TPLP **25** (3): 340–373, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068425000043

FreeCHR – An Algebraic Framework for Constraint Handling Rules Embeddings[*]

SASCHA RECHENBERGER and THOM FRÜHWIRTH
Institute of Software Engineering and Programming Languages, Ulm University, Ulm, Germany
(*e-mails:* sascha.rechenberger@uni-ulm.de, thom.fruehwirth@uni-ulm.de)

submitted 28 March 2024; revised 4 February 2025; accepted 21 March 2025

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. We hereby 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. We give proofs for soundness and completeness with its original definition. We also propose a first abstract execution algorithm and prove correctness with the operational semantics. Finally, we show the practicability of our approach by giving two possible implementations of this algorithm in Haskell and Python. Under consideration in Theory and Practice of Logic Programming.

KEYWORDS: embedded domain-specific languages, rule-based programing languages, constraint handling rules, operational semantics, initial algebra semantics

1 Introduction

Constraint Handling Rules (CHR) is a rule-based programing language that is designed to be embedded into a general-purpose 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 (Frühwirth (2006); De Koninck *et al.* (2006)), it has been used, for example, to solve scheduling problems (Abdennadher and Marte (2000)) implement concurrent and multi-agent systems (Thielscher (2002, 2005); Lam and Sulzmann (2006, 2007)). In general, CHR is ideally suited for any problem that involves the transformation of collections of data, as programs consist of a set of rewriting rules, hiding away the process of finding suitable candidates for rule application.

[*] This is an extended version of a paper presented at the 2023 RuleML+RR Conference. The authors are grateful to Anna Fensel, Ana Ozaki and Ahmet Soylu, the conference program co-chairs, for encouraging us to submit the paper to the journal Theory and Practice of Logic Programming and for their help in overseeing the review process.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

Hereby, we get a purely declarative representation of the algorithm without the otherwise necessary boilerplate code.

The literature on CHR as a formalism consists of a rich body of theoretical work concerning CHR, including a rigorous formalization of its declarative and operational semantics (Frühwirth (2009); Sneyers *et al.* (2010); Frühwirth (2015)), relations to other rule-based formalisms (e.g., Betz (2007) and Gall and Frühwirth (2018)) and results on properties like confluence (Christiansen and Kirkeby (2015)).

Implementations of CHR exist for a number of languages, such as Prolog (Schrijvers and Demoen (2004)), C (Wuille *et al.* (2007)), C++ (Barichard (2024)), Haskell (Lam and Sulzmann (2007); Chin *et al.* (2003)), JavaScript (Nogatz *et al.* (2018)) and Java (Abdennadher *et al.* (2002); Van Weert *et al.* (2005); Ivanović (2013); Wibiral (2022)).

While the implementations adhere to the formally defined operational semantics, they are not direct implementations of a common formal model. Therefore, the two aspects of CHR (formalism and programing language) are not strictly connected with each other and there is hence no strict guarantee that the results on the formalism CHR are applicable on the programing language CHR. Although, such a strict connection is probably not entirely achievable (unless we define and use everything inside a proof assistant like *Coq* or *Agda*), it is desirable to have both formal definition and implementation as closely linked as possible. Additionally, to being able to directly benefit from theoretical results, implementors of CHR embeddings and users of the CHR language can also use the formally defined properties to define more profound tests for their software.

Another apparent issue within the CHR ecosystem is that many of the implementations of CHR are currently unmaintained. Although some of them are mere toy implementations, others might have been of practical use. One example is JCHR (Van Weert *et al.* (2005)) which would be a useful tool if it was kept on par with the development of Java, especially with modern build tools like *Gradle*. Having a unified formal model from which every implementation is derived could ease contributing to implementations of CHR as it provides a strict documentation and description of the system a priori. Also, different projects might even be merged which would prevent confusion due to multiple competing, yet very similar implementations, as it can be observed in the *miniKanren* ecosystem (e.g., there exist about 20 implementations of *miniKanren* dialects only for Haskell[2]).

A third major issue is that many implementations, like the aforementioned JCHR or CCHR (Wuille *et al.* (2007)), are implemented as external EDSL, this is, they rely on a separate compiler which translates CHR code into code of the host language. This makes it significantly less convenient to use and hence also less likely to be adopted into the standard distribution of the host language. This is somewhat demonstrated by the *K.U.Leuven CHR system* which is implemented as a library in Prolog and distributed as a standard package with *SWI-Prolog*,[3] or by the library *core.logic*[4] which implements *miniKanren* for the LISP dialect *Clojure*.

To solve the three issues stated above, we introduce the framework *FreeCHR* which formalizes the embedding of CHR, using *initial algebra semantics*. This common concept in functional programing is used to inductively define languages and their semantics

[2] https://minikanren.org/#implementations
[3] https://www.swi-prolog.org/pldoc/man?section=chr
[4] see https://github.com/clojure/core.logic

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

(Hudak (1998); Johann and Ghani (2007)). 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. To our knowledge, the presented execution algorithm and its implementations, although simple, will be the first implementations of CHR for which there are formal proofs of correctness. Also, by FreeCHR-instances being internal embeddings, we get basic tooling like syntax highlighting and type-checking for free (Fowler and Parsons (2011)) and are able to implement CHR as a library, without the need for any external tooling like source-to-source compilers.

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. 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, Section 4.2 lifts the *very abstract* operational semantics ω_a of CHR to the *very abstract* operational semantics $\omega_a^\star$ of FreeCHR and Section 4.3 proves soundness and completeness of $\omega_a^\star$ with respect to ω_a.

