gen, float)]);
            thread.prepend([new pl.type.State(
                point.goal.replace(unif),
                point.substitution,
                point
            )]);
        }
    }
}
```

In the first line of this function, the arguments of the selected atom are collected in the variables `lower`, `upper` and `rand`. The first two arguments are then verified to be non-variable. If one of them is a variable, an instantiation error is thrown. Similarly, the type of the arguments is checked below, and a type error is thrown if needed. If there are no errors, a random number is generated between the two given numbers and a new choice point is inserted, where the third argument is bound to the random value.

It is important to note that functions which implement these predicates do not return any value. Returning a value that evaluates to `true` instead of `undefined` indicates to Tau Prolog that the function is asynchronous. For a predicate to be unsuccessful, it suffices not to push a new choice point on the stack.

Example 4

The `sleep/1` predicate from the `os` module of the Tau Prolog's operating system package sleeps the execution thread a specified number of milliseconds. For instance, the goal "`?- sleep(1000), X=a.`" gives the answer "`X=a`" after a second.

```
function(thread, point, atom) {
    var time = atom.args[0];
    if(pl.type.is_variable(time)) {
        thread.throw_error(pl.error.instantiation(thread.level));
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
    } else if(!pl.type.is_integer(time)) {
        thread.throw_error(pl.error.type("integer", time, thread.level));
    } else {
        setTimeout(function() {
            thread.success(point);
            thread.again();
        }, time.value);
        return true;
    }
}
```

If there are no errors, the `sleep/1` predicate uses the `setTimeout` function to perform an action after a few seconds, and returns the value `true` (only in case the function is going to succeed). This tells Tau Prolog that an asynchronous task has been run, and that it should no longer apply resolution steps, as the predicate itself will resume the inference process at some point. In the case of `sleep/1`, after a few seconds, a new choice point will be pushed into the stack, and the `Thread.prototype.again` method will be invoked to resume the inference.

2.3 Resolution and unification

The `pl.type.Thread` prototype provides the `step` method to perform a resolution step. The first choice point of the stack is removed and the leftmost atom is selected from it. Tau Prolog provides clause indexing by first argument. The corresponding clauses are collected, and a new choice point is pushed onto the stack for each clause whose head unifies with the atom. Hence, backtracking is automatically done by executing inference steps (when a predicate does not succeed, it does not push anything on the stack and the next choice point will correspond to a previous state).

```
Thread.prototype.step = function() {
    if(this.points.length === 0)
        return;
    var point = this.points.pop();
    var atom = point.goal.select();
    var definition_module = this.lookup_module(atom);
    var clauses = this.lookup_clauses(definition_module, atom);
    if(clauses instanceof Function) {
        return clauses(this, point, atom);
    } else {
        var states = [];
        for(var i in clauses) {
            var clause = clauses[i].rename(this);
            var occurs_check = this.get_flag("occurs_check");
            var mgu = pl.unify(atom, rule.head, occurs_check);
            if(mgu !== null) {
                var state = new State();
                state.goal = point.goal.replace(rule.body).apply(mgu);
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
                state.substitution = point.substitution.apply(mgu);
                state.parent = point;
                states.push(state);
            }
        }
        this.prepend(states);
    }
}
```

Unification is implemented following the algorithm described by Martelli and Montanari (1982). The `pl.unify` method takes two Prolog objects, and a flag to indicate whether the occurs check should be performed or not. User-defined Prolog term types can extend unification by implementing the `unify` method in its prototype (that takes the second Prolog object and the occurs check flag as arguments). When two Prolog objects unify, the `unify` method returns a substitution. Otherwise, it returns `null`.

Example 5

Tau Prolog's JavaScript package defines a new term type, `pl.type.JSValue`, for manipulating JavaScript values that cannot be directly converted to Prolog. A `JSValue` unifies with other Prolog term if both are `JSValue` with the same value, or if one of them is a `JSValue` that contains a JavaScript object and the other one is a Prolog term of the form {`prop_1: value_1, prop_2: value_2, ..., prop_n: value_n`}, where `prop_i` is an atom whose identifier represents a property of the JavaScript object, and `value_i` is a Prolog term that unifies with the value of that property, for $1 \leq i \leq n$.

