FreeCHR: An Algebraic Framework for CHR-Embeddings

Sascha Rechenberger(✉) and Thom Frühwirth

Institute for Software Engineering and Programming Languages, Ulm University, Albert-Einstein-Allee 11, 89069 Ulm, Germany
{sascha.rechenberger,thom.fruehwirth}@uni-ulm.de

Abstract. We introduce the framework *FreeCHR* which formalizes the embedding of *Constraint Handling Rules* (CHR) into a host language, using the concept of *initial algebra semantics* from category theory, to establish a high-level implementation scheme for CHR as well as a common formalization for both theory and practice. We propose a lifting of the syntax of CHR via an endofunctor in the category **Set** and a lifting of the very abstract operational semantics of CHR into FreeCHR, using the free algebra, generated by the endofunctor and give proofs for soundness and completeness w.r.t. their original definition.

Keywords: embedded domain-specific languages · declarative programming languages · constraint handling rules · operational semantics · category theory · initial algebra semantics

1 Introduction

Constraint Handling Rules (CHR) is a rule-based programming language that is usually embedded into a general-purpose programming language.

Having a CHR implementation available enables software developers to solve problems in a declarative and elegant manner. Aside from the obvious task of implementing constraint solvers [6,10], it can be used to solve scheduling problems [2], implement concurrent and multi-agent systems [20,21,31,32], for applications in music [13,29] and possibly game development [19]. In general, CHR is ideally suited for any problem that involves the transformation of (multi-) sets of data, as programs consist of a set of rewriting rules, hiding away the process of finding suitable candidates for rule application. Hereby, we get a purely declarative representation of the algorithm without the otherwise necessary boilerplate code.

Implementations of CHR exist for a number of languages, such as Prolog [28], C [36], C++ [3], Haskell [5,21], JavaScript [23] and Java [1,17,34,35]. These implementations do not follow a common approach, though, which complicates and often stops maintenance, altogether. There is also a rich body of theoretical work concerning CHR, formalizing its declarative and operational semantics

A. Fensel et al. (Eds.): RuleML+RR 2023, LNCS 14244, pp. 190–205, 2023.
https://doi.org/10.1007/978-3-031-45072-3_14

[11,12,30]. However, there is yet no formal connection between CHR as an implemented programming language and CHR as a formalism.

We introduce the framework *FreeCHR* which formalizes the embedding of CHR, using *initial algebra semantics*. This concept which is commonly used in functional programming is used to inductively define languages and their semantics [16,18]. FreeCHR provides both a guideline and high-level architecture to implement and maintain CHR implementations across host languages and a strong connection between the practical and formal aspects of CHR. Also, by FreeCHR-instances being internal embeddings, we get basic tooling, like syntax highlighting and type-checking for free [9].

Ultimately, the framework shall serve a fourfold purpose, by providing

- a general guideline on how to implement a CHR system in modern high-level languages,
- a guideline for future maintenance of FreeCHR instances,
- a common framework for both formal considerations and practical implementations
- and a framework for the definition and verification of general criteria of correctness.

In this work, we will give first formal definitions of FreeCHR, upon which we will build our future work. A follow-up paper will cover first instantiations of FreeCHR. Section 2 will provide the necessary background and intuitions. Section 3 introduces the syntax and semantics of Constraint Handling Rules and generalizes them to non-Herbrand domains. Section 4 introduces the framework FreeCHR. Section 4.1 lifts the syntax of CHR programs to a **Set**-endofunctor and introduces the free algebra, generated by that functor, Sect. 4.2 lifts the *very abstract* operational semantics ω_a of CHR to the *very abstract* operational semantics $\omega_a^\star$ of FreeCHR and Sect. 4.3 proves the correctness of $\omega_a^\star$ w.r.t. ω_a. Section 4.4 gives a short preview of future work, concerning the implementation and verification of FreeCHR instances. Example instances of FreeCHR in Haskell and Python can be found on *GitHub* [24].

An extended preprint of this paper, including complete proofs and additional examples, is available on arxiv.com [27].

2 Endofunctors and F-Algebras

In this section, we want to introduce endofunctors and F-algebras. Both concepts are taken from category theory and will be introduced as instances in the category of sets **Set**.

We do not assume any previous knowledge of category theory, but to readers more interested in the topic in general, we recommend [22] as introductory literature.

2.1 Basic Definitions

The *disjoint union* of two sets A and B

$$A \sqcup B = \{l_A(a) \mid a \in A\} \cup \{l_B(b) \mid b \in B\}$$

is the union of both sets, with additional labels l_A and l_B added to the elements, to keep track of the origin set of each element. We will also use the labels l_A and l_B as *injection* functions $l_A : A \to A \sqcup B$ and $l_B : B \to A \sqcup B$ which construct elements of $A \sqcup B$ from elements of A or B, respectively.

For two functions $f : A \to C$ and $g : B \to C$, the function

$$[f, g] : A \sqcup B \to C$$
$$[f, g]\,(l(x)) = \begin{cases} f(x), & \text{if } l = l_A \\ g(x), & \text{if } l = l_B \end{cases}$$

is called a *case analysis* function of the disjoint union $A \sqcup B$, and is a formal analogue to a `case ... of` expression. Furthermore, we define two functions

$$f \sqcup g : A \sqcup B \to A' \sqcup B'$$
$$(f \sqcup g)(l(x)) = \begin{cases} l_{A'}(f(x)), & \text{if } l = l_A \\ l_{B'}(g(x)), & \text{if } l = l_B \end{cases}$$

$$f \times g : A \times B \to A' \times B'$$
$$(f \times g)(x, y) = (f(x), g(y))$$

which lift two functions $f : A \to A'$ and $g : B \to B'$ to the disjoint union and the Cartesian product, respectively.

