ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering
and Computer Science
Institute of Software Engineering
and Compiler Construction

CHR.js: Compiling Constraint Handling Rules to JavaScript

Master Thesis at the University of Ulm

Submitted by:
Falco Nogatz
falco.nogatz@uni-ulm.de

Reviewer:
Prof. Dr. Thom Frühwirth
Prof. Dr. Enno Ohlebusch

Consultant:
Daniel Gall

2015

Version October 12, 2015

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

Constraint Handling Rules (CHR) is a high-level programming language extension which introduces declarative multiset semantics. Although originally designed in the early 1990s, the number of implementations is still small. While there are adaptions for popular imperative programming languages such as C and Java, its most popular host language is Prolog. As a result, the dissemination of CHR is currently restricted almost entirely to research community.

In this thesis we present an implementation of CHR in JavaScript. By embedding it into the dominating web programming language which recently got adopted for server-side frameworks as well, we open this declarative approach to a broad range of developers and new use cases.

The embedding of CHR in JavaScript gives the chance to easily create applications with CHR in combination with front-end functions. As a result, we created a web-interface to explore the evaluation of Constraint Handling Rules interactively This visual CHR tracer is based on the created interpreter called *CHR.js*, whose main target is the full extensibility of CHR. To remain competitive with existing CHR implementations, we created a transpiler for precompilation of CHR.js source code.

Contents

1

Introduction

Any application that can be written in JavaScript, will eventually be written in JavaScript.

— Jeff Atwood, Founder of Stack Overflow[1]

The first chapter introduces the research presented in this thesis and its general structure. We present the motivation to port Constraint Handling Rules to JavaScript, define the implementation goals and give a short introduction to the used methodology and targeted environments.

[1] Coding Horror: The Principle of Least Power, `http://blog.codinghorror.com/the-principle-of-least-power/` (2007)

1.1. Motivation

Since its creation in the early 1990s, Constraint Handling Rules (CHR) has been evolved from a language extension specifying constraint-solvers to a general-purpose programming language. Today's applications of CHR cover many and varied areas in of computer science. For example it is used for spatial and temporal reasoning, software testing and compiler construction. It combines techniques like forward and backward chaining, bottom-up and top-down evaluation, integrity constraints, tabling and memorization. Its general-purpose approach is the reason, Thom Frühwirth, the creator of Constraint Handling Rules, often refers to it as "the lingua franca of computer science" [Frü09b].

While CHR combines a wide range of techniques, to obtain this title it lacks of one important requirement: propagation. Although there are implementations for most of the popular programming languages, including Java, C and Haskell, its most prominent hosting language remains Prolog restricting its current employment almost entirely to the research community.

Measured against dissemination and popularity, the opposite applies for JavaScript: In the last years it made the step from the predominant language of the web to a general-purpose programming language. Today it is certainly the most favored programming language at GitHub, the largest Git repository hosting service in the world. More than one third of the popular applications on GitHub are written in or related to JavaScript [BVHC15]. Douglas Crockford, who developed the JavaScript Object Notation (JSON), once stated that that every personal computer in the world had at least one JavaScript interpreter installed on it and in active use[Cro01]. It is very likely that this assumption is accurate.

By porting CHR to JavaScript, we can benefit from the broad propagation of runtime environments. Whereas currently the runtime system of the host language has to be installed first (for example SWI-Prolog or Java), a CHR implementation written entirely in JavaScript can be used in any browser without modifications. In addition CHR itself is very suitable to the typical event-based architecture of JavaScript, which can be easily described in a declarative way by Constraint Handling Rules.

1.2. Related Work

There are several implementations of CHR in different languages. The particular host language is referred to in parentheses, therefore we speak for example of CHR(Prolog), CHR(Java) and CHR(C) systems for the host languages Prolog, Java and C. We aim to create a CHR(JavaScript) implementation, which does not exist so far.

1.2.1. Existing CHR systems

When developed, CHR was set up as a language extension for Prolog. A historical overview of CHR implementations can be found in [Sch05]. As of today, Prolog is still the predominant host language for CHR, with the KU Leuven CHR [SD04] as the state-of-the-art CHR system [Sne15]. It is available for all major Prolog implementations, including SWI-Prolog and SICStus Prolog.

The first formalization of the compilation of CHR in Prolog in 1998 [HF98] has since then been the basis for implementations and optimizations in other languages like HAL [HDLBSD05, Duc05].

Recently CHR has been ported to imperative programming languages as well. A basic compilation scheme for CHR, which can be used for various programming paradigms, is discussed in [Sch05]. In [VWWSD08], the different conceptual and technical difficulties encountered when embedding CHR into an imperative host language, are presented.

Today there are two major CHR implementations in imperative languages: *CCHR* [WSD07], a CHR(C) system, is today's fastest CHR implementation. Its CHR(Java) counterpart, called *JCHR* [VWSD], is based on *JACK* [AKSS02], a general constraint library for Java.

1.2.2. Logic Programming in JavaScript

While there is currently no CHR implementation for JavaScript, there are at present several Prolog implementations in JavaScript for instance Yield Prolog [Tho13] and

JScriptLog [Hol09]. Pengines [LW14], a recent approach to use Prolog on websites, communicates over HTTP with a server running a SWI-Prolog instance and would likewise be a possibility to call CHR constraints in the browser. Nevertheless it does not solve the original problem as a server accepting these Prolog remote procedure calls persists necessary. Therefore Pengines is comparable to WebCHR [Kae07], which also dispatches calls on CHR constraints via HTTP to a backend server, in this particular case running SICStus Prolog 4.

J. F. Morales et al. presented a lightweight compiler of (constraint) logic programming languages to JavaScript in 2012 [MHCH12]. It does not support CHR and is based on a special module system which requires implementation of (C)LP rules in a JavaScript dialect. In contrast our aim is to embed CHR in existing JavaScript source code and either execute on runtime or pre-compile to plain old JavaScript.

1.3. Implementation Goals

Our contribution in this thesis is the implementation of a CHR(JavaScript) system. It orientates at the following implementation goals:

Resembling to existing CHR systems The syntax used by the Javascript adaption of CHR should be familiar for CHR- as well as JavaScript-experienced developers. The user should be able to use regular JavaScript variables and expressions. On the other hand it may be possible to easily adapt existing CHR source code, written for example for CHR(Prolog) systems, by only changing syntactic elements specific for the host language.

Support for different runtime environments Because today every browser ships with a JavaScript interpreter, we want to ensure the suitability of our implementation for every modern browser. In addition, the server framework node.js shall be supported.

Extensible tracing options Being executable in the web, CHR can be opened to the public. We want to improve the understanding of CHR programs by providing

various tracing options. With these tracing options it should be possible to create a web application to interactively trace and debug CHR programs in the browser.

Efficiency Although the previous implementation goals have first priority, we want to create an efficient CHR system. Unlike existing CHR implementations, JavaScript is an interpreted programming language and in general just-in-time compiled, so it might not be possible to compete with C or Java implementations.

Supplementary to the previous points our implementation is licensed as open source. The GitHub repositories are referenced in Appendix A.

1.4. Scope of this Thesis

Because JavaScript is a multi-paradigm programming language, influenced by both traditional imperative programming languages like C as well as functional languages, we examine how existing compilation approaches for these programming paradigms are suitable for JavaScript. Due to its special execution cycle as an event loop, we present optimizations particularly suitable for JavaScript.

To support the full semantics of CHR as well as extensible tracing options, we introduce a compilation scheme for a CHR interpreter. This JavaScript module, called *CHR.js*, uses just-in-time (JIT) compilation of Constraint Handling Rules. Unlike existing CHR systems, this allows to specify and adapt rules dynamically at execution time. This is a necessary requirement to create a feature-rich web-based tracer.

The big disadvantage of a JIT compilation pattern is obvious: The expressiveness of dynamic rule declarations comes with a trade-off in efficiency and execution time. So our contribution consists of a second compilation scheme, which is more comparable to existing CHR systems in C and Java. The resulting JavaScript module called *babel-plugin-chr* pre-compiles CHR.js source code to native JavaScript, but does not support all of the features of the JIT version. Especially the support for tracing options and parallelism have been dropped for better single-thread performance.

To benchmark this pre-compiled CHR application, we present a benchmark suite called *CHR-Benchmarks*. It compares existing CHR implementations and supports the most popular CHR systems of SWI-Prolog, CCHR and JCHR.

As a fourth software component we present a web-based CHR tracer, online available at `http://chrjs.net/`. It is an example application of the *CHR.js* interpreter.

1.5. Road Map

This thesis is divided in several parts. We start with an introduction in the two relevant programming languages Constraint Handling Rules in Chapter 2 and JavaScript in Chapter 3. There we present the execution cycle of JavaScript and explain and present multiple means to handle asynchronous functions. These two chapters are the basis for the general compilation scheme of the *CHR.js* interpreter.

In Chapter 4 we start with the language definition of CHR embedded in JavaScript and present the general compilation scheme for the *CHR.js* interpreter. To demonstrate the expressiveness of the *CHR.js* module, we outline the interactive and web-based tracer for CHR `http://chrjs.net/` in Chapter 5.

We continue in Chapter 6 with an improved but synchronous compilation scheme which is used in the *babel-plugin-chr* pre-compiler. Here we also present the applied optimizations to increase the JavaScript performance specifically for the V8 JavaScript Engine.

Finally the benchmark suite for existing CHR systems is presented. We compare the execution time of ahead-of-time compiled *CHR.js* programs with equivalent in Prolog, Java and C host languages. To get a relation to the languages' native implementations, we also examine programs written just in JavaScript and C.

In Chapter 7 we discuss the two compilation schemes in general and consider the question, if they could be combined into a single module.

2

Background: Constraint Handling Rules

> *CHR has the potential to become a lingua franca, a hub which collects and dispenses research efforts from and to the various related fields.*
>
> — Jon Sneyers, Co-Author of the KU Leuven CHR System[1]

This chapter introduces the examined programming language CHR. After a presentation of the syntax and semantics of CHR, we introduce the compilation schemes of existing CHR systems. This is the basis for the definition of the *CHR.js* language extension and its two underlying compilation schemes presented in Chapter 4.

We restrict ourselves to a short introduction of the syntax and semantics of CHR, as they are the basis for the compilation process. For more detailed introductions to CHR we

[1] In [Sne09]

refer to [Frü98, SVWSDK10, Frü09a]. The material in this chapter consists of the usual definitions as in the above literature.

2.1. Introduction

Constraint Handling Rules, which is usually referenced to as CHR, was first defined by Thom Frühwirth in 1991 [Frü91]. Originally created to implement new constraint solvers in a high level programming language, it has its origins in the logic and constraint programming. Today CHR is used as a general-purpose programming language with a wide range of applications, though still be used only as a language extension to its so-called host language.

As the name “Constraint Handling Rules” suggests, CHR adds a method to specify a set of rules that rewrites multisets of constraints into simpler ones until a final state is reached, where no rule can be applied. Unlike its most popular host language Prolog, CHR is a committed-choice language and consists in general of multi-headed and guarded rules.

2.2. Syntax

Before we go into more detail about the syntax and semantics of CHR, we want to define the requirements to its host language.

2.2.1. Host Language Requirements

We assume that every host language has at least one data type with the following two operations:

1. the creation of a new, unique value and
2. equality testing based on two given identifiers.

Functions, operations and identifiers defined in the host language are so-called *built-in constraints*. The host language must specify at least the basic constraints `true` and `fail` with the following meanings: `true` is a constraint which is trivially satisfied, `fail` is the contradictory constraint.

2.2.2. Constraints

In contrast to built-in constraints, *CHR constraints* (or *user-defined constraints* or *constraints* for short) are defined by their occurrence in CHR rules. A CHR constraint is denoted by its *functor*, a pair of the constraint's name and its arity, i.e. the number of its arguments. Names do not have to be unique, while constraints with the same functor generally represent the same entity. As an example, the functors of the required basics constraints as stated above are `true/0` and `fail/0`. The requirements of the constraint's name are dependent on the host language, in general lower-case identifiers are preferred.

2.2.3. Constraint Store

All constraints which are known to be true will be placed in a so-called *constraint store*. For example, in a trivial CHR program with no rule at all, every given constraint will be placed in the constraint store as seen to be true.

Overall the remaining content of the constraint store represents the result of the given CHR program.

2.2.4. Program and Rules

By defining rules it is possible to manipulate the content of the constraint store. A CHR *program* is defined by a finite number of CHR rules in a concrete order. Although already included in the acronym ”CHR“, we will refer to a single rule in the program as ”CHR rule“ in the following.

Each rule can be a *simplification*, *propagation* or *simpagation* rule. It contains an optional name, a head, an optional guard and a body. The head is a conjunction of CHR constraints, the guard a conjunction of built-ins, while the body can be a conjunction of CHR as well as built-in constraints. The syntax and meaning of the execution of each rule depends on its type.

A rule is optionally preceded by $Name$ @ where $Name$ is an identifier. The name of a rule does not effect the execution of the CHR program at all and is only used for tracing and debugging purposes, however no two rules may have the same name.

In the following we denote constraints in the head of a rule in the form H_i; they are called *head constraints*. Constraints of the guard are called *guard constraints* and denoted by G_i, *body constraints* as B_i. The number $i > 0$ is used to reference a particular constraint. Every rule is applied if and only if all guard constraints $(G_1, \ldots, G_n)$ are satisfied. If the optional guard is omitted, it is considered to be `true/0`.

Simplification Rule

The syntax of a *simplification rule* is:

$$Name \quad @ \quad H_1, \ldots, H_n \quad \Longleftrightarrow \quad G_1, \ldots, G_m \quad | \quad B_1, \ldots, B_l$$

With the simplification rule we can specify replacements in the constraint store: The constraints $(H_1, \ldots, H_n)$ are removed from the constraint store, the constraints $(B_1, \ldots, B_l)$ of the body are added.

Propagation Rule

The syntax of a *propagation rule* is:

$$Name \quad @ \quad H_1, \ldots, H_n \quad \Longrightarrow \quad G_1, \ldots, G_m \quad | \quad B_1, \ldots, B_l$$

In opposite to the simplification rule, the head constraints $(H_1, \ldots, H_n)$ stay in the constraint store while the constraints $(B_1, \ldots, B_l)$ of the body are added. Because in this way the constraint store is just supplemented, we define, that a propagation rule fires only a single time for a given ordered set of head constraints $(H_1, \ldots, H_n)$.

Simpagation Rule

The syntax of a *simpagation rule* is:

$$Name \quad @ \quad H_1, \ldots, H_k \quad \backslash \quad H_{k+1}, \ldots, H_n \quad \Longleftrightarrow \quad G_1, \ldots, G_m \quad | \quad B_1, \ldots, B_l$$

This rule is a mix of the simplification and propagation rule: The head constraints $(H_1, \ldots, H_k)$ remain in the constraint store; they are called *kept constraints*. The head constraints $(H_{k+1}, \ldots, H_n)$ of a simpagation rule are removed from the constraint store and therefore called *removed constraints*.

Generalized Simpagation Form

In the simpagation rule, neither the set of kept nor removed constraints is allowed to be empty (i.e. $1 \leq k \leq n$). However it is obvious that the simplification rule is only a special case of a simpagation rule with $k = 0$. The same applies for the propagation rule by using $k = n$. For simplicity we therefore define a *generalized simpagation form*:

$$Name \quad @ \quad H_1, \ldots, H_k \quad \backslash \quad H_{k+1}, \ldots, H_n \quad \Longleftrightarrow \quad G_1, \ldots, G_m \quad | \quad B_1, \ldots, B_l$$

with $0 \leq k \leq n$ and $n \geq 1$

$(H_1, \ldots, H_n)$ are head constraints, with at least the set of kept or removed constraints is not empty. Although no CHR system allows the specification of rules in the generalized simpagation form, we will refer to it in the description of the compilation process in Chapter 4.

2.3. Semantics

The traditional way to prove characteristics of CHR programs is to conclude from its logical semantics. Although it is possible to translate each CHR rules into an equivalent logical formula [Bou04], this declarative semantics is lacking definitions on how to handle non-determinism. In this section, we therefore concentrate on the operational semantics as defined in [Abd97, AF98]. The *refined operational semantics* as presented

in Section 2.3.2 is applied in every major CHR system and therefore the basis for our implementation in equal.

2.3.1. The Theoretical Operational Semantics ω_t

The operational semantics ω_t of CHR are based on a transition system. A single state represents a moment in the execution of CHR, the transitions between correspond to the applied CHR rules.

The state transition system presented in this section is based on [Sch05], an adopted version of [DSDLBH04].

Execution State

The *execution state* is a 4-tuple $\sigma = < G, S, B, T >$ with the following components:

- *Goal* G, a conjunction of both user-defined and built-in constraints;
- *Store* S, a conjunction of user-defined constraints that can be matched with constraints in the rule heads;
- *Built-ins* B, a conjunction of built-in constraints that have been passed to the host language;
- *Propagation history* T, a set of sequences that represents the already applied rules with their related constraints.

The propagation history is used to prevent trivial non-termination of propagation rules as described in Section 2.2.4. We assume every rule to have a (generated) unique name. Similar applies for every user-defined constraint c: We denote every CHR constraint c with a unique integer i and denote $c\#i$ for this particular constraint. In this way, it is possible to save the applied rules with the identifiers of the matched constraints in the propagation history T.

Given an initial goal G, the program execution starts with the initial state $< G, \emptyset, true/0, \emptyset >$.

State Transition Rules

For the transition rules we cite [Sch05] in Figure 2.1. The symbol $\uplus$ defines the multiset union, meaning constraints of the same form remain multiple times in the resulting set.

Figure 2.1.: The transition rules of the operational semantics (based on [Sch05], p. 20)

1. Solve:
$\langle \{c\} \uplus G, S, B, T \rangle_n \rightarrowtail_{solve} \langle G, S, c \wedge B, T \rangle_n$

2. Introduce:
$\langle \{c\} \uplus G, S, B, T \rangle_n \rightarrowtail_{introduce} \langle G, \{c\#n\} \uplus S, B, T \rangle_{(n+1)}$

3. Apply:
$\langle G, H_1 \uplus H_2 \uplus S, B, T \rangle_n \rightarrowtail_{apply} \langle C \uplus G, H_1 \uplus S, \theta \wedge B, T' \rangle_n$
where there exists a rule r in the program P of the form

$$r \mathbin{@} H_1' \setminus H_2' \iff g \mid C$$

and T' is the supplemented propagation history.

In the last transition rule we denote the substitution θ to make the rule heads H_1' and H_2' match existing constraints in the store $H_1 \uplus H_2 \uplus S$. The matching algorithm depends on the host language (where this substitution is added to as requirement $\theta \wedge B$). In logic programming languages this is simply unification. In JavaScript we use the *destructuring* mechanism, as specified in the JavaScript introduction in Section 3.5.5.

The **solve** transition rule resolves a built-in constraint by adding it to the built-ins store B. The **introduce** transition rule adds a new constraint c to the CHR constraint store S. In the last rule, a CHR rule given in generalized simpagation form is applied. Therefore, the head constraints are matched with existing ones in the constraint store as described before. Before applying the CHR rule in this **apply** transition, the guard g must be proven. If this transition applies for a CHR rule r, this rule r is *fired*.

Starting from the initial state, these transitions are applied until either no more transitions are applicable (*successfull derivation*) or the built-ins B evaluate to `fail/0` (*failed derivation*).

This sequence of transitions is not deterministic at all: If multiple transition rules can apply for the current execution state, the order of their applications is not specified in the operational semantics ω_t. Due to this non-determinism, neither the result state of a program is well-defined nor the property if the program terminates at all.

2.3.2. The Refined Operational Semantics ω_r

In order to achieve a more deterministic execution, the *refined operational semantics* ω_r has been proposed in 2004 [DSDLBH04]. Today all major CHR systems implement this refined operational semantics.

The overall idea of the refined operational semantics orientates at the application of CHR in stack-based programming languages that CHR is compiled to: Constraints are treated as procedure calls, meaning that once a new constraint is added to the store, all matching rules for this constraint will be applied. If such a rule adds another constraint, we pause the execution of the origin constraint and call all the matching rules according to the new constraint. As soon as this is completed, we resume with the first constraint. The overall execution stops as soon as all appropriate rules are fired or a failed derivation has occurred.

In this way we define an order for the applications of the **apply** transition. However there is a second source for non-determinism: Being a multiset, the choice of the next goal in G is not defined. The refined operational semantics ω_t therefore defines G as an ordered sequence of goals, so we are no longer free to choose any constraint from G to either **solve** or **introduce** into the store.

The execution state of the refined operational semantics is very similar to the one of ω_t: $\sigma = < A, S, B, T >$, with S, B and T with respect to ω_t. Unlike in the previously presented execution state, A is a sequence of goals called *execution stack*, with a strict order in which the top-most constraint is called the *active constraint*. Because ω_r is

an instance of the ω_t semantics, there is a mapping between the execution states and transition rules of both semantics, as presented in [DSDLBH04].

Although the refined operational semantics clarify some execution orders, it is still non-deterministic [Sch05]. One source of non-determinism is that we did not specify an order to get partner constraints, if more than one possibility exists. *Partner constraints* are these constraints in a rule's head that must be present, to make a rule applicable in joint with the active constraint.

