CHR.js: A CHR Implementation in JavaScript

Falco Nogatz[1], Thom Frühwirth[2], and Dietmar Seipel[1]

[1] University of Würzburg, Department of Computer Science
Am Hubland, D – 97074 Würzburg, Germany
{falco.nogatz,dietmar.seipel}@uni-wuerzburg.de
[2] Ulm University, Institute of Software Engineering and Programming Languages
D – 89069 Ulm, Germany
thom.fruehwirth@uni-ulm.de

Abstract

Constraint Handling Rules (CHR) is usually compiled to logic programming languages. While there are implementations for imperative programming languages such as C and Java, its most popular host language remains Prolog. In this paper, we present CHR.JS, a CHR system implemented in JavaScript, that is suitable for both the server-side and interactive client-side web applications. CHR.JS provides (i) an interpreter, which is based on the asynchronous execution model of JavaScript, and (ii) an ahead-of-time compiler, resulting in synchronous constraint solvers with better performances. Because of the great popularity of JavaScript, CHR.JS is the first CHR system that runs on almost all and even mobile devices, without the need for an additional runtime environment. As an example application we present the CHR.JS Playground, an offline-capable web-interface which allows the interactive exploration of CHRs in every modern browser.

1 Introduction

Constraint Handling Rules (CHR) [7] has its origins in the field of constraint logic programming. However, today's applications cover many different areas, ranging from traditional reasoning and time tabling problems to data mining, compiler construction, and computational linguistics [8]. Although it is usually used together with a hosting language, CHR has been evolved to a general-purpose programming language since its creation in the early 1990s. One reason for this development has been the implementation of CHR systems in different programming languages. Among others, there are CHR systems for most popular languages, including Java [22, 1, 18] and C [23]. While these implementations in imperative programming languages are typically faster, CHR's most popular host language remains Prolog. As a result, CHR is more common in the research community than for commercial applications that could benefit from its forward-chaining rewrite rules.

As today's applications increasingly become interactive, one of the main challenges is the handling of state and its mutations. The handling and coordination of multiple events (e.g., mouse clicks, sensor data) could be described using CHRs, modelling the program's state as the actual content of the constraint store. From this point of view, the combination of CHR with the increasingly interactive environment of web applications seems promising. Besides this, the programming language and field of constraint logic programming could benefit from a web-based implementation that can be easily run on most of the current devices, without the need for an additional installation step. JavaScript seems like an intended target to be a CHR host language: measured against dissemination and popularity, it is currently one of the most popular programming languages. Douglas Crockford, who developed the JavaScript Object Notation (JSON), once stated that every personal computer in the world had at least one JavaScript interpreter installed on it and in active use [3].

JavaScript is already a popular target language for compilation, too. There are currently more than 300 languages that compile to JavaScript.[1] By porting CHR to JavaScript, we can benefit from this broad distribution of runtime environments. For the implementation of CHR.JS, we define some design goals:

Resemblance to existing CHR systems. The CHR.JS syntax should feel natural for users with experience in other CHR systems.

Syntax based on JavaScript. The definition of CHRs should conform to design patterns in JavaScript. We strive for a natural integration of both languages.

Support for different runtime environments. CHR.JS should be portable across multiple runtime environments, including all modern web browsers and the server-side JavaScript framework node.js [2].

Extensible tracing options. By being executable on the web, CHR can be opened to the public. We want to improve the understanding of CHR programs by providing various tracing options.

Efficiency. The CHR system should be efficient. But unlike most of the other hosting languages of existing CHR implementations, JavaScript is an interpreted programming language and compiled just-in-time (JIT), so it might not be possible to compete with C or Java implementations.

Overview. The remainder of the paper is organised as follows. In Section 2, we shortly introduce the syntax and semantics of CHR. As a motivational example and to emphasise the usefulness of the CHR.JS system, we present the web-based CHR tracer called CHR.JS *Playground* in Section 3. In Section 4, existing approaches on the compilation of CHR into imperative programming languages are introduced. We define the integration of CHR with the JavaScript language in Section 5. The compilation scheme for asynchronous CHRs, which is used by the interpreter of CHR.JS, is presented in Section 6. Next, Section 7 introduces the compilation scheme for synchronous CHRs, used by the AOT compiler. The performance of CHR.JS is compared to several other CHR systems in Section 8. Finally, we conclude with a summary and discussion of future work in Section 9.

2 Constraint Handling Rules

In this section, the syntax and semantics of CHR are shortly summarised. For a more detailed introduction, we refer to [7].

Constraints are expressions of the form $c(t_1, \ldots, t_n)$ with $n \geq 0$, where c is an n-ary constraint symbol and $t_1, \ldots, t_n$ are terms of the host language. In addition to these *CHR constraints*, there are *built-in constraints*, which are data structures of usually the same form but which defined in the host language. E.g., in Prolog these are predicates – either defined by the user or built-in –, and functions in JavaScript.