2.2 Endofunctors

A **Set**-endofunctor[1] F maps all sets A to sets FA and all functions $f : A \to B$ to functions $Ff : FA \to FB$, such that $F\,\mathbf{id}_A = \mathbf{id}_{FA}$ and $F(g \circ f) = Fg \circ Ff$, where $\mathbf{id}_X(x) = x$ is the identity function on a set X[2]. A signature $\Sigma = \{\sigma_1/a_1, ..., \sigma_n/a_n\}$, where σ_i are operators and a_i their arity, generates a functor

$$F_\Sigma X = \bigsqcup_{\sigma/a \in \Sigma} X^a \qquad F_\Sigma f = \bigsqcup_{\sigma/a \in \Sigma} f^a$$

with $X^0 = \mathbb{1}$ and $f^0 = \mathbf{id}_{\mathbb{1}}$, where $\mathbb{1}$ is a singleton set. Such a functor F_Σ models *flat* (i.e., not nested) terms over the signature Σ.

[1] Since we only deal with endofunctors in Set, we will simply call them *functors*.
[2] We will omit the index of **id**, if it is clear from the context.

2.3 *F*-Algebras

Since an endofunctor F defines the syntax of terms, an evaluation function $\alpha : FA \to A$ defines the *semantics* of terms. We call such a function α, together with its *carrier* A, an F-algebra (A, α).

If there are two F-algebras (A, α) and (B, β) and a function $h : A \to B$, we call h an *F-algebra homomorphism*, iff. $h \circ \alpha = \beta \circ Fh$, i.e., h preserves the structure of (A, α) in (B, β), when mapping A to B. In this case, we also write $h : (A, \alpha) \to (B, \beta)$.

A special F-algebra is the *free F-algebra* $F^\star = (\mu F, \mathbf{in}_F)$, for which there is a homomorphism $(\!|\alpha|\!) : F^\star \to (A, \alpha)$ for any other algebra (A, α). We call those homomorphisms $(\!|\alpha|\!)$ *F-catamorphisms*. The functions $(\!|\alpha|\!)$ encapsulate structured recursion on values in μF, with the semantics defined by the function α which is itself only defined on flat terms. The carrier of $F^\star$, with $\mu F = F\mu F$, is the set of inductively defined values in the shape defined by F. The function $\mathbf{in}_F : F\mu F \to \mu F$ inductively constructs the values in μF.

3 CHR over Non-herbrand Domains

First implementations of CHR were embedded into the logical programming language Prolog, where terms like `3+4` or `f(a,b,c)` are not evaluated, as is the case in most other programming languages, but interpreted as themselves. This is called the *Herbrand* interpretation of terms. Since we want to embed CHR in any programming language, we need to generalize to non-Herbrand interpretations of terms. We will formalize this, using initial algebra semantics.

3.1 Host Language

We first define a *data type* in the host language. A data type determines the syntax and semantics of terms via a functor Λ_T and an algebra τ_T. The fixed point $\mu\Lambda_T$ contains terms which are inductively defined via Λ_T and the catamorphism $(\!|\tau_T|\!)$ evaluates those terms to values of T.

Definition 3.1 (Data types). *A* data type *is a triple $\langle T, \Lambda_T, \tau_T \rangle$, where T is a set, Λ_T a functor and (T, τ_T) a Λ_T-algebra.*

We write $t \equiv_T t'$ for $t \in \mu\Lambda_T$ and $t' \in T$, iff. $(\!|\tau_T|\!)(t) = t'$.

Example 3.1 (Boolean data type). The signature

$$\Sigma_2 = \{(n \leqslant m)/0 \mid n, m \in \mathbb{N}_0\} \cup \{(n < m)/0 \mid n, m \in \mathbb{N}_0\} \cup \{\wedge/2, \mathit{true}/0, \mathit{false}/0\}$$

defines Boolean terms[3]. Σ_2 generates the functor

$$\Lambda_2 X = \mathbb{N}_0 \times \mathbb{N}_0 \sqcup \mathbb{N}_0 \times \mathbb{N}_0 \sqcup X \times X \sqcup \mathbb{1} \sqcup \mathbb{1}$$

[3] We will generally overload symbols like *false*, *true*, $\wedge$, $\neg$, ..., if their meaning is clear from the context.

the fixed point, $\mu\Lambda_2$, of which is the set of valid nested Boolean terms like $(0 < 4 \wedge 4 \leqslant 6)$. Let $\langle \mathbb{2}, \Lambda_2, \tau_2 \rangle$, with $\mathbb{2} = \{true, false\}$, be a data type. If we assume τ_2 to implement the usual semantics for Boolean terms and comparisons, $(0 < 4 \wedge 4 \leqslant 6)$ will evaluate as

$$(\!|\tau_2|\!)(0 < 4 \wedge 4 \leqslant 6) = (\!|\tau_2|\!)(0 < 4) \wedge (\!|\tau_2|\!)(4 \leqslant 6) = true \wedge true = true$$

For a set T, both Λ_T and τ_T are determined by the host language which is captured by the next definition.

Definition 3.2 (Host environment). *A mapping*

$$\mathcal{L}T = \langle T, \Lambda_T, \tau_T \rangle$$

where $\langle T, \Lambda_T, \tau_T \rangle$ is a data type, is called a host environment.

A host environment is implied by the host language (and the program, the CHR program is part of) and assigns to a set T a data type, effectively determining syntax and semantics of terms that evaluate to values of T.

3.2 Embedding CHR

With the formalization of our host environment, we can define the syntax and semantics of CHR.