While the open sources for non-determinism could be removed by specifying a more restricted operational semantics, they are often used by compilers to apply optimizations [HDLBSD05]. The non-determinisms also allow the usage of various data structures, for example for a highly efficient constraint store.

2.3.3. Existing Extensions and Variations

The three basic rule types presented in Section 2.2 are part of every CHR system. In addition extensions and variations of the basic CHR syntax were proposed. In general their aim is to modify or supplement the rules in order to change the execution cycle and semantics of the CHR program. A comprehensive overview of existing extensions and variations can be found in [SVWSDK10].

Because the aim of this work is to implement a basic compilation of CHR in JavaScript, we do not support any language extension so far. Because of its notability for an optimized compilation in future, we only want to give a short introduction into so-called *pragmas*.

Rule Pragmas

Pragmas are a method for the user to modify the compilation of the CHR program by special rule annotations. A pragma allows to explicitly set restrictions to fire a given rule. Unlike built-in constraints of the guard, their fulfilment is not proved at runtime but affects the runtime code generated by the compiler. They are supported by the major CHR(Prolog) systems, for example SICStus Prolog [CF14].

Historically there are two pragmas: `passive` and `already_in_head`. The latter one is a remnant of the original syntax of Frühwirth in 1991 [Frü91], when his proposition was called “Simplification Rules” and included only simplification and propagation, but no simpagation rule. With the introduction of explicit simpagation rule the `already_in_head` pragma became obsolete, but is still supported by some CHR systems.

Listing 2.1: Example for `passive` pragma

```
1 r1  @  a , b       ==>  c
2 r2  @  a , b # Id  ==>  d pragma passive(Id)
```

With the help of the `passive` pragma it is possible to explicitly define the entry points of a rule. Considering the example in Listing 2.1, the rule `r1` would generate an entry point for both constraints `a/0` and `b/0`. In opposite, rule `r2` generates only an entry point for `a/0`, because the `passive` pragma states explicitly that `b/0` can not be active in this rule. If the constraint `b/0` is added to a constraint store which already contains `a/0`, only rule `r1` would fire.

2.4. Example Program: `gcd/1`

In this section we give an example of a typical CHR program: The calculation of the greatest common divisor (GCD) using the subtraction-based Euclidean algorithm. It will be used as the reference example throughout the entire thesis.

Listing 2.2: Euclidean algorithm as pseudo-code

```
1 function gcd(a, b)
2   while b > 0
3     if a > b
4       a := a - b
5     else
6       b := b - a
7   return a
8 end
```

Consider the substraction-based, non-recursive Euclidean algorithm given as pseudo-code in Listing 2.2. It takes two positive integers `a` and `b` and returns its GCD.

To give an appropriate definition of this algorithm in CHR, we represent every input number as `gcd/1` constraint. In order to calculate the GCD of the numbers `6` and `9`, the goal would therefore be the set `gcd(6), gcd(9)`. An advantage of this representation is obvious: The CHR program can easily be used to calculate the GCD of more than two input values. The two rules to calculate the GCD in CHR by using the subtraction-based Euclidean algorithm are presented in Listing 2.3. The calculated result is the single `gcd/1` constraint that remains in the constraint store.

Listing 2.3: Euclidean algorithm in CHR

```
1 gcd1  @  gcd(0)           <=>  true.
2 gcd2  @  gcd(N) \ gcd(M) <=>  0 < N, N <= M | gcd(M-N).
```

The first simplification rule `gcd1` removes all `gcd/1` constraints with an argument of `0`. This rule is similar to the loop condition specified in line 2 of Listing 2.2.

The second simpagation rule `gcd2` can be applied for every two `gcd/1` constraints with positive arguments (which we assume), because the guard `0 < N, N <= M` will always be true for one ordering of the active with its partner constraint. With the help of the rule `gcd2` the `gcd/1` arguments will be subtracted by each other until one is zero and therefore is removed by the first rule `gcd1`.

A detailed successful derivation of the goal `gcd(6), gcd(9)` under both the theoretical operational semantics ω_t as well as the refined operational semantics ω_r can be found in [Sch05].

2.5. Compiling CHR

Several approaches to compile CHR have already appeared in the literature. In this section we present the basic compilation schemes of CHR depending on the targeted host language. This will be later used to reason implementation details of our CHR.js contribution and compare it to existing CHR systems.

2.5.1. Head Normal Form

Before we go into existing CHR systems and their compilation schemes, we want to present a transformation step that is used by all major CHR systems: bringing each rule in a so-called *head normal form (HNF)* [VWWSD08].

The HNF is special instance of the generalized simpagation form. It has the following generic form:

$$\begin{aligned} Name \quad @ \quad & c_1^{[j_1]}(X_{1,1},\ldots,X_{1,a_1}),\ \ldots,\ c_k^{[j_k]}(X_{k,1},\ldots,X_{k,a_k}) \quad \backslash \\ & c_{k+1}^{[j_{k+1}]}(X_{k+1,1},\ldots,X_{k+1,a_{k+1}}),\ \ldots,\ c_n^{[j_n]}(X_{n,1},\ldots,X_{n,a_n}) \quad \Longleftrightarrow \\ & G_1,\ \ldots,\ G_m \quad | \quad B_1,\ \ldots,\ B_l \end{aligned}$$

Similar to the generalized simpagation form presented in section 2.2.4, the constraints in the rule head are numbered from left to right. As stated before, for a simplification rule is $k = 0$, a propagation rule is represented by $k = n$.

In addition, we define the *occurrence number* j_i of a constraint in $c_i^{[j_i]}$. This is the j_i'th occurrence of the constraint c_i in the CHR program. The occurrences are numbered from top to bottom in order of rules and from right to left in a particular rule. In this way, removed occurrences of a constraint that appears in both the kept and removed heads are applied first, which is conform to the refined operational semantics ω_r of section 2.3.2.

Another requirement of the head normal form is the uniqueness of the variables in the rule's head. If a variable occurs more than once in the rule's head, we introduce fresh variables and make the equalities explicit in the guard of the rule. Moreover, even finite arguments are replaced by variables and bound in the rule's guard.

The Listing 2.4 shows the normalized version of the `gcd/1` example program of section 2.4.

Listing 2.4: Normalized Version of the Euclidean algorithm of Listing 2.3

```
1 gcd1  @  gcd[1](P)              <=> P = 0 | true.
2 gcd2  @  gcd[3](N) \ gcd[2](M) <=> 0 < N, N <= M | gcd(M-N).
```

This head normal form is used as input of the compilation schemes presented in the next sections. Because we only want to introduce the basic compilation idea of existing CHR systems, we assume basic knowledge of their targeted host languages.

2.5.2. CHR in Logic Programming Languages

Due to its origin as a language extension to built constraint solvers, the implementation of CHR in logic programming languages is well covered in literature. Since the reference implementation in SICStus Prolog in 1999 [HF99], several optimization techniques have been presented for the compilation into logic host languages. The implicit execution stack of Prolog maps very well to the ordered execution stack A 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 therefore executed before any other added constraint.

Consider a rule given in HNF as presented in section 2.5.1. To represent constraints handle their occurences and the CHR properties, we introduce the following Prolog predicates:

- c_i**/**a_k, e.g. `gcd/1`
  Predicate to call a constraint, that means to create the specified constraint, add it to the constraint store and apply all appropriate rules.
- **insert_in_store_**c_i**/**$(a_k + 1)$, e.g. `insert_in_store_gcd/2`
  This predicate is used to add the given constraint to the store. This generates a unique identifier which is returned as the last component, which is why it is of arity $a_k + 1$. This identifier is passed to the following predicates as an additional argument.
- c_i**_occurences/**$(a_k + 1)$, e.g. `gcd_occurrences/2`
  A predicate which is created by the compiler to apply all occurrences j_i of the constraint c_i/a_k.
- c_i**_occurrence_**j_i**/**$(a_k + 1)$, e.g. `gcd_occurrence_1/2`
  This predicate implements the concrete handler for the j_i'th occurrence of the constraint c_i.

Although the predicates `insert_in_store_`c_i and c_i`_occurrences` might be defined multiple times, if a constraint is defined with multiple arities, they are unambiguous in Prolog because of their distinct functors.

The general compilation scheme for a constraint c_i/a_k is shown in Listing 2.5.

Listing 2.5: Generated Prolog code for the Euclidean Algorithm

```
1  gcd(I) :-
2    insert_in_store_gcd(I,ID),
3    gcd_occurrences(I,ID).
4
5  insert_in_store_gcd(I,ID) :-
6    chr_store_add(gcd(I),ID).
7
8  gcd_occurrences(I,ID) :-
9    gcd_occurrence_1(I,ID),
10   gcd_occurrence_2(I,ID),
11   gcd_occurrence_3(I,ID).
```

The c_i**_occurrence_**$j_i$$/(a_k + 1)$ clauses will loop through all partner constraints for the constraint c_i with respect to the given rule. It also proves the guard and applies changes to the constraint store. The three generated occurrence predicates are presented in Listings 2.6, 2.7 and 2.8.

Listing 2.6: Generated Prolog code for the `gcd/1` occurrence in rule `gcd1`

```
1  gcd_occurrence_1(I,ID) :-
2    % check that the constraint has not been removed by
3    %   a previous occurence
4    chr_constraint_alive(ID),
5
6    % check that the rule has not already been applied
7    chr_not_in_history(gcd,1,[ID]),
8
```

```
9    % prove the guard of the HNF
10   I == 0,
11
12   chr_add_to_history(gcd,1,ID),
13   chr_kill_constraint(ID).
```

Listing 2.7: Generated Prolog code for the right `gcd/1` occurrence in rule `gcd2`

```
1  gcd_occurrence_2(I,ID) :-
2    chr_constraint_alive(ID),
3
4    % find partner constraints
5    chr_lookup(gcd(N),ID_1),
6
7    chr_not_in_history(gcd,2,[ID_1,ID]),
8    0 < N,
9    N <= I,
10   chr_add_to_history(gcd,2,[ID_1,ID]),
11   chr_kill_constraint(ID),
12   K is I-N,
13   gcd(K).
```

Listing 2.8: Generated Prolog code for the left `gcd/1` occurrence in rule `gcd2`

```
1  gcd_occurrence_3(I,ID) :-
2    chr_constraint_alive(ID),
3    chr_lookup(gcd(M),ID_2),
4    chr_not_in_history(gcd,3,[ID,ID_2]),
5    0 < ID,
6    ID <= M,
7    chr_add_to_history(gcd,3,[ID,ID_2]),
8    chr_kill_constraint(ID_2),
9    K is M-I,
```

```
10  gcd(K).
```

The predicates with a `chr_` prefix are used to handle the runtime components of a CHR program, namely the constraint store and propagation history, and implementation details. Their signatures are as follows:

- `chr_store_add(Constraint,ID)`
- `chr_lookup(Constraint,ID)`
- `chr_constraint_alive(ID)`
- `chr_kill_constraint(ID)`
- `chr_not_in_history(Constraint_Name,Occurrence_Number,IDs)`
- `chr_add_to_history(Constraint_Name,Occurrence_Number,IDs)`

It is obvious that the basic compilation scheme presented in Listing 2.5 can be optimized at various places. For example it is not necessary to test if the given constraint is still alive in line 17, because this is the very first occurrence of for `gcd/1`. More optimization techniques can be found in [HF99], [HDLBSD05] and [Sch05].

2.5.3. CHR in Imperative Programming Languages

There are two major CHR systems written in imperative programming languages:

- JCHR [VWSD], a CHR system for Java and
- CCHR [WSD07], a CHR system for C, which is currently the fastest implementation of CHR.

Both implementations are discussed in [VWWSD08], where a general compilation scheme for imperative programming languages is introduced. This work also presents the challenges of porting CHR, which was originally designed to work with logic programming languages, to C and Java. Some language constructs, which are very common for Prolog users, are unlikely in these languages:

- **Logical Variables**
  Many traditional CHR examples are based on logical variables. They allow the specification of constraints with not-yet-known properties. In imperative programming languages, variables can be reassigned multiple times. In opposite, Java and C are static typed, that means we can not arbitrarily assign values but have to declare the variable's type at compilation time.

- **Pattern Matching**
  In CHR a given active constraint is used as a pattern and gets compared with any head constraint to find appropriate rules. This one-way pattern matching is basically implemented with the help of unification (`=/2` in Prolog). Imperative programming languages generally do not implement a comparable mechanism. Instead, they support type-based comparisons.

- **Search**
  The search for appropriate partner constraints (c.f. `chr_lookup/2` in Section 2.5.2) can be easily implemented in Prolog by using backtracking. Imperative programming languages have to use optimized data structures to support fast queries against the constraint store and propagation history.

Before we go into detail about the compilation process for imperative languages, we introduce the adapted CHR syntax used by JCHR and CCHR. These examples will later be used to explain a similar adoption of CHR for JavaScript.

Syntax of JCHR

Similar to our CHR.js implementation goals, the aim of JCHR and CCHR was to create a CHR system that is familiar to both Java respectively C and CHR developers. Therefore the syntax established by the CHR(Prolog) systems of SICStus Prolog and SICStus Prolog have been slightly adapted to fit into the targeted host languages.

Listing 2.9 shows the specification of the `gcd/1` constraint handler in JCHR. The CHR rules are specified in a special `rules {...}` block. Their syntax is very similar to Prolog.

Listing 2.9: Euclidean algorithm `gcd/1` handler in JCHR

```
1 public handler gcd {
2   public constraint gcd(long);
3
4   rules {
5     gcd(0) <=> true.
6     gcd(N) \ gcd(M) <=>
7       0 < N, N <= M | gcd(LongUtil.sub(M, N)).
8   }
9 }
```

When this code is compiled with K.U. Leuven JCHR, it creates a new class `GcdHandler` that can be used in a normal Java program. An example application of the `GcdHandler` is shown in Listing 2.10.

To use JCHR, the user has to define the constraint handler in a separate file first. This file gets compiled with JCHR, which creates the related `*Handler` class that can be used in any Java program. The `*Handler` class provides special methods to add constraints or query the constraint store.

Listing 2.10: Example usage of the compiled `GcdHandler` of Listing 2.9

```
1  import java.util.Collection;
2
3  public class Example {
4    public static void main(String[] args) {
5      GcdHandler handler = new GcdHandler();
6
7      handler.tellGcd(6);
8      handler.tellGcd(9);
9
10     // lookup all gcd constraints in the store
11     Collection<GcdHandler.GcdConstraint> gcds =
12       handler.getGcdConstraints();
```

```
13
14      // there should remain a single GCD
15      assert gcds.size() == 1;
16    }
17  }
```

Syntax of CCHR

CCHR has a very similar approach: The CHR rules are specified in a special `cchr {...}` block. While in Java a file per class is required, we can embed the CHR rules directly to the C file. The `gcd/1` example adapted for CCHR is used shown in Listing 2.11. The `main` function calculates the GCD of `6` and `9`.

Listing 2.11: Euclidean algorithm `gcd/1` in CCHR

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  #include "gcd_cchr.h"
6
7  cchr {
8    constraint gcd(uint64_t);
9
10   gcd1 @ gcd(0ULL) <=> true;
11   gcd2 @ gcd(N) \ gcd(M) <=>
12     0 < N, N <= M | uint64_t K=M-N, gcd(K);
13 }
14
15 int main(int argc, char **argv) {
16   uint64_t a1 = 6;
17   uint64_t a2 = 9;
```

```
18
19    cchr_runtime_init();
20    cchr_add_gcd_1(a1);
21    cchr_add_gcd_1(a2);
22    cchr_runtime_free();
23    return 0;
24  }
```

The general syntax of the CHR rules remains similar to Prolog. The only significant change is that semicolons are used as delimiters between multiple rules. The constraint solver can be used with functions prefixed by `cchr_`, which are comparable to the methods provided by the JCHR-generated `*Handler` class.

Basic Compilation Scheme

The compilation scheme for logic programming languages presented in Section 2.5.2 can be adopted to a procedural computation style by replacing Prolog predicates with methods. We use imperative pseudo-code to introduce a basic compilation scheme based on [VWWSD08]. Although this scheme is conform to the refined operational semantics ω_r, it is not very efficient. Optimization techniques are presented in the same place, but dependent on the particular target language.

Similar to the compilation in logic programming languages, we create a method for every CHR constraint. They again call methods for every occurcence of the active constraint. This scheme is shown in Listing 2.12.

Listing 2.12: Basic compilation scheme in imperative languages

```
1 procedure c_i(X_1,...,X_{a_k})
2   ID = chr_create_constraint(c_i,[X_1,...,X_{a_k}])
3   chr_store_add(ID)
4   c_i_occurences(c_i(X_1,...,X_{a_k}),ID)
5 end
```

```
6
7  procedure c_i_occurrences(c_i(X_1,...,X_{a_k}),ID)
8    c_i_occurrence_1(X_1,...,X_{a_k},ID)
9    ...
10   c_i_occurrence_j_i(X_1,...,X_{a_k},ID)
11 end
```

Listing 2.13 shows the compilation scheme for the c_i_occurrence_j_i method, representing the j_i'th occurrence of the constraint c_i in the CHR program specified in HNF according to Section 2.5.1.

Listing 2.13: Compilation of the j_i'th occurrence of the constraint c_i

```
1  procedure c_i_occurrence_j_i(c_i(X_{i,1},...,X_{i,a_k}),ID_i)
2    // find partner constraints
3    foreach (c_1(X_{1,1},...,X_{1,a_1})#ID_1 in lookup(c_1)
4      ⋱
5        foreach (c_{i-1}(X_{i-1,1},...,X_{i-1,a_{i-1}})#ID_{i-1} in lookup(c_{i-1})
6          foreach (c_{i+1}(X_{i+1,1},...,X_{i+1,a_{i+1}})#ID_{i+1} in lookup(c_{i+1})
7            ⋱
8              foreach (c_n(X_{n,1},...,X_{n,a_n})#ID_n in lookup(c_n)
9                if all_different([ID_1,...,ID_{i-1},ID_i,ID_{i+1},...,ID_n])
10                 if chr_all_alive([ID_1,...,ID_{i-1},ID_i,ID_{i+1},...,ID_n])
11                   if G_1 and ... and G_m
12                     if chr_not_in_history(c_1,j_i,[ID_1,...,ID_n])
13                       chr_add_to_history(c_1,j_i,[ID_1,...,ID_n])
14                       kill(ID_{k+1})
15                       ...
16                       kill(ID_n)
17                       B_1
18                       ...
19                       B_l
20                     endif
```

```
21                  endif
22                endif
23              endif
24            end
25          ...
26        end
27      end
28    ...
29  end
30 end
```

The nested `foreach` loops are basically the procedural equivalent to the backtracking-based `chr_lookup/2` in the Prolog version. For every combinations of the active constraint and its partner constraints we prove that all found constraints are still alive (l. 10) and the guard is satisfied (l. 11). If the rule is applicable, it is added to the propagation history, the constraint store is updated and the body constraints are called (ll. 13-19). Because the body constraints $(B_1, \ldots, B_l)$ are called sequentially and before the next combination of partner constraints is processed, the refined operational semantics ω_r is enforced.

2.6. Summary

In this chapter we introduced CHR, the source programming language for our compiler. Its general syntactic elements have been presented as well as the refined operational semantics ω_r, which is implemented by all major CHR systems as of today.

To get an overview of the existing CHR systems, we have examined the syntax and call semantics of CHR(Prolog), CHR(Java) and CHR(C) implementations. Moreover, basic compilation schemes for these logic and imperative host languages were presented. They are basis for our approach of compiling CHR into JavaScript with the help of CHR.js.

The CHR program `gcd/1` to calculate the greatest common divisor with the help of the subtraction-based Euclidean algorithm will be picked up as reference example in the following chapters.

3

Background: JavaScript

In node.js everything runs in parallel, except your code.

— Felix Geisendörfer, Core Committer of node.js[1]

This chapter introduces the target programming language JavaScript. Our aim is to present its execution cycle in modern runtime environments. Because of their importance for the compilation process, a short introduction in often used JavaScript elements is given. At the end we define the targeted runtime environments and give an outlook in very recent improvements of the programming language JavaScript, from which the compilation process could benefit in future.

[1] debuggable: Understanding node.js, `http://debuggable.com/posts/understanding-node-js:4bd98440-45e4-4a9a-8ef7-0f7ecbdd56cb`

3.1. Introduction

JavaScript is a multi-paradigm, untyped and interpreted programming language. It was created in 1995 by Brendan Eich [Eic05], who was working for Netscape, the corporation behind the Netscape Navigator. Although the name “JavaScript” was chosen to suggest a connection to the popular programming language Java, both languages have very little in common.

The language is standardized in the ECMAScript language specification, with JavaScript as its most popular implementation. Due to its strong connection to the web, the use of JavaScript has been increased similar to the world wide web, and as of today, the majority of all websites are equipped with dynamic contents provided by JavaScript. Especially the usage of JavaScript for building asynchronous website with AJAX (*Asynchronous JavaScript and XML*) has increased its popularity. Therefore, all major browsers support JavaScript out-of-the-box.

JavaScript is traditionally implemented as an interpreted language, but more recent browsers perform just-in-time compilation. Due to the better and better performance of JavaScript applications, it has been evolved to a general-purpose programming language not only suitable for the web. Today’s most popular approach for a general use of JavaScript is node.js [TV10], which allows to run it server-sided and even build desktop applications and hardware drivers in JavaScript.

Especially in the era of browser wars at the end of the 1990s, JavaScript was very platform-dependent. The first standardization by the European Computer Manufacturers Association (*ECMA* for short) in 1997 [EK97] achieved a first portability between browsers of different organisations.

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 finalized in June of 2015, not all JavaScript implementations support the full syntax improvements. Because the changes are backwards-compatible to version 5 (December 2009), the improvements are adopted gradually.

3.2. Targeted Runtime Environments

As stated before, all of today's major browsers support JavaScript out-of-the-box. Nevertheless they use different implementations as the runtime environment, which differ in optimization techniques, execution time and implementation-specific language extensions.

The major implementations of JavaScript runtime environments are:

- **V8**, used by the web browsers Google Chrome and Opera, as well as in node.js;
- **SpiderMonkey**, used by most Mozilla products, including the Firefox web browser;
- **JavaScriptCore**, used in the Safari web browser;
- **Nashorn** and **Rhino**, which allows to embed JavaScript in Java applications.

Because CHR.js is written for usage in both web browsers and server-side environments using node.js, our approach is geared to the `V8` implementation of JavaScript. However, it only uses standard ECMAScript and is therefore executable in all of the other runtime environments, too. Differences might only occur in specific optimizations targeting the just-in-time compilation of V8.

The standardization of ES6 has been recently finalized in June 2015, thus we only use several new language elements. However, the design of the CHR.js language as specified in Chapter 4 already orientates on the ES6-specific syntax element of so-called tagged template strings. If the used runtime environment does not support the used ES6 syntax, we recommend the usage of a transpiler. With the help of a transpiler it is possible to translate new ES6 syntax to plain old ECMAScript 5, which is widely supported.

Many ES6 features were added to Google's V8 with the release of version `4.5` in July 2015. We therefore assume a version greater or equal for CHR.js. This V8 version is shipped with the Google Chrome web browser since version `45`, released in September 2015. node.js uses V8 `4.5` since its version `4.0`, which was equally released in September 2015. We do not recommend to use CHR.js with the prior node.js releases

of version `0.x` (the node.js version numbers `1.x`, `2.x` and `3.x` were skipped because of compatibility reasons), as they are shipped with very old V8 versions.

3.3. The JavaScript Event Loop

The syntax of JavaScript is influenced by C. Being a multi-paradigm programming language with functional and object-orientated influences, the code is easily readable for everyone familiar with these kind of languages. Although we assume a basic knowledge of JavaScript in this work, we want to introduce the execution cycle of JavaScript as it is different to the languages listed before and unlikely to most other imperative programming languages.

Listing 3.1: Example code of nested functions

