Lightweight compilation of (C)LP to JavaScript *

Jose F. Morales[1], Rémy Haemmerlé[2],

Manuel Carro[1,2], *and Manuel V. Hermenegildo*[1,2]

[1] *IMDEA Software Institute, Madrid (Spain)*

[2] *School of Computer Science, Technical University of Madrid (UPM), (Spain)*

(*e-mail:* {josef.morales,manuel.carro,manuel.hermenegildo}@imdea.org)

(*e-mail:* remy@clip.dia.fi.upm.es, {mcarro,herme}@fi.upm.es)

Abstract

We present and evaluate a compiler from Prolog (and extensions) to JavaScript which makes it possible to use (constraint) logic programming to develop the client side of web applications while being compliant with current industry standards. Targeting JavaScript makes (C)LP programs executable in virtually every modern computing device with no additional software requirements from the point of view of the user. In turn, the use of a very high-level language facilitates the development of high-quality, complex software. The compiler is a back end of the Ciao system and supports most of its features, including its module system and its rich language extension mechanism based on *packages*. We present an overview of the compilation process and a detailed description of the run-time system, including the support for modular compilation into separate JavaScript code. We demonstrate the maturity of the compiler by testing it with complex code such as a CLP(FD) library written in Prolog with attributed variables. Finally, we validate our proposal by measuring the performance of some LP and CLP(FD) benchmarks running on top of major JavaScript engines.

KEYWORDS: Prolog; Ciao; Logic Programming System; Implementation of Prolog; Modules; JavaScript; Web

1 Introduction

The Web has evolved from a network of hypertext documents into one of the most widely used OS-neutral environments for running rich applications —the so-called Web-2.0—, where computations are carried both locally at the browser and remotely on a server. A key factor in the success of the Web has been the development of *open* standards backed up by mature implementations. One of these is JavaScript (ECMA International 2009), which was initially designed as a simple dynamic language embedded in HTML documents in order to offer basic dynamic

∗ The research leading to these results has received funding from the Madrid Regional Government under CM project P2009/TIC/1465 (PROMETIDOS), and from the Spanish Ministry of Economy and Competitiveness under project TIN-2008-05624 *DOVES*. The research by Rémy Haemmerlé has also been supported by PICD, the Programme for Attracting Talent / young PHDs of the Montegancedo Campus of International Excellence.

content. Factors such as openness, simplicity, flexibility, full browser integration, and attention to the security and privacy concerns that naturally arise in the execution of untrusted code, have helped the language gain very significant popularity despite its initial low efficiency. Performance was initially not competitive with plug-in based technology like Java-based *applets* (Lindholm and Yellin 1996), but current JavaScript engines in major browsers use JIT compilation to optimize the *hot spots* in the program using type or trace information (Google ; Gal et al. 2009). While still not optimal for computationally intensive tasks, performance is good enough in many cases, specially those requiring mostly just graphical user interaction. The language has also raised significant interest in the research community, as witnessed for example by recent work studying the formalization of the full core language (Maffeis et al. 2008). Overall it is enabling a disruptive paradigm shift that is gradually replacing OS-dependent application development with fully portable Web applications which can run in a variety of devices.

While all this represents significant advances in the technology for developing Web applications, it is suboptimal to rely on a single language to solve all problems. While the whole spectrum of programming languages is normally available on the server side, server-side execution is not always appropriate: for example, the client side may not be allowed to transmit sensitive data outside the client, and there are always constraints on network capacity or usage (e.g., local search on a large set of personal data). Also, server-side execution requires computing power and storage on the server dedicated hardware, which can have an unacceptable cost and/or be a bottleneck for large numbers of clients.

While large applications have been written directly in JavaScript (despite the lack of analysis tools or a module system), targeting it as a back end language for cross-compilation is nowadays a popular option[1] in order to execute existing code in web browsers (since manual rewriting is costly for large projects) or to use libraries and features available in other languages.

At the same time, there has been interest and significant activity almost since the start of the Web in programming web applications in Prolog and other (constraint) logic programming dialects and/or using the Web as a portable graphical interface for (C)LP programs (including, e.g., complex tools such as analyzers or theorem provers). The major Prolog implementations have focused to date on server-side execution. One of the first popular frameworks for developing Web applications in Prolog is PiLLoW (Cabeza and Hermenegildo 2001), where a server running Prolog code communicates with browsers using HTTP server mechanisms (CGIs) or the HTTP protocol. This same approach was also taken and extended by SWI-Prolog (Wielemaker et al. 2008). For client-side execution, most systems have targeted Java or the Java VM. Some of the most notable systems, Jinni and more recently Lean Prolog, are derived from BinProlog (Tarau 2011a). However, in most cases such systems are developed from scratch and present at least moderate incompatibilities with server-side systems. Moreover, as technology shifts from

[1] See `https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS`.

Java towards JavaScript as client-side language, those systems may suffer from obsolesce. There has been one attempt that we are aware of at implementing Prolog in JavaScript, JScriptLog (`http://jlogic.sourceforge.net/`), but it is an interpreter and is meant to be just a demonstrator, supporting only a subset of Prolog.