Section 5 introduces a simple execution algorithm for a formal instance with *very abstact* operational semantics and proves their correctness. This is the second main contribution of this paper, as it gives an example use of a proven correct guideline to implement FreeCHR.

This is finally demonstrated with the two case studies in Section 6. The section provides two implementations of the defined instance in Haskell and Python. The languages were chosen, as they can be considered to be on the opposite sides of a static-dynamic spectrum of programing languages. They will hence give a good intuition of how the host language flavors FreeCHR.

Finally, Section 7 discusses the limitations of FreeCHR as presented in this paper, Section 8 discusses related work, Section 9 will give an overview over planned and ongoing future work and Section 10 concludes the paper.

2 Preliminaries

2.1 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 Milewski (2019) as introductory literature.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

2.1.1 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 $A \sqcup B$. It can be understood as a formal analog to a `case-of` expression. Furthermore, we define two functions

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

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

2.1.2 Endofunctors

A **Set**-endofunctor[5] 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.[6] A signature $\Sigma = \{\sigma_1/a_1, \ldots, \sigma_n/a_n\}$, where σ_i are operators and a_i their arity, generates a functor

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

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* (this is, not nested) terms over the signature Σ.

Example 1.
The signature

$$\Gamma = \{\mathbf{0}/0, \oplus/2, \mathbf{1}/0, \otimes/2\}$$

models two constants $\mathbf{0}$ *and* $\mathbf{1}$*, as well as two binary operators* $\oplus$ *and* $\otimes$*. It generates the functor*

$$F_\Gamma X = \mathbb{1} \sqcup X \times X \sqcup \mathbb{1} \sqcup X \times X$$
$$F_\Gamma f = \mathbf{id}_\mathbb{1} \sqcup f \times f \sqcup \mathbf{id}_\mathbb{1} \sqcup f \times f$$

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

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

2.1.3 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* if and only if $h \circ \alpha = \beta \circ Fh$, that is 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.

Example 2.
The initial F_Γ-algebra

$$(\mu F_\Gamma, [\mathbf{0}, \oplus, \mathbf{1}, \otimes])$$

can be used to construct the nested expressions like $\mathbf{1} \oplus (\mathbf{1} \otimes \mathbf{0})$ of μF_Γ. The F_Γ algebras

$$(\mathbb{2}, [\mathit{false}, \vee, \mathit{true}, \wedge]) \qquad (\mathbb{N}_0, [0, +, 1, \cdot])$$

give to F_Γ the semantics of Boolean and arithmetic expressions, respectively. Using the catamorphism $(\!|[\mathit{false}, \vee, \mathit{true}, \wedge]|\!) : \mu F_\Gamma \to \mathbb{2}$, we can evaluate the expression from above as

$$\begin{aligned}
&(\!|[\mathit{false}, \vee, \mathit{true}, \wedge]|\!)(\mathbf{1} \oplus (\mathbf{1} \otimes \mathbf{0})) \\
=&(\!|[\mathit{false}, \vee, \mathit{true}, \wedge]|\!)(\mathbf{1}) \vee (\!|[\mathit{false}, \vee, \mathit{true}, \wedge]|\!)(\mathbf{1} \otimes \mathbf{0}) \\
=&\mathit{true} \vee (\!|[\mathit{false}, \vee, \mathit{true}, \wedge]|\!)(\mathbf{1}) \wedge (\!|[\mathit{false}, \vee, \mathit{true}, \wedge]|\!)(\mathbf{0}) \\
=&\mathit{true} \vee (\mathit{true} \wedge \mathit{false}) \\
=&\mathit{true} \vee \mathit{false} \\
=&\mathit{true}
\end{aligned}$$

Analogously, if we use $(\!|[0, +, 1, \cdot]|\!) : \mu F_\Gamma \to \mathbb{N}_0$, we get

$$\begin{aligned}
&(\!|[0, +, 1, \cdot]|\!)(\mathbf{1} \oplus (\mathbf{1} \otimes \mathbf{0})) \\
=&(\!|[0, +, 1, \cdot]|\!)(\mathbf{1}) + (\!|[0, +, 1, \cdot]|\!)(\mathbf{1} \otimes \mathbf{0}) \\
=&1 + (\!|[0, +, 1, \cdot]|\!)(\mathbf{1}) \cdot (\!|[0, +, 1, \cdot]|\!)(\mathbf{0}) \\
=&1 + (1 \cdot 0) \\
=&1 + 0 \\
=&1
\end{aligned}$$

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

Technically, the constants $\mathbf{0}$, $\mathbf{1}$, false, true, 0 *and* 1 *are supposed to be functions from the singleton set* $\mathbb{1}$ *to the respective carrier. Since* $\mathbb{1}$ *only has one element, there is essentially no difference between such a function and the constant itself. We will hence use the constant directly.*

2.2 Labeled transition systems

In this section, we will lay out our definition and notation of *labeled transition systems* and notions of soundness and completeness.

Definition 1 (Labeled transition system).
A labeled transition system (LTS) is a structure $\langle S, A, R\rangle$, *where* S *is the set of states called the* domain *of the system,* A *is a set of labels and* $R \in S \times A \times S$ *a ternary* transition relation.