Definition 3.3 (CHR programs). *CHR programs are sets of multiset-rewriting rules of the form*

$$[N\ @]\ K\ \setminus\ R \iff [G\ |]\ B$$

For a set C, called the domain *of the Program, for which there is a data type $\mathcal{L}C = \langle C, \Lambda_C, \tau_C \rangle$, $K, R \in LC$ are called the* kept *and* removed head*, respectively. $LX = \bigcup_{i \in \mathbb{N}} X^i$ maps a set X to the set of finite sequences (or lists) over X, with $X^0 = \varepsilon$ being the empty sequence. The optional $G \in \mu\Lambda_2$ is called the* guard*. If G is omitted, we assume $G \equiv_2 true$. $B \in \mathcal{M}\mu\Lambda_C$ is called the* body*. The functor $\mathcal{M}$ maps a set X to the set $\mathcal{M}X$ of multisets over X and functions $X \to Y$ to functions $\mathcal{M}X \to \mathcal{M}Y$. N is an optional identifier for the rule.*

The members of the kept and removed head are matched against values of the domain C. The guard G is a term that can be evaluated to a Boolean value. The body B is a multiset over terms which can be evaluated to values of C. This includes any call of (pure) functions or operators which evaluates to Booleans, or values of C, respectively.

Definition 3.3 corresponds to the *positive range-restricted ground* segment of CHR which is commonly used as the target for embeddings of other (rule-based) formalisms (e.g., colored Petri nets [4]) into CHR [11, Chapter 6.2]. *Positive* means that the body of the rule contains only user constraints (i.e., values from C) which guarantees that computations do not fail. *Range-restricted* means that

instantiating all variables of the head (K and R) will ground the whole rule. This also maintains the *groundness* of the segment of CHR which requires that the input and output of a program are ground. $\mathbf{PRG}_C$ denotes the set of all such programs over a domain C.

Example 3.2 (Euclidean algorithm). The program GCD = {*zero*@..., *subtract*@...}

$$\begin{aligned} \mathit{zero} \ @ \ 0 \ &\Leftrightarrow \ \varnothing \\ \mathit{subtract} \ @ \ N \ \setminus \ M \ &\Leftrightarrow \ 0 < N \wedge 0 < M \wedge N \leqslant M \mid M - N \end{aligned}$$

computes the greatest common divisor of a collection of natural numbers. The first rule removes any numbers 0 from the collection. The second rule replaces for any pair of numbers N and M greater 0 and $N \leqslant M$, M by $M - N$. Note that we omitted the kept head and guard of the *zero* rule.

Definition 3.4 (C-instances of rules). *For a* positive range-restricted *rule*

$$r = R \ @ \ k_1, ..., k_n \ \setminus \ r_1, ..., r_m \ \Leftrightarrow \ G \mid B$$

with universally quantified variables $v_1, ..., v_l$, and a data type $\mathcal{L}C = \langle C, \Lambda_C, \tau_C \rangle$, we call the set

$$\begin{aligned} \Gamma_C(r) = \{ \ & R \ @ \ k_1\sigma, ..., k_n\sigma \ \setminus \ r_1\sigma, ..., r_m\sigma \ \Leftrightarrow \ G\sigma \mid \mathcal{M}(\!|\tau_C|\!)(B\sigma) \\ & \mid \sigma \textit{ instantiates all variables } v_1, \ ..., \ v_l, \\ & \ k_1\sigma, ..., k_n\sigma, r_1\sigma, ..., r_m\sigma \in C, \\ & \ G\sigma \in \mu\Lambda_2, \\ & \ B\sigma \in \mathcal{M}\mu\Lambda_C \ \} \end{aligned}$$

*the C-*grounding *of r. Analogously, for a set $\mathcal{R}$ of rules, $\Gamma_C(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} \Gamma_C(r)$ is the C-grounding of $\mathcal{R}$. An element $r' \in \Gamma_C(r)$ (or $\Gamma_C(\mathcal{R})$ respectively) is called a C-*instance *of a rule $r \in \mathcal{R}$.*

A C-instance (or grounding) is obtained, by instantiating any variables and evaluating the then ground terms in the body of the rule, using the Λ_C-catamorphism $(\!|\tau_C|\!)$. The functor $\mathcal{M}$ is used, to lift $(\!|\tau_C|\!)$ into the multiset.

Example 3.3. Given a body $\{M - N\}$ and a substitution $\sigma = \{N \mapsto 4, M \mapsto 6\}$, the body is instantiated like

$$\mathcal{M}(\!|\tau_C|\!)(\{M - N\}\,\sigma) = \mathcal{M}(\!|\tau_C|\!)(\{6 - 4\}) = \{(\!|\tau_C|\!)(6 - 4)\} = \{2\}$$

With Example 3.3, we can also easily see that, if we use a data type $\mathcal{L}\mu\Lambda_C = \langle \mu\Lambda_C, \Lambda_C, \mathbf{in}_{\Lambda_C} \rangle$, we get the Herbrand interpretation of terms over C. Hence, f.i., an expression $3 + 4 \in \mu\Lambda_{\mathbb{N}_0}$ is evaluated to itself, as is the case in Prolog.