Our ambitious objective is to enable client-side execution of *full-fledged (C)LP programs* by means of their *compilation* to JavaScript, i.e., to support essentially the full language available on the server side. Our starting point is the Ciao system (Hermenegildo et al. 2012), which implements a multi-paradigm Prolog dialect with numerous extensions through a sophisticated module and program expansion system (*packages*). Such packages facilitate syntactic and semantic language extensions, all of which are also to be supported in our approach. The module system also offers a precise distinction between static and dynamic parts, which is quite useful in the translation process. Other approaches often put emphasis on the feasibility of the translation or on performance on small programs, while ours is focused on completeness and integration:

- We share the language front end and implement the translation by redefining the (complex) last compilation phases of an existing system. In return we support a full module system including packages, as well as the existing analysis and program transformation tools.
- We provide a minimal but scalable runtime system (including built-ins) and a compilation scheme based on the WAM (Ait-Kaci 1991; Warren 1983) that can be progressively extended and enhanced as required.
- We offer both high-level and low-level foreign language interfaces with JavaScript to simplify the tasks of writing libraries and integrating with existing code.

This allows us to read and compile (mostly) unmodified Prolog programs (as well as all the Ciao extensions such as different flavors of (C)LP or functional notation and higher-order, to name a few), run real benchmarks, and, in summary, be able to develop full applications, where interaction with JavaScript or HTML is performed via Prolog libraries and client-side execution in the browser does not require manual recoding. To the extent of our knowledge, ours is the first approach and full implementation which can achieve these goals.

The paper is organized as follows. In Section 2 we provide an overview of the JavaScript language and introduce our solution for generating code in a modular way. In Section 3 we describe the cross-compilation process. In Section 4 we show the language interface with JavaScript. We present experimental results in Section 5. Finally, Section 6 presents our conclusions.

2 Making JavaScript a target for modular compilation

JavaScript is a simple, lexically scoped, imperative language. Its syntax is close to that of C or Java but internally it is closer to Scheme or Self. Data objects can be native (numbers, strings, booleans, etc.), records (mutable maps from primitive data to values), or closures (anonymous functions). Records contain a distinguished field called `prototype` that is the basis for object-oriented programming in JavaScript.

When a field is not found in a record, it is searched recursively following the prototype field. This allows records to share fields (e.g., to implement a class with methods shared by all its instances). Functions act also as object constructors and are records themselves, with a special `prototype` field. Given a function `ctor`, `new ctor(args)` creates a new empty object whose prototype will be `ctor.prototype`, and then executes the function `ctor` with `this` bound to the object. Prototypes may form a chain, which is useful for implementing inheritance. The internal prototype field of a record cannot be accessed directly and the only valid operation on it is `x instanceof C`, which is true if `C` is found in the *prototype chain* of `x`.

One of the main drawbacks of JavaScript for developing scalable and reusable code is the *lack of proper namespaces or module system*, where all symbols apparently live in a single common global namespace. For that reason, some proposals for adding modularity to JavaScript programs exist (e.g., Prototype, CommonJS). However, we found them either too complex or not complete enough for our purposes. Nevertheless, closures, the scoping rules, and records can be used to manually achieve effective symbol hiding. We use this mechanism to implement the necessary symbol tables for encoding *modular* Prolog programs, as described below.

Runtime for Symbol Tables. Inspired by the implementation of Prolog predicate tables, we defined a thin runtime layer (Fig. 1) to implement a symbol table which associates symbol names (strings) to their definitions. For each symbol we also store associated information, like *export* tables, used to implement modules. Additionally, we allow the definition of symbols associated with JavaScript classes (coordinating initialization of base classes and prototype chains). We will use them to implement the data type hierarchy for terms and to avoid *tagging*. We will later (Section 3) populate tables with actual definitions (modules, predicates, functors, JavaScript closures, etc.).

Symbols. Fig. 1 illustrates our approach for representing symbols as JavaScript objects. The most important components are the `status`, which stores the initialization state of the symbol, an `exports` table which associates names (strings) with values, and a `nested` table for associated nested symbols. Symbols are initially created (constructor in Line 2) with an `UNDEFINED` status. Initially we create a single *root* symbol (named `$r`) in the global scope. The m`.query(`n`)` method (Line 11) obtains the symbol associated with the name n in the nested table of m. If it does not exist, it is created. Symbol objects behave as pointers or references to definitions. Keeping track of nested symbols will later be useful to ensure correct initialization, as well as providing a simple way to store tables for modular programs (e.g., `$r.query("lists").query("append/3")`, assuming that the symbol is associated with the predicate `lists:append/3`).

Defining and Registering Symbols. In order to support complex dependencies, symbol definitions are completed in two different passes. First, symbols are registered in the *nested* table of another symbol. The r`.def(`n`,`c`)` method (Line 6) queries the symbol n in r, changes the symbol state from undefined to `NOT_READY`, and executes the closure c. Fig. 2 presents a schematic view of a symbol definition, where we provide the structure of the definition closure. The closure effectively

```
var UNDEFINED=0, NOT_READY=1, PREPARING=2, READY=3;
function $s(name) {                        // Symbol constructor
    this.name=name; this.exports={}; this.status=UNDEFINED;
    this.nested={}; this.link=null; this.mlink=null;
}
$s.prototype.def=function(name, def) {  // Define a symbol
    var m=this.query(name);
    m.status=NOT_READY;                     // mark the symbol as not ready
    def(m); return m;
}
$s.prototype.query=function(name) {      // Query a (sub)symbol
    var m=this.nested[name];
    if (m === undefined) { m=new $s(name); this.nested[name]=m; }
    return m;
}
$s.prototype.prepare=function() {        // Prepare the symbol
    if (this.status !== NOT_READY) return this;
    this.status=PREPARING;                 // preparing the symbol (not ready)
    if (this.ctor !== undefined) {         // the symbol defines a class
        if (this.base !== null) {
            this.base.prepare();           // prepare base symbol, status is READY
            $extends(this.ctor, this.base.ctor); // setup prototype chain
        }
        if (this.mlink !== null) this.mlink(this.ctor); // instance methods
    }
    if (this.link !== null) this.link(); // link local from imported symbols
    this.status=READY;                     // mark the symbol as ready
    // prepare nested symbols
    for (var k in this.nested) if (this.nested[k]) this.nested[k].prepare();
    return this;
}
function $extends(c, base) {              // (auxiliary for subclassing)
    // copy class methods from base to c
    for (var k in base) if (base.hasOwnProperty(k)) c[k]=base[k];
    // ensure that the object c.prototype has the prototype base.prototype
    function ctor() {}; ctor.prototype=base.prototype; c.prototype=new ctor;
    c.prototype.constructor=c;
}
```