For an LTS $\omega = \langle S, A, \mapsto\rangle$, we write $s \overset{a}{\mapsto} s'$, if $(s, a, s') \in (\mapsto)$ and $s_0 \overset{a_1}{\longmapsto} s_1 \overset{a_2}{\longmapsto} s_2 \overset{a_3}{\longmapsto} \ldots \overset{a_n}{\longmapsto} s_n$ for $s_0 \overset{a_1}{\longmapsto} s_1 \wedge s_1 \overset{a_2}{\longmapsto} s_2 \wedge \ldots \wedge s_{n-1} \overset{a_n}{\longmapsto} s_n$. Also, we will write $s_1 \overset{a}{\mapsto}{}^{*} s_n$ if there is an $a \in A$, such that $s_1 \overset{a}{\mapsto} s_2 \wedge \ldots \wedge s_{n-1} \overset{a}{\mapsto} s_n$, for $n \geq 0$.

Definition 2 (θ-Soundness).
Given two labeled transition systems $\omega = \langle S, A, \mapsto\rangle$ *and* $\omega' = \langle S, A', \hookrightarrow\rangle$. *For function* $\theta : A \to A'$, *we call* ω θ*-sound with respect to* ω' *if and only if*

$$s \overset{a}{\mapsto} s' \Longrightarrow s \overset{\theta(a)}{\hookrightarrow} s'$$

for all $s, s' \in S$ *and* $a \in A$[7].

θ-soundness means that any transition defined by ω is also defined by ω', when mapping labels with θ. This is typically used to verify a more constrained system (ω) against a more general one (ω').

Dually, there exists a concept of *completeness*.

Definition 3 (θ-Completeness).
Given two labeled transition systems $\omega = \langle S, A, \mapsto\rangle$ *and* $\omega' = \langle S, A', \hookrightarrow\rangle$. *For a function* $\theta : A \to A'$, *we call* ω θ-complete *with respect to* ω' *if and only if*

$$s \overset{a}{\mapsto} s' \Longleftarrow s \overset{\theta(a)}{\hookrightarrow} s'$$

for all $s, s' \in S$ *and* $a \in A$.

θ-completeness means that ω defines all transitions that are defined by ω', but using only the mapped labels of ω.

We will also need a slightly relaxed notion of soundness.

Definition 4 (θ-Soudness up to repeated transition).
Given two labeled transition systems $\omega = \langle S, A, \mapsto\rangle$ *and* $\omega' = \langle S, A', \hookrightarrow\rangle$. *For a function* $\theta : A \to A'$, *we call* ω θ*-sound up to repeated transition with respect to* ω' *if and only if*

[7] We will generally just use the symbol $\mapsto$ for the relation if the respective transition system is clear from the context.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

$$s \overset{a}{\mapsto} s' \Longrightarrow s \overset{\theta(a)^*}{\longrightarrow} s'$$

for all $s, s' \in S$, $a \in A$ *and* $l \geq 0$.

This definition says that an LTS is also θ-sound with respect to another if we can accomplish a transition in one LTS with zero or more transitions in the other.

3 CHR over non-Herbrand domains

The first implementations of CHR were embedded into the logical programing language Prolog, where terms like `3+4` or `f(a,b,c)` are not evaluated, as is the case in most other programing languages, but interpreted as themselves. This is called the *Herbrand* interpretation of terms. Since we want to embed CHR in any programing language, we need to generalize the language 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 that are inductively defined via Λ_T and the catamorphism $(\!|\tau_T|\!)$ evaluates those terms to values of T.

Definition 5 (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 $t' \in T$ if and only if $(\!|\tau_T|\!)(t) = t'$.

Example 3 (Boolean data type).
The signature

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

defines Boolean terms[8]. Σ_2 *generates the functor*

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

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

$$(\!|\tau_2|\!)(0 < 4 \wedge 4 \leq 6) = (\!|\tau_2|\!)(0 < 4) \wedge (\!|\tau_2|\!)(4 \leq 6) = \mathit{true} \wedge \mathit{true} = \mathit{true}$$

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

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

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

Definition 6 (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 by 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 7 (CHR programs).
CHR programs are sets of multiset-rewriting rules of the form

$$N \mathbin{@} K \setminus R \iff G \mid B$$

For a set C*, called the* domain *of the program, for which there is a data type* $\mathcal{L}C = \langle C, \Lambda_C, \tau_C \rangle$*,* $K, R \in \texttt{list}C$ *are called the* kept *and* removed head*, respectively. The backslash* ($\setminus$) *is used to separate the kept and removed part of the head. Either* K *or* R *must be non-empty. If either head part is empty, the separator is omitted as well.*

$\texttt{list}X = \bigcup_{i \in \mathbb{N}_0} X^i$ *maps a set* X *to the set of lists over* X*, with* $X^0 = \varepsilon$ *being the empty sequence, and functions* $f : A \to B$ *to functions*

$$\texttt{list}f\,(a_1, \ldots, a_n) = (f(a_1), \ldots, f(a_n))$$

The optional $G \in \mu\Lambda_2$ *is called the* guard*. If* G *is omitted, we assume* $G \equiv_2 true$ *and omit the pipe* ($\mid$) *which separates the guard from the body as well.* $B \in \texttt{mset}\mu\Lambda_C$ *is called the* body*. The functor* `mset` *maps sets* X *to the set of multisets over* X *and functions* $f : X \to Y$ *to functions*

$$\texttt{mset}f\,(\{a_1, a_2, \ldots\}) = \{f(a_1), f(a_2), \ldots\}$$

N is an optional name for the rule, which is generally used for debugging and tracing. The @ *symbol is used to separate the name of the rule from the rule itself. It is also omitted if the rule is not given a name.*

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 functions or operators which evaluate to Boolean, or values of C, respectively.

In literature, rules with an empty kept head are called *simplification* rules and rules with an empty removed head *propagation* rules. Rules which have both a removed and kept head are called *simpagation* rules.