All CHR constraints that are known to be true are placed in a multi-set which is called the *constraint store*. By defining rules, it is possible to manipulate its contents. There are three types of rewrite rules which are applied until a final state is reached:

[1] List of languages that compile to JS, https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js

$$gcd1 \ @ \ gcd(0) \ \Leftrightarrow \ true$$
$$gcd2 \ @ \ gcd(N) \ \setminus \ gcd(M) \ \Leftrightarrow \ 0 < N, N \leq M \mid gcd(M - N)$$

Figure 1: CHR rules to calculate the greatest common divisor given as *gcd/1* constraints.

- the *propagation rule* is of the form $K_1, \ldots, K_n \ \Rightarrow \ G_1, \ldots, G_m \mid B_1, \ldots, B_l$,
- the *simplification rule* is of the form $R_1, \ldots, R_n \ \Leftrightarrow \ G_1, \ldots, G_m \mid B_1, \ldots, B_l$,
- the *simpagation rule* is a combination of both and of the form $K_1, \ldots, K_k \setminus R_{k+1}, \ldots, R_n \ \Leftrightarrow \ G_1, \ldots, G_m \mid B_1, \ldots, B_l$,

with K_i and R_i CHR constraints, G_i built-in constraints, and B_i built-in or CHR constraints. The CHR constraints denoted by K_i are kept in the constraint store, the R_i are removed, and the B_i are added. A rule is only applied if there are constraints in the constraint store which match with K_i resp. R_i, and if the *guard* specified by the built-in constraints G_i is satisfied. A rule is optionally preceded by *Name* @, where *Name* is its identifier.

Unlike its most popular host language Prolog, CHR is a committed-choice language and consists of multi-headed and guarded rules. There are multiple operational semantics for CHR. CHR.JS is based on the refined operational semantics ω_r as defined in [4], like most current CHR implementations. Most importantly, our implementation fixes the execution order of the given rules, while there remains non-determinism in the search of partner constraints.

Running Example. Figure 1 shows the classical *gcd/1* handler. Its two rules with the names *gcd1* and *gcd2* define a constraint solver that calculates the greatest common divisor (GCD) of all positive integers specified as *gcd/1* constraints. Given the two constraints $gcd(36)$ and $gcd(8)$, the rule *gcd2* is applied, replacing $gcd(36)$ by $gcd(28)$. After five more steps, the constraint $gcd(0)$ is created and removed by rule *gcd1*. The only remaining constraint $gcd(4)$ is the result.

3 CHR.js Playground: Web-based Tracing for CHR

In our experience, there are two typical scenarios when working with a CHR system, both with different requirements: either one wants to simply (i) use the constraint solving mechanism, then with its best performance; or the user's aim is (ii) to interact with the rules and the constraint store. The latter is often the case when developing and tracing CHRs, or for educational purposes.

As an example application, our contribution contains an interactive web-based tracer for CHR. Figure 2 presents a screenshot of the created web application. It is inspired by collaborative code sharing platforms like JSFiddle[2] and SWISH[3] [21]. On the left-hand side the code editor is used to edit CHRs and to define built-in constraints in native JavaScript. In the right panel, queries can be specified, there is an optional tracer with step-by-step execution, and the current constraint store is visualised. A public, hosted instance is available online at `http://chrjs.net/playground`.