Fig. 1. Minimal runtime code for modular symbol tables in JavaScript.

hides all local variable and function names from outer scopes. It receives the symbol object as parameter *m* to fill its definition. Then, other nested symbols can be defined (Line 3), and entries in the export table filled (Line 4) to selectively make inner definitions available (both closures or data). When the symbol has an associated class, we connect `ctor` with the class constructor and, optionally, `base` with the symbol containing the base class (Line 5). The rest of the definition is delayed in other closures, that will be invoked by *preparing* the symbol (explained below). Definitions of class methods, which must be delayed until the constructor is ready, are delayed in the `mlink` closure (Line 6). On the other hand, values of exported

```
1  r.def(name, function(m) {
2    var u, ...;                           // placeholders for imported symbols
3    m.def(name', ...); ...                // nested symbols
4    m.exports.k = ...; ...                // exported symbols
5    m.ctor = ...; m.base = ...;           // (constructor and base, optional)
6    m.mlink = function(c) { c.prototype.m = ...; ... };
7    m.link = function() { p.prepare(); u = p.exported.k; ... };
8  });
```

Fig. 2. Structure of a definition closure.

entries of imported symbols, which are available once the symbol is *prepared*, are filled in the `link` closure (Line 7).

Preparing Symbols. The `prepare` method (Fig. 1-Line 16) changes the state of `NOT_READY` symbols to `READY`. First, it prepares the base and fixes the prototype chain of the constructor (Line 32), and fills the methods invoking `mlink`. Then, values of imported symbols are filled invoking `link`. We assume that each `link` closure calls the `prepare` method of the required symbols. Finally, all nested symbols are prepared.

3 Compiler and system architecture for cross-compilation

We base our compiler on two design decisions. First, we share a common front end with the bytecode back end of Ciao. Second, we reuse most of the WAM compilation algorithm, and a significant part of the WAM emulator. A global view of this architecture is shown in Fig. 3. We will elaborate on both points below.

Sharing the Ciao front end clearly simplifies maintenance of the system, and avoids undesired or unexpected language incompatibilities. More importantly, it reuses the Ciao package mechanism for language extensions. As mentioned before, such packages provide a collection of syntactic additions (or restrictions) to the input language, translation rules for code generation to support new semantics, and the necessary run-time code. Packages are separated into compile-time and run-time parts. The compile-time parts (termed *compilation modules*) are invoked during

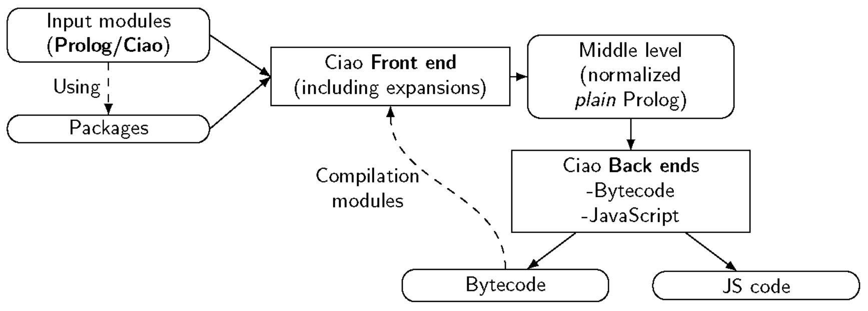
Input modules (Prolog/Ciao)
Using
Packages
Ciao Front end (including expansions)
Middle level (normalized plain Prolog)
Compilation modules
Ciao Back ends -Bytecode -JavaScript
Bytecode
JS code

Fig. 3. Overview of the Multi Back End Architecture for Ciao.

compilation, and are not necessary during execution. On the other hand, the run-time parts are only required for execution. This phase distinction has a number of practical advantages, such as reducing executable size. Another important objective achieved is a stratified separation of modules that makes the code more amenable to static analysis. In this way, compilation modules are dynamically loaded by the compiler and invoked during compilation, but not subject to analysis, and the source modules can be determined statically. Interestingly, in our context this separation between compile-time expansion code and run-time code provided by the design of the packages system also enables cross-compilation without sacrificing extensibility. Thanks to this, rather than bootstrapping the full system in JavaScript, we can use the full-fledged compiler, which we have parameterized to use different back ends as needed during the same compilation. In this way, compilation modules can be compiled and loaded with the bytecode back end, while the source modules can be compiled independently in the back end selected for target executables.