Definition 7 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 as by Betz (2007)) into CHR (Frühwirth, 2009, Chapter 6.2). *Positive*

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

means that the body of the rule contains only user constraints (this is, 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 4 (Euclidean algorithm).
The program $\mathrm{GCD} = \{zero, subtract\}$[9]

$$zero \ @ \ 0 \ \Leftrightarrow \ \varnothing$$
$$subtract \ @ \ N \ \setminus \ M \ \Leftrightarrow \ 0 < N \wedge 0 < M \wedge N \leq M \mid M - N$$

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

Definition 8 (C-instances of rules).
For a positive range-restricted rule

$$R \ @ \ k_1, \ldots, k_n \ \setminus \ r_1, \ldots, r_m \ \Leftrightarrow \ G \mid B$$

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

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

*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 all variables and evaluating the then ground terms in the body of the rule, using the Λ_C-catamorphism $(\!|\tau_C|\!)$. The functor `mset` is used to lift $(\!|\tau_C|\!)$ into the multiset.

Example 5.
Given a body $\{M - N\}$ and a substitution $\sigma = \{N \mapsto 4, M \mapsto 6\}$, the body is instantiated like

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

With Example 5, 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, for instance, an expression $(3 + 4) \in \mu\Lambda_{\mathbb{N}_0}$ is evaluated to itself, as it is the case in Prolog.

[9] We will typically use the rule name as a symbol for the rule as a whole.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

$$\frac{}{S \overset{R}{\longmapsto} S}\ \text{EMPTY} \qquad \frac{r \in R \qquad S \overset{\{r\}}{\longmapsto} S' \qquad S' \overset{R}{\longmapsto} S''}{S \overset{R}{\longmapsto} S''}\ \text{STEP}$$

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

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

Example 6 (C-instances).
If we instantiate the rule

$$subtract\ @\ N\ \backslash\ M\ \Leftrightarrow\ 0 < N \wedge 0 < M \wedge N \leq M\ |\ M - N$$

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*

$$(subtract)\sigma_1 = subtract\ @\ 4\ \backslash\ 6\ \Leftrightarrow\ 0 < 4 \wedge 0 < 6 \wedge 4 \leq 6\ |\ 2$$
$$(subtract)\sigma_2 = subtract\ @\ 0\ \backslash\ 6\ \Leftrightarrow\ 0 < 0 \wedge 0 < 6 \wedge 0 \leq 6\ |\ 6$$

Both instances are elements of the $\mathbb{N}_0$*-grounding* $\Gamma_{\mathbb{N}_0}(\text{GCD})$ *of the program in Example 4.*

Classically, the guard G contains constraints which are defined with respect to a constraint theory $\mathcal{CT}$. We typically write $\mathcal{CT} \models G$[10] to denote that the guard is satisfiable with respect to $\mathcal{CT}$ 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

$$\tau_2 \models G \Longleftrightarrow G \equiv_2 true \qquad \tau_2 \models \neg G \Longleftrightarrow G \equiv_2 false$$

Note that we always need a data type $\mathcal{L}2$. In Prolog, for instance, 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[11] over the elements of C.

Definition 9 (Very abstract operational semantics of CHR).
The very abstract operational semantics of CHR programs over a domain C *is given by the labeled transition system*

$$\omega_a = \langle \texttt{mset}C, \mathbf{PRG}_C, \mapsto \rangle$$

where the transition relation $(\mapsto)$ *is defined by the inference rules in Figure 1.*

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

Example 7 (ω_a-transitions).
Intuitively, both

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

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

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

hold. Hence, we can prove the transition $\{4,6\} \overset{\{subtract\}}{\longmapsto} \{4,2\}$, *but not* $\{0,6\} \overset{\{subtract\}}{\longmapsto} \{0,6\}$.

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

Example 8 (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*

$$\{4,6\} \overset{\{subtract\}}{\longmapsto} \{4,2\} \overset{\{subtract\}}{\longmapsto} \{2,2\} \overset{\{subtract\}}{\longmapsto} \{2,0\} \overset{\{zero\}}{\longmapsto} \{2\}$$

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.

4.1 Syntax

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

Definition 10 (Syntax of FreeCHR programs).
The functor[12]

$$\mathrm{CHR}_C D = \mathtt{list}2^C \times \mathtt{list}2^C \times 2^{\mathtt{list}C} \times (\mathtt{mset}C)^{\mathtt{list}C} \sqcup D \times D$$
$$\mathrm{CHR}_C f = \mathbf{id} \sqcup f \times f$$

describes the syntax of FreeCHR programs.

The set $\mathtt{list}2^C \times \mathtt{list}2^C \times 2^{\mathtt{list}C} \times (\mathtt{mset}C)^{\mathtt{list}C}$ is the set of single rules. The kept and removed head of a rule are sequences of functions in $\mathtt{list}2^C$ 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 $2^{\mathtt{list}C}$ 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 $(\mathtt{mset}C)^{\mathtt{list}C}$ and maps the matched values to a multiset of newly generated values.