```
pl.type.JSValue.prototype.unify = function(obj, occurs_check) {
    if(pl.type.is_js_object(obj) && this.value === obj.value)
        return new pl.type.Substitution();
    if(pl.type.is_term(obj) && obj.indicator === "{}/1") {
        var left = [], right = [];
        var pointer = obj.args[0];
        var props = [];
        while(pl.type.is_term(pointer) && pointer.indicator === ",/2") {
            props.push(pointer.args[0]);
            pointer = pointer.args[1];
        }
        props.push(pointer);
        for(var i = 0; i < props.length; i++) {
            var bind = props[i];
            if(!pl.type.is_term(bind) || bind.indicator !== ":/2")
                return null;
            var name = bind.args[0];
            if(!pl.type.is_atom(name) || !this.value.hasOwnProperty
                                                        (name.id))
                return null;
            var value = pl.fromJavaScript.apply(this.value[name.id]);
            right.push(bind.args[1]);
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
            left.push(value);
        }
        return pl.unify(left, right, occurs_check);
    }
    return null;
};
```

Suppose there is an object "`var o = {x: 1, y: false, z: {w: [2,"a"]}};`" defined in the global scope of JavaScript. Then, the following goals are possible (the predicate `get_prop/2` is described in detail in Section 3):

```
?- get_prop(o, Object).
Object = javascript<object>.

?- get_prop(o, Object1), get_prop(o, Object2), Object1 = Object2.
Object1 = javascript<object>, Object2 = javascript<object>.

?- get_prop(o, {x: X, y: Y, z: {w: W}}).
X = 1, Y = false, W = [2,a].

?- get_prop(o, Object), Object = {x: X, y: false}, Object = {z: {w: W}}.
Object = javascript<object>, X = 1, W = [2,a].
```

2.4 Callback-based approach

A *callback function* is a function passed into another function as an argument, which is then invoked inside the outer function to complete some action. The `pl.type.Session` and `pl.type.Thread` prototypes have three main methods to load programs and query goals, all of them based on executing a user's callback when the main action has finished:

- The `consult` method takes the input program and a callback and, if there are no syntax errors, it adds the parsed rules to the knowledge base, executing the callback afterwards. The `consult` method can take a string with the Prolog program, an URL or path to a Prolog file, or the identifier of a `<script type="text/prolog">` tag inserted on the web page.
- The `query` method takes an input goal as a string and a callback and, if there are no syntax errors, it adds the goal to the stack of choice point and executes the callback. At first glance, consulting a program or querying a goal may not seem like an asynchronous task, but it is necessary to incorporate certain Prolog features such as term and goal expansion (Carlsson *et al.* 1988), which require running Prolog code while parsing.
- The `answer` method takes a callback and pushes it to an internal stack of calls. The callback will be executed when the next computed answer is found. If there are no previous calls on the stack, the `again` method is invoked to start the search.

After querying a goal, the search for a computed answer does not begin until the `answer` method is invoked.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
Thread.prototype.answer = function(callback) {
    this.__calls.push(callback);
    if(this.__calls.length == 1)
        this.again();
};
```

The `again` method performs resolution steps while there are choice point on the stack, as long as there are no errors and the resolution limit is not reached. When an answer is found, the user's callback is executed to handle it and the callback is removed from the stack. If the `step` method, that performs a resolution step, returns `true`, the search stops. This tells the thread that an asynchronous action has been taken, and the predicate will resume the search (by explicitly calling the `again` method) when it is done. This model allows predicates to perform asynchronous tasks, freeing up the browser until the search must continue. The Example 4 shows the implementation of an asynchronous predicate that sleeps the thread a given amount of time.

```
Thread.prototype.again = function() {
    while(this.__calls.length > 0) {
        while(this.current_limit > 0
           && this.points.length > 0
           && this.head_point().goal !== null
           && !pl.type.is_error_state(this.head_point()))
                  if(this.step() === true)
                      return;
        var callback = this.__calls.shift();
        var answer = this.points.pop();
        (function(answer, callback) {
            setTimeout(function() {
                callback(answer);
            }, 0);
        })(answer, callback);
    }
};
```