As mentioned before, our back end is partially based on the WAM. The combination of a WAM-based engine and compiler is one of the most efficient approaches to implementing Prolog. Such engines and compilers are carefully crafted to optimize code execution and data movements, which makes them relevant and applicable even when the target of the compilation is a high-level language and the back end a highly optimized compiler. Although it has been shown that a basic WAM can be refined from more abstract specifications (Börger and Rosenzweig 1990), the mechanization of such process is not trivial. For that reason, it is currently unrealistic to expect that such kinds of optimizations can be introduced automatically from a high-level compiler. Thus, our approach reuses parts of the WAM design and compiler in order to implement relevant optimization opportunities.

Fig. 4 shows an overview of our back end, highlighting the points where it differs from the WAM code generation performed for the C-based engine. The first step consists of the normalization of (already expanded) Prolog code into simple Horn clauses, and the generation of symbolic WAM code. It is at this split point that separate schemes for register assignment, data representation, code generation, etc. are selected, as well as a separate runtime system. This process is described in the following sections.

3.1 Representing terms and modules

Our translation departs from the WAM in that, instead of defining an explicit heap and using *tagged* words, we use JavaScript objects to implement terms. Some advantages of this choice (discussed further in Section 5) are that garbage collection is then performed by the JavaScript engine and that interoperability with JavaScript code (Section 4) is simplified. However, we maintain many of the WAM concepts within these JavaScript objects. We use subclassing to build a hierarchy of term constructors. In the following we write $t \leq u$ if t is a subclass of u. We define `term_base` as the base class for all terms, and two other base classes for variables (`var_base` $\leq$ `term_base`), and non-variables (`nonvar_base` $\leq$ `term_base`). The `instanceof` operator can check if a given object belongs to any particular class, which would al-

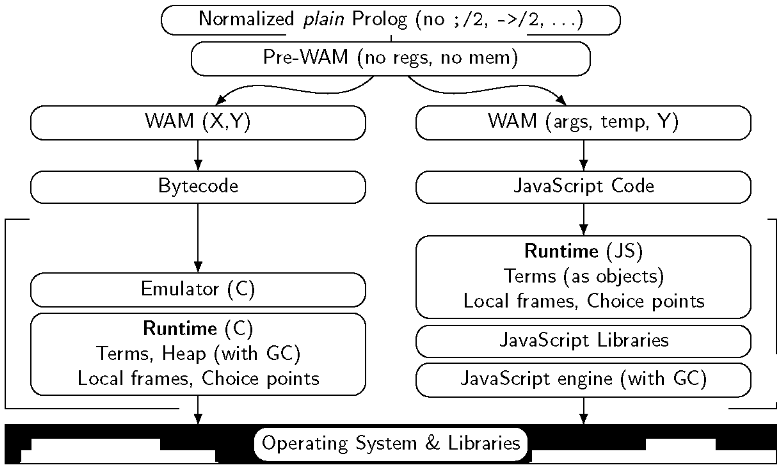
Normalized *plain* Prolog (no ;/2, ->/2, ...)
Pre-WAM (no regs, no mem)
WAM (X,Y)
WAM (args, temp, Y)
Bytecode
JavaScript Code
Emulator (C)
Runtime (C)
Terms, Heap (with GC)
Local frames, Choice points
Runtime (JS)
Terms (as objects)
Local frames, Choice points
JavaScript Libraries
JavaScript engine (with GC)
Operating System & Libraries

Fig. 4. Comparison of the bytecode and JavaScript WAM-based back ends

```
$m.def("term_base", function($m) { // Base for terms
  function term_base() {}; $m.ctor = term_base;
};
$m.def("var_base", function($m) { // Base for variables
  function var_base() {}; $m.ctor = var_base; $m.base = $r.query("term_base");
};
$m.def("nonvar_base", function($m) { // Base for non-variables
  function nonvar_base() {}
  $m.ctor = nonvar_base;
  $m.base = $r.query("term_base");
  $m.mlink = function($c) {
    $c.prototype.unify = function(w, a0) {
      return a0.unify_nonvar(w, this);
    };
    $c.prototype.deref = function() { return this; };
  };
};
```

Fig. 5. Base classes for *var* and *nonvar* terms.

ready provide the conditional code necessary to implement all operations on terms, like unification. However, looking for a particular base class is definitively slower than fast *switch on tag* operations. In this back end we redefine most operations on terms using *dynamic dispatching* to simulate *switch on tag* operations. Fig. 5 shows an implementation of these classes. Some methods in this code require a reference to the state of the *worker* (the set of control stacks containing trail entries, choice-points, and frames — see Section 3.2) denoted as w. From that base hierarchy, we define the rest of the elements of the domain of terms as follows.

Variables (and Unification). Variables are defined in Fig. 6. Since some WAM optimizations (e.g., conditional trailing) require comparing the relative age of variables and there is no such order for JavaScript objects, we define a global timestamp. Variables are therefore tuples (*ref*, *timestamp*). Each time a variable is created, the

```
 1 $m.def("t_var", function($m) { // Variables
 2   function v() { this.ref = this; this.timestamp = timestamp++; }
 3   $m.ctor = v;
 4   $m.base = $r.query("var_base");
 5   $m.mlink = function($c) {
 6     $c.prototype.deref = function() { // Dereference
 7       if (!this.is_unbound()) return this.ref.deref();
 8       return this;
 9     };
10     $c.prototype.is_unbound = function() { return this.ref === this; };
11     $c.prototype.unify_nonvar = function(w, a0) { // Unify with nonvar
12       if (!this.is_unbound()) return this.ref.unify_nonvar(w, a0);
13       return this.unify(w, a0);
14     };
15     $c.prototype.unify = function(w, a0) { // Unify
16       if (!this.is_unbound()) return this.ref.unify(w, a0);
17       a0 = a0.deref();
18       if (a0 instanceof v) {
19         if (this.timestamp > a0.timestamp) {
20           this.ref = a0; w.trail(this);
21         } else {
22           a0.ref = this; w.trail(a0);
23         }
24       } else { this.ref = a0; w.trail(this); }
25       return true;
26     };
27     $c.prototype.unbind = function() { this.ref = this; }; // Unbind
28   };
29 });
```