```
1 function f()
2   g()
3   h()
4   return 0
5 end
6
7 f()
```

In most traditional imperative programming languages there is a strict stack-based execution cycle which will be illustrated with help of the pseudo-code example of Listing 3.1: Assuming a function `f` is initially called and that in its body it calls another function `g`, the current program state (that means program counter, stack pointer, etc.) is saved at the stack and `g` gets executed. Once the execution of `g` is terminated, we return to the save program state and continue the execution of `f`, for example by calling the next function `h` in the body of `f`.

This stack-based execution cycle has disadvantages when it comes to blocking tasks: If the function `g` in this example is a query to a remote database, the whole program gets blocked until the result of `g` is returned and the execution can continue with the

application of `h`. We therefore call `f` a *blocking function*. If `h` does not need the result of the blocking function `g`, it could be executed in parallel.

In most programming languages this consideration results in a concurrency model based on multiple threads. However, in JavaScript this problem is solved with the help of the *event loop*: It adds a second layer, the *message queue*, to the sequence of tasks and therefore allows the concurrency of blocking functions.

The message queue stores an ordered list of messages to be processed by the runtime environment. Each message is handled as described in the stack-based execution cycle, which means it is processed until the function returns. The difference is that in the function's body we can register event handlers which might add messages to the message queue in future. In the example given before, the blocking function `g` which sends a request to a remote database would register an event handler that shall be executed once the response is received.

The function `g` registers the event handler and returns immediately. The execution of the function `f` is resumed, that means `h` gets called. Once `f` terminates, the next message of the message queue is taken and processed. This constant lookup for messages in the message queue can be described as a loop which is given in pseudo-code in Listing 3.2.

Listing 3.2: Blocking implementation of the event loop

```
1 while queue.waitForMessage()
2   queue.processNextMessage()
3 end
```

This code example clarifies why this execution cycle is called *event loop*: It simply is a loop constantly looking for new messages representing events. While the event loop is used for a better handling of blocking functions, for itself it is blocking – the `queue.waitForMessage()` stops as long as the message queue is empty and also the `queue.processNextMessage()` is blocking. This is reasonable, because the event loop is in general implemented as part of the runtime environment, that means in another programming language as JavaScript. Because JavaScript runtimes environments are single-threaded, this also illustrates that every message is processed until it

terminates. That means no two functions are executed at the same time. Even when a new message arrives, they are handled in the order of the message queue. On the other hand, all work that is not done by JavaScript (for example database queries, HTTP requests, I/O), is executed in the background concurrently and only adds a message to the message queue once it is finished. This behaviour resulted in the quotation of Felix Geisendörfer mentioned at the beginning of this chapter: “In node.js everything runs in parallel, except your code.”

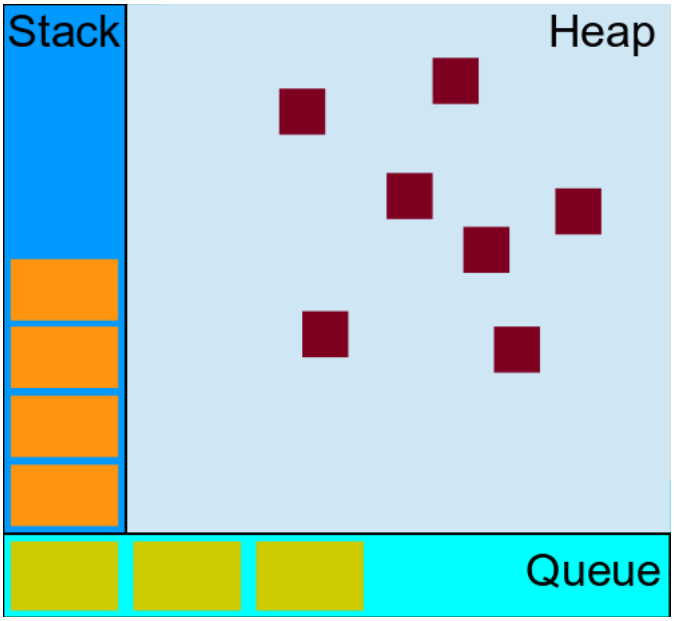

Figure 3.1.: Visual representation of the event loop[2]

Figure 3.1 illustrates the behaviour of the event loop, including the traditional stack, the message queue and the heap, a mostly unstructured region of memory where instantiated JavaScript objects are stored.

[2]By the Mozilla Developer Network, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`

3.3.1. Synchronous and Asynchronous Functions

As stated before, even with the event loop it is not possible to run multiple JavaScript functions in parallel.[3] The event loop execution cycle only improves the way to handle functions that are waiting for non-JavaScript processes to finish (like I/O) and therefore blocking.

Consider the JavaScript example given in Listing 3.3. This function is blocking as well, although it does not require any external process. To make it even worse, its execution blocks any other message on the event loop as long as the `for`-loop is not completely finished. However, this can not be improved with the use of the message queue if we really want to process all `1000000` items – either the stack or the message queue is used to loop through the items.

Listing 3.3: Blocking function in JavaScript

```
1 function f () {
2   var limit = Math.pow(10,10)
3   for (var i = 0; i < limit; i++) {
4     // do something
5   }
6 }
```

In other functions it is possible to decide whether to process functions via the stack-based execution or the message queue fits better. To make use of the message queue mechanism, we use an *asynchronous function*. This is a function which gets a callback as one of its parameters. This *event handler* is called with the returned value. Listing 3.4 presents a JavaScript code example of an asynchronous, non-blocking database query. Following the JavaScript conventions, the event handler is passed as the very last argument to the function `queryDbAsync`.

[3] In Chapter 5, where the implementation of the interactive CHR tracer *chrjs.net* is presented, we introduce a method to run multiple JavaScript functions in parallel using Web Workers.

Listing 3.4: Asynchronous database query in JavaScript

```
function queryDbAsync (query, callback) {
  database.request(query, function onFinish (result) {
    // function is called when the query has been finished

    // pass the result to the given callback
    callback(result)
  })
}
```

In opposite, Listing 3.5 shows the equivalent blocking database call. Functions which return their result with the help of the `return` statement, are called *synchronous functions*.

Listing 3.5: Synchronous database query in JavaScript

```
function queryDbSync (query) {
  var result = database.request(query) // blocking, which might
                                       //   take some time
  // ... finally
  return result
}
```

The distinction between asynchronous and synchronous as well as non-blocking and blocking functions matters especially when it comes to browser performance and user feedback: When the event loop is blocked by a sychronous function, in most browsers the user can not interact with the website any more, because JavaScript is single-threaded and used for the rendering of the website too. It is therefore recommended to outsource processor-intensive calculations to non-JavaScript processes or spawn child-processes of JavaScript (natively possible in node.js or with the help of Web Workers in the browser). Another possibility is to give the event loop the chance to switch to another task by explicitly adding a new message to the event loop that will later resume the time-consuming calculation.

3.3.2. Queueing Messages

JavaScript provides multiple ways to add new messages to the message queue:

- `setTimeout(callback, time)`,
- `setImmediate(callback)` (not in the ECMAScript standard) and
- `process.nextTick(callback)` (only in node.js).

With the help of these functions, it is possible to add the event handler provided as parameter `callback` to the event queue. In case of `setTimeout`, the `callback` is invoked after the given `time`, the other two functions call the event handler immediately after all other operations have been finished. Listing 3.6 presents a variation of the code given in Listing 3.3. The overall calculation is split into smaller parts. Because of the usage of `setImmediate`, we give other events the chance to be executed in between.

Listing 3.6: Queued implementation of Listing 3.3

```
1  var limit = Math.pow(10,10)
2  var chunkLimit = Math.pow(10,5)
3  function f (k) {
4    for (var i = k; i < k+chunkLimit; i++) {
5      // do something
6    }
7
8    if (i == limit)
9      return
10
11   setImmediate(function () {
12     f(i)
13   })
14 }
```

Another way to structure asynchronous functions is the usage of *Promises*, which are introduced in Section 3.5.6.

3.3.3. Stack Size

Unlike most C and C++ compilers, JavaScript has no tail call optimisation (TCO).[4] With TCO it is possible to optimize the execution of a function with tail recursion. Instead of generating a new stack frame the current one is reused.

The actual available stack size depends on the used JavaScript runtime environment and the used system. Splitting large calculations into asynchronous functions is way to prevent exceeding the stack size. This is a result of the fact, that the event loop is continued only when the current calculation has been finished. That means the stack is completely cut down when polling a new message from the message queue.

This method to prevent a JavaScript `RangeError` will be illustrated in Section 3.4 again.

3.3.4. Garbage Collection

It is worth mentioning garbage collection (GC) here. Similar to other imperative programming languages, JavaScript runtime environments perform GC to free memory by destroying unreferenced variables and objects in the heap. As JavaScript does not have explicit memory management, this process can not be controlled manually and is instead managed by the runtime environment. The time when the GC is executed depends on its implementation. The only thing specified among the various runtime environments is the GC not being active within the execution of a function. This is another reason dividing a large calculation in smaller chunks is worth considering.

3.4. Example Program: `gcd(a,b)`

In this section we want to discuss the implementation of the Euclidean algorithm to compute the greatest common divisor as presented in Section 2.4. We will present different implementations with respect to the function properties presented before. This

[4] Tail call optimisation has been specified in the ECMAScript 6 standard in June of 2015. However it is currently implemented by no JavaScript engine. Only Babel, a JavaScript transpiler, supports TCO in parts.

section will later be referred to when we discuss our implementation approaches of CHR.js and possible optimizations in Chapter 6. Unlike the CHR implementation, we restrict ourselves to the calculation of the GCD of exactly two positive integer values.

3.4.1. Iterative Implementation

The first implementation approach simply translates the pseudo-code of Listing 2.2 into equivalent JavaScript. It is presented in Listing 3.7. An example call is given in line 11.

Listing 3.7: Iterative implementation of `gcd()`

```
1  function gcd (a, b) {
2    while (b > 0) {
3      if (a > b) {
4        a = a - b
5      } else {
6        b = b - a
7      }
8    }
9    return a
10 }
11 // Example call:
12 console.log(gcd(6,9))
```

This implementation is blocking and synchronous. If it is called with large input values, the `while` loop of line 2 is executed until the GCD is found. While this approach is straight-forward and very efficient, it will block the entire application until finished.

3.4.2. Recursive Implementation

Listing 3.8 presents a recursive implementation of the `gcd` function. Instead of mutating the variables `a` and `b` directly, we pass the adapted argument into a recursive call.

Although this implementation is tail-recursive, it does not get optimized as already discussed in Section 3.3.3. As a consequence, it is not possible to call this implementation of `gcd` with fairly large inputs without JavaScript throwing a `MaxRange` error. This is due to the fact that this implementation is blocking like the previous one, but because of the recursion a new stack frame is generated for each call of `gcd`.

Listing 3.8: Recursive implementation of `gcd()`

```
function gcd (a, b) {
  if (b == 0)
    return a
  if (a > b)
    return gcd(a-b, b)
  return gcd(a, b-a)
}
```

This can not be avoided by simply pass a callback as suggested in Listing 3.9. Because the definition of `gcd` still does not include any asynchronous function that adds a new message to the message queue (c.f. Section 3.3.2) it remains blocking and recursive, equally resulting in an exceeded stack size for large input values.

Listing 3.9: Recursive implementation of `gcd()`, with `callback` parameter

```
function gcd (a, b, callback) {
  if (b == 0)
    callback(a)
  else if (a > b)
    gcd(a-b, b, callback)
  else
    gcd(a, b-a, callback)
}
```

3.4.3. Asynchronous Implementation

As suggested before, we have to explicitly pause the calculation and resume it in a new event. This can be achieved by using `setTimeout(callback, 0)`, which is supported by all JavaScript runtime environments. We adapt the recursive implementation of Listing 3.8 by adding `setTimeout` calls in a way that the execution is continued in one of the next steps of the event loop. This asynchronous implementation is presented in Listing 3.10.

Listing 3.10: Asynchronous implementation of `gcd()`

```
1  function gcd (a, b, callback) {
2    if (b == 0) {
3      callback(a)
4    } else if (a > b) {
5      setTimeout(function () {
6        gcd(a-b, b, callback)
7      }, 0)
8    } else {
9      setTimeout(function () {
10       gcd(a, b-a, callback)
11     }, 0)
12   }
13 }
```

By this adoption we can prevent the exceeding of the stack size because only a single stack frame is generated per message in the event loop. On the other hand, this comes with an enormous decrease of performance: According to the specification of HTML 5 [HH11], the minimum delay of `setTimeout` calls is four milliseconds.

A common approach to profit from both the advantages of asynchronous functions using the message queue as well as the good performance of the stack-based execution is to call `setTimeout` only for every n'th invocation, with n as near as possible to the maximum stack size. This approach is illustrated with the second asynchronous

implementation of Listing 3.11. Because the stack size is platform-dependent, we use $n = 1000$ in this example.

Listing 3.11: Asynchronous implementation with moderate use of `setTimeout()`

```
var i = 0
var chunkLimit = 1000
function gcd (a, b, callback) {
  i++
  if (b == 0) {
    callback(a)
  } else if (a > b) {
    if (i % chunkLimit) {
      gcd (a-b, b, callback)
    } else {
      setTimeout(function () {
        gcd(a-b, b, callback)
      }, 0)
    }
  } else {
    if (i % chunkLimit) {
      gcd(a, b-a, callback)
    } else {
      setTimeout(function () {
        gcd(a, b-a, callback)
      }, 0)
    }
  }
}
```

3.5. Syntax

The implementation examples of the previous section already gave an insight to the syntax of JavaScript. In opposite to CHR, describing the syntax of the targeted language JavaScript is out of the scope of this work. In this section we want to concentrate only on syntactic elements that are worth a closer examination – either because of their importance for the compilation process or because they are very recently defined as part of the ECMAScript 6 standard and therefore not yet well known, even for experienced JavaScript developers.

3.5.1. Prototype-based Inheritance

The prototype-based inheritance model of JavaScript is of the first category. It is a major construct of the language, but often confusing for developers familiar with class-based programming languages like Java and C++.

JavaScript does not provide a traditional class implementation.[5] Instead, JavaScript has only one construct: objects. Each object has an internal link to another object called its *prototype*, a general form of the object. This prototype has a prototype of its own, and so on. This link from the initial object over all of its prototypes is called the *prototype chain*. It ends once an object with `null` as its prototype is reached, which has by definition no prototype.

When the property `'b'` of an object `a` is accessed (written as `a.b` or `a['b']`), the property `'b'` is searched for along the prototype chain until the first occurrence is found. So `'b'` is either specified in the object `a` itself or in the nearest prototype along the chain containing a `'b'` property.

The values of the objects can be of any type. It is even possible to define a function, then the `this` keyword refers to the object. This way it is possible to define methods on objects. Listing 3.12 shows an example of the definition of an object, including a function as property.

[5] Although the ECMAScript 6 standard defines the `class` keyword, it is just syntactic sugar and wraps around JavaScript's prototype-based inheritance model.

Listing 3.12: JavaScript object example

```
var person = {
  prename: 'John',
  surname: 'Doe',
  name: function () {
    return this.prename + ' ' + this.surname
  }
}
console.log(person.name()) // 'John Doe'
```

With the definition of a person as presented in Listing 3.12 it would be necessary to define the `name()` function for each and every person. This could be prevented by using a named function (that means to simply define the function `name()` in the global scope, as the `this` reference can be bound dynamically). A better approach is to create a `Person` prototype, which is similar to the definition of a class in languages like Java and C++.

A new prototype can be created by simply define a new function. When called with the `new` keyword, memory for the new instance gets allocated. Methods of this prototype can be defined by adding them to the `prototype` property of this new defined function. An example definition of the `Person` prototype is presented in Listing 3.13.

The advantage of specifying methods in the `prototype` property is that the function is created only once and can be used by any instance. A disadvantage can be the process of looking up the prototype chain to access a property not defined in the object itself.

With prototype it is possible to realize an equivalent of traditional class inheritance: It is possible to explicitly define the prototype chain of an object or prototype, which results in an inheritance model similar to Java and C++.

Unlike these programming languages, JavaScript does not provide a way to for information hiding. Keywords like `public`, `protected` and `private` simply does not exist. Every property defined along the prototype chain can be accessed by the instantiated object. Although it would be possible prevent the exposure of internal variables us-

ing closures, this has enormous downsides regarding memory usage and possible optimizations.

Listing 3.13: Example for the creation of a prototype

```
function Person (prename, surname) {
  this.prename = prename
  this.surname = surname
}

Person.prototype.name = function () {
  return this.prename + ' ' + this.surname
}

var john = new Person('John', 'Doe')
console.log(john.name()) // 'John Doe'
```

This resulted in naming conventions used by JavaScript developers: private properties and functions are in general prefixed by an underscore _. Because in the CHR.js module we want to allow using constraint and rule names beginning with an underscore, we use the convention to start private properties with a capital letter.

A more detailed introduction of the prototype-based inheritance model is out of scope of this work. However, because CHR rules, constraints and even the used compilers have been implemented using prototypes, a basic comprehension of JavaScript prototypes is necessary to understand our contribution.

In the next subsections we will shortly introduce further JavaScript elements which were added very recently to the ECMAScript standard in June of 2015 and are therefore not yet very common, but intensively used for our compilation process.

3.5.2. Template Strings

With JavaScript versions prior to ECMAScript 6 it is not easily possible to define multi-line strings. Neither "" nor ″ support strings spreading over multiple lines without

modification in the source code. Because we want to make the specification of CHR rules as easy as possible, we decided to use ES6 *template strings* (`) for these purposes. An example creation of a multi-line string containing multiple rules in presented in Listing 3.14. It includes placeholders specified within curly braces `${ ... }`. This syntax will be later use to evaluate built-in constraints within rules as well as a comfortable method to generate source code in the compiler.

Listing 3.14: JavaScript multi-line strings using template strings

```
1 var name = 'John Doe'
2 var activity = 'travel'
3
4 var greeting = `Hello ${ name },
5 how was your ${ activity }?`
```

3.5.3. Tagged Template Strings

Template strings can not only be used to create multi-line strings containing placeholders. It is possible to prefix the template string with a function name (the *template handler*) which takes the string and values of the placeholders as arguments. This syntax, called *tagged template strings*, was added in ES6 to support to directly embed domain-specific languages (DSL) into JavaScript. Because in our contribution we want to enhance the native JavaScript syntax with CHR as DSL, we use this new syntax to specify the CHR rules. An example usage of tagged template strings is given in Listing 3.15.

Listing 3.15: Example usage of tagged template strings