The following is a general scheme of how to use Tau Prolog to load a program and look for an answer. This outline can be cumbersome for the programmer, since up to three callbacks have to be nested to execute a goal. Furthermore, as can be seen, a callback or an object that contains callbacks can be passed for the different scenarios that may arise (success, failure, error or resolution limit).

```
const session = pl.create();
const program = "your prolog program";
const goal = "your prolog goal";
session.consult(program, {
    success: function() {
        session.query(goal, {
            success: function(goal) {
                session.answer({
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
                success: function(answer) { /* Answer */ },
                error:   function(err)    { /* Uncaught error */ },
                fail:    function()       { /* Failure */ },
                limit:   function()       { /* Limit exceeded */ }
            })
        },
        error: function(err) { /* Error parsing goal */ }
    });
},
error: function(err) { /* Error parsing program */ }
});
```

As seen in Kambona *et al.* (2013), this problem is commonly known as the *callback hell*. To overcome the drawbacks of this interface based on callbacks, Tau Prolog distributes a package that extends the prototypes to add a new interface based on promises, that we introduce in the next section. However, notice that, unlike Tau Prolog, browsers can freeze while running goals that takes so long in a non-asynchronous Prolog implementation like jsProlog.

2.5 Promises and reactive programming

A promise object represents the eventual completion or failure of an asynchronous operation and its resulting value. Tau Prolog's promises package extends the `pl.type.Session` and `pl.type.Thread` prototypes to add new methods for consulting programs and querying goals, returning promises.

The Tau Prolog's promises package defines three new methods:

- `promiseConsult`: consults a program and returns a promise that is resolved when the program loads successfully, or rejected when there is an error. It takes the same arguments as the `consult` method.
- `promiseQuery`: queries a goal and returns a promise that is resolved when the goal loads successfully, or rejected when there is an error. It takes the same arguments as the `query` method.
- `promiseAnswer`: finds the next computed answer and returns a promise that is resolved when it finds an answer or there are no more answers, or is rejected when there is an error or the limit of inferences has been reached. It takes the same arguments as the `answer` method.

Also, the package defines a fourth method, `promiseAsnwers`, to find all computed answers, returning an asynchronous generator.

Example 6
The following Node.js script loads a Prolog program that defines the `append/3` predicate, which concatenates two lists, and writes to the standard output all the computed answers for goal "`?- append(X, Y, [a,b,c])`".

```
const pl = require("tau-prolog");
require("tau-prolog/modules/promises.js")(pl);
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

```
(async() => {
    const program = `
        append([], X, X).
        append([H|T], X, [H|S]) :- append(T, X, S).
    `;
    const goal = "append(X, Y, [a,b,c]).";
    const session = pl.create();
    await session.promiseConsult(program);
    await session.promiseQuery(goal);
    for await (let answer of session.promiseAnswers())
        console.log(session.format_answer(answer));
})();
```

This script prints out the expected computed answers "`X=[], Y=[a,b,c]`", "`X=[a], Y=[b,c]`", "`X=[a,b], Y=[c]`" and "`X=[a,b,c], Y=[]`".

Bainomugisha *et al.* (2013) define *reactive programming* as a declarative programming paradigm that facilitates the development of event-driven and interactive applications by providing abstractions to express time-varying values and automatically managing dependencies between such values.

It is possible to easily use the promises interface of Tau Prolog together with reactive programming libraries, as long as they support the conversion of asynchronous generator functions to *observables*, such as RxJS (Clow 2018) from version 7.0.0. An observable represents the idea of an invokable collection of future values or events.

Example 7

RxJS is a library for reactive programming using observables, to make it easier to compose asynchronous or callback-based code. The following Node.js script does the same result as the script in Example 6, but creating an observable `ap` from the goal "`?- append(X, Y, [a,b,c])`" to which we subscribe to handle each of the answers.