Fig. 6. Variable definition and methods.

timestamp is incremented.[2] Given a variable x, x`.deref()` Line 6 obtains the dereferenced value. It does so by invoking `deref` until the variable is unbound. If the variable points to a *nonvar* object, it simply returns that object, as specified by the `deref` of *nonvar*s. Dynamic dispatching is used in a similar way to implement unification. The process includes two cases. First x`.unify(w,` y`)` (Line 15) unifies x and y, and returns a boolean indicating whether the unification succeeded and updates the state `w` accordingly. The other case is x`.unify_nonvar(w,` y`)` (Line 11). It assumes that y is dereferenced to a *nonvar*. For *nonvar* objects, x`.unify(w,` y`)` is defined as y`.unify_nonvar(w,` x`)`, which is implemented by each of the derived classes.

Functor, Predicate Symbols, and Modules. They represent atoms and structures ($\leq$`nonvar_base`). A constructor contains as many arguments as its arity, copying them to `a0`, `a1`, fields. Additionally it contains static (i.e., shared by all the objects in the class) entries for the *name* and *arity*. Symbols for predicates

[2] Timestamps on variables are necessary also to implement lexical comparisons of terms, such as `compare/3`.

```
$r.def("t_string", function($m) { // String primitive
  function s(a0) { this.a0 = a0; }
  $m.ctor = s;
  $m.base = $r.query("t_nonvar");
  $m.mlink = function($c) {
    $c.prototype.unbox = function() { return this.a0; }; // Unbox
    $c.prototype.unify_nonvar = function(w, other) { // Unify with nonvar
      if (!(other instanceof s)) return false; // not a string, fail
      return this.a0 === other.a0; // proceed if the strings are the same
    };
  };
});
```

Fig. 7. Primitive term definition for `t_string`.

include an additional `execute` method containing the compiled body. The body compilation process will be described in Section 3.2. During compilation, we generate a new class per predicate or functor symbol (e.g., `append/3`, `./2`, `[]/0`), nested within their corresponding module (usually, `user` for functor symbols). Atoms are a special case of functor symbols with arity 0. In the same way that we associate the body of a predicate with the head functor definition, we associate the content of a module with its atom. We do so by storing nested symbols for predicates and functors inside them.

Primitive Terms. These are the terms that implement term wrappers for primitive values or arbitrary JavaScript objects. They are defined as $t \leq$ `nonvar_base`. Numbers (whose class is called `t_num`) and native strings (`t_string`) are two examples. An example definition for native strings, plugged into the runtime layer, is shown in Fig. 7. Code using strings can import the string constructor with `var s=$r.query("t_string").ctor` (only once in its context). Then, it can be used anywhere with `new s("...")`. Note that primitive data creation acts as a *boxing* operation, while a method `unbox` is sometimes necessary or convenient.

3.2 Control stacks and code generation

In order to implement backtracking, we adapt some of the registers and stacks of the WAM, with some modifications. Our machine state is defined in a `worker`, that contains:

- `goal`: the goal being resolved (a term).
- `undo`: a stack that implements the trail. Each entry in the trail is a variable. Trailing pushes a variable onto the stack, untrailing pops entries up to a certain point, and undoes variable changes by invoking the `unbind` method on each of them.
- `choice`: the current choice point, which contains the *failure continuation* and a copy of all the worker registers (including a reference to `goal`).
- `frame`: the current local frame, which contains Y frame variables, saved frame, and the success continuation.

The combination of choice, frame, and goal are similar to the frame structure in B-Prolog (Cs and Zhou 2007) (and also to the original Dec10-Prolog abstract machine): there are no X registers and argument registers (arguments of `goal`) and local JavaScript variables (for temporaries) are used instead. The code is generated and executed in a similar way to (Morales et al. 2004). The WAM code is split into chunks of consecutive instructions separated by predicate calls. Each chunk is compiled as a closure, which after execution returns the next closure to be executed. Before each predicate call, we set a success continuation that points to the next chunk. When there are no more chunks, we return the next continuation saved in the worker. As usual, choice points are created when executing nondeterministic code. Failure is implemented by untrailing, restoring the worker registers, and jumping to the failure continuation. The main changes are that the timestamp is also saved and restored.

3.3 Attributed variables

Most current Prolog systems offer (at least some primitive handling of) *attribute variables* in order to be able to extend unification (and also to allow more flexible control of execution). Attributed variables, as introduced by Huitouze (1990), are special variables that can be associated to a term called an *attribute*. Classical built-ins view attributed variables as normal variables. However the unification of such a variable with an instantiated term or another attributed variable is redefined according to a user-defined predicate. This mechanism is very powerful and allows the efficient implementation of coroutines (Holzbaur 1992), constraint solvers (Holzbaur 1995), and other high-level language extensions (Holzbaur and Frühwirth 1999) directly in Prolog. To implement such extensions, a number of Ciao packages make extensive use of attributed variables.