```
1 function text (parts, name, activity) {
2   return parts[0] + name + parts[1] + activity + parts[2]
3 }
4
5 var greeting = text`Hello ${ 'John' },
6 how was your ${ 'travel' }?`
```

The first parameter of the template handler is an array of the input string being split at each placeholder. Therefore in the example of Listing 3.15 `parts` is `['Hello ', ',\nhow was your ', '?']`.

3.5.4. Arrow Functions

Template strings are basically syntactic sugar and could be expressed with JavaScript syntax prior to ECMAScript 6, too.[6] In general this holds true *arrow functions*. They provide an easy way to define (often anonymous) functions and are similar to lambda expressions of functional languages. The traditional way to calculate the squares of an array of numbers is shown in Listing 3.17. Arrow functions provide a shorter syntax, which Listing 3.16 illustrates. There the `square` function has been replaced by the arrow function `(x) => x*x`.

Listing 3.16: Example usage of arrow functions

```
1 console.log([ 1, 2, 3 ].map((x) => x*x)) // [1, 4, 9]
```

Listing 3.17: ES5 equivalent of Listing 3.16

```
1 function square (x) {
2   return x * x
3 }
4 console.log([ 1, 2, 3 ].map(square)) // [1, 4, 9]
```

We will use the arrow function syntax together with template strings to easily specify guards in CHR rules. This syntax is also used as a convenient way to implement the optimized compiler in Chapter 6.

3.5.5. Desctructuring Assignments

As mentioned in Section 2.5.3, imperative programming languages generally do not have an equivalent of the unification used for the pattern matching in CHR(LP) systems.

[6] This is what transpilers like *Babel* typically do: Translate very new language features into equivalent code for older runtime environments.

With ES6, a new JavaScript expression has been introduced which might fill this gap: *destructuring assignments*. With these it is possible to define a pattern at the left-hand side of an assignment. Some examples for destructuring assignments are provided in Listing 3.18.

Listing 3.18: Example usage of destructuring assignments

```
1  // a = 1, b = 2
2  [a, b] = [1, 2]
3
4  // a = 1, b = 2, rest = [3, 4, 5]
5  [a, b, ...rest] = [1, 2, 3, 4, 5]
6
7  // a = 1, b = 2
8  ({a, b} = {a:1, b:2})
9
10 // a = 1, b = 2, rest = { c: 3, d: 4 }
11 // part of ECMAScript 7
12 ({a, b, ...rest} = {a:1, b:2, c:3, d:4})
```

With destructuring assignments it is even possible to provide nested patterns which makes it suitable for an unification replacement in CHR.js.

3.5.6. Promises

As a last syntax element of JavaScript we want to elaborate in this work, we consider *promises*, which provide a convenient way to structure asynchronous functions. The asynchronous implementation of `gcd()` in Listing 3.10 already illustrates a problem many JavaScript programs face: callbacks are often nested (here: `callback` within `gcd` within `setTimeout`, line 5ff.) which leads to source code which is hard to maintain. For the developer, the event loop has the disadvantage of program flow being not easily readable from the source code.

One way out of the so-called “callback hell” is the usage of *promises*. Simply put, they provide a syntax to specify the program flow by chainable functions instead of nested callbacks. The resulting waterfall model is a better representation of the execution cycle provided by the event loop.

A basic example of a promise is shown in Listing 3.19. By calling `new Promise()`, a promise is created which provides a `then` function (and is therefore called a *thenable object*) which is used to process the result. A promise is called to *resolve* to the result. Unlike traditional event handlers using callbacks, a promise is guaranteed to resolve only once. To start the promise chain, we will use the predefined function `Promise.resolve()`, which simply creates a promise that always resolves, but has `undefined` as result.

Listing 3.19: Example definition of a promise

```
var promise = new Promise(function(resolve, reject) {
  // do something, probably asynchronous
  var result = 'e.g. dataset'

  if (/* success */) {
    resolve(result);
  }
  else {
    reject(new Error('Failed'));
  }
})

promise.then(function (result) {
  // process result
})
```

While this example might not look like an improvement to the traditional callback version, the advantages are clear once multiple promises are used. As illustrated by the example

in Listing 3.20, the promises can be chained by their `then` function. The `catch` function fetches any error that might occur while processing the promise chain.

Listing 3.20: Promise chaining

```
promise1.then(function (result1) {
  return promise2
}).then(function (result2) {
  return promise3
}).then(function (finalResult) {
  // do something
}).catch(function (error) {
  // somewhere in the chain an error occurred
})
```

In CHR.js we will use promises to represent each constraint. In this way it is easily possible to process multiple input constraints – either in sequence or even in parallel, as presented in Section 4.5.

To compare the different implementation ideas of `gcd(a,b)`, we present in Listing 3.21 an implementation similar to Section 3.4 using only promises.

Listing 3.21: Implementation of `gcd()` using promises

```
function p (v) { // v = [a, b]
  if (v[1] === 0) {
    return v[0]
  }
  if (v[0] > v[1]) {
    return Promise.resolve([v[0]-v[1], v[1]]).then(p)
  }
  else {
    return Promise.resolve([v[0], v[1]-v[0]]).then(p)
  }
}
```

```
12
13 function gcd (a, b) {
14   return Promise.resolve([a, b]).then(p)
15 }
16
17 // Example call:
18 gcd(6, 9).then(function (res) {
19   console.log(res)
20 })
```

The definition of `p` in Listing 3.21 makes use of the fact, that when a promise returns another one, it is handled like called within the `then()` property. As a result, the following two expressions are equal:

```
1  // Variation 1: Nested promises
2  promise1.then(function (v1) {
3    // process v1
4    return promise2.then(function (v2) {
5      // process v2
6      return promise3.then(function (v3) {
7        ...
8      })
9    })
10 })
11 // Variation 2: Chained promises
12 promise1.then(function (v1) {
13   // process v1
14 }).then(function (v2) {
15   // process v2
16 }).then(function (v3) {
17   ...
18 })
```

3.6. Code Structuring

JavaScript has no native support for the inclusion of scripts and modules. In the world of the web, required third-party libraries were simply added to the `<head>` section of the HTML document. This is a very simple mechanism to load JavaScript modules on websites and very frequently used even today. The only thing web developers have to consider are naming conventions, because all loaded JavaScripts expose their interfaces to the global scope.

The server-side JavaScript runtime environment *node.js* had to provide a different way to split a large JavaScript code base across multiple files. In opposite to browser environments, it provides a `require()` function to include JavaScript code. This resulted in a simple but powerful module system. Because files and modules are in one-to-one correspondece, it is possible to load the file `rules.js` of the same directory by calling `require('./rules.js')`. On the other hand it supports global packages which are specified only by their name, that means the explicit directory (here: `./`) is omitted. Global packages can be installed with the help of *npm*, a package manager for node.js.

In the code examples we will use the `var identifier = require(location)` syntax to refer to the used module by its `location`. If the code fragments should be executed in a browser environment – where the `require()` function is not defined – it can by simply used by the given `identifier`. This is a result of the fact, that the libraries we created as part of this work use the same names for both browser-builds and builds for node.js.

3.7. Summary

In this chapter we introduced JavaScript, the targeted language of our compilation process. We presented its execution cycle based on event loop and message queue. Its comprehension is necessary for design decisions in the development of a feature-rich CHR(JavaScript) system. It is also the basis for optimizations. At the end the

advantage and disadvantage of synchronous and asynchronous, blocking and non-blocking implementations has to be balanced, as the `gcd()` example of 3.4 has shown.

In Section 3.5 we introduce some syntax elements of the recent ECMAScript 6 standard that are important for both the compiler construction as well as the CHR.js language design. The implementation of tagged template strings to support DSLs was essentially for our approach of embedding CHR into JavaScript. The introduction of these syntax elements already indicates the language design of CHR.js, which will be discussed in the following chapters.

4

CHR.js – A CHR(JavaScript) Interpreter

> *They say we only use a fraction of our brain's true potential. Now that's when we're awake. When we're asleep, we can do almost anything.*
>
> — Dominick "Dom" Cobb, in Inception (2010)

In the previous chapters we introduced the two programming languages relevant for our work. The short introduction to JavaScript already contained most of the components necessary to compile CHR into JavaScript. In this chapter we want to add the last ingredients: the definition of the CHR.js language, its parsing grammar and the compilation idea.

As a result, we present a feature-rich implementation of a CHR(JavaScript) system. The created JavaScript module, called *CHR.js*, will be used as an interpreter for CHR

embedded in JavaScript. Although this just-in-time compilation approach has disadvantages regarding execution performance, it allows us to manipulate the execution of CHR programs at runtime, which is a requirement for the implementation of graphical and web-based tracers. An example application, which uses most of the tracing abilities provided by CHR.js, will be presented in Chapter 5.

For an improvided performance, in Chapter 6 we will introduce a precompiler which takes JavaScript source code with embedded CHR.js and translates the contained CHR rules in a compilation step, similar to the approach used by JCHR and CCHR. Keeping this in mind, the definition of the CHR.js language in this chapter covers both use-cases, so that at the end the JavaScript code containing CHR rules can be either executed directly in an interpreter mode or being precompiled to improve performance.

4.1. Overview

The CHR.js module consists of the following components:

Runtime Components No matter how the given CHR rules look like, to execute a CHR.js program a number of components is always needed. We will introduce JavaScript prototypes to represent a single constraint, the constraint store, the propagation history, etc.

The Parsing Grammar To analyse a given string consisting of one or more CHR rules, our contribution contains a grammar that describes the CHR.js language in a formal way and is as well the basis for the parsing step of the compilation.

Compiler After the source code has been analysed by the parser, the compiler generates actual JavaScript code to handle the specified constraints.

In the following sections we introduce the three components and give usage examples. The CHR.js manual, a more technical introduction for developers and users, can be found in Appendix C.1.

4.1.1. Integration

The concrete way to integrate CHR.js into a JavaScript project depends on the runtime environment (c.f. 3.6). In browser-based applications, the `chr.js` (or `chr.min.js` as minified version) must be included. It exposes a global `CHR` variable which provides access to the library.

For the usage with node.js the module has been published at *npm*, JavaScript's most popular package manager, under the name "chr". By calling `var CHR = require('chr')` the library can be assigned to the variable `CHR`, too.

In the following sections we assume that the module has already been loaded in the `CHR` variable.

4.2. Runtime Components

In [VWWSD08], the data structures and their methods needed for imperative host languages have been presented: a representation for a single constraint, the constraint store and the propagation history. Our implementation seizes on the mentioned interfaces and adapt them to work best with JavaScript. In most cases, this includes the definition whether a method should be synchronous or asynchronous. The data structures are defined by prototypes (c.f. Section 3.5.1).

In addition to these data structures, we add a helper object for common tasks used in the runtime and data structures to manage the rules. The latter is not needed in the work of [VWWSD08], because there the implementation of an ahead-of-time (AOT) compilation is discussed.

The runtime components are exposed as properties to the `CHR` object, meaning the constraint constructor can be used via `CHR.Constraint`, the propagation history via `CHR.History`, etc.[1] In addition, the node.js module provides a single endpoint for the runtime components which can be used by calling `require('chr/runtime')`.

[1] The related sources can be found in the `/src` directory, for example `src/constraint.js` for the constraint constructor.

4.2.1. Constraint: `CHR.Constraint`

The `Constraint` prototype stores all data of an actual constraint. It can be created by calling `new Constraint(name, arity, argumentsArray)`. Unlike the other runtime components, the `Constraint` prototype is only used for data storage. In order to avoid so-called *hidden classes*, which can result in big performance decreases, this prototype is used. In general a hidden class is generated for every object that is not instantiated by using a prototype. Because V8's JIT-compiler has no knowledge about the general construction and usage of this object, it can not be optimized. We avoid these hidden classes by explicit JavaScript prototypes where possible.

Properties

`name` The name of the constraint as string.

`arity` The constraint's arity as positive integer.

`functor` The functor of the constraint is often used and therefore added automatically at construction.

`args` The arguments of the constraint, given as array.

`id` The `id` of a constraint is generated when it is added to the constraint store. Until then it remains `null`.

`alive` Flag, whether the constraint is still alive or had been removed from the constraint store.

Methods

The `Constraint` prototype exposes only a single method: `toString()`. It generates a string representation similar to CHR(Prolog) systems, for example `gcd(6)`.

4.2.2. Store: `CHR.Store`

The constraint store is the data structure that contains all alive constraints. It has no constructor parameters.

Properties

`length` The number of stored constraints. This property should be read only.

`invalid` Flag whether the constraint store is invalid, that means the built-in `false/0` has been added.

Methods

The following methods are adaptions of the proposition of [VWWSD08].

`reset()` Method to truncate the constraint store. In practice it does not matter if the constraint store is reset or a fresh `Store` instance is used. However the reuse of the existing instance leads to better performance in V8.

`add(constraint)` Adds a constraint to the store.

`kill(id)` Method to kill and delete a constraint from the store.

`alive(id)` Checks whether the constraint specified by the given identifier is still alive.

`allAlive(arrayOfIds)` Checks whether all constraints specified by an array of identifiers are still alive.

`args(id)` Returns the arguments of the constraint specified by the given identifier.

`lookup(name,arity)` Returns an array of ids of all alive constraints matching the given name and arity.

Additionally the constraint store exposes the following methods:

`invalidate()` Method to reset the store and set the `invalid` flag.

`forEach(callback)` Similar to JavaScript's native `Array.forEach(callback)` this method iterates over the stored and alive constraints and executes the given callback for each of them. The order of the iteration is not guaranteed.

`map` As an equivalent to JavaScript's native `Array.map(callback)` this method iterates over all stored and alive constraints and returns the result of the callback for each. This execution of this method is synchronous and returns an array.

`toString()` Method which returns a ASCII-based table representation of the contents of the constraint store.

Events

The constraint store emits several events. It is possible to add event listeners by calling `store.on(eventName, eventHandler)`, where `eventHandler` is a function invoked once the event appears.

Two events are supported: `add` and `remove`. Both pass the added or killed constraint to the specified constraint handler. A usage example is presented in Listing 4.1.

Listing 4.1: Registering event listeners for the constraint store

```
1 var store = new CHR.Store()
2 store.on('add', function (constraint) { /* process it */ })
3 store.on('remove', function (constraint) { /* process it */ })
```

While we will use JavaScript promises for the asynchronous behaviour of CHR.js, we use traditional callbacks for the events of the constraint store. Promises can be executed only once, whereas it is reasonable use-case to fetch the removal of constraints with multiple event listeners. Also we expect that these listeners are only used for debugging output and to trigger user interface updates, which are not required to block the further CHR.js execution.

4.2.3. Propagation History: `CHR.History`

The propagation history is used to prevent the repetitive execution of rules with identical constraints. It takes no constructor argument and has no public properties at all. In addition to the proposition of [VWWSD08], we add a method to explicitly ask for the existence of an entry in the propagation history, not just its absence.

Methods

`add(rule,arrayOfIds)` Adds an entry to the history for the given rule. The used ids of the constraints have to be specified as an array.

`notIn(rule,arrayOfIds)` Asks whether the given tuple is not in the propagation history.

`has(rule,arrayOfIds)` Asks whether the given tuple is already in the propagation history.

Because a rule is only applied if it was not fired with the identical constraints before, the `notIn` method is generally used.

4.2.4. Rule: `CHR.Rule`

The `CHR.Rule` constructor is used to store a single CHR rule and all of its properties. In general it is instantiated automatically after using the parser. However it is possible for the user to create a new `CHR.Rule` instance manually. Then it expects the object generated by the parser as its first argument. Options, for example to explicitly define the variable scope of the rule, can be passed as a second argument to the constructor. Given the object generated by the CHR.js parser, it automatically generates the source code to make the rule applicable for a `Constraint` instance.

Properties

The `Rule` object is used to provide an easy access to the contained constraint occurrences. So a rule which has the constraint with the functor $a/0$ in its head will generate a dynamic property `rule['a/0']`, which is an array with all occurrences of this constraint (as it is possible to occur multiple times in the rule's head). Because we want to avoid naming conflicts in the properties, all static properties of the `Rule` prototype begin with an uppercase letter. By definition CHR constraints must start with a lowercase letter.

The order of the occurrence handlers saved in the `rule[functor]` array coincide to the order of the refined operational semantics ω_r, meaning from right to left in case of multiple occurrences. However it is possible to re-order or even add or remove occurrence handlers manually to dynamically change the semantics of a CHR.js program.

`Scope` The outer scope of the variable of the rule. With this property it is possible to explicitly add variables to the scope. For example, by setting `rule.Scope = { a: 42 }`, the rule gets access to the variable `a`. It is only possible to add variables to the scope because JavaScript provides no mechanism to hide variables in it.

`Replacements` It is possible to define placeholders for built-ins in rules. This mechanism is presented in detail in Section 4.3.2. In the Replacements property these placeholders can be specified.

`Name` Every rule has a (at least generated) name which is stored in this property.

Methods

`ForEach(callback)` Similar to JavaScript's native `Array.forEach(callback)` function this method iterates over all saved functors and applies the given callback for each.

`Fire(chr,constraint)` Method that applies the rule for the given constraint. Because for every functor an array of occurrences is saved, this method iterates over the occurrences appropriate for the given `constraint` and applies this rule. As

the rule has to be executed in the global context of the `CHR` instance `chr`, it has to be passed as an argument. Otherwise the rule would have no knowledge about other rules in the program.

This `Fire()` method returns a promise which is resolved when all occurrences have been applied.

Breakpoints

The `Rule` prototype has another special property: `Breakpoints`. Similar to the event listeners of the constraint store they provide a way to interfere with the execution of the CHR program. Unlike event listeners, these breakpoints are not called as fire-and-forget but are real promises. That means that the application of the rule is paused until the breakpoint promise is resolved.

The `Rule` prototype implements the following Breakpoint:

`onTry` Pauses the rule application for each occurrence that is applicable. An object is passed to the promise which refers to the current rule, occurrence, the matching constraint and information about the location of this rule in the original CHR.js source code.

By default the `Rule.Breakpoints.onTry` property is assigned to `Promise.resolve()` which always immediately resolves.

4.2.5. CHR.js Program: `CHR.Rules`

A CHR.js program consists of a number of CHR rules. They are stored in the `Rules` prototype. Because a single rule should be addressable as `rules[ruleName]`, each rule is saved as a property to this object. To avoid naming conflicts, the properties of the `Rules` prototype therefore begin with an uppercase letter as well.

This prototype should never be manually instantiated. It is automatically created for every `CHR` instance. There, the properties, methods and breakpoints of the `Rules` prototype can be addressed via `chr.Rules`.

Properties

The only property of the `Rules` prototype is `Order`. It stores the order to use the saved rules as an array. Similar to the `Rule` prototype it is possible to change this order at runtime to dynamically change the behaviour of the given CHR program.

Methods

`Add(ruleObj,replacements)` Method that creates a new `Rule` according to the `ruleObj`, which is an object generated by the CHR.js parser. As the second argument the replacements used by the `Rule` prototype can be specified.

`Reset()` Removes all saved rules. This is a comfortable way to reset a given `CHR` instance by calling `chr.Rules.Reset()`.

`ForEach(callback)` Similar to JavaScript's native `Array.forEach(callback)` function this method iterates over all saved rules and applies the given callback for each.

`SetBreakpoints(promise)` Method to assign the given promise to all saved rules, that means to set `rule.Breakpoints.onTry` on each.

`RemoveBreakpoints()` Similar to `SetBreakpoints()`, but resets all rule breakpoints to `Promise.resolve()`.

4.2.6. Helpers

For common tasks we include an object with helper methods. It is accessible via `chr.Helpers`. Because it exposes only static methods, it does not have a constructor and prototype. In node.js, the helper methods can also be used by calling `require('chr/runtime').Helper`.

Static Methods

The `Helpers` object contains the following static methods:

`allDifferent(array)` Function which returns `true` if all values in the given array are unique, otherwise `false`.

`forEach(arrayOfArray,iteratorCallback,endCallback)` This function is a supplement for JavaScript's native `Array.forEach(iteratorCallback)` function. Instead of iterating over all values in an array, this version expects an array of arrays and iterates over all permutations of the contained values. An example application and its output is presented in Appendix B.1.

Dynamic Caller

In addition to the two static methods presented before, a third function, which is very important for the execution of CHR.js, is defined in the `Helper` object: `dynamicCaller(constraintName)`. It is a higher-order function in JavaScript, returning a function itself. By calling `dynamicCaller(constraintName)` a function with the name `constraintName` is created. This function then can be used to add a new constraint to the constraint store and trigger the rule application. It will be exposed as `chr[constraintName]`. The `gcd/1` constraint can therefore be called with `chr.gcd(6)`.

Listing 4.2: Definition of `Helper.dynamicCaller` (extract)

```
1  function dynamicCaller (name) {
2    return function () {
3      // 'functor', 'arity' and 'args' are defined
4      //   based on the given call arguments
5
6      var constraint = new Constraint(name, arity, args)
7      this.Store.add(constraint)
8
9      var rules = []
10     this.Rules.ForEach(function (rule) {
11       if (rule[functor]) {
12         rules.push(rule)
```