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

$$\rho\colon \mathtt{list}2^C \times \mathtt{list}2^C \times 2^{\mathtt{list}C} \times (\mathtt{mset}C)^{\mathtt{list}C} \longrightarrow D \qquad \nu\colon D \times D \to D$$

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

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

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 1 (Free CHR_C-algebra).
With

$$\mu\mathrm{CHR}_C = \mathtt{list}2^C \times \mathtt{list}2^C \times 2^{\mathtt{list}C} \times (\mathtt{mset}C)^{\mathtt{list}C} \sqcup \mu\mathrm{CHR}_C \times \mu\mathrm{CHR}_C$$

and labels/injections

$$\begin{aligned} \mathit{rule} &: \mathtt{list}2^C \times \mathtt{list}2^C \times 2^{\mathtt{list}C} \times (\mathtt{mset}C)^{\mathtt{list}C} \longrightarrow \mu\mathrm{CHR}_C \\ \odot &: \mu\mathrm{CHR}_C \times \mu\mathrm{CHR}_C \longrightarrow \mu\mathrm{CHR}_C \end{aligned}$$

$\mathrm{CHR}_C^\star = (\mu\mathrm{CHR}_C, [\mathit{rule}, \odot])$ *is the free* CHR_C*-algebra.*

Proof.
We show that $\mathrm{CHR}_C^\star = (\mu\mathrm{CHR}_C, [\mathit{rule}, \odot])$ is the *free* CHR_C-algebra, by constructing the CHR_C-catamorphism

$$(\!|[\rho, \nu]|\!) : \mu\mathrm{CHR}_C \longrightarrow (A, [\rho, \nu])$$

for any CHR_C-algebra $(A, [\rho, \nu])$. By definition of the free CHR_C-algebra,

$$(\!|[\rho, \nu]|\!)([\mathit{rule}, \odot]\,(p)) = [\rho, \nu]\,((\mathrm{CHR}_C(\!|[\rho, \nu]|\!))(p)) \tag{1}$$

needs to be true for any $p \in \mu\mathrm{CHR}_C$.

Case 1 $(p = \mathit{rule}(k, r, g, b))$.

$$\begin{aligned} & (\!|[\rho, \nu]|\!)([\mathit{rule}, \odot]\,(k, r, g, b)) = [\rho, \nu]\,((\mathrm{CHR}_C(\!|[\rho, \nu]|\!))(k, r, g, b)) \\ \Leftrightarrow\ & (\!|[\rho, \nu]|\!)(\mathit{rule}(k, r, g, b)) = [\rho, \nu]\,(\mathbf{id}(k, r, g, b)) \\ \Leftrightarrow\ & (\!|[\rho, \nu]|\!)(\mathit{rule}(k, r, g, b)) = [\rho, \nu]\,(k, r, g, b) \\ \Leftrightarrow\ & (\!|[\rho, \nu]|\!)(\mathit{rule}(k, r, g, b)) = \rho(k, r, g, b) \end{aligned}$$

Case 2 $(p = p_1 \odot p_2)$.

$$\begin{aligned} & (\!|[\rho, \nu]|\!)([\mathit{rule}, \odot]\,(p_1, p_2)) = [\rho, \nu]\,((\mathrm{CHR}_C(\!|[\rho, \nu]|\!))(p_1, p_2)) \\ \Leftrightarrow\ & (\!|[\rho, \nu]|\!)(p_1 \odot p_2) = [\rho, \nu]\,((\!|[\rho, \nu]|\!) \times (\!|[\rho, \nu]|\!))(p_1, p_2)) \\ \Leftrightarrow\ & (\!|[\rho, \nu]|\!)(p_1 \odot p_2) = [\rho, \nu]\,((\!|[\rho, \nu]|\!)(p_1), (\!|[\rho, \nu]|\!)(p_2)) \\ \Leftrightarrow\ & (\!|[\rho, \nu]|\!)(p_1 \odot p_2) = \nu((\!|[\rho, \nu]|\!)(p_1), (\!|[\rho, \nu]|\!)(p_2)) \end{aligned}$$

Therefore, (1) holds if and only if

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

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

$$\frac{}{S \overset{p}{\mapsto} S}\ \text{PASS/FINAL} \qquad \frac{S \overset{p_i}{\mapsto} S' \quad S' \overset{p_1 \odot \ldots \odot p_i \odot \ldots \odot p_l}{\mapsto} S''}{S \overset{p_1 \odot \ldots \odot p_i \odot \ldots \odot p_l}{\mapsto} S''}\ \text{STEP}_i$$

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

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

for all CHR_C-algebras $(A, [\rho, \nu])$. Hence, there is a unique homomorphism

$$(\!|[\rho, \nu]|\!) : \mathrm{CHR}_C^\star \to (A, [\rho, \nu])$$

for any CHR_C-algebra $(A, [\rho, \nu])$, making $\mathrm{CHR}_C^\star$ the free CHR_C-algebra. □

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.

We can easily see that $\odot$ is associative up to isomorphism.[13] We thus will not explicitly write parentheses and generally use chained expressions like $p_1 \odot \ldots \odot p_l$ for some $l \in \mathbb{N}$.

Example 9 (Euclidean algorithm (cont.)).
The program $gcd = zero \odot subtract$ *with*

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

implements the euclidean algorithm, as defined in Example 4. λ-abstractions are used for ad-hoc definitions of functions.

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 labeled transition system, where states are multisets over elements of C.

Definition 11 (Very abstract operational semantics $\omega_a^\star$).
The very abstract operational semantics of FreeCHR is defined as the labeled transition system

$$\omega_a^\star = \langle \mathtt{mset}C, \mu\mathrm{CHR}_C, \mapsto \rangle$$

where the transition relation $(\mapsto)$ *is defined by the inference rules in Figure 2.*

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 $p_1 \odot \ldots \odot p_l$ (for $1 \leq i \leq l$) and from S' to S'' with the whole composition. The idea is that, without loss of generality p_i is a

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

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

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.

We want to demonstrate the *very abstract* operational semantics of FreeCHR on a small example.