In the back end we implement attributed variables obeying the Ciao interface. They are enabled by the `attr` package, which follows the proposal by Demoen (2002) for hProlog. The interface is based on `get_attr/3` (that gets the attribute of a variable), `put_attr/3` (that sets the attribute of a variable), and `attr_unify_hook/2` (which is invoked when two attributed variables are unified, or an attributed variable is unified with a *nonvar* term). The runtime code included by the back end for attributed variables provides a definition for the built-ins, as well as a new class of terms for attributed variables.

Once this interface is in place and supported at the JavaScript level (including a number of additional support predicates) the system can exploit all the attribute variable-based extensions present in Ciao, including constraint solvers, extended control, etc. The coroutining `freeze/2` predicate and the Ciao CLP(FD) solver are examples of such extensions which will be used extensively in the experimental evaluation.

4 Interfacing with JavaScript code

We have focused so far on the runtime and code generation. In practice, these are useless without a process to make the source and target layers interoperable. For this reason, we require a foreign interface between JavaScript and Prolog. This interface is the basis for both embedding Prolog into existing JavaScript code and implementing the standard set of libraries interfacing with the O.S. (in this case, through the browser).

A JavaScript to Prolog interface is straightforward if we follow the compilation algorithm, which already defines how terms are built and unified, and how goals are resolved. Since terms are represented by JavaScript objects and since memory is reclaimed automatically, there are no major complications. Only an API which abstracts implementation details is necessary.

Accessing JavaScript data and code from the Prolog side is more involved. Any external object can be seen as an atomic type (so that two terms bound to JavaScript objects unify iff they are actually bounded to the same object). In many cases we want to read or modify object fields or invoke some of its methods. In (Wielemaker and Anjewierden 2002), a single set of predicates is used to perform those operations (`new/2`, `send/2`, `get/3`). In our approach, we follow (Pineda and Bueno 2002) where objects are seen as modules and methods as predicates of those modules. In practice, this allows using the same syntax and similar semantics as when specifying interfaces with formal properties (Ciao-style *assertions* (Hermenegildo et al. 2005)) and as in other Ciao foreign interfaces (e.g., for C). Consider for example:

```
:- pred document(-element) + js:foreign("return␣document;").
:- js:foreign_class element {
  :- pred body(-element) + js:foreign("return␣this.body;").
  :- pred set_innerHtml(+X) :: string + js:foreign("this.innerHtml=X;").
}.
```

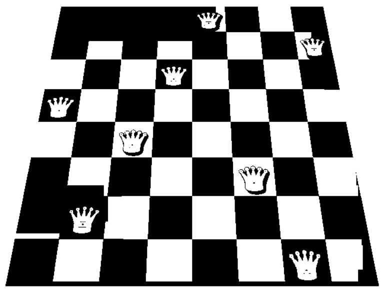
Fig. 8. Graphical representation for a solution of Queens-8.

where each `predicate` assertion indicates (among other possible properties) the expected types and modes of the arguments and the computational properties of the code (+ field at end of the assertion). Foreign code in JavaScript is specified with the `js:foreign` property. Such foreign code is assumed to be deterministic (Ciao `is_det` property) unless otherwise noted. In the example, modes (`+` and `-`) are used in the usual way to express input and output arguments (Ciao `isomodes` library), and types/-classes (`element`, `string`) are specified in the modes or in a `::` field. The assertions inside the `foreign_class` block specify the internal methods associated with objects that are `element`s. Each `foreign_class` defines a term wrapper for foreign data that includes the required glue code predicates. Given the previous interface, the following is a simple, html-oriented *hello world* program:

```
main :- document(D), D:body(B), B:set_innerHtml("Hello World").
```

It queries the document body and replaces its text with the given string. Using the same idea we have easily created more complex code like that generating Fig. 8.[3]

[3] The on-line version of this program is available at `http://cliplab.org/~jfran/ptojs/queens_ui/queens_ui.html` and can be tested on, e.g., a smart phone (a QR code is provided for convenience).

5 Experimental evaluation

We have measured experimentally the performance of the compiler back end and the system runtime and libraries by compiling a collection of unmodified, small and medium-sized benchmarks to JavaScript and comparing their execution time (under the V8 engine and Chrome 17 (Google)) with that of the Ciao virtual machine. Although raw performance is currently not our main goal, this gives us an initial indication of the size of problem that is amenable to client-side execution with the current implementation. We have chosen a) the following classical benchmarks:

qsort	Implementation of QuickSort.
tak	Computation of the Takeuchi function.
fft	Fast Fourier transform.
primes	Sieve of Eratosthenes.
nreverse	Naive reversal of a list using `append/3`.
deriv	Symbolic derivation of polynomials.
poly	Raises symbolically the expression `1+x+y+z` to the n^{th} power.
boyer	Simplified Boyer-Moore theorem prover kernel.
crypt	Cryptoarithmetic puzzle involving multiplication.
guardians	Prison guards playing game.
jugs	Jugs problem.
knights	Chess knight tour, visiting only once every board cell.
11-queens	N-Queens with $N = 11$.
query	Makes a natural language query to a knowledge database with information about country names, population, and area.

as well as b) the following collection of more complex problems:

A collection of CLP(FD) programs. We use a CLP(FD) library based on indexicals (Codognet and Diaz 1996) written in Prolog *using attributed variables* (plus syntactic extensions, etc.). We tested the classical `SEND+MORE=MONEY`, a `sudoku` solver, the `bridge` optimization problem, and the first solution to N-**Queens** with $N = 50$.

sat-freeze A benchmark based on an implementation of the DPLL algorithm for solving the Boolean satisfiability problem (SAT) (Howe and King 2010). The solver implements *watched literals* using `freeze/2` for delayed control.