```
13        }
14      })
15
16      var self = this
17      return rules.reduce(function (curr, rule) {
18        return curr.then(function () {
19          return rule.Fire(self, constraint)
20        })
21      }, Promise.resolve())
22    }
23  }
```

An extract of the definition of `dynamicCaller` is presented in Listing 4.2. After the constraint is created and added to the constraint store, all rules are iterated to get all occurrences for this functor. In line 17ff a new JavaScript promise is created by reducing the array of applicable occurrence handlers to a single promise chain. This basically transforms the array to an expression of the form:

```
1 Promise
2   .resolve()
3   .then(function () {
4     return occurrenceHandler[0]
5   })
6   .then(function () {
7     return occurrenceHandler[1]
8   })
9   ...
```

4.3. Parsing CHR.js Rules

The demand for a further prototype `CHR` is not obvious, because the `CHR.Rules` prototype already stores all information about a CHR.js program – what else is needed? The reason for a superior `CHR` prototype comes from the consideration that we also want to provide an easy syntax to define CHR rules in JavaScript. Although `Rules.Add()` already provides a way to add new rules, we aim for a shorter syntax using tagged template strings (c.f. Section 3.5.2). In the following we assume a `CHR` instance has been assigned to the `chr` variable, meaning:

```
1 var chr = new CHR()
```

This CHR.js instance both stores the runtime properties and works as a template handler to define new rules. It is worth mentioning that it is possible to create multiple `CHR` instances in parallel to generate different solvers.

In sections 2.5.3 and 2.5.3 we presented usage examples for JCHR and CCHR. Both have in common, that the CHR rules are defined in a special code block respectively class and can therefore be easily embedded into existing plain-old Java and C code. In our contribution we follow this approach and specify the CHR rules within a special tagged template string. As a result, the overall source code remains valid JavaScript code, so that existing development tools like transpilers and static code analysers can still be used.

4.3.1. Example Program: `gcd/1`

We start the definition of the CHR.js language by considering our classical example: The calculation of the greatest common divisor. Its definition in CHR is known from Section 2.4. In Listing 4.3 the equivalent definition in CHR.js is shown.[2]

[2] In JavaScript the backslash `\` is an escape character in strings, for example used to specify the linebreak `\n`. Therefore, to use a backslash in a string, it is necessary to escape it too, that means to write `\\`. We omit the second backslash in our code listings for better readability and comparison to other CHR systems. Being a very common pitfall for syntax errors, CHR.js supports also the normal slash `/` instead of the backslash in simpagation rules.

Listing 4.3: Rules for `gcd/1` in CHR.js

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

This syntax is very similar to JCHR and CCHR, the CHR rules are specified in a tagged template string with the template handler `chr`.

In the next two subsections we want to give a deeper introduction to the syntax supported by the CHR.js language. We focus on the specification of CHR rules with the help of the `chr()` function first. The syntax to call constraints once defined and work with a CHR.js program is presented in Section 4.5.

4.3.2. Syntax

The syntax for the definition of rules is very similar to most existing CHR systems. Simple rules can be specified the same way as in CHR(Prolog), as long as one uses the JavaScript equivalent of Prolog operators.[3] Moreover, different rules are either separated by a newline or semicolon instead of a point.

Listing 4.4 shows the basic definition of the three CHR rule types. Here and if not other stated we assume that the template handler `chr` is an instantiated CHR.js solver.

Listing 4.4: Example definition for all three rule types

```
chr`
  propagation    @ a , b ==> c, d
  simplification @ e , f <=> g, h
  simpagation    @ i \ j <=> k, l
`
```

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

Built-in Constraints

Unlike JCHR, CCHR and CHR(Prolog) systems, we do not need to explicitly declare CHR constraints in CHR.js. Our contribution goes the other way round: It is always assumed that the identifiers provided in the head of the rules are CHR constraints, following the syntax introduced in 2.2.4. Because in the guard only built-ins are allowed, we do not need any explicit declaration here.

The rule's body is the only part of a rule where both built-in and CHR constraints are allowed. We therefore define that any identifier in the body is handled as a CHR constraint. To use built-ins in the body, we introduce a special syntax.

As presented in Section 3.5.2, (tagged) template strings already provide a way to include native JavaScript expressions into a DSL: placeholders. Using the `${...}` syntax, built-in constraints can be referenced.

At this point it is important to remember that the `${...}` syntax passes an expression, not the JavaScript source code to the template handler. That means, that for `${ 1+2 }` the template handler only gets the value `3`; for `${ console.log('Test') }` the `console.log` command is immediately executed and its result (which is `undefined`) is passed to the template handler. Actually our intention is to pass a function that is executed only when the rule fires.

Due to this consideration, we make use of the short arrow function syntax introduced in Section 3.5.4. Instead of passing the expression directly as in `${ expression }`, it is specified in an anonymous function: `${ () => expression }`, which is invoked for each rule application. A working example which uses the built-in `console.log()` is shown in Listing 4.5.

Listing 4.5: Example usage of a synchronous built-in

```
1 chr`
2   hello <=> ${ () => console.log('Hello!') }
3 `
```

The arrow function specified with the `${ () => expression }` syntax is synchronous and blocking. This is fine if we want to execute a synchronous command like `console.log()`. But in the case of an asynchronous function this is not sufficient. This use-case is not unusual: Consider the program should send an HTTP request in case of the rule application and not continue until this request has been finished.

Therefore asynchronous functions are supported too. They have a similar syntax with the difference that the function within `${ ...}` gets an additional parameter `cb` – a callback which is executed once the asynchronous function has been finished. Listing 4.6 illustrates the usage of an asynchronous built-in. By calling JavaScript's native `setTimeout(cb, 1000)`, the callback is invoked after one second, which will continue the CHR.js execution.

Listing 4.6: Example usage of an asynchronous built-in

```
1 chr`
2   hello <=> ${ (cb) => { setTimeout(cb, 1000) } }
3 `
```

The example of Listing 4.6 already gives an impression how the tracer of Chapter 5 will be implemented: The given callback is invoked as recently as the user wants to continue to the next tracing step.

Guards

In the guard of a rule only built-ins are allowed. We therefore do not need to use the `${ ...}` syntax introduced before. However it is recommended because in this way the user can make use of a great variety of existing JavaScript tools as the code within the braces is recognized as native JavaScript code. By specifying the guard directly in the string it is not possible to make use of syntax highlighting or static code analysis.

For the usage of asynchronous guards one has to use the placeholder syntax, because all directly embedded built-ins are considered synchronous.

Scope

JavaScript has function scope, that means that variables declared within this function via the `var` keyword are local to this function. The same applies for variables passed as arguments to a function.

When compiling CHR.js it is necessary to keep track of the scope of the variables. In other words, variables within a rule should not be exposed to the global scope. Then again the rules must have access to the variables of the global scope.

Listing 4.7: Scope of variables in rules

```
1 var Name = 'John'
2 chr`
3   global @ a       <=> ${ ()     => console.log(Name) }
4   local  @ b(Iden) <=> ${ (Iden) => console.log(Iden) }
5   hidden @ c(Name) <=> ${ (Name) => console.log(Name) }
6 `
```

Listing 4.7 illustrates three cases when accessing a variable `Name` resp. `Iden`. The first rule `global` has access to the globally defined `Name` variable. In the `local` rule, a fresh identifier `Iden` is used. In order to have access to local variables defined in the head in the body of a rule, the variable has to be added as an argument to the anonymous function. This is a result of the variable hiding presented in rule `hidden`, where the local `Name` variable covers the one of the global scope.

This syntax allows the manipulation of global variables from within rules, too. In the code example of Listing 4.8 the global `counter` variable is incremented in each rule application. Together with the guard this allows to restricts the rule to five applications.

Listing 4.8: Access and modification of global variables

```
1 var counter = 0
2 chr`
3   inc <=> ${ () => i < 5 } | ${ console.log(++counter) }
4 `
```

On the other hand, even the declaration of rules can depend on runtime properties. Considering the code fragment of Listing 4.9, it depends on the value of `flag` which of the rules are interpreted.

Listing 4.9: Dynamic definition of rules

```
if (flag) {
  chr`one_rule      @ a ==> b`
} else {
  chr`another_rule  @ a ==> c`
}
```

4.3.3. Rule Definition without Tagged Template Strings

Although the definition of rules with the help of tagged template string is comfortable, the CHR.js module can be equally used with ECMAScript 5 syntax. This is both useful for runtime environments which do not yet support the ES6 features of tagged template strings and arrow functions, and in some cases for an improved structure and reusability of existing code. Listing 4.10 illustrates the usage of the `chr` function, previously used as template handler, with traditional call semantics.

Listing 4.10: Example rule definitions in ES5

```
chr('a ==> b')
chr('hello ==>', function () { console.log('Hello') })

function gt0 (N) {
  return N > 0
}
chr('dec(N) <=>', gt0, '| dec(N-1)')
```

Internally this method works very similar to the tagged template approach. The given strings are concatenated again with numbered placeholders of the form `${ number}`. In this way a single string containing the whole rule is created. In the examples of

Listing 4.10, the rule in line 2 would therefore be translated to the string `'hello ==> ${1}'`, the rule of line 7 to `'dec(N) <=> ${1} | dec(N-1)'`.

4.3.4. Parsing Expression Grammar

Because both call semantics result in a very similar string, a unified grammar can be used to analyse the structure of the CHR rules.

In order to parse a given string of CHR rules, we formalise its structure using a Parsing Expression Grammar (PEG, [For04]). Its syntax is very similar to context-free grammars, EBNF or regular expressions: similar to EBNF or Regular Expressions: `/` separates alternatives, `?` indicates optional terms. We assume basic knowledge about grammars here and omit a more detailed introduction into PEG.

There already exists a PEG.js grammar to parse plain JavaScript code.[4] Because our contribution extends the JavaScript language[5], we use this JavaScript PEG grammar as a basis. Listing 4.11 presents our CHR-related extensions. The __ entity (two underscores) is used to allow whitespaces between most of the expressions, for example after commas.

Listing 4.11: Parsing Expression Grammar for CHR.js (extract)

```
1 // the overall program
2 Program
3   = __ CHRSourceElements? __
4 CHRSourceElements
5   = CHRSourceElement (__ CHRSourceElement)*
6 CHRSourceElement
7   = Rule ";"?
8 Rule
9   = RuleName?
```

[4] Part of [Maj11], https://github.com/pegjs/pegjs/blob/master/examples/javascript.pegjs

[5] As mentioned before, it is possible to include JavaScript code directly in the guard. In order to parse these expressions, a grammar for the JavaScript language is needed.

```
10      RuleWithoutName
11  RuleName
12    = RuleIdentifier __ "@" __
13  // the rule name must start with a lower-case letter
14  //   and must not contain a "@";
15  // DoubleStringCharacter is part of the JavaScript grammar
16  RuleIdentifier
17    = ($(!"@" [a-z0-9_] DoubleStringCharacter))+
18  RuleWithoutName
19    = PropagationRule
20    / SimplificationRule
21    / SimpagationRule
22  PropagationRule
23    = Constraints __
24      "==>" __
25      Guard?
26      Body
27  SimplificationRule
28    = Constraints __
29      "<=>" __
30      Guard?
31      Body
32  SimpagationRule
33    = Constraints __
34      ("\\" / "/") __  // allows both backslash and slash
35      Constraints __
36      "<=>" __
37      Guard?
38      Body
39  Constraints
40    = Constraint (__ "," __ Constraint)*
```

```
41 Constraint
42   = ConstraintName
43     ("(" __ Parameters __ ")")?
44 ConstraintName
45   = [a-z] [A-z0-9_]*
46 Parameters
47   = Parameter (__ "," __ Parameter)*
48 Parameter
49   = SimplePrimaryExpression  // defined in JavaScript grammar
50 CallParameters
51   = CallParameter (__ "," __ CallParameter)*
52 CallParameter
53   = Expression  // defined in JavaScript grammar
54 BodyConstraint
55   = ConstraintName
56     ("(" __ BodyParameters __ ")")?
57 BodyParameters
58   = BuiltIns
59 // Guard is in general a list of JavaScript expressions;
60 // the bitwise or operator "|" must be prevented here
61 Guard
62   = BuiltInsNoBitwiseOR __ "|" !"|" __
63 Body
64   = BodyElement (__ "," __ BodyElement)*
65 BodyElement
66   = Replacement
67   / BodyConstraint
68 BuiltIns
69   = Expression  // defined in JavaScript grammar;
70                 // basically all JavaScript expressions
71                 //   are allowed here
```

```
72 BuiltInsNoBitwiseOR
73   = ...          // similiar to BuiltIns, but all
74                  //   JavaScript expressions not containing
75                  //   the bitwise or "|"
76 // DecimalIntegerLiteral, FunctionExpression and Expression
77 //   are part of the JavaScript grammar
78 Replacement
79   = ReplOpenSym __ $(DecimalIntegerLiteral) __ ReplCloseSym
80   / ReplOpenSym __ FunctionExpression __ ReplCloseSym
81   / ReplOpenSym __ Expression __ ReplCloseSym
```

Parser Generation

This Parsing Expression Grammar can be used by *PEG.js* [Maj11], a parser generator for JavaScript. It allows to specify transformations using JavaScrip code blocks for each expression. With the help of PEG.js it is therefore possible to create an abstract syntax tree (AST) of the given CHR.js source code.

The transformation codes has been omitted in Listing 4.11. As an example, Listing 4.12 shows the complete parsing rule for CHR's propagation rule. Names have been added to the sub-expressions and a `{...}` was added at the end.

Listing 4.12: PEG.js code for a propagation rule

```
1 PropagationRule
2   = headConstraints:Constraints __
3     "==>" __
4     guard:Guard?
5     bodyConstraints:Body {
6       var desc = {
7         type: 'PropagationRule',
8         kept: headConstraints,
9         removed: [],
```

```
10          body: bodyConstraints,
11          guard: guard || []
12        };
13        desc = formatRule(desc);
14        return desc;
15      }
```

In this way an object is created containing the kept, removed and body constraints as properties. The array of removed constraints is empty here, because it is a propagation rule.

As part of this step, the rule is also transformed into head normal form as introduced in Section 2.5.1. The object generated when parsing the `gcd/1` CHR.js source code from Listing 4.3 is presented in Appendix B.2.

4.4. Compilation

As already mentioned in Section 4.2.4 this parser object is passed to the runtime components which generate executable code.

As a consequence to the considerations of Section 3.3 we use JavaScript promises to ensure the asynchronous behaviour of CHR.js. In other words, given the CHR.js rules for the `gcd/1` constraint of Listing 4.3 we expect to create a `chr.gcd` function which returns a promise. An example usage of this function looks like this:

```
1 chr.gcd(6).then(function () {
2   // CHR.js execution finished
3 })
```

A more advanced usage example will be presented in Section 4.5, which illustrates the advantages of using promises. The `chr.gcd` function is generated by the `Helper.dynamicCaller` function.

As described in Section 4.2.6, for each occurrence of a constraint in a rule's head a function is generated which returns a promise. With the help of `Helper.dynamicCaller`, these functions, stored in the dynamically created properties of the `Rule` prototype, are called and combined to a single promise using promise chaining.

4.4.1. Compilation Scheme for Occurrence Handlers

For each occurrence of a constraint in a rule head a separate function, which we call *occurrence handler*, is generated. In this section we present the basic compilation scheme for a single occurrence handler. It adopts the idea of [VWWSD08], which we presented in Section 2.5.3. Instead of nested loops we use JavaScript promises whenever it is possible.

Listing 4.13: Basic compilation scheme for occurrence c_i

```
1  function occurrence_c_i_j_i (constraint,replacements) {
2    var self = this  // references the CHR instance
3
4    // bind arguments of the current constraint
5    var Arg_1 = constraint.args[0]
6    ...
7    var Arg_a_i = constraint.args[a_i - 1]
8
9    // get list of possible partner constraints,
10   //   including the current constraint
11   var constraintIds = [
12     self.Store.lookup("c_1", a_1),
13     ...
14     self.Store.lookup("c_i-1", a_i-1),
15     [ constraint.id ],
16     self.Store.lookup("c_i+1", a_i+1),
17     ...
18     self.Store.lookup("c_n", a_n)
```

```
19  ]
20
21  return new Promise(function (resolve, reject) {
22    self.Helper.forEach(constraintIds,
23    function (ids, callback) {
24      if (!self.Store.allAlive(ids))
25        return callback()
26
27      if (self.History.has("name", ids))
28        return callback()
29
30      // bind variables of partner constraints
31      var C_0_0 = self.Store.args(ids[0])[0]
32      var C_0_1 = self.Store.args(ids[0])[1]
33      ...
34      var C_n_a_n = self.Store.args(ids[n][a_n])
35      // these variables are typically referenced in the
36      //   guards G_p or body constraints B_q
37
38      // define guards as array of promises
39      var guards = [
40        new Promise(function (s, j) { (G_i) ? s() : j() }),
41        ...
42        new Promise(function (s, j) { (G_m) ? s() : j() })
43      ]
44
45      Promise.all(guards)
46      .then(function () {
47        self.History.add("name", ids)
48
49        // kill the removed constraints
```

```
50          self.Store.kill(ids[k + 1])
51          ...
52          self.Store.kill(ids[n])
53
54          // call body constraints in promise chain
55          Promise.resolve()
56          .then(function () {
57            return self[B_1]()
58          })
59          ...
60          .then(function () {
61            return self[B_l]()
62          })
63          .then(function () {
64            // body has been processed, so finish
65            //   this occurrence handler
66            callback()
67          })
68          .catch(function () {
69            // reject promise if an error occurred
70          })
71        })
72        .catch(function () {
73          // invoked if at least one guard promise
74          //   has been rejected
75          callback()
76        })
77      }, resolve)  // resolve promise when all permutations
78                   //   of partner constraints have been executed
79    })
80  }
```

The generated occurrence handlers are stored in the array `chr.Rules[ruleName][functor]`. In case of the `gcd/1` CHR.js program, the occurrence handlers are therefore stored in `chr.Rules.gcd1['gcd/1]` and `chr.Rules.gcd2['gcd/1]`. The first array has only a single item, the latter two, because there are two `gcd/1` occurrences in the rule `gcd2`. By calling the `toString()` function on each it is possible to display the JavaScript code generated by the CHR.js compiler. The generated source code for the `gcd/1` example program is presented in Appendix B.3.

4.4.2. Correctness

The compilation scheme presented in the previous section follows the refined operational semantics (c.f. Section 2.3.2). It also resembles closely to the compilation scheme presented in [VWWSD08].

The occurrence handler returns a single promise which is defined to resolve only once. This is ensured by using `Helper.forEach`, which takes the `resolve` function as its last parameter in line 76 of Listing 4.13. For every permutation of partner constraints the guards are proven (line 44). Only if they are all resolved, the rule gets applied. Using the propagation history `self.History`, we prevent multiple applications of the same rule with identical constraints.

The promise chain of lines 54-66 guarantees the body constraints to be added in the correct order. The callback specified with the help of `then` is not fired before the application of the constraint B_q has been finished.

By using promises we also prevent the exceeding of the stack size as described in Section 3.3. The use of classical recursive functions would result in JavaScript range errors for large input values.

4.4.3. Termination

The code generated by the presented compiler is not guaranteed to terminate. This is a result of the fact that it is easily possible to define non-terminating CHR rules, for example:

```
1 chr`a <=> a`
```

Although the `a/0` always looks the same, the actual constraint instances are not identical, which is why the propagation history does not prevent the non-termination here.

4.5. Advanced Usage

To conclude this chapter, we want to give an advanced usage example of the CHR.js interpreter. The basic setup of the examined constraint solver is presented in Listing 4.14.

Listing 4.14: Advanced usage example of CHR.js

```
1  var CHR = require('chr')
2  var chr = new CHR()
3  var start
4
5  function elapsedTime () {
6    return new Date().getTime() - start
7  }
8
9  var latencyA = 300
10 var latencyB = 500
11
12 chr.Store.on('add', function (constraint) {
13   console.log(
14     constraint.toString() + ' [' + elapsedTime() + 'ms]'
15   )
```

```
16 })
17
18 chr`
19   r1 @ a(N) ==> N < 5 | a(N+1)
20   r2 @ b(N) ==>
21     N < 5 | ${ (cb) => setTimeout(cb, latencyB) }, b(N+1)
22 `
23
24 chr.Rules.r1.Breakpoints.onTry = function (data, next) {
25   setTimeout(next, latencyA)
26 }
27
28 module.exports = chr
```

With the code example of Listing 4.14 we want to discuss the general usage of CHR.js programs as well as different call semantics for promises. It contains most of the features presented in the previous sections.

In the first two lines, the constraint solver is instantiated and declared as `chr`. Line 25 is specific to node.js. It exports the create solver as a module, so it can be easily used in existing projects.

Since we want to track the execution time of the CHR.js program, we define a variable `start` in line 3. The function `elapsedTime()` returns the elapsed time since the start in milliseconds. In lines 9-10 constants are defined that will be later used to pause the rule applications.

To get notified as soon as a new constraint is added to the constraint store, in lines 12-16 a listener for the `'add'` event of the constraint store is defined. The actual definition of CHR rules happens in lines 18-22. The rule `r1` takes a `a/1` constraint and if its argument is less than 5, its successor is propagated. Rule `r2` works very similar but executes JavaScript's `setTimeout` function first. As a result the `b(N+1)` is propagated not until `latencyB` milliseconds have been elapsed.

A similar behaviour is also achieved for the `a/1` constraints of rule `r1` after defining the `onTry` breakpoint in lines 24-26. However this breakpoint pauses the CHR.js execution as well as if the rule `r1` is not applicable in case of an unsatisfied guard, which is not the case for `r2`.

4.5.1. Sequential Execution

The classical call syntax is to chain the promises as presented in Listing 4.15.

Listing 4.15: Sequential execution syntax