[2] JSFiddle, `https://jsfiddle.net/`
[3] SWISH, `https://swish.swi-prolog.org/`

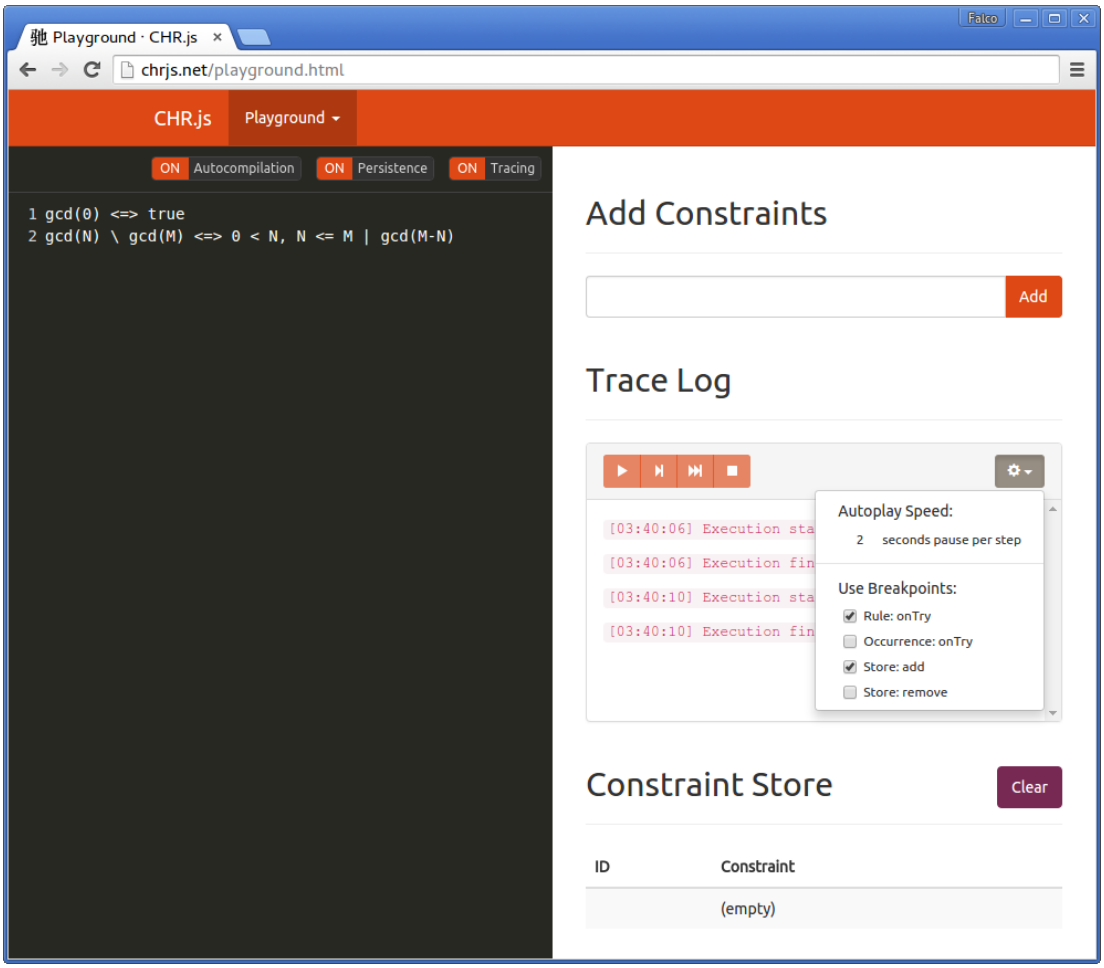

Figure 2: Screenshot of the CHR.JS Playground at `chrjs.net` run in the Google Chrome browser with tracing enabled.

Although it is a web application, the CHR.JS Playground can be used standalone offline, because unlike SWISH no remote Prolog server is needed to run the specified queries. The stated CHRs are compiled on the fly. The CHR.JS Playground provides a persistent mode so it is possible to define CHRs that work on an already existing constraint store. To the best of our knowledge, this is the first CHR implementation that runs on mobile devices.

4 Related Work

Several approaches to compile CHR have already appeared in the literature. Since it was created as a language extension for Prolog in the first place [11, 12, 13], Prolog is the target language predominantly discussed. The implicit execution stack of Prolog maps very well to the ordered execution stack of the refined operational semantics ω_r: if a new constraint is added, all of its

occurrences are handled as a conjunction of Prolog goals and are therefore executed before any other added constraint.

For imperative target languages, there are two major CHR systems: (i) K.U.Leuven JCHR [18], a CHR system for Java, and (ii) CCHR [23], a CHR system for C, which is currently the fastest implementation of CHR. Both implementations are discussed in [19]. A historical overview of CHR implementations can be found in [17].

Basic Compilation Scheme for Imperative Languages and Optimisations. The compilation scheme of CHR for logic programming languages as presented in [13] can be adopted to a procedural computation style by replacing Prolog predicates with methods or function calls. The resulting basic compilation scheme for imperative programming languages is presented in [19] and independent from the actual target language.

Although this general scheme is conform to the refined operational semantics ω_r, it is fairly inefficient. Multiple optimisations have been proposed in the past with [19] providing an exhaustive overview. These optimisations have been categorised into general ones (e.g., indexing, avoiding loop invariants, and guard simplifications) and optimisations in respect of recursions. Since JavaScript does not – unlike, for example, C and some implementations of the Java Virtual Machine – support tail call optimisations (TCO), they are of special interest for the implementation of CHR.JS.[4] Most non-trivial CHR programs contain recursions, and the discussed basic compilation scheme adds even more layers of recursive function calls due to its use of helper functions for each occurrence handler. On the other hand, the maximum call stack size is very limited in all major JavaScript systems, ranging from about 10k to 50k. Van Weert et al. present an adapted compilation scheme using a technique called *trampoline* [10] to avoid stack overflows for CHRs with tail calls. For the CHR.JS compiler we make use of the *explicit stack*, where the host language's stack is replaced by a self-maintained continuation queue on the host's heap.

5 Seamless Integration of CHR into JavaScript

Following our design goals, CHR.JS should be easy to use and adapt for both JavaScript and CHR experts. However, JavaScript has a different syntax than CHR and the well-known hosting language Prolog. In this section, we present how CHR can be embedded into JavaScript code and used together with JavaScript's synchronous and asynchronous functions. Even though this is discussed in particular for CHR, this section is useful for any reader interested in the seamless integration of a rule-based language into JavaScript.

5.1 Embed CHR in JavaScript using Tagged Template Strings

The syntax of rule definitions is very similar to most existing CHR systems. Simple rules can be specified the same way as in CHR for Prolog (i.e. $\Rightarrow$ is encoded as `==>`, and $\Leftrightarrow$ as `<=>`), as long as one uses the JavaScript equivalents of Prolog operators.[5] Moreover, different rules are either separated by a newline or a semicolon instead of a period. Unlike K.U.Leuven JCHR, CCHR and CHR systems for Prolog, CHR constraints do not need to be declared in advance.