Example 3.4 (C-instances). If we instantiate the rule

$$r = \mathit{subtract} \ @ \ N \ \setminus \ M \ \Leftrightarrow \ 0 < N \wedge 0 < M \wedge N \leqslant M \mid M - N$$

with $\sigma_1 = \{N \mapsto 4, M \mapsto 6\}$ and $\sigma_2 = \{N \mapsto 0, M \mapsto 6\}$, respectively, we get the $\mathbb{N}_0$-instances

$$r\sigma_1 = \mathit{subtract} \ @ \ 4 \ \setminus \ 6 \ \Leftrightarrow \ 0 < 4 \wedge 0 < 6 \wedge 4 \leqslant 6 \mid 2$$
$$r\sigma_2 = \mathit{subtract} \ @ \ 0 \ \setminus \ 6 \ \Leftrightarrow \ 0 < 0 \wedge 0 < 6 \wedge 0 \leqslant 6 \mid 6$$

Both instances are elements of the $\mathbb{N}_0$-grounding $\Gamma_{\mathbb{N}_0}(\text{GCD})$ of the program in Example 3.2.

Classically, the guard G contains constraints which are defined w.r.t. a constraint theory $\mathcal{CT}$. We typically write $\mathcal{CT} \models G$[4] to denote that the guard is satisfiable w.r.t. $\mathcal{CT}$ and $\mathcal{CT} \models \neg G$ otherwise. Since in our case $G \in \mu\Lambda_2$, $\mathcal{CT}$ is essentially τ_2, as it determines the semantics of Boolean terms. We thus write

$$\tau_2 \models G \Longleftrightarrow G \equiv_2 \mathit{true} \qquad \tau_2 \models \neg G \Longleftrightarrow G \equiv_2 \mathit{false}$$

Note that we always need a data type $\mathcal{L}2$. In Prolog, f.i., 2 corresponds to the set $\{\texttt{true}, \texttt{false}\}$, representing successful, or failed computations, respectively.

Finally, the operational semantics of CHR is defined as a state transition system, where the states are multisets[5] over the elements of C.

Definition 3.5 (*Very abstract* operational semantics of CHR). *Figure 1 shows the very abstract operational semantics ω_a for CHR programs $\mathcal{R}$ over a domain C. With $\uplus_C : \mathcal{M}C \times \mathcal{M}C \to \mathcal{M}C$ being the union of multisets over elements of C, the inference rule* APPLY *reads as follows: if we have a C-instance*

$$R@c_1, ..., c_n \setminus c_{n+1}, ..., c_{n+m} \Leftrightarrow G|B \in \Gamma_C(\mathcal{R})$$

of a rule $r \in \mathcal{R}$ and the guard G evaluates to true, we can replace the subset $\{c_{n+1}, ..., c_{n+m}\}$ by the body B. We write

$$\mathcal{R} \vdash S_0 \mapsto^* S_l$$

if there are rules $R_1@r_1, ..., R_l@r_l \in \mathcal{R}$, with $1 \leqslant l$, such that

$$\mathcal{R} \vdash S_0 \mapsto_{R_1} S_1 \wedge S_1 \mapsto_{R_2} S_2 \wedge ... \wedge S_{l-1} \mapsto_{R_l} S_l$$

where S_i are multisets over ground elements of C.

$$\frac{R@c_1, ..., c_n \setminus c_{n+1}, ..., c_{n+m} \Leftrightarrow G|B \in \Gamma_C(\mathcal{R}) \qquad \tau_2 \models G}{\mathcal{R} \vdash \{c_1, ..., c_{n+m}\} \uplus \Delta S \mapsto_R \{c_1, ..., c_n\} \uplus B \uplus \Delta S} \ \text{APPLY}$$

Fig. 1. *Very abstract* operational semantics for ground and pure CHR

[4] As we only work with ground values, we do not need to quantify any variables.
[5] There may be some additional decoration in more refined operational semantics.

Rules are applied until no more are applicable to the state, i.e., we have reached a *final* state.

Example 3.5 (ω_a-transitions). Intuitively, both

$$\tau_2 \models 0 < 4 \wedge 0 < 6 \wedge 4 \leqslant 6 \qquad \text{and} \qquad \tau_2 \models \neg(0 < 0 \wedge 0 < 6 \wedge 0 \leqslant 6)$$

hold. Hence, we can prove the transition $\text{GCD} \vdash \{4, 6\} \mapsto_{subtract} \{4, 2\}$, but not $\text{GCD} \vdash \{0, 6\} \mapsto_{subtract} \{0, 6\}$.

The following example shows the execution of the Euclidean algorithm as a final example of the operational semantics of CHR.

Example 3.6 (Euclidean algorithm (cont.)). The rules of GCD are applied until exhaustion, leaving only the greatest common divisor of all numbers of the input. For an input $\{4, 6\}$, the program will perform a sequence

$$\{4, 6\} \mapsto_{subtract} \{4, 2\} \mapsto_{subtract} \{2, 2\} \mapsto_{subtract} \{2, 0\} \mapsto_{zero} \{2\}$$

of transformations.

4 FreeCHR

The main idea of FreeCHR is to model the syntax of CHR programs as a functor CHR_C. We then use the *free* CHR_C-algebra to define the operational semantics of FreeCHR which we can directly use to verify instances of FreeCHR.

4.1 Syntax

We now present the fundamental definition of our work which allows us to view CHR-programs over a domain C.

Definition 4.1 (Syntax of FreeCHR programs). *The functor*[6]

$$\begin{aligned} CHR_C D &= L2^C \times L2^C \times 2^{LC} \times (\mathcal{M}C)^{LC} \sqcup D \times D \\ CHR_C f &= \mathbf{id} \sqcup f \times f \end{aligned}$$

describes the syntax of FreeCHR programs.

The set $L2^C \times L2^C \times 2^{LC} \times (\mathcal{M}C)^{LC}$ is the set of single rules. The kept and removed head of a rule are sequences of functions in $L2^C$ which map elements of C to Booleans, effectively checking individual values for applicability of the rule. The guard of the rule is a function in 2^{LC} and maps sequences of elements in C to Booleans, checking all matched values in the context of each other. Finally, the body of the rule is a function in $(\mathcal{M}C)^{LC}$ and maps the matched values to a multiset of newly generated values.