```
const Rx = require("rxjs");
const pl = require("tau-prolog");
require("tau-prolog/modules/promises.js")(pl);

(async() => {
    const program = `
        append([], X, X).
        append([H|T], X, [H|S]) :- append(T, X, S).
    `;
    const goal = "append(X, Y, [a,b,c]).";
    const session = pl.create();
    await session.promiseConsult(program);
    await session.promiseQuery(goal);
    const ap = Rx.from(session.promiseAnswers());
    const sub = ap.subscribe(x => console.log(session.format_answer(x)));
})();
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

Both, promises and observables, can take the place of answer handlers, but there are key differences. For example, observable subscriptions are cancellable, promises not: unsubscribing removes the listener from receiving further answers, and notifies the subscriber function to cancel work.

The JavaScript interface of SWI-Prolog pengines is based on HTTP requests and callbacks, and therefore, the same observable-based reactive interface can be built on top of it, just like Tau Prolog does, providing a generalized interface for Prolog systems.

3 Web interaction with Tau Prolog

In this section, we focus on the main packages available on Tau Prolog for the web interaction: the DOM package and the Tau Prolog foreign function interface.

3.1 DOM manipulation

As defined by World Wide Web Consortium *et al.* (2004), the *Document Object Model* (DOM) 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 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 `dom` module also includes predicates to go through the DOM starting at the HTML objects retrieved with the previous predicates: `parent_of/2` and `sibling/2`.
- 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`. If we try to insert an element which is already part of the DOM, the predicate fails and the element is not inserted again. Otherwise, the element is inserted and the predicate succeeds.
- 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.

3.2 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`, `mouseover`, `mouseout`, etc.), an event and

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

a goal, and bind the HTML object with said 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.

Example 8
In this example, the `keypress` event has been added to the page body, so when a key is pressed, the HTML object whose id is `output` displays what key has been pressed.

```
get_by_id(output, Output),
get_by_tag(body, B),
bind(B, keypress, Event, (
        event_property(Event, key, Key),
        set_html(Output, Key)
    )
).
```

Notice that the `event_property/3` predicate and the `Event` term 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).

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 behavior 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 Mozilla Contributors (2021b).

3.3 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. `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/3`: gets a property of a JavaScript object. `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. `global(Context)` is true if and only if `Context` is the global JavaScript object (`window` in browser or `global` in Node.js).

There are also versions of `get_prop/2` and `apply/3` predicates where the context is just the global object.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

Example 9
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.

Note that when a value cannot 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. The JavaScript foreign function interface of Tau Prolog is really simple but very effective. As we will see in the following section, it allows to create interfaces with complex JavaScript APIs, such as drawing on `<canvas>` HTML elements.

4 Tau Prolog programming environment

Although the main purpose of Tau Prolog is to embed Prolog code in web pages, Tau Prolog also has an online interactive interpreter which is very convenient for debugging, testing and sharing Prolog code. In this section, we show the programming environment for Tau Prolog.

4.1 Tau Prolog sandbox

The *Tau Prolog Sandbox*[10] (see Figure 1) is an online interactive interpreter of Prolog which runs the latest version of all Tau Prolog's packages available 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).

Below the Program editor option, 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 (ISO/IEC 13211-1 1995). After consulting a Prolog program, under the HTML editor choice users can insert HTML code and run programs related to it.