We ran on a MacBook Pro, Intel Core 2 Duo (2.66 GHz and 3MB L2 cache). The execution times and the slowdown ratios are shown in Fig. 9 and Table 1.

Since the target is a dynamic language where we do not have precise control of data sizes, placement, memory movements, or assembler instructions (unlike, e.g., in a translation to C or a bytecode engine written in C), the gap between the source code and what is finally executed is large and slowdowns are to be expected. Indeed, the geometric mean of the slowdown for all the benchmarks is 10.00. Also, in our experience the actual performance is highly dependent on the engine which executes the JavaScript code (see later for details).

A more careful study splits the benchmarks into several groups. The first group (from `qsort` to `primes`) requires fast management of control and backtracking, but

	Benchmark	Ciao t_0 (ms)	JS (V8) t_1 (ms)	Ratio t_1/t_0
Group 1	**qsort (x1000)**	28.39	267.00	9.43
	tak (x10)	55.70	149.00	2.67
	fft	13.60	74.00	5.44
	primes (x100)	4.21	27.00	6.41
Group 2	**crypt (x10)**	6.60	44.00	6.67
	guardians	6.63	79.00	11.91
	jugs (x10)	17.80	89.00	5.00
	knights	371.00	1636.00	4.40
	11-queens	286.00	1672.00	5.84
	query (x100)	17.30	524.00	30.28
Group 3	**nreverse**	4.02	95.00	23.63
	deriv (x1000)	6.42	118.00	18.38
	poly (x10)	22.50	295.00	13.05
	boyer	27.20	1281.00	47.09
Group 4	**sendmore-fd**	20.40	217.00	10.63
	sudoku-fd	110.00	1310.00	11.90
	bridge-fd	2322.00	22857.00	9.84
	50-queens-fd	151.00	2123.00	14.05
	sat-freeze (x10)	376.00	2969.00	7.89

Table 1. Performance comparison of the bytecode and JavaScript back ends.

does not create complex data, and exhibits the best performance results. The second group (from `nreverse` to `boyer`) heavily depends on data creation and unification and performs worse. In particular, we tracked down this difference to a concrete issue in the state of our compilation scheme: we currently perform less indexing than what the WAM can do. To validate this assumption, we disabled indexing in the **bytecode** version of `boyer`. This yielded code which is 7.68 slower, which made the JavaScript / bytecode speed ratio to be in the ballpark of the first group. The performance of the benchmarks in the third (search problems) and fourth groups (constraints) can be explained in a similar way by their internal dependency on complex data manipulation (e.g., internal data structures for FD implementation) or indexing (e.g., `query`).

Practicality. We believe that the absolute performance achieved is sufficient for a large range of interactive, web-bound, non computationally-intensive tasks. And even for the case of more computationally-intensive tasks, if they are, however, not straighforward to program in a traditional language (e.g., they involve constraint solving, reasoning, etc.) we believe that our technology can be very useful. As just a

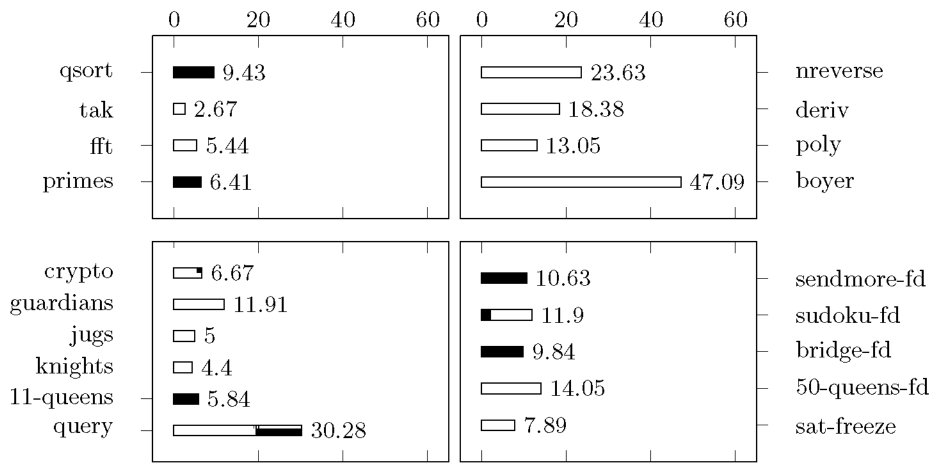

Fig. 9. Slowdown comparison of the different groups of benchmarks.

simple example, rule-based form validations with non-trivial, interdependent rules can be complex enough to be much more natural to program in a constraint/logic-based language. When confronted with the dilemma of server-side vs. client-side execution, data transmission delays[4] may make client-side execution be preferable even if client-side execution is slower than on the server side. And, as mentioned before, there are also other issues such as privacy or the cost of server-side computation and storage, which favor client-side execution irrespective of performance.

As stated before, our first goal has been the construction of a framework that is as complete as possible and easy to maintain, and which offers a high degree of compatibility for existing code. We believe our results show that our system allows non-trivial code (such as that implementing CLP(FD) or the SAT solver example, just to give two examples of code foundations which can be interesting to execute in a browser) to be easily run in browsers practically unmodified, alongside with other JavaScript code.

Nevertheless, we expect to obtain further performance improvements using more sophisticated compilation techniques (e.g., applying indexing and other WAM optimizations more aggressively and eventually program analysis). At the same time, the approach will obviously benefit from future improvements in the JavaScript platforms (which we have seen to improve significantly during our work).