[6] That CHR_C is indeed a functor can easily be verified via equational reasoning.

The set $D \times D$ represents the composition of FreeCHR programs by an execution strategy, allowing the construction of more complex programs from, ultimately, single rules.

By the structure of CHR_C, a CHR_C-algebra with carrier D is defined by two functions

$$\rho : \ L2^C \times L2^C \times 2^{LC} \times (\mathcal{M}C)^{LC} \longrightarrow D \qquad \nu : \ D \times D \to D$$

as $(D, [\rho, \nu])$. The free CHR_C-algebra $\mathrm{CHR}_C^\star$ provides us with an inductively defined representation of programs which we will later use to lift the very abstract operational semantics ω_a.

Lemma 4.1 (Free CHR_C-algebra). *With*

$$\mu CHR_C = L2^C \times L2^C \times 2^{LC} \times (\mathcal{M}C)^{LC} \sqcup \mu CHR_C \times \mu CHR_C$$

and labels/injections

$$rule : L2^C \times L2^C \times 2^{LC} \times (\mathcal{M}C)^{LC} \longrightarrow \mu CHR_C$$
$$\odot : \mu CHR_C \times \mu CHR_C \longrightarrow \mu CHR_C$$

$CHR_C^\star = (\mu CHR_C, [rule, \odot])$ *is the* free CHR_C*-algebra.*

Proof sketch. From the specialized homomorphism property

$$(\!|[\rho, \nu]|\!)([rule, \odot]\,(p)) = [\rho, \nu]\,((\mathrm{CHR}_C(\!|[\rho, \nu]|\!))(p))$$

we construct the CHR_C-catamorphism

$$(\!|[\rho, \nu]|\!) : \mu\mathrm{CHR}_C \longrightarrow A$$
$$(\!|[\rho, \nu]|\!)(rule(k, r, g, b)) = \rho(k, r, g, b)$$
$$(\!|[\rho, \nu]|\!)(p_1 \odot p_2) = \nu((\!|[\rho, \nu]|\!)(p_1), (\!|[\rho, \nu]|\!)(p_2))$$

for any CHR_C-algebra $\alpha = (A, [\rho, \nu])$.

The free CHR_C-algebra corresponds to the definition of abstract syntax trees of programs, while the catamorphism $(\!|\alpha|\!)$ corresponds to an interpretation that preserves the semantics of α.

Also, we can easily see that $\odot$ is associative up to isomorphism[7]. We thus will not explicitly write parentheses and generally use chained expressions like $p_1 \odot ... \odot p_l$ for some $l \in \mathbb{N}$.

Example 4.1 (Euclidean algorithm (cont.)). The program $gcd = zero \odot subtract$ with

$$zero = rule(\varepsilon, (\lambda n.n = 0), (\lambda n.true), (\lambda n.\varnothing))$$
$$subtract = rule((\lambda n.0 < n), (\lambda m.0 < m), (\lambda n\ m.n \leqslant m), (\lambda n\ m.\{m - n\}))$$

implements the euclidean algorithm, as defined in Example 3.2. λ-abstractions are used for ad-hoc definitions of functions.

[7] By $\mathbf{assoc}(a, (b, c)) = ((a, b), c)$ and $\mathbf{assoc}^{-1}((a, b), c) = (a, (b, c))$.

4.2 Operational Semantics

We now lift the *very abstract* operational semantics ω_a of CHR to the very abstract operational semantics $\omega_a^\star$ of FreeCHR. We assume that our programs are defined over a domain C, where there is a data type $\mathcal{L}C = \langle C, \Lambda_C, \tau_C \rangle$. Like ω_a, $\omega_a^\star$ is defined as a state transition system, where states are multisets over elements of C.

Definition 4.2 (*Very abstract* operational semantics $\omega_a^\star$). *Let $S, S' \in \mathcal{M}C$ and $p \in \mu CHR_C$. We write*

$$p \vdash S \overset{\omega_a^\star}{\longmapsto} S'$$

if we can derive a transition from state S to S' with the program p. Figure 2 shows inference rules, defining the very abstract *operational semantics $\omega_a^\star$ of FreeCHR.*

$$\frac{}{p \vdash S \overset{\omega_a^\star}{\longmapsto} S} \text{ PASS/FINAL} \qquad \frac{p_i \vdash S \overset{\omega_a^\star}{\longmapsto} S' \quad p_1 \odot ... \odot p_i \odot ... \odot p_l \vdash S' \overset{\omega_a^\star}{\longmapsto} S''}{p_1 \odot ... \odot p_i \odot ... \odot p_l \vdash S \overset{\omega_a^\star}{\longmapsto} S''} \text{ STEP}_i$$

$$\frac{k_1(c_1) \wedge ... \wedge k_n(c_n) \wedge r_1(c_{n+1}) \wedge ... \wedge r_m(c_{n+m}) \wedge g(c_1, ..., c_{n+m}) \equiv_2 true}{rule(k, r, g, b) \vdash \{c_1, ..., c_{n+m}\} \uplus \Delta S \overset{\omega_a^\star}{\longmapsto} \{c_1, ..., c_n\} \uplus b\,(c_1, ..., c_{m+n}) \uplus \Delta S} \text{ APPLY}$$

Fig. 2. *Very abstract* operational semantics of FreeCHR

The rule PASS/FINAL states that a program is always allowed to do nothing to a state. STEP$_i$ states that we can derive a transition from S to S'', if we can transition from S to S' with the i-th program in the composition $p_1 \odot ... \odot p_l$ (for $1 \leqslant i \leqslant l$) and from S' to S'' with the whole composition. The idea is that, w.l.o.g., p_i is a rule that is applied to the current state, whereafter execution is continued. APPLY is the translation of the original APPLY-rule of ω_a. k_i and r_j denote the i-th and j-th elements of the sequences k and r.