[4] The JavaScript language is standardised as *ECMAScript* [5]. The most recent version of the ECMAScript specification is the sixth version, often referred to as *ES6* or *ES2015* or by its codename *Harmony*. Although it defines TCO, it has not yet been implemented by the major browsers.

[5] For example the equality check in JavaScript is performed using `==` or `===` (type-safe), instead of the often used unification `=` in Prolog.

```
var chr = new CHR()
chr` gcd1 @ gcd(0) <=> true
     gcd2 @ gcd(N) \\ gcd(M) <=> 0 < N, N <= M | gcd(M-N)`
```

Figure 3: CHR.JS rules to calculate the greatest common divisor.

CHR.JS goes the other way round and uses a special syntax for the built-ins in the guard and body which must be enclosed by a dollar sign followed by curly brackets `${...}`. As of now, there is no support for logical variables.

JavaScript and Prolog have a thing common: both lack easy support for multiline strings. With SWI-Prolog since version 6.3.17 this has been addressed using *quasi-quotations* [20]. They provide a special syntax to directly embed text that spawns multiple lines, which makes them suitable to embed external domain–specific languages (DSL) into Prolog source code without any modification [16]. A similar technique has been recently added to JavaScript. So called *tagged template strings* allow to embed a DSL directly into JavaScript using the following syntax:

```
tag`text with ${ embedded_expr } even across multiple lines`
```

where `tag` is a user-defined function (the *template handler*) and expressions can be embedded using the `${...}` notation. Figure 3 presents the *gcd/1* handler as introduced in Section 2 in CHR.JS syntax. Since in JavaScript the backslash \ is an escape character in strings, it has to be escaped by a second backslash.

The CHR.JS constructor `CHR()` creates the template handler `chr`, which can be used to embed the CHRs in JavaScript as a tagged template string. The template handler generates an abstract syntax tree (AST) for the given CHR program. In order to parse the rules, we have formalised their structure using a Parsing Expression Grammar (PEG) [6] and its JavaScript parser generator *PEG.js*[6]. PEG's syntax is similar to Definite Clause Grammars (DCG). Since the template string contains JavaScript fragments (e.g., the guard `0 < N`), we extended the JavaScript meta-grammar shipped with PEG.js to be able to parse CHR rules, too. A more detailed presentation of these PEG rules, the rule grammar, and the generation of the AST is given in [15, Sec. 4.3].

5.2 Synchronous and Asynchronous Execution

For each constraint c/n in the rule head a caller function `chr.`c`(`$arg_1, \ldots, arg_n$`)` is created by CHR.JS. This way, the calculation of the GCD of the numbers 36 and 8 can be invoked by calling `chr.gcd(36)` and `chr.gcd(8)`. Because JavaScript has no static typing, the same caller function `chr.`c is used for all n, i.e. `gcd/2` is invoked by simply specifying two arguments.

In most traditional imperative programming languages there is a strict stack-based execution cycle. To avoid blocking functions, concurrency models based on multiple threads have been established. Since JavaScript is single-threaded, this has been solved using the *event loop*. For functions taking long to execute it is possible to define a callback which is invoked once the function has been finished. These functions are called *asynchronous*.

The GCD of our running example can be calculated without any blocking function. However, if the application of a rule depends, e.g., on the result of a possibly long-running database

[6] PEG.js, https://pegjs.org/

query, we can make use of CHR.JS' support for asynchronous functions. The caller function `chr.`$c(arg_1, \ldots, arg_n)$ provides a method `.then(callback)` to specify the callback function which is used once the constraint solving process has been completely finished. It is realised using the increasingly more popular approach of using *Promises* [9]:

```
chr.gcd(36).then(function () {    // call when finished handling gcd(36)
  chr.gcd(8).then(function () {   // call when finished handling gcd(8)
    console.log(chr.Store.toString()) // prints constraint store
  }) })                           // output:   gcd(4)
```

In JavaScript, a Promise represents the eventual completion or failure of an asynchronous operation. Once it is created, a callback function `cb` can be attached via `.then(cb)`. It is called as soon as the asynchronous function has been finished and takes the computed value as its first argument.

If there is only a single asynchronous function in one of the rule's guard or body, the complete CHR.JS constraint solver becomes asynchronous. Due to JavaScript's execution model using the event loop, there is simply no way to wrap an asynchronous function inside another function and make it synchronous, that means blocking, again. So we have to categorise two kinds of CHR programs that we want to use with CHR.JS: those which contain asynchronous functions, and those which do not. This is reflected also in our two compilation schemes: CHR.JS provides an interpreter, which uses and supports asynchronous functions. Its compilation scheme is presented in Section 6. On the other hand we provide an AOT compiler which only supports synchronous functions. Both have their justifications: the asynchronous version is flexible and avoids stack overflows by design, because only a single stack frame is generated per message in the event loop. On the other hand the synchronous version is faster and more natural for users already familiar with CHR.[7]

6 An Asynchronous CHR Interpreter using Promises

While an asynchronous function can not be brought back into a synchronous form, the other way round is always possible. JavaScript's `Promise.resolve(v)` creates a Promise object that always immediately resolves to the specified value `v`:

```
var p = Promise.resolve([1,2,3]) // creates a new Promise p
p.then(function (ret) {          // p is "then-able" now like chr.gcd(8)
  console.log(ret) })            // output:   [1,2,3]
```

We have modified the basic compilation scheme of [19, Sec. 5.2] to use Promises, i.e. to use asynchronous callbacks specified via `.then()`. The constraints in the rule body and guard must be called asynchronously. The modified loop variant in the compilation scheme for a single occurrence
`occurrence_`c_i`_`j_i`(...)` is presented in Figure 4.