```
1  start = new Date().getTime() // start start time
2  chr.a(0)               // first add a(0)
3  .then(function () {
4    return chr.b(0)      // and then b(0)
5  }).then(function () {
6    console.log('Execution finished [' + elapsedTime() + 'ms]')
7  })
```

This results in the following output as expected:

Listing 4.16: Output of the sequential execution of Listing 4.15

```
1  a(0) [24ms]
2  a(1) [325ms]
3  a(2) [626ms]
4  a(3) [928ms]
5  a(4) [1230ms]
6  a(5) [1534ms]
7  b(0) [1535ms]
8  b(1) [2036ms]
9  b(2) [2537ms]
10 b(3) [3039ms]
11 b(4) [3540ms]
```

```
12 b(5) [4040ms]
13 Execution finished [4541ms]
```

First the `a(0)` constraint is handled until its execution finishes with the creation of `a(5)` after 1.5 seconds, then `b(0)` is processed in the same way in 2.5 seconds. The shown times match the specified latencies: 300 milliseconds on each application of `r1`, 500 milliseconds for `r2`. Note that program is finished another 500 milliseconds after the last constraint addition, because the `latencyB` is used in the `onTry` breakpoint even if the related guard fails.

4.5.2. Parallel Execution

The sequential usage is the usual way to use CHR.js. However due to the fact that our contribution uses non-blocking JavaScript promises, we can process multiple constraints concurrently. JavaScript natively provides a `Promise.all(arrayOfPromises)` function which creates a new promise based on an array of promises. It is resolved as soon as all of the given promises have been resolved. The appropriate call syntax for this parallel execution is illustrated in Listing 4.17.

Listing 4.17: Parallel execution syntax

```
1 start = new Date().getTime()
2 Promise.all([
3   chr.a(0),
4   chr.b(0)
5 ]).then(function () {
6   console.log('Execution finished [' + elapsedTime() + 'ms]')
7 })
```

Here, the time in which the execution of a rule is paused due to the asynchronous `setTimeout` function is used by the event loop to apply other possible rules. So for the parallel execution the output looks like this:

Listing 4.18: Output of the parallel execution of Listing 4.17

```
1  a(0) [13ms]
2  b(0) [13ms]
3  a(1) [314ms]
4  b(1) [514ms]
5  a(2) [615ms]
6  a(3) [916ms]
7  b(2) [1015ms]
8  a(4) [1216ms]
9  b(3) [1516ms]
10 a(5) [1518ms]
11 b(4) [2017ms]
12 b(5) [2518ms]
13 Execution finished [3020ms]
```

Because of the smaller latency, the `a/1` constraints are generated in shorter intervals. The overall execution time drops to 2.5 seconds, the overall execution time of `b(0)`, plus another 500 milliseconds for the last `onTry` breakpoint of rule `r2`. At the end, this parallel execution results in a better execution time because the event loop perfectly fills empty capacities if a single rule is paused.

4.6. Summary

In this chapter we presented the most important part of our contribution: the CHR.js JavaScript module, a just-in-time compiler and runtime environment for CHR embedded in JavaScript. Compared to existing CHR systems it adds a large number of tracing and debugging options as well as the possibilities to manipulate CHR rules on-the-fly.

The execution semantics presented in Section 4.5 illustrate the benefits of using JavaScript promises to model the rule application in an asynchronous way. The proper-

ties of CHR programs that are executed in parallel are only marginally investigated so far. CHR.js can therefore be a tool to examine these properties empirically.

Nevertheless this flexibility comes at a price: The execution time of the just-in-time compiled code is not comparable to existing CHR systems so far. This originates from the asynchronous approach, which has minimal latencies as described in Section 3.4.3. All browser JavaScript implementations add a latency of four milliseconds for each asynchronous call, which will have effects on the execution time in many test cases that are used for performance comparisons.[6] Also it is clear, that the asynchronous execution using the message queue can never beat a single-stack approach.

In Chapter 6 we will present a second compilation scheme which goes the other way around and completely avoids the message queue for improved performance. This compilation scheme is used by the precompiler, which on the other hand produces code not as flexible as the CHR.js module.

[6] This applies only to HTML5 documents, which is why these latencies are not included in the timestamps presented in the outputs of Section 4.5, which has been executed in node.js.

5

chrjs.net – Web-Based Tracing for CHR

I hate almost all software. It's unnecessary and complicated at almost every layer. (...) The only software that I like is one that I can easily understand and solves my problems. The amount of complexity I'm willing to tolerate is proportional to the size of the problem being solved.
The only thing that matters in software is the experience of the user.

— Ryan Dahl, Creator of node.js[1]

[1] I hate almost all software (2011), Google Plus blog post originally located at `https://plus.google.com/115094562986465477143/posts/Di6RwCNKCrf`, archived version at `https://github.com/ponyfoo/articles/blob/master/reference/ryan-dahl-rant-quote.md`

As an example application of the CHR.js module, our contribution includes an interactive web-based tracer for CHR. This project, which has the name *CHR.js-website* provides a graphical editor for Constraint Handling Rules. It displays the constraint store and several tracing outputs in a user-friendly way. A public, hosted instance is online available at `http://chrjs.net`.

In opposite to existing web applications that can be used to execute CHR in the browser – namely WebCHR [Kae07] and Pengines [LW14] –, our contribution is a standalone HTML website which does not require any server-side framework. The complete processing, from compilation to execution of CHR, is done in the client's web browser. This example application therefore demonstrates the strengths of CHR.js very well.

In this chapter we first present the *CHR.js-website* application. Section 5.1 introduces the user interface and features of the *CHR Playground*, which is the central page of the project. In Section 5.2 we present its architecture, with a special focus on modifications and concrete applications of the used CHR.js module. Because this thesis focusses on the translation of CHR into JavaScript, we omit an in-depth look in anything related to the user interface, that means the general website structure and the used HTML and CSS. The readers interested in these technologies, can find more information in the project's sources (c.f. Appendix A).

5.1. The CHR Playground

The *CHR.js-website* provides an online playground to explore CHR. It is inspired by collaborative code sharing platforms like JSFiddle[2], JS Bin[3] and RequireBin[4]. They all have in common that their user interface is dominated by a large text field to work on source code. Similar to this, our contribution consists of a webpage to edit CHR programs and display relevant information like the current content of the constraint store and tracing output.

[2] `http://jsfiddle.net/`
[3] `http://jsbin.com/`
[4] `http://requirebin.com/`

5.1.1. Screenshots

Figure 5.1 presents a screenshot of the created website. On the left-hand side the code editor is used to edit Constraint Handling Rules. This section can also be used to define helper functions in native JavaScript in a special code block. This syntax is presented in Section 5.2.1.

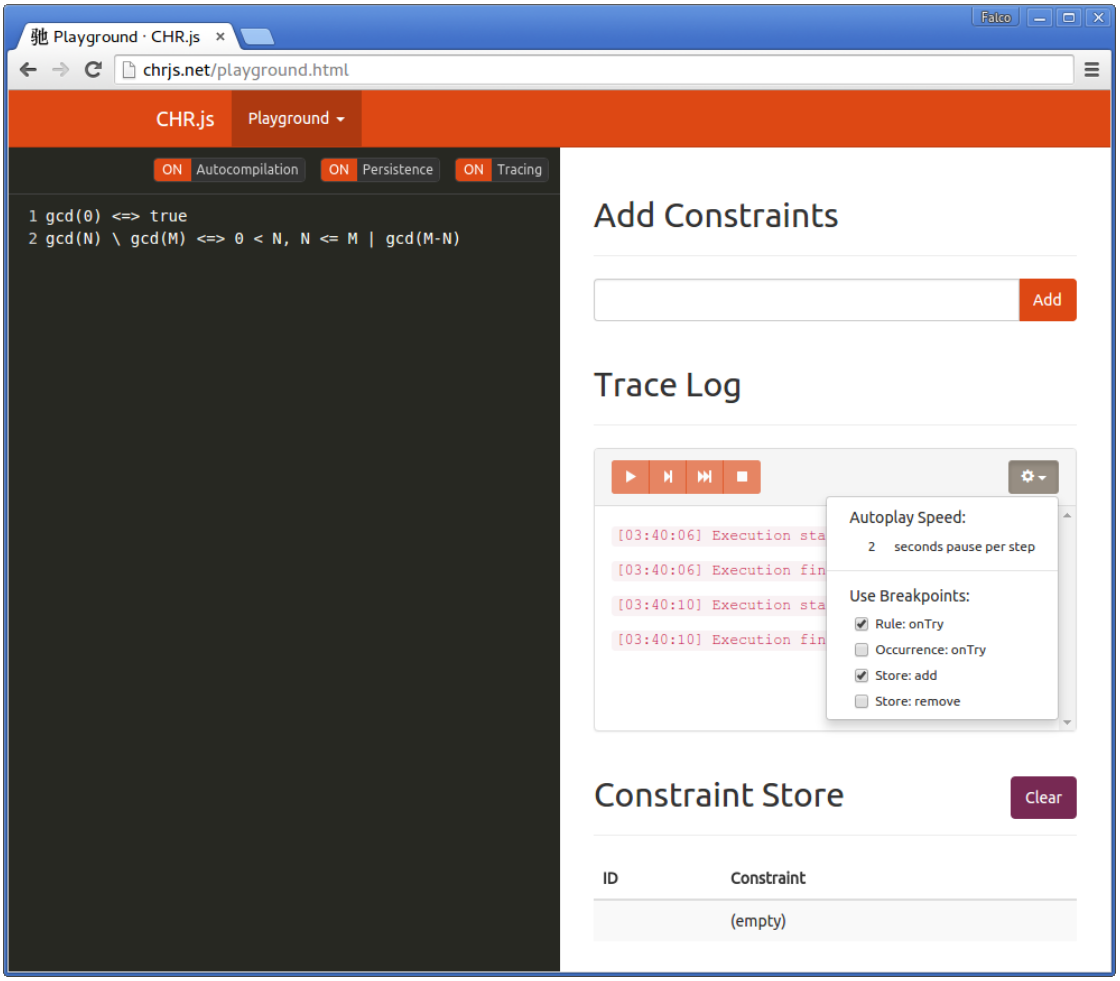

Figure 5.1.: Screenshot of `chrjs.net` with tracing enabled

The right-hand side contains all information related to the execution of the defined CHR program: In the first block queries can be specified. The optional tracing output is followed by a table which presents the current content of the constraint store.

Figure 5.2 shows the mobile view of the CHR Playground. The user interface remains usable even on small screens, so that CHR can be used from mobile phones as well.

5.1.2. Implementation Goals

Because the CHR Playground is heavily inspired by existing collaborative code editing tools, we want to support similar use-cases. Our implementation goals are:

Real-time Compilation Using CHR.js there is no need to reload the webpage on each change in the CHR source code. Instead our aim is to provide a coding experience similar to local development. By compiling the CHR code instantly, the user should be able to process new constraints without any further action.

Real-time Execution This leads to the next requirement: Just like the compilation does not require a page reload, the new constraints should be added and processed instantly.

Consistent Constraint Store By compiling changed CHR code instantly, it is possible to loose the tight coupling of the code and the constraint store. Our aim is to be able to change the code and constraint store independently.

Easy Sharing To allow an easy mechanism for collaboration, we want to integrate a sharing service. If a CHR program is valid, a unique `http://chrjs.net` URL to directly link to this program should be created.

Figure 5.2.: Screenshot of `chrjs.net` in mobile view

5.2. Architecture

The *CHR.js-website* uses Jekyll[5] as a static site generator. With Jekyll it is possible to use templates, for example for the top menu bar. At the end it still produces static HTML sites, so no server-side language is needed.

5.2.1. Adapted Grammar

The most important component of the CHR Playground is the CHR.js module presented in Chapter 4. However the specification of CHR rules is inverted: Instead of explicitly call the `chr` template handler respectively function, we want to make the definition of rules in the Playground as intuitive as possible. The CHR rules are the essential part, so simply putting a rule like `a ==> b` in the code editor should be accepted.

On the other hand, if `a ==> b` becomes a valid expression in the Playground, we have to provide a special syntax to define native JavaScript functions and expressions that can be used along rules. This is achieved by the *preamble*, a JavaScript code block at the very beginning of the rules. The syntax is inspired by PEG.js: JavaScript expressions are encapsulated in `{...}`, a block of curly braces.

We therefore extend the general CHR.js grammar (c.f. Section 4.3.4) to work with a given preamble. Instead of the `Program` rule, we define a new PEG element `ProgramWithPreamble`. Its definition is presented in Listing 5.1.

Listing 5.1: Parsing Expression Grammar for the CHR Playground

```
1 ProgramWithPreamble
2   = __ Preamble? __
3     Program          // defined in CHR.js grammar
4
5 Preamble
6   = ReplacementOpeningSymbol __
7     PreambleSource __
```

[5] Project website: `http://jekyllrb.com/`

```
8     ReplacementClosingSymbol
9
10  PreambleSource
11    = SourceElements  // defined in JavaScript grammar
12    / __              // allowed to be empty
```

5.2.2. Parallelization Using Web Workers

Both parsing and compilation of CHR rules as well as execution of a CHR program can take some time. For source code which is not valid according to the grammar presented in Section 5.2.1 the PEG.js parser is not even guaranteed to terminate. The same applies for the execution of a CHR program. As illustrated in Section 4.4.3 a non-terminating program can be easily defined.

Therefore we have to prevent blocking the browser's event loop by non-terminating or even prolonged processes. Because JavaScript is single-threaded, the user is not able to interact with the website's user interface until the execution has been finished. In many cases this will crash the complete web browser.

Modern browsers thus provide a mean to run scripts in background threads: *web workers*. They can be used for tasks which do not need to interact with the user interface. In our contribution, we outsource two common and time-consuming tasks into web workers: the parsing of user's source code and the actual execution of the CHR program.

To separate these tasks, CHR.js provides the following browser-specific builds (in the project's `/dist` directory) to fit in as many architectures as possible:

`chr.js` (200 kB) The complete CHR.js module as a standalone file.

`chr.min.js` (135 kB) The minified version of `chr.js`.

`chrparser.js` (135 kB) A standalone module that exposes a single function to parse strings according to the (adapted) CHR.js Parsing Expression Grammar.

`chrparser.min.js` (95 kB) The minified version of `chrparser.js`.

`chr-wop.js` (40 kB) The CHR.js module without anything related to the parsing. Any CHR.js function which expects a source string also works by directly providing the AST object created by the parser.

Using the `chr-wop.js` and `chrparser.min.js` scripts, the parsing and execution tasks have been outsourced to web workers. A JavaScript controller coordinates these child processes and respawns a new instance as soon as it exceeds a specified time.

5.2.3. Limitations

The usage of web workers comes with some limitations. As mentioned before it is not possible to interact with the website's user interface. While this is not a typical use-case for the CHR Playground, the restricted access to other global objects has more practical relevance: As of yet, one can not use built-ins in CHR rules which need references to the `window`, `global` or `console` object. In our experience, this currently prevents the usage of JavaScript's `alert()` and `console.log()` functions. Because these are in particular functions often used in a testing environment, our implementation contains workarounds. Unfortunately they are not supported by all major browsers so far.

5.3. Interactive Tracing

Some of the design decisions made in the development of the CHR.js module are based on the needs of the CHR Playground: The `onTry` Breakpoints of Rules, introduced in Section 4.2.4, were originally modelled as event emitters similar to the `add` and `remove` events of the `CHR.Store` prototype. Because of the need for a tracer, which should continue only if the user clicks a button, a non-blocking event handler is not sufficient. Therefore all breakpoint functions receive a callback as their last parameter, which should be called as soon as the execution is resumed.

The graphical tracer implemented in the CHR Playground currently highlights the used rule and tried occurrence. Additionally the added and removed constraints are logged.

The tracer supports two modes: It can be executed either manually, which means that the user can pause and resume the execution of a CHR program with appropriate buttons or in an auto-play mode where each execution step pauses a predefined amount of time. It is possible to switch between these modes at any time.

With the help of the tracer it is even possible to execute originally non-terminating CHR programs. They can easily be aborted from the user interface.

5.4. Future Work

Using web workers to run the CHR.js execution in separate thread leads to a consideration we already formulated in Section 4.5.2: How can multiple CHR programs be executed in parallel? To improve the performance of a web-based CHR systems, the usage of web workers is an easy way to elude JavaScript's single-thread execution cycle.

However this consideration is not necessary for the CHR Playground `http://chrjs.net`: Its main focus is the easy exploration of CHR programs and their properties, it is unlikely that the *CHR.js-website* project will be used for actual, CPU-intensive calculations. More often it will be used to interactively execute small CHR programs.

To improve the user experience, the code editor should have built-in syntax highlighting for CHR. The author of this work already started the development of a language extension for the used graphical code editor *CodeMirror*. The used syntax to specify the syntax highlighting of a new programming language is in fact another grammar definition. Unfortunately *CodeMirror* does not support Parsing Expression Grammars so far. By adding support for PEG we could easily integrate syntax highlighting for CHR.js into the used online code editor.

5.5. Summary

This chapter presented the web-based CHR tracer `http://chrjs.net`. It allows the specification and execution of Constraint Handling Rules directly in the web browser. It

therefore has the potential to increase the popularity of CHR in general. It is also very comfortable for users of existing CHR systems as they can easily adapt their existing CHR programs CHR.js, because the CHR Playground is only a low threshold compared to the installation requirements of other CHR systems.

We concentrated on the ideas and general architecture behind the example application. A main goal for future improvements is the definition of syntax highlighting for CHR. From this development other CHR system would benefit as well.

6

Benchmarks and Optimizations

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified*
>
> — Donald Knuth, in The Art of Computer Programming (1968)

At the end of Chapter 4 we already looked into the execution time of our implementation. While in Section 4.5 the different call semantics using promises – sequential and parallel – have been discussed, we want to challenge the performance of the CHR.js runtime at all. Therefore in Section 6.2 we take up again the different implementations techniques presented in Section 3.4. This is the basis for the decision, which execution

model (synchronous/asynchronous or promises) fits best for a more efficient CHR.js precompiler.

In Section 6.3.3 we present the basic compilation idea for a CHR.js precompiler. The programs generated by this precompiler are compared to existing CHR systems. In Section 6.4 the result of the benchmark is discussed.

6.1. Benchmark Setup

Before we present the results of several benchmarks, we want to describe the test set-up. Our benchmark suite called `CHR-Benchmarks` is based on [WSD07] and the CCHR implementation[1]. It can be used to compare the execution times of CHR systems with regard to the problem size.

Installation and usage instructions can be found in Appendix C.3. We used the `CHR-Benchmarks` suite to compare the different CHR systems as well as different JavaScript implementation ideas.

The system versions were:

- JCHR: *1.5.1*
- SWI-Prolog: *6.6.4*
- CHR.js: *2.0, with node.js of version 4.0*
- CCHR: *(No version provided)*
- C: *GCC of version 4.84*

All benchmarks are run on an Intel Core i7 9xx Dual Core CPU with 8 GB of RAM, using Ubuntu Server 14.04.3 64bit (Linux Kernel 3.13.0). No other background jobs have been executed.

Basically the 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. The latter will be used to compare the different implementations.

[1] Copy available at `https://svn.ulyssis.org/repos/sipa/cchr/`

6.2. Comparison of `gcd(a,b)` Implementations in JavaScript

In Section 3.4 we already examined different implementation ideas for the calculation of the greatest common divisor. To recall, the presented techniques are:

- Iterative implementation (Listing 3.7)
- Recursive implementation (Listing 3.8)
- Asynchronous implementation (Listing 3.10)
- Asynchronous implementation with moderate use of `setTimeout()` (Listing 3.11)
- Implementation using promises (Listing 3.21, in Section 3.5.6)

Because the asynchronous implementation of Listing 3.10 is very similar to the recursive implementation (the calculated result gets passed in a callback instead of directly returned), it has very similar properties: comparable execution time and the problem of exceeding the stack size very fast. Therefore we omit the asynchronous implementation in our comparison.

The four implementations have been executed in order to calculate `gcd(5,1000*N)`, with `N` the problem size. The recursive implementation exceeds the stack size limit for `N = 60`, which is why the 10 seconds limit never gets reached.

The results of the benchmarks are presented in Figure 6.1. As expected, the iterative implementation is by far the fastest. The asynchronous implementation with moderate use of `setTimeout()` is similar to the iterative implementation, because the `setTimeout()` function is called only every 10000'th iteration. This number depends on the system's hardware capabilities.

The promise-based and recursive implementations are the slowest, with more than four orders of magnitude difference to the iterative implementation. As a result, the asynchronous implementation model used in CHR.js seems to be comfortable but not efficient at all. With the use of the compilation scheme presented in Section 4.4.1, it is not possible to be competitive with existing CHR systems.

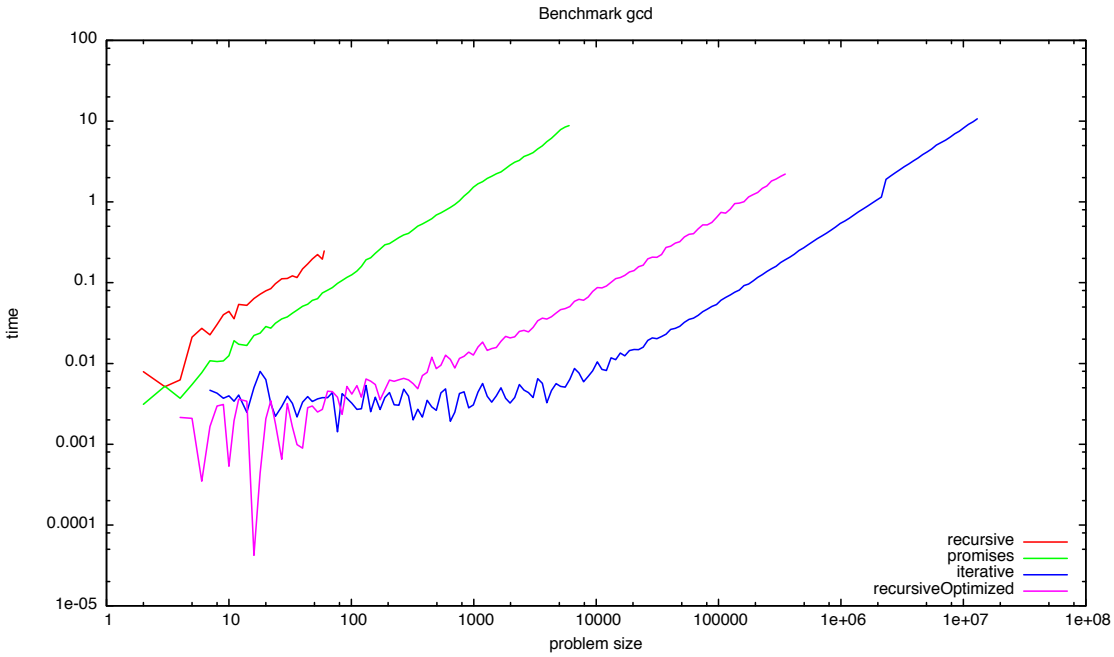

Figure 6.1.: Benchmarks of the different `gcd(a,b)` implementations in JavaScript

6.3. Precompile CHR.js Source Code

To improve the execution time, we created an ahead-of-time compiler for CHR.js code. It takes JavaScript source code with embedded CHR.js rules and precompiles the rules to native JavaScript. In this way, the translation process is invoked only once, whereas CHR.js' just-in-time compilation is executed for each call of the script.

In addition to this, the compilation scheme can be optimized, because unlike with just-in-time compilation, all CHR rules are known at compilation time.

6.3.1. Babel – A JavaScript Transpiler

When ECMAScript 6 was released in June of 2015, some new language features were already implemented in web browsers and even used in JavaScript projects in production. This was only possible because of *transpilers* – programs, that replace new language features by standardized, widely supported older ES5 syntax. The most

popular JavaScript transpiler, Babel[2], is widely adopted and as of today used by a large number of companies, including Facebook, Yahoo and Apple.

We created a plugin for Babel called *babel-plugin-chr*. Its installation and usage instructions can be found in Appendix C.4. This plugin replaces the `require('chr')` and `new CHR()` expressions as well as all applications of the `chr()` function respectively template handler. As a result, the transpiled code does not contain any references to CHR.js but instead directly contains all compiled CHR rules and the runtime components.

We do not get into detail about the plugin source codes here. It basically searches for the expressions mentioned above in a given abstract syntax tree and replaces them by optimized CHR.js source code.

6.3.2. Trampolines

As presented in Section 6.2, the usage of promises results in a large performance decrease. We therefore avoid the usage of promises in our compilation scheme for the precompiler.

Instead, an idea presented in [VWWSD08] is used: the usage of *trampolines*. In general, a trampoline is a loop that iteratively invokes functions. That means, a global loop asks for the next to be performed function and executes it. The program is finished as soon as there is no function to be invoked.

Basically this results in a user-defined stack. The global loop mentioned before asks for elements on the stack, while functions can add new ones. We adapt this mechanism to create a global stack of constraints that should be processed. In this way the functions generated by the compiler simply add elements to the stack but do not need to be called recursive.

The basic idea of this trampoline is presented in Listing 6.1.

[2]Project Homepage: `http://babeljs.io/`

Listing 6.1: CHR trampoline, used by `babel-plugin-chr`