4.3 Soundness and Completeness of $\omega_a^\star$

To prove the soundness and completeness of $\omega_a^\star$ w.r.t. ω_a, we first need to embed FreeCHR into the *positive range-restricted ground* segment of CHR. This is a common approach to relate rule-based formalisms to CHR [11, Chapter 6].

Definition 4.3 (Embedding FreeCHR into CHR). *Let*

$$\Theta : \mu CHR_C \longrightarrow \mathbf{PRG}_C$$
$$\Theta\,(rule\,(k, r, g, b)) = \{R@v_1, ..., v_n \setminus v_{n+1}, ..., v_{n+m} \Leftrightarrow G | b\,(v_1, ..., v_{n+m})\}$$
$$\Theta\,(p_1 \odot ... \odot p_l) = \Theta\,(p_1) \cup ... \cup \Theta\,(p_l)$$

be the function embedding FreeCHR programs into the positive range-restricted ground *segment of CHR, with universally quantified variables* v_1,...,v_{n+m}, R *a uniquely generated rule name and*

$$G = k_1\left(v_1\right) \wedge ... \wedge r_m\left(v_{n+m}\right) \wedge g\left(v_1, ..., v_{n+m}\right)$$

We now prove soundness, up to the embedding Θ, of $\omega_a^\star$ w.r.t. ω_a, i.e., if we can prove a derivation under $\omega_a^\star$ for a program p, we can prove it under ω_a for $\Theta(p)$.

Theorem 4.1 (Soundness of $\omega_a^\star$). $\omega_a^\star$ *is* sound *w.r.t.* ω_a*, i.e., for* $S, S' \in \mathcal{MC}$ *and* $p \in \mu CHR_C$*,*

$$p \vdash S \overset{\omega_a^\star}{\longmapsto} S' \Longrightarrow \Theta\left(p\right) \vdash S \mapsto^* S'$$

Proof sketch. We prove soundness by induction over the inference rules of $\omega_a^\star$. As induction base cases, we show

$$p \vdash S \overset{\omega_a^\star}{\longmapsto} S \Longrightarrow \Theta(p) \vdash S \mapsto^* S \qquad \text{(PASS/FINAL)}$$

and

$$rule(k, r, g, b) \vdash S \overset{\omega_a^\star}{\longmapsto} S' \Longrightarrow \Theta(rule(k, r, g, b)) \vdash S \mapsto^* S' \qquad \text{(APPLY)}$$

As induction step, we show, for $p = p_1 \odot ... \odot p_l$

$$p \vdash S \overset{\omega_a^\star}{\longmapsto} S'' \Longrightarrow \Theta(p) \vdash S \mapsto^* S'' \qquad (\text{STEP}_i)$$

assuming the induction hypotheses

$$\forall i \in \{1, ..., l\} .p_i \vdash S \overset{\omega_a^\star}{\longmapsto} S' \Longrightarrow \Theta(p_i) \vdash S \mapsto^* S'$$
$$p \vdash S' \overset{\omega_a^\star}{\longmapsto} S'' \Longrightarrow \Theta(p) \vdash S' \mapsto^* S''$$

We established that we can prove any derivation we can prove with a program p under $\omega_a^\star$, with a program $\Theta(p)$, under ω_a. We also want to prove completeness up to Θ, i.e., we can prove any derivation with a program $\Theta(p)$ under ω_a with a program p under $\omega_a^\star$.

Theorem 4.2 (Completeness of $\omega_a^\star$). $\omega_a^\star$ *is* complete *w.r.t.* ω_a*, i.e., for* $S, S' \in \mathcal{MC}$ *and* $p \in \mu CHR_C$*,*

$$\Theta\left(p\right) \vdash S \mapsto^* S' \Longrightarrow p \vdash S \overset{\omega_a^\star}{\longmapsto} S'$$

Proof sketch. We prove completeness by induction over ω_a transition steps. As the induction base case, we show

$$\Theta(p) \vdash S \mapsto^* S \Longrightarrow p \vdash S \overset{\omega_a^\star}{\longmapsto} S$$

As the induction step, we show

$$\Theta(p) \vdash S \mapsto_R S' \wedge S' \mapsto^* S'' \Longrightarrow p \vdash S \overset{\omega_a^\star}{\longmapsto} S''$$

assuming the induction hypothesis

$$\Theta(p) \vdash S' \mapsto^* S'' \Longrightarrow p \vdash S' \overset{\omega_a^\star}{\longmapsto} S''$$

With Theorem 4.1 and Theorem 4.2, we have established that, up to Θ, FreeCHR is as expressive as CHR. Analogously to classical CHR, we are now able to define more refined operational semantics for FreeCHR and prove their soundness w.r.t. $\omega_a^\star$.

4.4 Instantiation

State transitions can be modeled as functions that map a state to its successor. Hence, an instance of FreeCHR defines a CHR_C-algebra

$$\left((\mathcal{MC})^{\mathcal{MC}}, [\textsc{rule}, \textsc{compose}]\right)$$

by providing the functions

$$\begin{aligned} \textsc{rule} &: L2^C \times L2^C \times 2^{LC} \times (\mathcal{MC})^{LC} \longrightarrow (\mathcal{MC})^{\mathcal{MC}} \\ \textsc{compose} &: (\mathcal{MC})^{\mathcal{MC}} \times (\mathcal{MC})^{\mathcal{MC}} \longrightarrow (\mathcal{MC})^{\mathcal{MC}} \end{aligned}$$

The function RULE transforms a single rule into a multiset transformation. COMPOSE implements the execution strategy which defines how two programs are composed[8]. A function

$$\textsc{compose}(\textsc{rule}(...), ..., \textsc{rule}(...)) : \mathcal{MC} \longrightarrow \mathcal{MC}$$

then needs to be applied to a state repeatedly, until a fixed point is reached. Simple example implementations in Haskell and Python can be found on *Github* [24].