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

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

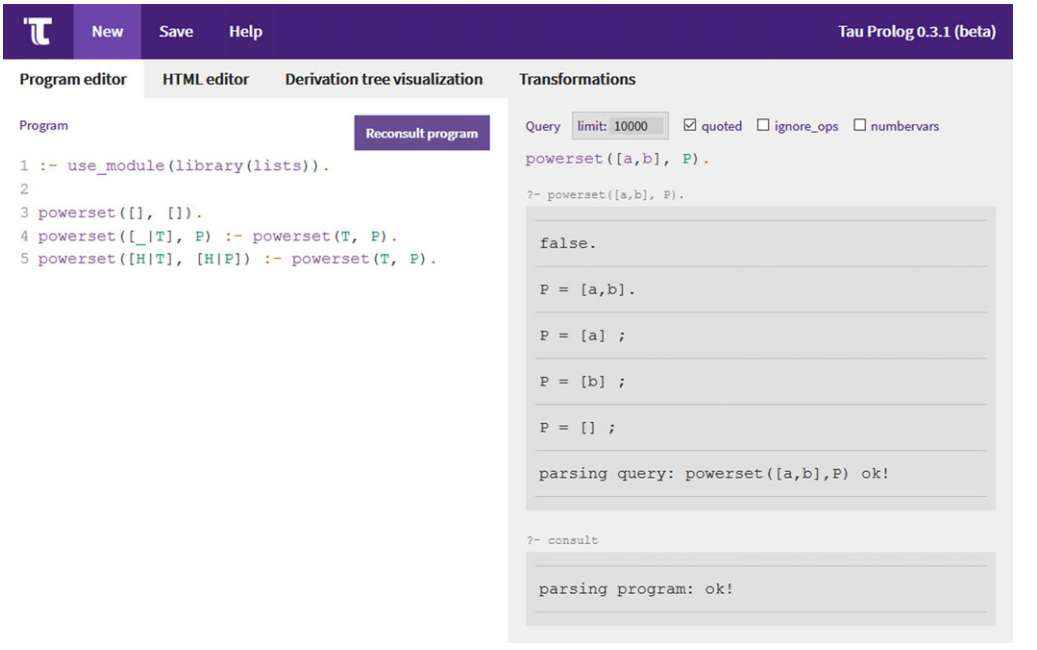

Fig. 1. Screenshot of the Tau Prolog Sandbox online tool running a Prolog program.

Example 10

The Canvas API provides a means for drawing graphics via JavaScript and the HTML `<canvas>` element. The following Prolog program creates an animation in a `<canvas>` element. The `draw/2` predicate takes the context of the canvas and the degrees, and draws an arc on it. The `main/0` predicate draws arcs from 0 to 360 degrees every 10 milliseconds.

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

draw(Ctx, Degrees) :-
    apply(Ctx, clearRect, [0, 0, 200, 200], _),
    Radians is Degrees * pi / 180,
    apply(Ctx, beginPath, [], _),
    apply(Ctx, arc, [100, 100, 80, 0, Radians], _),
    apply(Ctx, stroke, [], _),
    get_by_id(radians, Rad), set_attr(Rad, value, Radians),
    get_by_id(degrees, Deg), set_attr(Deg, value, Degrees).

main :-
    get_by_id(canvas, Canvas),
    apply(Canvas, getContext, ['2d'], Ctx),
    between(0, 360, Degrees),
    draw(Ctx, Degrees),
```

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

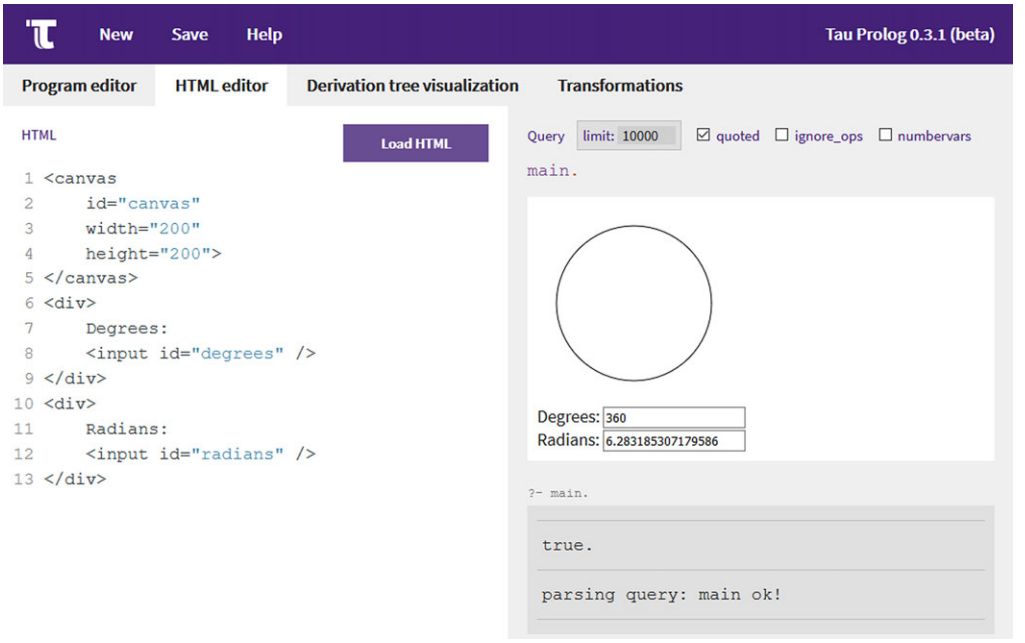

Fig. 2. Screenshot of the Tau Prolog Sandbox online tool under the HTML editor.