```
var chr = {
  Store: new Store(),
  Tells: []
}

function tell () {
  var current
  while (chr.Tells.length > 0) {
    current = chr.Tells.pop()
    if (current.type === 'constraint') {
      chr.Store.add(current.constraint)
    }
    dispatchTell(current)(current)
  }
}
```

In `chr.Tells` an array of constraints that have to be handled is stored. It behaves as a classical stack: new constraints are added to the top of the stack, while the top-most is used as the next element. By using this mechanism, we also ensure the correctness according to the refined operational semantics ω_r: As soon as a constraint is added as part of the body of a rule, it is processed immediately.

However, this requires to save the current program state as the next stack element equally. We therefore define a new JavaScript prototype `State` as presented in Listing 6.2. It contains the currently examined constraint as well as information about the program properties, that means which occurrence handlers have been executed (`step`), which are the current partner constraints (`lookup`) and what is the variable scope. With these informations it is possible to return to continue a previous step in the execution.

Listing 6.2: Definition of the `State` prototype

```
function State (constraint, type, step, lookup, scope) {
  this.type = type
  this.constraint = constraint
  this.step = step
  this.lookup = lookup
  this.scope = scope
}
```

6.3.3. Basic Compilation Scheme using Trampolines

The compilation scheme is similar to the one using promises in Section 4.4.1. Instead of promise handling we manage the trampoline stack `CHR.Tells`.

Listing 6.3: Trampoline compilation scheme for occurrence c_i

```
function c_i_j_i (current) {
  var lookup, scope, constraints

  // check if this is an 'intermediate' state, which
  //   means we have to restore the State entities
  if (current.type === 'intermediate') {
    lookup = current.lookup
  } else {
    // otherwise the Lookup for partner constraints
    //   have to be created
    lookup = new Lookup(
      chr,
      [
        { name: 'c_1, arity: a_1 },
        ...
        { name: 'c_{i-1}', arity: a_{i-1} },
```

```
17        { constraint: current.constraint },
18        { name: 'c_{i+1}', arity: a_{i+1} },
19        ...
20        { name: 'c_n', arity: a_n }
21      ]
22    )
23  }
24
25  // bind arguments of the current constraint
26  scope = {
27    Arg_1: current.constraint.args[0],
28    ...
29    Arg_a_i: current.constraint.args[a_i-1]
30  }
31
32  if (/* guards that depend only on Arg_1 ... Arg_a_i */) {
33    // get next permutation of partner constraints
34    while (constraints = lookup.next()) {
35      // bind variables of partner constraints
36      scope.C_0_0 = constraints[0].args[0]
37      scope.C_0_1 = constraints[0].args[1]
38      ...
39      scope.C_n_a_n = constraints[n - 1].args[a_n]
40
41      if (!(/* remaining guards */)) {
42        continue
43      }
44
45      // kill the removed constraints
46      constraints[k + 1].kill()
47      ...
```

```
48        constraints[n].kill()
49
50        // save current state
51        chr.Tells.push(
52          saveState(current.constraint, a_n, lookup, scope)
53        )
54
55        // add body constraints
56        chr.Tells.push(
57          constraintState(B_1),
58          ...
59          constraintState(B_l)
60        )
61
62        // continue with these constraints
63        return true
64      }
65    }
66  }
```

The used `saveState` and `constraintState` functions generate the appropriate `State` instances. The referenced `Lookup` prototype is similar to the `Helper.forEach` function. For better optimizations in the JavaScript runtime environment we use a new prototype here.

For each constraint, a new function is created that executes all occurrence handlers that are generated according to the previous scheme.

The generated precompiled source code for the `gcd/1` example of Listing 4.3 is presented in Appendix B.4. Although it does not use promises, we generated a `then`-able function which imitates a promise. As a result, the generated code can be used similar to normal CHR.js code. With one exception: All built-ins have to be synchronous, because the occurrence handler now have to be blocking.

6.4. Benchmark Result

The generated code using trampolines is significant faster than the promise-based implementation. We compared our solution to existing CHR systems, executing the `gcd/1` problem on each. The problem to calculate the greatest common divisor fits very well as a test, because it is a linear program involving at most two constraints. Our implementation goal was to generate fast occurrence handlers – runtime components for better constraint lookup or propagation history indexing can easily be changed.

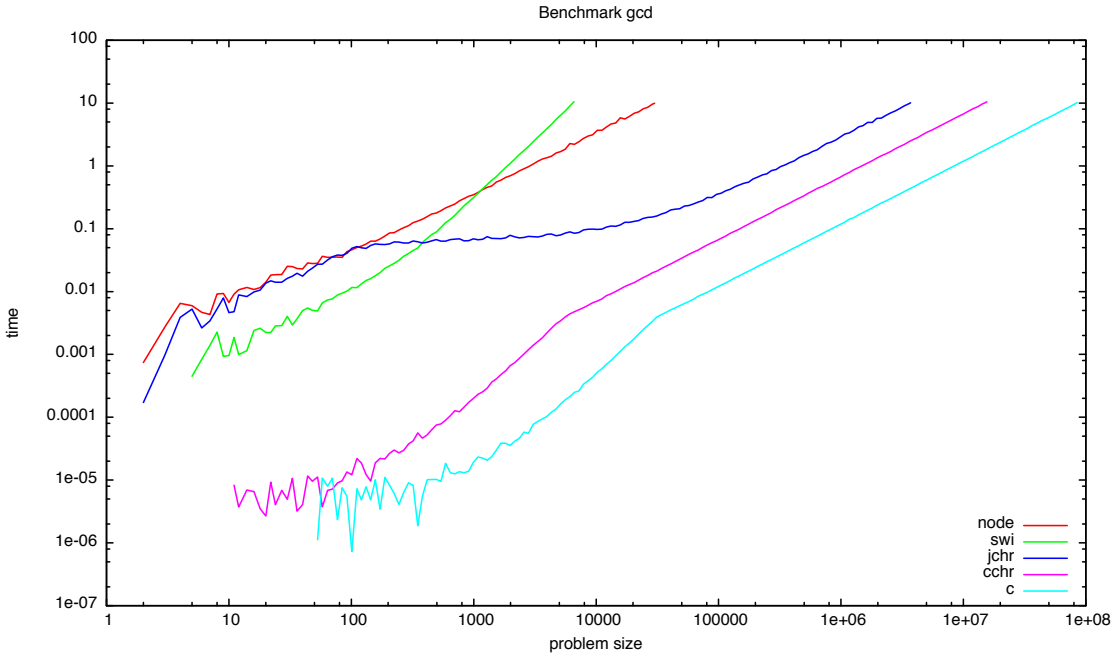

Figure 6.2.: Benchmark for `gcd/1`

Figure 6.2 illustrates the benchmark results. The precompiled CHR.js implementation in node.js is at the beginning similar to JCHR. With bigger problem sizes, it is faster than SWI-Prolog. All in all it seems to be competitive with existing CHR systems. This is in particular remarkable, as it translates CHR into JavaScript, which is just-in-time compiled by V8 as well. In opposite to JCHR and CCHR, there is no ahead-of-time compilation of this JavaScript code at all.

6.5. Summary

In this chapter we introduced an alternative compilation scheme based on trampolines. Because promises are a relative slow mean to handle asynchronous functions, we switched to a global stack `CHR.Tells` to store new constraints and intermediate execution states. This prevents exceeding the stack size because of recursive functions, but it also loses the ability to use asynchronous built-ins in the rule's guard or body. As second disadvantage, the improved execution time comes with the price of losing the flexibility to dynamically change the rule set.

Our implementation is comparable to the existing SWI-Prolog CHR system. The compiled versions using JCHR and CCHR are still three orders of magnitude faster than our precompiled version.

7

Conclusion

In this chapter the results of the thesis are summarized and future improvements of the `CHR.js` software package are discussed.

7.1. Summary

In this work we presented the two programming languages Constraint Handling Rules and JavaScript. Our aim was to create a new system for the usage of CHR in JavaScript. Therefore we introduced the syntax and semantics of CHR in Chapter 2, along with the basic compilation schemes for existing CHR(Prolog), CHR(Java) and CHR(C) systems.

In Chapter 3 we gave an introduction in the target programming language JavaScript. We defined the targeted JavaScript runtime environments and presented its execution cycle, the event loop. Unlike other existing CHR systems, this leads to the differentiation

whether a function is synchronous or asynchronous. These two choices will also dominate our implementation approaches, because the original CHR.js library will completely will support both type of functions, whereas its precompiler can be only used with synchronous functions for better performance.

This Chapter also includes different example definitions for the `gcd(a,b)` program. They are used to evaluate the different techniques to structure possibly asynchronous functions. At the end of Chapter 3 multiple JavaScript constructs are introduced, which are the basis for the compiler construction.

Chapter 4 contains the main part of our contribution: The definition of the CHR.js module, a customizable CHR(JavaScript) interpreter. All program properties are defined using generalized JavaScript prototypes, which allows the modification of CHR rules even at runtime. The promise-based compilation scheme has been presented in Section 4.4.1.

An example application that uses most of the features provided by CHR.js is the web-based tracing utility `chrjs.net`, which was presented in Chapter 5. It allows the compilation and execution of CHR in a *CHR Playground*, a standalone HTML-only website.

In the last Chapter 6 we introduced a second compilation scheme based on trampolines instead of promises. Although it is reasonable faster than the CHR.js JIT-compiled version, it loses most of its features. With modifications its execution time is comparable to the one SWI-Prolog.

7.2. Conclusion

The precompiled version of CHR.js has been highly optimized against the V8 JavaScript engine that is used in node.js and Google Chrome. We used several tools to generate code, that can be optimized by V8's just-in-time JavaScript compilation.

Figure 7.1 shows the CPU file recorded by the Google Developer Tools to track the execution of the generated `gcd/1` code. All functions have been rewritten to be able to be optimized using V8's hot-function detection resulting in very efficient code. Even with

Figure 7.1.: Screenshot of the Google Developer Tools

these optimizations our implementation is just competitive with SWI-Prolog. The CHR systems JCHR and CCHR are faster so far.

However the implementation goals listed performance only as the last point. We wanted to create a feature-rich CHR(JavaScript) system which can be used in browser-based environments – and `CHR.js` fits very well. With the abilities to even modify CHR rules at runtime, it can be basis for further investigations of CHR properties. Especially examining properties of running CHR in parallel as presented in Section 4.5 seems to be worth performance decreases resulting from the promises-based execution approach. Our contribution focuses on user-friendliness and expressiveness instead of high performance.

7.3. Future Work

As a result of this considerations, not the improvement of the compilation process is of the highest priority when it comes to future improvements. To make better use of the CHR.js system, we suggest the implementation of syntax highlighting for *CodeMirror*, the online code editor for the CHR Playground. From this highlighting several other (even offline) editors could profit when developing CHR.

It is also worth considering the unification of the two presented compilation schemes. Two approaches seem reasonable: The precompiler could analyse the rules to decide

whether they contain only synchronous or asynchronous functions. This could result in a unified compilation scheme. However detecting whether a function is synchronous or asynchronous is something no static analyse can achieve so far, so this would result in a more explicit syntax for CHR.js.

The second method is the *lifting* of precompiled CHR.js code: It is worth considering, that CHR.js provides a mean to import already precompiled CHR rules in its JIT-compilated runtime environment. In this way, one could use precompiled CHR.js source code for better performance and import it to CHR.js if the modification of rules is needed.

A

Available Sources

The sources of this thesis and the created implementations are provided as a CD along with this thesis. Additionally the sources of the implementations are available online at GitHub. They are released under the terms of the license specified in the project's repository or if not specified, under the MIT license.

Thesis The thesis is located in the `/thesis` directory. It contains the LaTeX source files. The base document is `thesis.tex`. Make sure to install all packages needed for compilation:

- `tikz`
- `mathdots`
- `chngcntr`

Implementation All sources of the CHR.js module and the created benchmark and compilation tools are located at the `/implementation` directory of the CD. They are additionally released as open source. For an easy usage, all but the benchmark tools have been published on npm, the de-facto package manager for node.js.

CHR.js The CHR.js module is located at `/implementation/CHR.js`. It is also available at GitHub at `https://github.com/fnogatz/CHR.js`. On npm, which only support lowercase letters in names, it has been named `chr` (`https://www.npmjs.com/package/chr`).

In addition to the usage guide given in Chapter 4, a manual with further usage and developments instructions is provided in Appendix C.1.

CHR.js Babel Plugin The JavaScript module that pre-compiles CHR.js source code, as presented in Chapter 6, is called `babel-plugin-chr`. This follows the common naming convention for Babel plugins. It is located at `/implementation/babel-plugin-chr` and published on npm. Its GitHub repository is located at `https://github.com/fnogatz/babel-plugin-chr`.

chrjs.net Sources The source code for the web-based CHR.js tracer (Chapter 5) currently located at `http://chrjs.net` is put into the `/implementation/CHR.js-website` directory. Its GitHub repository is located at `https://github.com/fnogatz/CHR.js-website`. It depends on the static site generator *jekyll*. The CD contains a compiled version in the `/implementation/chrjs.net-static` directory.

CHR Benchmarks The benchmark tool used for the evaluation presented in Chapter 6 are located at `/implementation/CHR-Benchmarks`. Because it contains sources of CHR systems of other authors, it is not yet released on GitHub. However, the publishing is intended in future.

B

Code Examples

In this appendix we present some important functions and their output in detail.

B.1. Example Usage of `Helper.forEach`

The following Listing B.1 illustrates the behaviour of the runtime helper function `Helper.forEach` as introduced in section 4.2.6.

Listing B.1: Example Usage of `Helper.forEach`

```
1 var forEach = require('chr/runtime').Helper.forEach
2
3 function iteratorCallback (permutation, next) {
4   console.log(permutation)
5   next()
```

```
}

function endCallback () {
  console.log('Finished execution')
}

forEach([[1,2],[3],[4,5,6]], iteratorCallback, endCallback)

// Result:
/* [ '1', '3', '4' ]
   [ '1', '3', '5' ]
   [ '1', '3', '6' ]
   [ '2', '3', '4' ]
   [ '2', '3', '5' ]
   [ '2', '3', '6' ]
   Finished execution */
```

B.2. PEG.js Parsed Program `gcd/1`

The following Listing 4.12 shows the object created by the PEG.js parser for the `gcd/1` program of Listing 4.3. It uses the grammar introduced in section 4.3.4.

Listing B.2: Object generated by the PEG.js Parser

```
{
  "type": "Program",
  "body": [
    {
      "type": "SimplificationRule",
      "kept": [],
      "removed": [
        {
          "type": "Constraint",
          "name": "gcd",
          "parameters": [
```

```
          { "type": "Literal", "value": 0, "original": "0" }
        ],
        "original": "gcd(0)",
        "location": {
          "start": { "offset": 7, "line": 1, "column": 8 },
          "end": { "offset": 13, "line": 1, "column": 14 }
        },
        "arity": 1, "functor": "gcd/1"
      }
    ],
    "body": [
      {
        "type": "Constraint",   "name": "true",
        "parameters": [],       "original": "true",
        "arity": 0,             "functor": "true/0"
      }
    ],
    "guard": [],
    "constraints": [ "gcd/1", "true/0" ],
    "r": 0,
    "head": [
      {
        "type": "Constraint",
        "name": "gcd",
        "parameters": [
          { "type": "Literal", "value": 0, "original": "0" }
        ],
        "original": "gcd(0)",
        "location": {
          "start": { "offset": 7, "line": 1, "column": 8 },
          "end": { "offset": 13, "line": 1, "column": 14 }
        },
        "arity": 1, "functor": "gcd/1"
      }
    ],
```

```
      "replacements": [],
      "name": "gcd1",
      "original": "gcd1 @ gcd(0) <=> true",
      "location": {
        "start": { "offset": 0, "line": 1, "column": 1 },
        "end": { "offset": 22, "line": 1, "column": 23 }
      }
    },
    {
      "type": "SimpagationRule",
      "kept": [
        {
          "type": "Constraint",
          "name": "gcd",
          "parameters": [
            { "type": "Identifier", "name": "N", "original": "N" }
          ],
          "original": "gcd(N)",
          "location": {
            "start": { "offset": 30, "line": 2, "column": 8 },
            "end": { "offset": 36, "line": 2, "column": 14 }
          },
          "arity": 1, "functor": "gcd/1"
        }
      ],
      "removed": [
        {
          "type": "Constraint",
          "name": "gcd",
          "parameters": [
            { "type": "Identifier", "name": "M", "original": "M" }
          ],
          "original": "gcd(M)",
          "location": {
            "start": { "offset": 39, "line": 2, "column": 17 },
```