To prove that a FreeCHR instance $\left((\mathcal{MC})^{\mathcal{MC}}, \phi\right)$ is correct w.r.t. $\omega_a^\star$ (or another operational semantics), we use the catamorphism $(\!|\phi|\!)$. In particular, we show that, if applying $(\!|\phi|\!)(p)$ to a state S yields S', we must be able to prove a derivation from S to S' under $\omega_a^\star$ using a program p, i.e.,

$$(\!|\phi|\!)(p)(S) \equiv_{\mathcal{MC}} S' \Longrightarrow p \vdash S \overset{\omega_a^\star}{\longmapsto} S'$$

Vice versa, we show that, if we can prove a derivation from S to S' using a program p, we need to be able to compute S' from S by a finite number of applications of $(\!|\phi|\!)(p)$, i.e.,

$$p \vdash S \overset{\omega_a^\star}{\longmapsto} S' \Longrightarrow ((\!|\phi|\!)(p) \circ ... \circ (\!|\phi|\!)(p))(S) \equiv_{\mathcal{MC}} S'$$

[8] We generally assume that COMPOSE is associative. Hence, we will not nest COMPOSE-expressions.

5 Related Work

First work concerning the implementation of CHR systems introduced specific embeddings of CHR into different languages. This included first implementations for Prolog by Holzbaur and Frühwirth [15] and the Java Constraint Kit by Abdennadher et al. [1]. The operational semantics used up to this point were ad-hoc refinements of the *abstract operational semantics* ω_t [11], which was first formalized by Duck et al. [8] with the *refined operational semantics* ω_r. This was a major step towards standardizing implementations, by formally capturing the necessary practical aspects of real-life implementations. Further progress was made by introducing more general implementation schemes based on ω_r, for logical and imperative languages by Duck et al. [7] and van Weert et al. [33], respectively, which are now the basis of most modern CHR systems, like the K.U. Leuven System [28] or CHR.js [23].

First approaches to embed CHR as an internal language, i.e., by implementing CHR programs using constructs of the host-language [9], were introduced by Ivanović for Java [17] and Hanus for Curry [14]. The main inspiration for FreeCHR was introduced by Wibiral [35]. It first implemented the idea of explicitly composing CHR programs from single rules and the use of anonymous functions to model the functional dependency of the guard and body of rules on matched constraints.

Although we consider FreeCHR a framework for new implementations of CHR, the question, of whether it may be applied to existing implementations arises. Aside from the JavaCHR by Wibiral [35], CHR(Curry) by Hanus [14] seems to be a promising candidate for a possible future reframing in FreeCHR, as they are implemented as internal languages.

6 Conclusion

In this paper, we introduced the framework FreeCHR which formalizes the embedding of CHR, using initial algebra semantics. We provided the fundamental definition of our framework which models the syntax of CHR programs over a domain C as a **Set**-endofunctor CHR_C. We defined the *very abstract* operational semantics $\omega_a^\star$, using the free CHR_C-algebra $\mathrm{CHR}_C^\star$ and proved soundness and completeness w.r.t. the original *very abstract* operational semantics ω_a of CHR.

Ongoing work is first and foremost concerned with the introduction of formal and practical instances of FreeCHR, including the formalization in proof assistants like *Lean*, *Agda*, or *Coq*. We further plan to lift more definitions and results concerning CHR to FreeCHR, especially more refined operational semantics, as they are crucial for actual implementations (see, e.g., [25,26]). A future step is

to generalize the CHR_C functor and the accompanying definitions to more general definitions. The ultimate goal is to be able to model CHR in the context of arbitrary side-effects which are caused by effectful computations, including the addition of logical variables and failure of computations.

Acknowledgments. We thank our reviewers for their helpful comments. We also thank our colleagues Florian Sihler and Paul Bittner for proofreading and their constructive feedback.