Example 10 (Euclidean algorithm (cont.)).
Given the program gcd of Example 9 and the initial state $\{12, 9\}$, *we can prove the derivation*

$$\{12, 9\} \overset{gcd}{\longmapsto} \{3\}$$

as follows. We first use the STEP *transition to apply the subtract rule to the initial state.*

$$\frac{\{12, 9\} \overset{subtract}{\longmapsto} \{3, 9\} \qquad \{3, 9\} \overset{zero \odot subtract}{\longmapsto} \{3\}}{\{12, 9\} \overset{zero \odot subtract}{\longmapsto} \{3\}} \text{STEP}_2$$

and prove that this transition is valid

$$\frac{0 < 9 \wedge 0 < 12 \wedge 9 \leq 12 \equiv true}{\{12, 9\} \uplus \varnothing \overset{subtract}{\longmapsto} \{9\} \uplus \{3\} \uplus \varnothing} \text{APPLY}$$

We repeat this process for the transitions $\{3, 9\} \overset{gcd}{\longmapsto} \{3, 6\}$, $\{3, 6\} \overset{gcd}{\longmapsto} \{3, 3\}$ *and* $\{3, 3\} \overset{gcd}{\longmapsto} \{3, 0\}$. *At this point, we have to apply the zero rule to remove the* 0 *and use the* PASS/FINAL *transition to prove the empty transition* $\{3\} \overset{gcd}{\longmapsto} \{3\}$.

$$\frac{0 = 0 \wedge true \equiv true}{\{3, 0\} \uplus \varnothing \overset{zero}{\longmapsto} \{3\} \uplus \varnothing \uplus \varnothing} \text{APPLY} \qquad \frac{\{3, 0\} \overset{zero}{\longmapsto} \{3\} \qquad \dfrac{}{\{3\} \overset{zero \odot subtract}{\longmapsto} \{3\}} \text{PASS/FINAL}}{\{3, 0\} \overset{zero \odot subtract}{\longmapsto} \{3\}} \text{STEP}_2$$

With this done, we have fully proven the derivation.

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

To prove the soundness and completeness of $\omega_a^\star$ with respect to 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 (Frühwirth, 2009, Chapter 6).

Definition 12 (Embedding FreeCHR into CHR).
Let

$$\begin{aligned} \Theta : \mu\text{CHR}_C &\longrightarrow \mathbf{PRG}_C \\ \Theta\left(rule\left(k, r, g, b\right)\right) &= \{R@v_1, \ldots, v_n \setminus v_{n+1}, \ldots, v_{n+m} \Leftrightarrow G | b\left(v_1, \ldots, v_{n+m}\right)\} \\ \Theta\left(p_1 \odot \ldots \odot p_l\right) &= \Theta\left(p_1\right) \cup \ldots \cup \Theta\left(p_l\right) \end{aligned}$$

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

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

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

Example 11.
If we apply the embedding Θ to the program gcd (including the evaluation of function applications[14]*), we get the program*

$$zero \mathbin{@} \varepsilon \setminus n \Leftrightarrow n = 0 \wedge true \mid \varnothing$$
$$subtract \mathbin{@} n \setminus m \Leftrightarrow 0 < n \wedge 0 < m \wedge n \leq m \mid m - n$$

*with universally quantified variables n and m. The program corresponds to the head-normalization (*Duck (2005)*) of the program in Example 4.*

We now prove Θ-soundness of $\omega_a^\star$ with respect to ω_a, this is, 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 1 (Θ-Soundness of $\omega_a^\star$).
$\omega_a^\star$ is Θ-sound with respect to ω_a, this is, for $S, S' \in \texttt{mset}\, C$ and $p \in \mu\mathrm{CHR}_C$,

$$S \overset{p}{\mapsto} S' \Longrightarrow S \overset{\Theta(p)}{\longmapsto} S'$$

Proof.
We prove soundness by induction over the inference rules of $\omega_a^\star$.

*Induction Base Case (*PASS/FINAL*).*
Follows directly from EMPTY.

*Induction Base Case (*APPLY*).*
Given a proof

$$\frac{G \equiv_2 true}{\{c_1, \ldots, c_{n+m}\} \uplus \Delta S \overset{p}{\mapsto} \{c_1, \ldots, c_n\} \uplus b(c_1, \ldots, c_{n+m}) \uplus \Delta S}\,\text{APPLY}$$

with $r = rule((k_1, \ldots, k_n), (r_1, \ldots, r_m), g, b)$ and

$$G = k_1(t_1) \wedge \ldots \wedge k_n(t_n) \wedge r_1(t_{n+1}) \wedge \ldots \wedge r_m(t_{n+m}) \wedge g(t_1, \ldots, t_{n+m})$$

there is a C-instance

$$R \mathbin{@} c_1, \ldots, c_n \setminus c_{n+1}, \ldots, c_{n+m} \Leftrightarrow G \mid b(c_1, \ldots, c_{n+m}) \in \Gamma_C(\Theta(p))$$

such that $\tau_2 \models G$. By Definition 12 we know that $\{p'\} = \Theta(p)$. We can thus prove $S \overset{\Theta(p)}{\longmapsto} S'$ by

$$\frac{R @ c_1, \ldots, c_n \setminus c_{n+1}, \ldots, c_{n+m} \Leftrightarrow G | b(c_1, \ldots, c_{n+m}) \in \Gamma_C(\Theta(p')) \quad \tau_2 \models G}{\underbrace{\{c_1, \ldots, c_{n+m}\} \uplus \Delta S}_{=S} \overset{\{p'\}}{\longmapsto} \underbrace{\{c_1, \ldots, c_n\} \uplus b(c_1, \ldots, c_{n+m}) \uplus \Delta S}_{=S'}}\,\text{APPLY}$$