The functions `resolve()` resp. `s()` successfully conclude the current Promise, similar to a `return` statement in synchronous function calls. With the functions `reject()` resp. `j()` the Promise gets rejected, similar to a `throw` statement in synchronous function calls.

`Promise.all(arrayOfPromises)` executes a given array of Promises in parallel. Note that due to CHR's refined operational semantics ω_r this is only allowed for the guards. The body constraints have to be handled sequentially instead. In [15, Sec. 4.5], we began to examine the

[7] A more detailed discussion on using synchronous or asynchronous functions is given in [15, Sec. 3.4].

```
if (!Store.allAlive(constraintIds))        // = nested check of still alive ids   [l. 8]
  return resolve()
if (!Store.allDifferent(constraintIds))    // = pairwise comparison           [ll. 9-11]
  return resolve()
if (History.has(ruleId, constraintIds))    // = notInHistory()                   [l. 13]
  return resolve()
var guards = [                             // = g_1 and ... and g_ng             [l. 12]
  new Promise(function(s, j) { return g_1 ? s() : j() }),
  ...
  new Promise(function(s, j) { return g_ng ? s() : j() })
]
Promise.all(guards)                        // prove all guards
  .then(function () {                      // all guards satisfied
    History.add(ruleId, constraintIds)     // = addToHistory()                   [l. 14]
    Store.kill(constraintIds[r])           // = kill()                        [ll. 15-17]
    ...
    Store.kill(constraintIds[h])
    Promise.resolve()
      .then(function() { return b_1() })   // invoke Promise for body 1       [ll. 18-20]
      . ...
      .then(function() { return b_nb() })  // invoke Promise for body nb
      .then(function() { resolve() })
      .catch(function() { reject() })
  }).catch(function() { reject() })        // could not be fulfilled
```

Figure 4: The modified loop variant for the compilation of a single occurrence. The bracketed line numbers refer to the original compilation scheme of [19].

parallel execution of constraints, which remains an interesting field for future improvements and research.

Support for Event Listeners and Breakpoints. Similar to the CHR reference implementation by Christian Holzbaur [13], CHR.JS provides a runtime environment which is reused by all instances. For example, the global constraint store referenced in Figure 4 as `Store` can be accessed as part of the created `chr` object using `chr.Store`. During program execution it emits events for the addition and removal of a constraint. As usual in JavaScript, they can be received by adding event listeners as follows:

```
chr.Store.on('add', function (c) { /* added constraint c */ })
chr.Store.on('remove', function (c) { /* removed constraint c */ })
```

We use traditional callbacks instead of Promises for the event listeners, since Promises can be executed only once, whereas it is a reasonable use-case to fetch the removal of several constraints with multiple event listeners.

Similar to the constraint store, the defined CHR rules can be accessed and modified using `chr.Rules`. It holds an object for every defined CHR rule, for instance `chr.Rules['gcd1']` and `chr.Rules['gcd2']`. They have a special `Breakpoints` property which allows the binding of a Promise that will be called once the rule is tried to be applied. Unlike for event listeners, the further application of the CHR.JS program is paused until this Promise is resolved.

7 Ahead-of-time Compilation with an Explicit Stack

Having to wait for the event loop for every single Promise, the compilation scheme presented in Section 6 does not result in fast constraint solvers. However, without the use of asynchronous functions, the constraint solver easily ends in a stack overflow because of the heavy use of

recursion in the basic compilation scheme. This can be avoided using an optimising technique which is introduced in [19] as *explicit stack*. Instead of directly calling, e.g., the body constraints of a rule, an appropriate *continuation* is returned by the occurrence handler and pushed to the global list `stack`. The continuations are then handled by a global loop, similar to the event loop:

```
function trampoline() {
  while (constraint = stack.pop()) // pull first element
    constraint.continue() } // might push new continuations to the stack
```

We push the constraints that have been added in the rule's body to `stack` and add the continuation as the property `cont` of this particular constraint, so the call of `constraint.continue()` simply invokes it. Working with continuations is common in JavaScript, because functions are first-class citizens.