```
            "end": { "offset": 45, "line": 2, "column": 23 }
          },
          "arity": 1, "functor": "gcd/1"
        }
      ],
      "body": [
        {
          "type": "Constraint",
          "name": "gcd",
          "parameters": [
            {
              "type": "BinaryExpression",
              "operator": "-",
              "left": {
                "type": "Identifier",
                "name": "M"
              },
              "right": {
                "type": "Identifier",
                "name": "N"
              },
              "original": "M-N"
            }
          ],
          "original": "gcd(M-N)",
          "arity": 1, "functor": "gcd/1"
        }
      ],
      "guard": [
        {
          "type": "BinaryExpression",
          "operator": "<",
          "left": { "type": "Literal", "value": 0 },
          "right": { "type": "Identifier", "name": "N" }
        },
```

```
        {
          "type": "BinaryExpression",
          "operator": "<=",
          "left": { "type": "Identifier", "name": "N" },
          "right": { "type": "Identifier", "name": "M" }
        }
      ],
      "constraints": [ "gcd/1" ],
      "r": 1,
      "head": [
        {
          "type": "Constraint",
          "name": "gcd",
          "parameters": [
            { "type": "Identifier", "name": "N", "original": "N" }
          ],
          "original": "gcd(N)",
          "location": {
            "start": { "offset": 30, "line": 2, "column": 8 },
            "end": { "offset": 36, "line": 2, "column": 14 }
          },
          "arity": 1, "functor": "gcd/1"
        },
        {
          "type": "Constraint",
          "name": "gcd",
          "parameters": [
            { "type": "Identifier", "name": "M", "original": "M" }
          ],
          "original": "gcd(M)",
          "location": {
            "start": { "offset": 39, "line": 2, "column": 17 },
            "end": { "offset": 45, "line": 2, "column": 23 }
          },
          "arity": 1, "functor": "gcd/1"
```

```
152           }
153         ],
154         "replacements": [],
155         "name": "gcd2",
156         "original":
157           "gcd2 @ gcd(N) \ gcd(M) <=> 0 < N, N <= M | gcd(M-N)",
158         "location": {
159           "start": { "offset": 23, "line": 2, "column": 1 },
160           "end": { "offset": 74, "line": 2, "column": 52 }
161         }
162       }
163     ]
164   }
```

B.3. Generated Code for the Occurrence Handlers of `gcd/1`

The following Listings present the code generated by the CHR.js compiler. The result is based on the general compilation scheme presented in section 4.4.1. The numbering of the occurrences follows the head normal form (c.f. 2.5.1).

Listing B.3: Generated code for $gcd^{[1]}$

```
1  function (constraint,replacements) {
2    var self = this
3
4    if (constraint.args[0] !== 0) {
5      return
6    }
7
8    var constraintIds = [
9      [ constraint.id ]
10   ]
11
12   return new Promise(function (resolve, reject) {
13     self.Helper.forEach(constraintIds,
```

```
14      function iterateConstraint (ids, callback) {
15        if (!self.Store.allAlive(ids))
16          return callback()
17
18        if (self.History.has("gcd1", ids))
19          return callback()
20
21        self.History.add("gcd1", ids)
22        self.Store.kill(ids[0])
23
24        Promise.resolve()
25        .then(function () {
26          callback()
27        })
28        .catch(function() {
29          reject()
30        })
31      }, resolve)
32    })
33  }
```

Listing B.4: Generated code for $gcd^{[2]}$

```
1  function (constraint,replacements) {
2    var self = this
3
4    var M = constraint.args[0]
5
6    var constraintIds = [
7      self.Store.lookup("gcd", 1)
8    , [ constraint.id ]
9    ]
10
11   return new Promise(function (resolve, reject) {
12     self.Helper.forEach(constraintIds,
13     function iterateConstraint (ids, callback) {
```

```
14        if (!self.Store.allAlive(ids))
15          return callback()
16
17        if (self.History.has("gcd2", ids))
18          return callback()
19
20        var N = self.Store.args(ids[0])[0]
21
22        var guards = [
23          new Promise(function (s, j) { (0 < N) ? s() : j() })
24        , new Promise(function (s, j) { (N <= M) ? s() : j() })
25        ]
26
27        Promise.all(guards)
28        .then(function () {
29          self.History.add("gcd2", ids)
30          self.Store.kill(ids[1])
31
32          Promise.resolve()
33          .then(function () {
34            return self.gcd(M - N)
35          })
36          .then(function () {
37            callback()
38          })
39          .catch(function() {
40            reject()
41          })
42        })
43        .catch(function () {
44           callback()
45        })
46      }, resolve)
47    })
48  }
```

Listing B.5: Generated code for $gcd^{[3]}$

```
function (constraint,replacements) {
  var self = this

  var N = constraint.args[0]

  var constraintIds = [
    [ constraint.id ]
  , self.Store.lookup("gcd", 1)
  ]

  return new Promise(function (resolve, reject) {
    self.Helper.forEach(constraintIds,
    function iterateConstraint (ids, callback) {
      if (!self.Store.allAlive(ids))
        return callback()

      if (self.History.has("gcd2", ids))
        return callback()

      var M = self.Store.args(ids[1])[0]

      var guards = [
        new Promise(function (s, j) { (0 < N) ? s() : j() })
      , new Promise(function (s, j) { (N <= M) ? s() : j() })
      ]

      Promise.all(guards)
      .then(function () {
        self.History.add("gcd2", ids)
        self.Store.kill(ids[1])

        Promise.resolve()
        .then(function () {
          return self.gcd(M - N)
```

```
35          })
36          .then(function () {
37            callback()
38          })
39          .catch(function() {
40            reject()
41          })
42        })
43        .catch(function () {
44           callback()
45        })
46      }, resolve)
47    })
48  }
```

B.4. Precompiled Code for gcd/1

The following Listing shows the code generated by the babel-plugin-chr precompiler for CHR.js. The result is based on the compilation scheme presented in Section 6.3.3.

Listing B.6: Precompiled code for gcd/1

```
1  function _chr () {
2    "use strict";
3
4    /**
5      ... static implementation for CHR runtime components ...
6    */
7
8    function State (constraint, type, step, lookup, scope) {
9      this.type = type
10     this.constraint = constraint
11     this.step = step
12     this.lookup = lookup
13     this.scope = scope
```

```
14    }
15
16    function saveState (constraint, step, lookup, scope) {
17      return new State(constraint, 'intermediate', step, lookup, scope)
18    }
19
20    function constraintState (constraint) {
21      return new State(constraint, 'constraint', null, null, null)
22    }
23
24    function tell () {
25      var current
26      while (chr.Tells.length > 0) {
27        current = chr.Tells.pop()
28        if (current.type === 'constraint') {
29          chr.Store.add(current.constraint)
30        }
31        dispatchTell(current)(current)
32      }
33    }
34
35    function dispatchTell (current) {
36      switch (current.constraint.functor) {
37        case 'gcd/1':
38          return gcd_1
39        break
40      }
41    }
42
43    // creates a promise-like "then-able" function
44    function thenable () {
45      return {
46        then: function (cb) {
47          cb()
48          return thenable()
```

```
49        }
50      }
51    }
52
53    var chr = {
54      Store: new Store(),
55      Functors: {
56        'gcd/1': true
57      },
58      Tells: [],
59      gcd: gcd
60    }
61
62    function gcd () {
63      var l = arguments.length
64      var args = new Array(l)
65      for (var i = 0; i < l; i++) args[i] = arguments[i];
66
67      var arity = args.length;
68      var constraint = new Constraint('gcd', arity, args)
69
70      chr.Tells.push(constraintState(constraint))
71      tell()
72
73      return thenable()
74    }
75
76    function gcd_1_0 (current) {
77      if (current.type === 'intermediate') {
78        current.type = 'constraint'
79      }
80
81      if (current.constraint.args[0] === 0) {
82        current.constraint.kill()
83        return true
```

```
84      }
85    }
86
87    function gcd_1_1 (current) {
88      var lookup, constraints
89      if (current.type === 'intermediate') {
90        current.type = 'constraint'
91        lookup = current.lookup
92      } else {
93        lookup = new Lookup(
94          chr,
95          [
96            { name: 'gcd', arity: 1 },
97            { constraint: current.constraint }
98          ]
99        )
100     }
101
102     while (constraints = lookup.next()) {
103       if (gcd_1_1_h0(current, constraints)) {
104         return true
105       }
106     }
107   }
108
109   function gcd_1_1_h0 (current, constraints) {
110     var scope = {
111       N: constraints[0].args[0],
112       M: current.constraint.args[0]
113     }
114
115     if (!(0 < scope.N && scope.N <= scope.M)) {
116       return
117     }
118     current.constraint.kill()
```

```
119
120     chr.Tells.push(constraintState(
121       new Constraint('gcd', 1, [ scope.M-scope.N ])
122     ))
123     return true
124   }
125
126   function gcd_1_2 (current) {
127     var lookup, scope, constraints
128     if (current.type === 'intermediate') {
129       lookup = current.lookup
130     } else {
131       lookup = new Lookup(
132         chr,
133         [
134           { constraint: current.constraint },
135           { name: 'gcd', arity: 1 }
136         ]
137       )
138     }
139
140     scope = {
141       N: current.constraint.args[0]
142     }
143
144     if (0 < scope.N) {
145       while (constraints = lookup.next()) {
146         scope.M = constraints[1].args[0]
147
148         if (!(scope.N <= scope.M)) {
149           continue
150         }
151         constraints[1].kill()
152
153         // save current state
```

```
154         chr.Tells.push(
155           saveState(current.constraint, 2, lookup, scope),
156           constraintState(
157             new Constraint('gcd', 1, [ scope.M-scope.N ])
158           )
159         )
160
161         // continue with these constraints
162         return true
163       }
164     }
165   }
166
167   function gcd_1 (current) {
168     if (current.type !== 'intermediate'
169       || current.step === 0) {
170       if (gcd_1_0(current)) {
171         return
172       }
173     }
174
175     if (current.type !== 'intermediate'
176       || current.step === 1) {
177       if (gcd_1_1(current)) {
178         return
179       }
180     }
181
182     if (current.type !== 'intermediate'
183       || current.step === 2) {
184       if (gcd_1_2(current)) {
185         return
186       }
187     }
188   }
```

```
189
190     return chr
191 }
192
193 var chr = _chr()
```

C

User Manuals

In this appendix there are the user manuals for the created libraries and tools. In every project's directory there is an additional file `README.md` with a short installation and usage instructions. In case the usage was already explained in the main part of this work, we quit it here.

C.1. CHR.js

The CHR.js module and its usage was presented in Chapter 4.

C.1.1. Installation

CHR.js can be used in node.js as well as in browser environments.

node.js

The module has been published on npm under the name `chr`. It requires node.js as of version 4.0 and higher. It can be installed using npm with the following command:

```
npm install chr
```

Then the `CHR` constructor can be loaded via:

```
var CHR = require('chr')
```

Browser Environments

CHR.js provides pre-bundled and minified browser builds (in the project's `/dist` directory):

`chr.js` (200 kB) The complete CHR.js module as a standalone file.

`chr.min.js` (135 kB) The minified version of `chr.js`.

`chrparser.js` (135 kB) A standalone module that exposes a single function to parse strings according to the (adapted) CHR.js Parsing Expression Grammar.

`chrparser.min.js` (95 kB) The minified version of `chrparser.js`.

`chr-wop.js` (40 kB) The CHR.js module without anything related to the parsing. Any CHR.js function which expects a source string also works by directly providing the AST object created by the parser.

To use the module(s), the file(s) must be placed in the `<head>` part of the HTML document, for example:

```
<script src="chr.min.js"></script>
```

The bundled files `chr.js`, `chr.min.js` and `chr-wop.js` expose the `CHR` constructor to the global namespace. To avoid naming clash with an existing variable of this name, CHR.js comes with a compatibility mode:

```
var newName = CHR.noConflict()
```

An very basic example website using CHR.js is provided in `/dist/index.html`.

C.1.2. Usage Example with node.js

The following CHR rule generates all Fibonacci numbers upto a given index `Max` as constraints of the form `fib(Number,Value)`:

```
upto(Max), fib(A,AV), fib(B,BV) ==>
  B === A+1, B < Max | fib(B+1,AV+BV)
```

The CHR rule can be used in JavaScript after declaring it via the ‘chr()‘ function. This is illustrated by the following example which illustrates the usage of CHR.js in the node.js REPL.

```
var CHR = require('chr')              // load the module
var chr = CHR()                       // create new solver

// add the rule
chr('upto(Max), fib(A,AV), fib(B,BV) ==> \
       B === A+1, B < Max | fib(B+1,AV+BV)')

console.log(chr.Store.toString())     // print the content of the
                                      //   constraint store
/* results in:
    (empty)
*/

Promise.all([
  chr.fib(1,1),                       // the first Fibonacci is 1
  chr.fib(2,1)                        // the second is 1
```

```
])

console.log(chr.Store.toString())    // both have been stored
/* results in:
    ID  Constraint
    --  ----------
    1   fib(1,1)
    2   fib(2,1)
*/

// now generate the Fibonaccis upto the 5th element
chr.upto(5).then(function () {
  console.log(chr.Store.toString())
})
/* results in:
    ID  Constraint
    --  ----------
    1   fib(1,1)
    2   fib(2,1)
    3   upto(5)
    4   fib(3,2)
    5   fib(4,3)
    6   fib(5,5)
*/
```

C.1.3. Tests

CHR.js was created following the test-driven development approach. The correctness of CHR.js is ensured by currently more than 400 defined tests.

The tests can only be executed with node.js. The test suite is given in the `/test` directory. The tests can be started with:

```
npm run tape
```

The output follows the Test Anything Protocol (TAP).

C.2. CHR.js-website

The web-based CHR tracer introduced in Chapter 5 uses the static site generator Jekyll[1]. Using this generator, the CHR Playground can be easily deployed on GitHub using the `gh-pages` branch. If so, the `CNAME` file in the root directory has to be adjusted to point to the correct domain name.

The static website can be generated by calling the following in the project's root directory:

```
bundle exec jekyll serve --watch
```

This will place the generated static HTML webpages in the `/_site` directory.

C.3. CHR-Benchmarks

The benchmark suite presented in Section 6.1 can be used to compare different CHR systems. It has also been used to compare the different `gcd(a,b)` implementations in JavaScript as presented in Section 6.2.

C.3.1. Installation

The `CHR-Benchmarks` suite can be used to compare the execution runtimes of the following CHR systems:

- SWI-Prolog, using KU Leuven CHR

[1] Project website: `http://jekyllrb.com/`

- CCHR
- JCHR
- CHR.js
- Native C implementations

The project comes with a `Makefile` to install the systems and prepare the tests. It is recommended to call the `make` commands as root, because systems like SWI-Prolog have to be installed.

With

```
make install
make prepare
```

all systems mentioned before can be installed and the tests prepared. The preparation step includes the compilation of the tests.

C.3.2. Test Cases

The following tests have been specified:

fib The bottom-up calculation of the Fibonacci numbers.

gcd The calculation of the greatest common divisor of two integers, based on the subtraction-based Euclidean algorithm.

leq A constraint solver for less-equal constraints between variables.

primes An implementation of the Sieve of Eratosthenes to generate prime numbers.

ram A simulator of a Random Access Machine.

tak Implementation of the Takeuchi function.

Due to different features of the examined CHR systems not all tests have been implemented for every system.

C.3.3. Makefile

The `Makefile` contains a large number of targets. For every system (`swi`, `jchr`, `cchr`, `c` and `node`) there are several sub-targets, in particular:

`system.preinstall` Installation of dependencies and benchmark setup, for example the creation of temporary directories.

`system.install` Installs the actual system.

`system.prepare` Preparation tasks for the benchmarks. Usually this includes the compilation of the test source files, for example compile `*.jchr` files for JCHR.

`system.clean` Task to delete temporary directories and files. This should be called before the benchmark is executed.

`system.test` Runs each test once to check if it is executable. This will generally create no output. The tests have been passed if no error occurs.

`system.bench` Executes the benchmarks for this system.

Apart from these tasks there are further, system dependent sub-tasks, for example to benchmark only a single system and single test case.

C.4. Babel Plugin for CHR.js

The `babel-plugin-chr` is a plugin for Babel, a JavaScript transpiler. It replaces CHR.js rule definitions in JavaScript source code and replaces it with the appropriate precompiled code. Internally, it calls CHR.js to compile the given rules.

C.4.1. Installation

The `babel-plugin-chr` is published on npm and can therefore be installed with the following command:

```
1 npm install babel-plugin-chr
```

C.4.2. Usage

The plugin can be used with Babel's command line interface:

```
babel --plugins chr script.js
```

However we recommend to add the plugin to the `.babelrc` configuration file:

```
{
  "plugins": [ "chr" ]
}
```

This will automatically call the CHR.js precompiler to all of the project's JavaScript files.

Listings

Bibliography

[Abd97] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Principles and Practice of Constraint Programming-CP97*, pages 252–266. Springer, 1997.

[AF98] Slim Abdennadher and Thom Frühwirth. On completion of constraint handling rules. In *Principles and Practice of Constraint Programming—CP98*, pages 25–39. Springer, 1998.

[AKSS02] Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK:: A Java Constraint Kit. *Electronic Notes in Theoretical Computer Science*, 64:1 – 17, 2002. {WFLP} 2001, International Workshop on Functional and (Constraint) Logic Programming, Selected Papers.

[Bou04] Olivier Bouissou. A CHR library for SiLCC. *Diplomathesis. Technical University of Berlin, Germany*, 2004.

[BVHC15] Hudson Borges, Marco Tulio Valente, Andre Hora, and Jailton Coelho. On the Popularity of GitHub Applications: A Preliminary Note. *CoRR*, abs/1507.00604, 2015.

[CF14] Mats Carlsson and Thom Fr"uhwirth. *Sicstus PROLOG User's Manual 4.3*. Books On Demand - Proquest, 2014.

[Cro01] Douglas Crockford. JavaScript: The world's most misunderstood programming language. *Douglas Crockford's Javascript*, 2001.

[DSDLBH04] 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.

[Duc05] Gregory J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, PhD thesis, University of Melbourne, Australia, 2005.

[Eic05] Brendan Eich. Javascript at ten years. *ACM SIGPLAN Notices*, 40(9):129–129, 2005.

[EK97] ECMAScript ECMA-Kommittee. A general purpose, cross-platform programming langugae, Standard ECMA-262, 1997.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[Frü91] Thom Frühwirth. Introducing Simplification Rules. Technical Report ECRC-LP-63, European Computer-Industry Research Centre, Munchen, Germany, October 1991. Presented at the Workshop Logisches Programmieren, Goosen/Berlin, Germany, October 1991 and the Workshop on Rewriting and Constraints, Dagstuhl, Germany, October 1991.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(1):95–138, 1998.

[Frü09a] Thom Frühwirth. *Constraint Handling Rules*. August 2009.

[Frü09b] Thom Frühwirth. First steps towards a lingua franca for computer science: Rule-based Approaches in CHR. In *CHR 2009—Proceedings of the 6th International Workshop on Constraint Handling Rules*, page 1, 2009.

[HDLBSD05] Christian Holzbaur, Maria Garcia De La Banda, Peter J Stuckey, and Gregory J Duck. Optimizing compilation of constraint handling rules in HAL. *Theory and Practice of Logic Programming*, 5(4-5):503–531, 2005.

[HF98] Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules. In *ERCIM/COMPULOG Workshop on Constraints, CWI, Amsterdam*. Citeseer, 1998.

[HF99] 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.

[HH11] Ian Hickson and David Hyatt. HTML5. *W3C Working Draft WD-html5-20110525, May*, 2011.

[Hol09] Glendon Holst. JScriptLog, September 2009.

[Kae07] Martin Kaeser. WebCHR, 2007.

[LW14] Torbjörn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. *Theory and Practice of Logic Programming*, 14(4-5):539–552, 2014.

[Maj11] David Majda. PEG.js-Parser Generator for JavaScript. *URL: http://pegjs. majda. cz/.*, 2011.

[MHCH12] Jose F Morales, Rémy Haemmerlé, Manuel Carro, and Manuel V Hermenegildo. Lightweight compilation of (C) LP to JavaScript. *Theory and Practice of Logic Programming*, 12(4-5):755–773, 2012.

[Sch05] Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.

[SD04] Tom Schrijvers and Bart Demoen. The KU Leuven CHR system: Implementation and application. In *First workshop on constraint handling rules: selected contributions*, pages 1–5, 2004.

[Sne09] Jon Sneyers. Optimizing compilation and computational complexity of Constraint Handling Rules. In *Logic Programming*, pages 494–498. Springer, 2009.

[Sne15] Jon Sneyers. Constraint Handling Rules Website: Downloads, September 2015.

[SVWSDK10] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint handling rules. *Theory and practice of logic programming*, 10(01):1–47, 2010.

[Tho13] Jeff Thompson. Yield Prolog, May 2013.

[TV10] Stefan Tilkov and Steve Vinoski. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, (6):80–83, 2010.

[VWSD] 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.

[VWWSD08] Peter Van Weert, Pieter Wuille, Tom Schrijvers, and Bart Demoen. CHR for imperative host languages. In *Constraint Handling Rules*, pages 161–212. Springer, 2008.

[WSD07] Pieter Wuille, Tom Schrijvers, and Bart Demoen. CCHR: the fastest CHR Implementation, in C. In *Proc. 4th Workshop on Constraint Handling Rules (CHR'07)*, pages 123–137, 2007.

Name: Falco Nogatz Matrikelnummer: 718320

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Falco Nogatz