Induction Step (STEP$_i$)
Given a proof

$$\frac{S \overset{p_i}{\mapsto} S' \qquad S' \overset{p}{\mapsto} S''}{S \overset{p}{\mapsto} S''}\,\text{STEP}_i$$

[14] this is, β-reductions

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

with $p = p_1 \odot \ldots \odot p_i \odot \ldots \odot p_l$, for $1 \leq i \leq l$, we assume

$$\forall i \in 1, \ldots, l. S \overset{p_i}{\mapsto} S' \Longrightarrow S \overset{\Theta(p_i)}{\longmapsto} S' \tag{2}$$

$$S' \overset{p}{\mapsto} S'' \Longrightarrow S' \overset{\Theta(p)}{\longmapsto} S'' \tag{3}$$

as induction hypotheses.

Since we can, without loss of generality, assume that $p_i = rule(\ldots)$, we know from Definition 12 that $\{p'\} = \Theta(p_i) \subseteq \Theta(p)$. By induction hypothesis (2) we know from $S \overset{p_i}{\longmapsto} S'$ that $S \overset{\Theta(p_i)}{\longmapsto} S'$. By induction hypothesis (3) we know from $S' \overset{p}{\mapsto} S''$ that $S' \overset{\Theta(p)}{\longmapsto} S''$. We can thus prove $S \overset{\Theta(p)}{\longmapsto} S''$ by

$$\frac{p' \in \Theta(p) \qquad S \overset{\{p'\}}{\longmapsto} S' \qquad S' \overset{\Theta(p)}{\longmapsto} S''}{S \overset{\Theta(p)}{\longmapsto} S''}\text{STEP}$$

□

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 Θ-completeness, this is, we can prove any derivation with a program $\Theta(p)$ under ω_a with a program p under $\omega_a^\star$.

Theorem 2 (Θ-completeness of $\omega_a^\star$).
$\omega_a^\star$ is Θ-complete with respect to ω_a, this is, for $S, S' \in \texttt{mset}C$ and $p \in \mu\text{CHR}_C$,

$$S \overset{\Theta(p)}{\longmapsto} S' \Longrightarrow S \overset{p}{\mapsto} S'$$

Proof.
We prove Θ-completeness by induction over the inference rules of ω_a

*Induction Base Case (*EMPTY*).*
Follows directly from PASS/FINAL

*Induction Base Case (*APPLY*).*
Given a proof

$$\frac{R@c_1, \ldots, c_n \setminus c_{n+1}, \ldots, c_{n+m} \Leftrightarrow G | B \in \Gamma_C(\Theta(p)) \qquad \tau_2 \models G}{\{c_1, \ldots, c_{n+m}\} \uplus \Delta S \overset{\Theta(p)}{\longmapsto} \{c_1, \ldots, c_n\} \uplus B \uplus \Delta S}\text{APPLY}$$

By definition of ω_a, we know that $\Theta(p) = \{p'\}$. Therefore, $p = rule(k, r, g, b)$, and hence $G = k_1(c_1) \wedge \ldots \wedge r_m(c_{n+m}) \wedge g(c_1, \ldots, c_{n+m})$ and $B = b(c_1, \ldots, c_{n+m})$. From $\tau_2 \models G$ follows $G \equiv_2 true$. We can thus construct a proof

$$\frac{G \equiv_2 true}{\{c_1, \ldots, c_{n+m}\} \uplus \Delta S \overset{p}{\mapsto} \{c_1, \ldots, c_n\} \uplus B \uplus \Delta S}\text{APPLY}$$

*Induction Step (*STEP*).*
Given a proof

$$\frac{p' \in \Theta(p) \quad S \overset{\{r'\}}{\longmapsto} S' \qquad S' \overset{\Theta(p)}{\longmapsto} S''}{S \overset{\Theta(p)}{\longmapsto} S''}\text{STEP}$$

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

By Definition 12, we know from $r' \in \Theta(p)$ that for $p = p_1 \odot \ldots \odot p_i \odot \ldots \odot p_l$ (with $l \geq 1$) there is a p_i such that $\{p'\} = \Theta(p_i)$. We hence assume as induction hypotheses that

$$S \overset{\{p'\}}{\longmapsto} S' \Longrightarrow S \overset{p_i}{\mapsto} S' \tag{4}$$

$$S' \overset{\Theta(p)}{\longmapsto} S'' \Longrightarrow S' \overset{p}{\mapsto} S'' \tag{5}$$

From $S \overset{\{r\}}{\longmapsto} S'$ and induction hypothesis (4) follows $S \overset{p_i}{\mapsto} S'$. From $S' \overset{\Theta(p)}{\longmapsto} S''$ and induction hypothesis (5) follows $S' \overset{p}{\mapsto} S''$. We can hence construct the proof

$$\frac{S \overset{p_i}{\mapsto} S' \qquad S' \overset{p}{\mapsto} S''}{S \overset{p}{\mapsto} S''} \text{STEP}$$

□

With Theorem 1 and Theorem 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 with respect to $\omega_a^\star$, which is ongoing future work.

5 An instance with *very abstract* operational semantics

In this section, we want to present a simple execution algorithm and prove its adherence to $\omega_a^\star$ by using the inductive nature of our definition of syntax and semantics of FreeCHR. This will serve both as a first, albeit simple, guideline for implementations, as well as a case study on how a FreeCHR execution algorithm can be defined and verified.

5.1 Instances

Since the operational semantics define the execution of FreeCHR programs as a state transition system, an obvious way to implement FreeCHR is by mapping programs to functions mapping one state to its successor.