The initial caller function of a constraint, for instance `chr.gcd()`, generates the constraint object and initialises the `constraint.cont` property with the first occurrence handler `__gcd_1_0`:

```
function gcd() {
  var args = Array.prototype.slice.call(arguments)
  var arity = arguments.length, functor = "gcd/" + arity
  var constraint = new Constraint("gcd", arity, args);
  constraint.cont = [__gcd_1_0, 0]; stack.push(constraint)
  trampoline() }
```

The occurrence functions are of the form $\_\_c_i\_a_i\_j_i$, with c_i the current constraint name, a_i its arity and j_i an increasing number which identifies the occurrence. At the end, we add a generic handler with the occurrence number $(j_i + 1)$, which simply takes the constraint `c` from the stack and adds it to the constraint store:

```
function __gcd_1_3(c,n) { c.cont = null; chr.Store.add(c) }
```

If, for instance, a constraint $c/0$ occurs only in a rule's body but in no head, there is no rule that can be applied when c is added. So the only occurrence handler is the generic $\_\_c\_0\_0$, which adds c to the constraint store.

Instead of directly using the caller functions of the body constraints, we now add these constraints to the stack and assign their first occurrence handler as their first continuation. Therefore, a typical generated code fragment looks like this:

```
if (condition)
  c.cont = [nextOccurrenceHandler, 0]; stack.push(c); return
```

For instance, this is used with the rule's guard as the `condition`: if they are not satisfied, the next occurrence handler is called. As seen in the previous code fragments, we not only specify the continuation but also a number n which is initialised with 0. It is used to effectively iterate through all possible combinations of partner constraints in rules where the active constraint is kept. Instead of using a (possibly nested) loop, we can simply add the current occurrence handler as the next continuation but with an incremented n, so the next combination of partner constraints is used.

8 Experimental Evaluation

CHR.JS has been developed in a test-driven approach. Currently its compliance to the refined operational semantics ω_r is ensured by more than 420 functional tests. The correctness of CHR.JS is discussed in [15, Sec. 4.2.2]. Our implementation closely resembles the basic compilation scheme proposed in [19], whose completeness and correctness is shown in the same place. The functional tests has been used in a continuous integration environment with *Travis CI*[8].

8.1 Used Benchmarks and Systems

In order to compare our implementation to existing CHR systems, we used four benchmarks based on [23] and the CCHR implementation[9]:

- *gcd* calculates the greatest common divisor of 5 and $1000 \times N$ using the subtraction-based Euclidean algorithm as presented in our running example. It is a linear program involving at most two constraints.
- *fib* calculates the N'th Fibonacci number by bottom-up evaluation. This involves at most three constraints: an `upto/1` and the last two `fib/2`.
- *primes* generates all prime numbers upto N using the sieve of Eratosthenes.
- *ram* is a RAM simulator which counts down from N.

The original CCHR benchmarks specify two more problems, *tak* and *leq*. We do not make use of them, because both require logical variables.

Basically, our benchmark suite executes a given command as often as possible within a 10 seconds time slot. It measures the number of iterations as well as the average execution time. We do not consider the used memory.[10]

All benchmarks are run on an Intel Core i7 9xx Dual Core CPU with 8 GB of RAM, using Ubuntu Server 16.04.3 64bit (Linux Kernel 4.4.0) with low load. We compare with three other CHR implementations: K.U.Leuven JCHR (v1.5.1) in Java (OpenJDK v1.8.0), CCHR (no version number provided, GCC v5.4.0), and K.U.Leuven CHR in SWI-Prolog (v7.6.4). For reference we provide a native JavaScript solution run with node.js (v9.5.0). CHR.JS has been used in v3.3.1 and with node.js (v9.5.0).

The asynchronous CHR.JS interpreter as presented in Section 6 makes great use of JavaScript's Promises. In [15, Sec. 6.2], we have shown that this technique is not even competitive to equivalent synchronous, iterative implementations. For every `Promise`, the JavaScript systems in all of today's browsers add a latency of four milliseconds. We therefore only use AOT-compiled CHR.JS programs in the benchmarks.[11]

8.2 Benchmark Results

Figure 5 shows the results of the benchmarks. Table 1 lists the geometric averages to complete the calculations. The averages for JavaScript have been set to 1 and those of the other systems have been scaled relatively. For the average calculation we consider only problem sizes that (i) have been successfully finished by all five systems within 10 seconds, and (ii) the overall completion time for every system is not lower than 0.1 milliseconds.

[8] Travis Continuous Integration service, https://travis-ci.org/
[9] Copy available at https://svn.ulyssis.org/repos/sipa/cchr/
[10] Benchmarks available at https://github.com/fnogatz/CHR-Benchmarks
[11] CHR.JS provides a command line utility to pre-compile CHR programs: `chrjs --optimized program.in`

Table 1: Relative Geometric Averages of Benchmark Results

Program	SWI-Prolog	K.U.Leuven JCHR	CHR.JS	JavaScript	CCHR
gcd	280	6.8	53	[a]1.0	.08
fib	3.3	39	[b]9.9	[c]1.0	.05
primes	16	18	19	1.0	.03

[a]For very large $N \times 1000$, the problem exceeds the largest safe integer number in JavaScript, `Number.MAX_SAFE_INTEGER`. The calculation therefore needs more iterations due to the imprecision of the input value.

[b]For $N = 1470$ the N'th Fibonacci number exceeds the largest possible number in JavaScript, `Number.MAX_VALUE`. From there on the results are simply stated as `Infinity`. The CHR.JS program finishes but the addition `Infinity + Infinity` is faster than the additions before, resulting in a faster overall execution.