```
    sleep(10),
    false ; true.
```

Figure 2 shows the HTML editor tab after running the animation. In this example, we are using the JavaScript foreign function interface for drawing the canvas, and the DOM package to get the HTML elements and write into the `<input>` elements.

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 Tamaki (1984). 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. This tool is not exclusive of the Tau Prolog Sandbox, in fact, it is an extension of Tau Prolog that users can install in their own web pages, so we introduce it in the following section.

4.2 Graphical derivation trees

The graphical derivation trees[11] tool extends the `pl.type.Session` and `pl.type.Thread` prototypes adding a new method, `draw`, that takes the following arguments:

- the maximum number of answers to find in the derivation (to avoid infinite trees),
- the `<canvas>` HTML element, or its identifier,
- and (optionally) a JavaScript object with style properties.

[11] https://github.com/tau-prolog/draw-derivation-trees.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

The `draw` method must be called after querying a goal instead of the `answer` method.

Example 11
The following HTML document draws the derivation tree for goal "`?- powerset([a,b], P)`", where `powerset/2` generates all the subsets of a given set:

```
<html>
  <head>
    <script src="tau-prolog/core.js"></script>
    <script src="tau-prolog/draw-derivation-trees.js"></script>
    <script id="program.pl" type="text/prolog">
        powerset([], []).
        powerset([_|T], P) :- powerset(T, P).
        powerset([H|T], [H|P]) :- powerset(T, P).
    </script>
  </head>
  <body>
    <canvas id="derivation"></canvas>
    <script type="text/javascript">
        var session = pl.create();
        session.consult("program.pl", function() {
            session.query("powerset([a,b], P).", function() {
                session.draw(10, "derivation");
            });
        });
    </script>
  </body>
</html>
```

Figure 3 shows the graphical derivation tree generated by Tau Prolog for this goal. Each node displays the current goal and substitution in the inference process, where answers are leaves with empty goals "□". Note that only variables in the original goal are shown in substitutions. Nodes representing answers and errors are highlighted with different colors.

5 Benchmarks

In this section, we present some benchmarks to compare the performance of Tau Prolog with other Prolog systems. Table 1 shows the average runtime of executing Prolog benchmarks with Tau Prolog 0.3.4 running in Chromium 109.0.5414.119, SWI-Prolog 8.4.2, the latest version[12] of Scryer Prolog, the latest version[13] of Trealla Prolog, GNU Prolog 1.4.5, Ciao Prolog 1.22.0, the latest version[14] of Dogelog running in Chromium 109.0.5414.119, the latest version[15] of hitchhiker Prolog running in Chromium 109.0.5414.119 and

[12] https://github.com/mthom/scryer-prolog/tree/1118b37c9232d10b7c2c84648bf364d859210319.
[13] https://github.com/trealla-prolog/trealla/tree/c213efefe886c830d9cc98de2ff7e9ee6d2d28b7.