References

1. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: a java constraint kit. In: Electronic Notes in Theoretical Computer Science, vol. 64, pp. 1–17 (2002). https://doi.org/10.1016/S1571-0661(04)80344-X
2. Abdennadher, S., Marte, M.: University course timetabling using constraint handling rules. AAI **14**(4), 311–325 (2000). https://doi.org/10.1080/088395100117016
3. Barichard, V.: CHR++ (2022). https://gitlab.com/vynce/chrpp
4. Betz, H.: Relating coloured petri nets to constraint handling rules. In: CHR 2007, vol. 7, pp. 33–47 (2007). https://dtai-static.cs.kuleuven.be/projects/CHR/papers/chr2007/betz_petri_nets_chr07.pdf
5. Chin, W., Sulzmann, M., Wang, M.: A Type-Safe Embedding of Constraint Handling Rules into Haskell (2008). https://www.semanticscholar.org/paper/A-Type-Safe-Embedding-of-Constraint-Handling-Rules-Chin-Sulzmann/ea47790fc268710d73b2a6be0305e3f3453682e3
6. De Koninck, L., Schrijvers, T., Demoen, B., Fink, M., Tompits, H., Woltran, S.: INCLP(R) - interval-based nonlinear constraint logic programming over the reals. In: WLP 2006, vol. 1843-06-02. Technische Universität Wien, Austria (2006). https://lirias.kuleuven.be/1653773
7. Duck, G.J.: Compilation of constraint handling rules. Ph.D. thesis, University of Melbourne, Victoria, Australia (2005)
8. Duck, G.J., Stuckey, P.J., de la Banda, M.G., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 90–104. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27775-0_7
9. Fowler, M., Parsons, R.: Domain-Specific Languages. Addison-Wesley, Upper Saddle River (2011)
10. Frühwirth, T.: Complete propagation rules for lexicographic order constraints over arbitrary domains. In: Hnich, B., Carlsson, M., Fages, F., Rossi, F. (eds.) CSCLP 2005. LNCS (LNAI), vol. 3978, pp. 14–28. Springer, Heidelberg (2006). https://doi.org/10.1007/11754602_2
11. Fruehwirth, T.: Constraint Handling Rules. Cambridge University Press, Cambridge; New York (2009)
12. Frühwirth, T.: Constraint handling rules - what else? In: Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) RuleML 2015. LNCS, vol. 9202, pp. 13–34. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21542-6_2
13. Geiselhart, F., Raiser, F., Sneyers, J., Frühwirth, T.: MTSeq: multi-touch-enabled CHR-based Music Generation and Manipulation (2010). https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.170/home/raiser/publications/Geiselhart2010.pdf

14. Hanus, M.: CHR(Curry): interpretation and compilation of constraint handling rules in curry. In: Pontelli, E., Son, T.C. (eds.) PADL 2015. LNCS, vol. 9131, pp. 74–89. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19686-2_6
15. Holzbaur, C., Frühwirth, T.: Compiling constraint handling rules. In: ERCIM/COMPULOG Workshop on Constraints, Amsterdam (1998)
16. Hudak, P.: Modular domain specific languages and tools. In: ICSR 1998, pp. 134–142 (1998). https://doi.org/10.1109/ICSR.1998.685738
17. Ivanović, D.: Implementing Constraint Handling Rules as a Domain-Specific Language Embedded in Java (2013). https://doi.org/10.48550/arXiv.1308.3939
18. Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 207–222. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73228-0_16
19. Karth, I., Smith, A.M.: WaveFunctionCollapse is constraint solving in the wild. In: FDG 2017, FDG 2017, pp. 1–10. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3102071.3110566
20. Lam, E., Sulzmann, M.: Towards Agent Programming in CHR (2006). https://www.semanticscholar.org/paper/Towards-Agent-Programming-in-CHR-Lam-Sulzmann/43277216bafb824d651c802ad487dbce8f7f7478
21. Lam, E.S.L., Sulzmann, M.: A concurrent constraint handling rules implementation in Haskell with software transactional memory. In: DAMP 2007, Nice, France, pp. 19–24. ACM Press (2007). https://doi.org/10.1145/1248648.1248653
22. Milewski, B.: Category Theory for Programmers. Lightning Source UK, Milton Keynes (2019). https://github.com/hmemcpy/milewski-ctfp-pdf
23. Nogatz, F., Frühwirth, T., Seipel, D.: CHR.js: a CHR implementation in JavaScript. In: Benzmüller, C., Ricca, F., Parent, X., Roman, D. (eds.) RuleML+RR 2018. LNCS, vol. 11092, pp. 131–146. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99906-7_9
24. Rechenberger, S.: Example Instances of FreeCHR (2023). https://gist.github.com/SRechenberger/739683a23f8a9978ae601c6c815d61c4
25. Rechenberger, S.: FreeCHR-Haskell (2023). https://github.com/SRechenberger/free-chr-hs
26. Rechenberger, S.: FreeCHR-Python (2023). https://github.com/SRechenberger/freechr-python
27. Rechenberger, S., Frühwirth, T.: FreeCHR: An Algebraic Framework for CHR-Embeddings (2023). https://doi.org/10.48550/arXiv.2306.00642
28. Schrijvers, T., Demoen, B.: The K.U. Leuven CHR system: implementation and application. In: CHR 2004, pp. 1–5 (2004). https://lirias.kuleuven.be/retrieve/33588
29. Sneyers, J., De Schreye, D.: APOPCALEAPS: automatic music generation with CHRiSM. In: BNAIC 2010 (2010). https://lirias.kuleuven.be/1584193
30. Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: constraint handling rules: a survey of CHR research from 1998 to 2007. TPLP **10**(1), 1–47 (2010). https://doi.org/10.1017/S1471068409990123
31. Thielscher, M.: Reasoning about actions with CHRs and finite domain constraints. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 70–84. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45619-8_6
32. Thielscher, M.: FLUX: a logic programming method for reasoning agents. TPLP **5**(4–5), 533–565 (2005). https://doi.org/10.1017/S1471068405002358
33. van Weert, P.: Efficient lazy evaluation of rule-based programs. IEEE Trans. Knowl. Data Eng. **22**(11), 1521–1534 (2010). https://doi.org/10.1109/tkde.2009.208

34. Van Weert, P., Schrijvers, T., Demoen, B., Schrijvers, T., Frühwirth, T.: K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In: CHR 2005. Department of Computer Science, K.U.Leuven (2005). https://lirias.kuleuven.be/1654703
35. Wibiral, T.: JavaCHR – A Modern CHR-Embedding in Java. Ph.D. thesis, Universität Ulm (2022). https://oparu.uni-ulm.de/xmlui/handle/123456789/43506
36. Wuille, P., Schrijvers, T., Demoen, B.: CCHR: The fastest CHR implementation, in C. In: CHR 2007, pp. 123–137 (2007). https://lirias.kuleuven.be/retrieve/22123