[c]To avoid exceeding `Number.MAX_VALUE` we reset the numerical sequence every 1470th loop pass. For $N \approx 2 \cdot 10^9$ the JavaScript implementation takes about 10 seconds.

All benchmarks have in common that CCHR is the fastest CHR implementation. CHR.JS will not be able to compete with CCHR, since even the native JavaScript implementation of the problems is one order of magnitude slower than the C constraint solver.

The *gcd* and *fib* problems are similar in kind: both are actively working on only two constraints and one important CHR rule. By the use of a simpagation rule in both cases one of the constraints is removed and another gets added. But while for the *fib* problem it is guaranteed that always the older constraint is removed, this is not necessarily for *gcd*. In fact, the problem to solve `gcd(5), gcd(1000*N)` always removes the just now created `gcd/1` constraint.

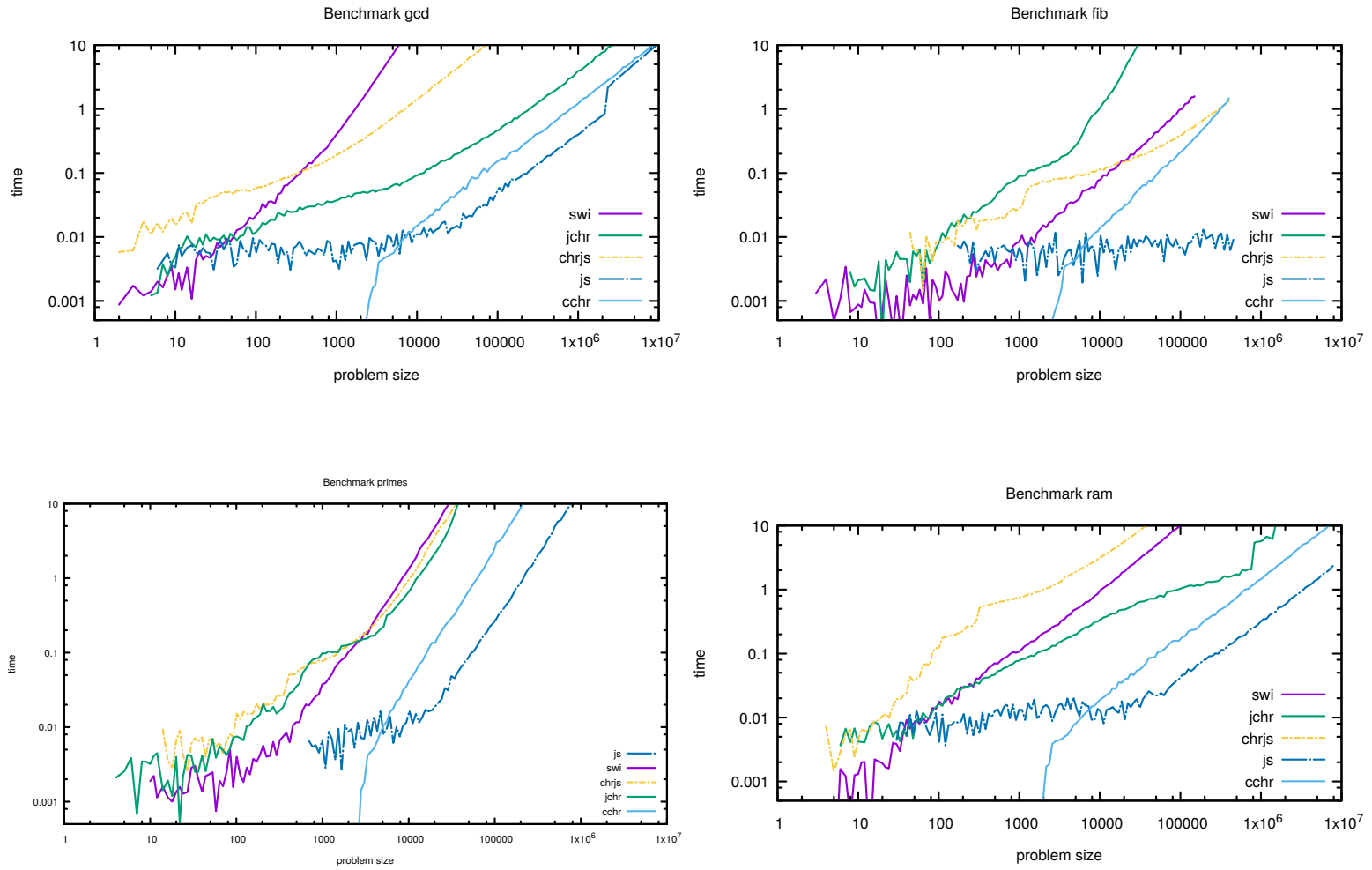

Figure 5: Results for the benchmarks *gcd*, *fib*, *primes*, and *ram*. Further annotations are given in Table 1. Note that all benchmarks have in common that the native JavaScript implementation is not linear as expected. Because the start-up time for node.js scripts is about 10ms, the linear dependence can be seen only for big enough problem sizes.