[14] https://www.xlog.ch/izytab/doclet/docs/18_live/10_reference/example02/package.html.
[15] https://github.com/CapelliC/hhprolog/tree/87074de0aea959ed0ac2c0eaa45a6aa13e3e37d5.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

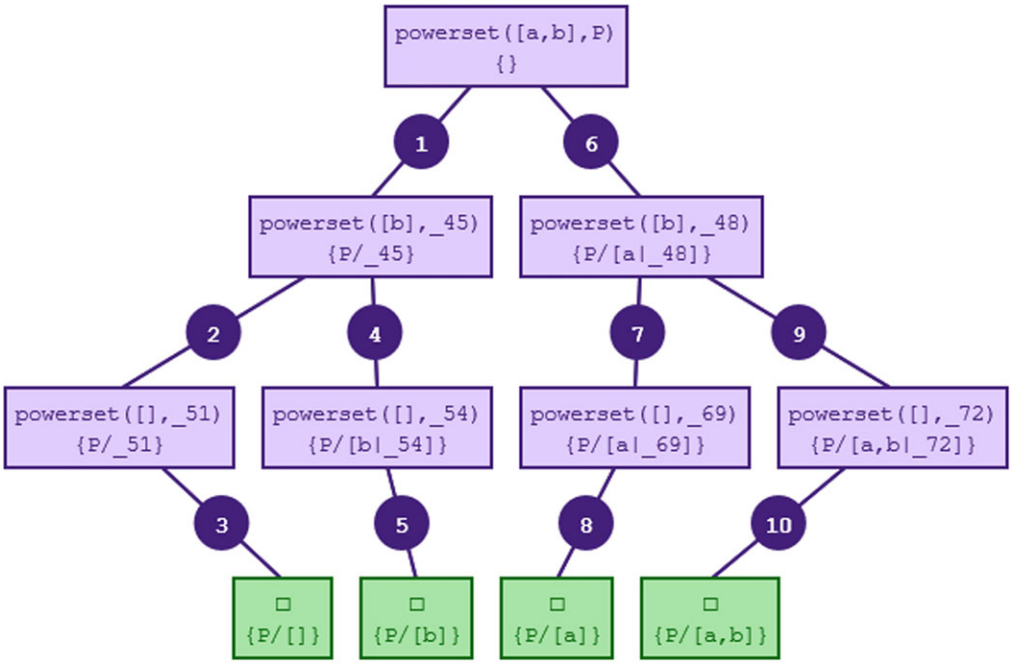

Fig. 3. Graphical derivation tree generated by Tau Prolog.

Table 1. *Average runtime (in seconds) of Prolog programs on different systems*

Benchmark	Tau	SWI	Scryer	Trealla	GNU	Ciao	Dogelog	hh	jsProlog
mergesort	0.785	0.010	0.006	0.008	0.004	0.001	0.016	*	160.352
zebra	0.920	0.009	0.027	0.032	0.005	0.004	0.019	0.113	62.851
queens	1.213	0.037	0.021	0.020	0.014	0.005	0.075	*	228.594
peano	4.208	0.005	0.003	0.006	0.004	0.000	0.014	0.061	**
sat	4.517	0.016	0.062	0.069	0.060	0.026	0.171	*	**
inorder	5.048	0.003	0.004	0.008	0.003	0.000	0.009	0.025	**
append	6.284	0.001	0.002	0.002	0.001	0.000	0.005	0.135	**

the latest version[16] of jsProlog running in Node 18 (since it has been necessary to increase the memory limit for some benchmarks). All tests have been executed in Ubuntu 22.04.1 LTS (64 bits) using a desktop computer equipped with an Intel® Core™ i7-8700 CPU @ 3.20 GHz and 32.00 GB RAM. Test programs considered for benchmarking can be found at https://github.com/tau-prolog/tau-prolog/tree/master/examples/tplp, and all goals were chosen to give a reasonably long overall time. For instance, the queens benchmark refers to finding all solutions to the 8 queens problem. Currently, we can see that classic Prolog systems are much faster than Tau Prolog. However, Tau Prolog is capable of performing common tasks in the browser (validating

[16] https://github.com/cubiwan/jsProlog/tree/5d6f00e7d5cc435a1bf706527e7c4781957170cd.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

forms, creating animations,[17] deploying simple games,[18] etc.) without any problem, as we show in Riaza (2022). Blank entries in the hitchhiker Prolog column (*) are due to the lack of extra-logical predicates (control constructs, arithmetic operators, etc.). Blank entries in the jsProlog column (**) correspond to errors due to memory limits.

6 Conclusions and future work

In this paper we have described the features and implementation details 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. Finally, we have shown some benchmarks to measure the performance of Tau Prolog. As future work, we hope to improve the performance and efficiency of Tau Prolog (by implementing, e.g. last call optimization), and to incorporate features of Constraint Handling Rules (Frühwirth 1998) and Constraint Logic Programming (Jaffar and Lassez 1987) to it. In this line, we consider implementing an interface for attributed variables, originally defined by Holzbaur (1990).

References

Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S. and Meuter, W. d. 2013. A survey on reactive programming. *ACM Computing Surveys (CSUR) 45,* 4, 1–34.

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

Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H. and Sjöland, T. 1988. *SICStus Prolog User's Manual*, Vol. 3. Swedish Institute of Computer Science Kista, Sweden.