Further Details on the Performance Results. One source of overhead identified is the cost of term representation. 32-bit WAM implementations typically require only $(1 + n)$ words to represent an f/n structure constructor, and 1 word for simpler objects like variables and constants. Objects in JavaScript are in principle much more complex, as values are records (dictionaries or hash tables) with an arbitrary number of fields. One key optimization in V8 are *hidden classes* and *inline caching* (brought in from efficient implementations of Self (Chambers et al.

[4] Note that the lower bounds to latency times (`ping`) are limited by the speed of light transmission over optical lines: 6.7 ms / 1000 km, and, in fact, much higher in practice.

1989)) to represent records efficiently and optimize property access. Even with those optimizations the system still requires 3 words plus data per object.

In (Morales et al. 2008) we observed that simply doubling the space required for tagged words, leaving the rest of the engine unaltered, can significantly affect performance in WAM-based machines. Marking some distinguished elements in the type lattice with tag bits is a clever optimization, hard to reproduce without a specialized heap representation. Moreover, we need additional data such as *timestamps* that is not required in the WAM, where the age of terms can be compared directly by their pointer addresses.

Alternatively, an explicit management of the heap as an array could improve term encoding (by making it closer to that of the WAM). This approach has been used in the compilation from C to JavaScript (as done in EMScripten (Zakai 2011), which has a 2.4-8.4 slowdown w.r.t. native code) or systems compiling to Java like Lean Prolog (Tarau 2011b). However, by adopting this approach we would lose some advantages of using a native JavaScript representation, including getting garbage collection for free. Additionally, larger and more complex runtime code would be required. The impact of this change is difficult to evaluate *a priori* and is left as future work.

Performance on Other Browsers. We tested the performance of our compilation on other major browsers (SpiderMonkey, Firefox 11; Nitro, Safari 5.1.2), and observed a slowdown of between 2.1÷ and 11.5÷ w.r.t. Google's V8. We have verified in synthetic benchmarks that, despite these virtual machines being as good as V8 for usual JavaScript programs (including a handcoded version of `tak`), they are less efficient in the creation and manipulation of many small objects. One reason is that objects in V8 are significantly smaller than in other engines, which makes SpiderMonkey and Nitro less convenient for our current translation scheme. We conjecture that V8 implements optimizations related to frequent, small object creation, as one of the benchmarks in the V8 suite is the compilation of the EarleyBoyer classic Scheme benchmarks using `sch2js`, which later evolved to be part of Hop (Loitsch and Serrano 2007). Nevertheless, for benchmarks which do not create large numbers of objects, the engines of the major browsers offer similar performance. It would also be interesting to explore whether with an explicit heap smaller performance gaps might be observed across engines.

6 Conclusions and future work

We believe our system makes a significant contribution towards the practical feasibility of client-side Web applications based (fully or partially) on (constraint) logic programming, while relying exclusively on Web standards. This reliance makes it possible to execute code on a variety of devices without any need for installation of additional plug-ins or proprietary code. We believe this is an important advantage, specially since a good number of the currently popular portable devices make such installation hard or impossible.

We made a strong effort to preserve source compatibility with existing Prolog code, and declaring special libraries and dialectic changes explicitly. For all this, the module and package system of Ciao was of great help. The current implementation represents a promising scaffolding on top of which a truly full-fledged system can be built. Our future work will be focused on several parallel lines. First, developing automatic methods for distributing code across browsers and servers, using AJAX or WebSockets for communication. Second, improving the compilation technology, specially using more WAM-level optimizations, analysis information, and the combination of such optimizations with JIT compilation, which has already been shown to significantly improve the execution of Prolog interpreters (Bolz et al. 2010). Third, gradually extending the current implementation of the Ciao libraries and language features, which would allow client-side execution of more and more complex programs.

The system is integrated in the Ciao repository and will be included in upcoming Ciao distributions. Examples and benchmark programs (including the Queens program of 8) are publicly available from `http://cliplab.org/~jfran/ptojs`.

References

Ait-Kaci, H. 1991. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press.

Bolz, C. F., Leuschel, M., and Schneider, D. 2010. Towards a jitting VM for prolog execution. In *PPDP'10 - Proc. of the 12th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming.* ACM, Hagenberg, Austria.

Börger, E. and Rosenzweig, D. 1990. From prolog algebras towards wam - a mathematical study of implementation. In *CSL'90.* Springer LNCS.

Cabeza, D. and Hermenegildo, M. 2001. Distributed WWW Programming using (Ciao) Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming 1,* 3 (May), 251–282.

Chambers, C., Ungar, D., and Lee, E. 1989. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not. 24,* 10 (Sept.), 49–70.

Codognet, P. and Diaz, D. 1996. Compiling constraints in clp(fd). *J. Log. Program. 27,* 3, 185–226.

Cs, C. and Zhou, N.-F. 2007. A Register-Free Abstract Prolog Machine with Jumbo Instructions. In *International Conference on Logic Programming.*

Demoen, B. 2002. Dynamic attributes, their hProlog implementation, and a first evaluation. Tech. Rep. CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium. October.

ECMA International. 2009. ECMAScript Language Specification, Standard ECMA-262, Edition 5. Tech. rep. September. Available at `http://wiki.ecmascript.org`.

Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation.* PLDI '09. ACM, New York, NY, USA, 465–478.

Google. V8 Javascript Engine. `https://developers.google.com/v8/design`.

Hermenegildo, M., Puebla, G., Bueno, F., and García, P. L. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr. 58,* 1–2.