In the CHR.JS AOT compiler we already have implemented the *late storage* optimisation as suggested in [19, Sec. 5.3]. So, the problem to solve `gcd(5), gcd(1000*N)` stores only a single constraint at all, since a constraint `gcd(P)` immediately invokes the continuation for `gcd(P-5)`, and so on.

In the *primes* benchmark, our implementation is at par with the CHR systems in SWI-Prolog and JCHR, but still two orders of magnitude slower than the native JavaScript implementation. The *primes* problem highly depends on an efficient way to find partner constraints. To the best of our knowledge, there is no JavaScript standard implementation of an efficient in-memory index that works with node.js and all major browsers. In the worst case, a single lookup for partner constraints in CHR.JS has to loop through all existing constraints.

The results in Figure 5 and Table 1 suggest that the CHR systems in SWI-Prolog, JCHR and our contribution CHR.JS have implemented various optimisations, resulting in fairly similar performances. There is no clear ranking of all the four systems – except for CCHR, whose claim to be the fastest CHR implementation could be verified.

9 Conclusion

In this work, we have presented CHR.JS, the first implementation of CHR in JavaScript. It is published at https://github.com/fnogatz/CHR.js (MIT License) and supports synchronous and asynchronous JavaScript functions in the rules' guards and bodies. As an example application, we have presented the CHR.JS Playground. This offline-capable web application uses event listeners and breakpoints, so that rules and the constraint store can be interactively edited and the execution of the CHR program can be traced.

To achieve CHR programs with reasonable performances, we have presented the compilation scheme for CHR to synchronous JavaScript. It makes use of continuation-passing to avoid recursions. Although the generated programs are at least one order of magnitude slower than their native JavaScript counterparts, there are several promising open optimisation ideas.

The presented approach of embedding CHR into JavaScript using tagged template strings might be useful for the integration of other rule-based DSLs, too. In particular, we would like to adapt the existing JavaScript interface of SWI-Prolog's Pengines [14] library, which allows the definition of remote procedure calls from web applications to Prolog engines, to also use a similar mechanism.

References

[1] Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JaCK: A Java Constraint Kit. *Electronic Notes in Theoretical Computer Science*, 64:1 – 17, 2002.

[2] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node.js in Action*. Manning Publications, 2017.

[3] Douglas Crockford. JavaScript: The world's most misunderstood programming language. *Douglas Crockford's Javascript*, 2001.

[4] Gregory J Duck, Peter J Stuckey, Maria Garcia De La Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *Logic Programming*, pages 90–104. Springer, 2004.

[5] ECMAScript ECMA-Kommittee. A general purpose, cross-platform programming langugage, Standard ECMA-262, 1997.

[6] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[7] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[8] Thom Frühwirth. *Constraint Handling Rules - What Else?*, pages 13–34. Springer International Publishing, Cham, 2015.

[9] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*, pages 1–10. IEEE, 2015.

[10] Steven E Ganz, Daniel P Friedman, and Mitchell Wand. Trampolined style. In *ACM SIGPLAN Notices*, volume 34, pages 18–27. ACM, 1999.

[11] Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules. In *ERCIM/COMPULOG Workshop on Constraints, CWI, Amsterdam*, 1998.

[12] Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In *Principles and Practice of Declarative Programming*, pages 117–133. Springer, 1999.

[13] Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. *Applied Artificial Intelligence*, 14(4):369–388, 2000.

[14] Torbjörn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. *Theory and Practice of Logic Programming*, 14(4-5):539–552, 2014.

[15] Falco Nogatz. CHR.js: Compiling Constraint Handling Rules to JavaScript. *Master Thesis, Ulm University, Germany*, 2015.

[16] Falco Nogatz and Dietmar Seipel. Implementing GraphQL as a Query Language for Deductive Databases in SWI–Prolog Using DCGs, Quasi Quotations, and Dicts. In *Proc. 30th Workshop on Logic Programming (WLP 2016)*, 2016.

[17] Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.

[18] Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. pages 47–62.

[19] Peter Van Weert, Pieter Wuille, Tom Schrijvers, and Bart Demoen. CHR for imperative host languages. In *Constraint Handling Rules*, pages 161–212. Springer, 2008.

[20] Jan Wielemaker and Michael Hendricks. Why It's Nice to be Quoted: Quasiquoting for Prolog. In *Proc. 23rd Workshop on Logic–based Methods in Programming Environments (WLPE)*, 2013.

[21] Jan Wielemaker, Torbjörn Lager, and Fabrizio Riguzzi. SWISH: SWI-Prolog for sharing. In *Proc. of the International Workshop on User-Oriented Logic Programming (IULP 2015)*, 2015.

[22] Armin Wolf. Adaptive constraint handling with CHR in Java. In *International Conference on Principles and Practice of Constraint Programming*, pages 256–270. Springer, 2001.

[23] Pieter Wuille, Tom Schrijvers, and Bart Demoen. CCHR: the fastest CHR Implementation, in C. In *Proc. 4th Workshop on Constraint Handling Rules (CHR07)*, pages 123–137, 2007.