Clow, M. 2018. Observers, reactive programming, and rxjs. In *Angular 5 Projects*. Springer, 291–307.

Ecma International. 2021a. ECMAScript 2022 internationalization api specification. https://tc39.es/ecma402/.

Ecma International. 2021b. ECMAScript 2022 language specification. https://tc39.es/ecma262/.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *The Journal of Logic Programming 37,* 1-3, 95–138.

García-Pradales, G., Morales, J. F., Hermenegildo, M., Arias, J. and Carro, M. 2022. An s (casp) in-browser playground based on ciao prolog.

Holzbaur, C. 1990. *Specification of Constraint Based Inference Mechanism Through Extended Unification*. Ph.D. thesis, Department of Medical Cybernetics & AI, University of Vienna.

ISO/IEC 13211-1 1995. Information technology – Programming languages – Prolog – Part 1: General core.

[17] http://tau-prolog.org/examples/draggable.
[18] http://tau-prolog.org/examples/snake.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 111–119.

Kambona, K., Boix, E. G. and De Meuter, W. 2013. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, 1–9.

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

Lager, T. and Wielemaker, J. 2014. Pengines: Web logic programming made easy. *Theory and Practice of Logic Programming 14,* 4-5, 539–552.

Latypova, V., Martynov, V. and Turganov, A. 2020. 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, 1–5.

Martelli, A. and Montanari, U. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4,* 2, 258–282.

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

Moura, P. 2022. *The Logtalk Handbook*, Release 3.54.0 ed. https://logtalk.org/manuals/TheLogtalkHandbook-3.54.0.pdf.

Mozilla Contributors. 2021a. Concurrency model and the event loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

Mozilla Contributors. 2021b. Event reference. https://developer.mozilla.org/en-US/docs/Web/Events.

Mozilla Contributors. 2021c. Object prototypes. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes.

Mozilla Contributors. 2021d. Using web workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

Riaza, J. A. 2022. Web development with tau prolog. In *Proceedings of the XXI Conference on Programming and Languages, PROLE'22*, P. Julián Iranzo, Ed. SISTEDES, Santiago de Compostela, Spain, 1–14.

Tamaki, H. 1984. Unfold/fold transformation of logic programs. In *Proc. of 2nd ILPC*, 127–138.

Tarau, P. 2018. A Hitchhiker's Guide to Reinventing a Prolog Machine. In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, R. Rocha, T. C. Son, C. Mears and N. Saeedloei, Eds. OpenAccess Series in Informatics (OASIcs), vol. 58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:16.

Thompson, J. 2021. Yield prolog. http://yieldprolog.sourceforge.net.

Wielemaker, J., Lager, T., and Riguzzi, F. 2015. SWISH: Swi-prolog for sharing. CoRR abs/1511.00915.

World Wide Web Consortium et al. 2004. Document object model (dom) level 3 core specification. https://www.w3.org/TR/DOM-Level-3-Core/.

Yarnpkg. 2021. Constraints - Yarn Package Manager. https://yarnpkg.com/features/constraints.

https://doi.org/10.1017/S1471068423000352 Published online by Cambridge University Press