Definition 13 (FreeCHR instance).
An instance *of FreeCHR is a* CHR_C*-algebra*

$$\left((\mathtt{mset}C)^{\mathtt{mset}C}, [\mathtt{rule}, \mathtt{compose}]\right)$$

with

$$\mathtt{rule} : \mathtt{list}2^C \times \mathtt{list}2^C \times 2^{\mathtt{list}C} \times (\mathtt{mset}C)^{\mathtt{list}C} \longrightarrow (\mathtt{mset}C)^{\mathtt{mset}C}$$
$$\mathtt{compose} : (\mathtt{mset}C)^{\mathtt{mset}C} \times (\mathtt{mset}C)^{\mathtt{mset}C} \longrightarrow (\mathtt{mset}C)^{\mathtt{mset}C}$$

A FreeCHR instance hence needs to define the two functions `rule` and `compose` which implement the execution of programs and hence the operational semantics of the program. `compose` implements the application of a single rule to a state if it is possible. `compose` implements the execution strategy of, fundamentally, a sequence of rules to a state. Both are only concerned with single execution steps. We hence also need the function

$$\mathtt{run} : (\mathtt{mset}C)^{\mathtt{mset}C} \longrightarrow (\mathtt{mset}C)^{\mathtt{mset}C}$$

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

which maps a function performing singular steps to a function applying this function until a final state is reached. In CHR this generally means, until no rules are applicable to the state.

Since an instance practically also defines operational semantics it makes sense to directly associate it with an LTS as well.

Definition 14 (LTS of a FreeCHR instance).
Let

$$I = \left((\texttt{mset}C)^{\texttt{mset}C}, [\rho, \nu]\right)$$

I implies an LTS

$$crI = \langle \texttt{mset}C, L, \mapsto \rangle$$

where the labels L are recursively defined as

$$L = \left\{\rho(k, r, g, b) \mid k, r \in \texttt{list}(2^C), g \in 2^{\texttt{list}C}, b \in (\texttt{mset}C)^{\texttt{list}C}\right\} \cup \{\nu(f, g) \mid f, g \in L\}$$

and the transition relation as

$$(\mapsto) = \left\{ s \overset{f}{\mapsto} s' \mid s \in \texttt{mset}C, f \in L, f(s) \equiv_{\texttt{mset}C} s' \right\}$$

With this, we can easily verify an instance against operational semantics, using the concepts from Section 2.2. We will do so from the perspective of the operational semantics, instead of from the perspective of the implementation. To verify, that an instance only performs valid transitions, we hence prove Θ-*completeness*

$$S \overset{p}{\mapsto} S' \Longleftarrow S \overset{(\!|\alpha|\!)(p)}{\longmapsto} S'$$

Consequentially, to prove that the implementation can perform all valid transitions, we prove Θ-*soundness up to repeated transition*

$$S \overset{p}{\mapsto} S' \Longrightarrow S \overset{(\!|\alpha|\!)(p)}{\longmapsto}{}^{*} S'$$

We need the relaxation *"up to repeated transition"* since the instances, and hence their respective LTS, are only defined for singular steps.

5.2 Execution algorithm

The algorithm presented in this section implements the operations required for a FreeCHR instance as non-deterministic imperative pseudo-code. The *rule* function is implemented by RULE, ⊙ by COMPOSE, in Algorithm 1.

*Definition 15 (Pure CHR-programs in **Set** (cont.)).*
Algorithm 1 shows the abstract implementation of a CHR_C*-algebra*

$$\mathbf{Exec}_a = \left((\texttt{mset}C)^{\texttt{mset}C}, [\text{RULE}, \text{COMPOSE}]\right)$$

The function RULE first checks whether there is a matching $\{c_1, \ldots, c_{n+m}\} \subseteq S$, such that head constraints $\texttt{kept}_i$ applied to c_i and $\texttt{removed}_j$ applied to c_{n+j} evaluate to *true*, as well as the guard, applied to all constraints of the matching (l. 2). If such a subset

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

Algorithm 1 Implementation of $Exec_a$

```
1:  function rule(kept, removed, guard, body, S)
2:    if ∃{c_1, ..., c_{n+m}} ⊆ S.kept_1(c_1)∧...∧removed_m(c_{n+m})∧guard(c_1, ..., c_{n+m}) ≡_2 true
      then
3:      return (S∪{c_{n+1}, ..., c_{n+m}}) ⊎ body(c_1, ..., c_{n+m})
4:    else
5:      return S
6:    end if
7:  end function
8:
9:  function compose((p_1, ..., p_l), S)
10:   if ∃p ∈ {p_1, ..., p_l} .p(S) ≠_msetC S then
11:     return p(S)
12:   else
13:     return S
14:   end if
15: end function
```

Algorithm 2 Implementation of run_f

```
1: function run(f, S)
2:     repeat
3:       S' ← S
4:       S ← f(S)
5:     until S = S'
6:     return S
7: end function
```

exists, the constraints c_{n+j} with $1 \leq j \leq m$ are removed and the multiset resulting from applying the body to the matching will be added to S (l. 3). The function COMPOSE checks if there is a $p \in \{p_1, \ldots, p_l\}$ with $p(S) \neq S$ (l. 10). If this is the case, p is applied to S (l. 11). If in either function the respective check does not succeed, S is returned unmodified (l. 5 and 13, respectively). The execution strategy implemented by COMPOSE is to non-deterministically select an applicable program and apply it to the state.

Note, that we will freely use currying on the last argument of either function, for instance, RULE(k, r, g, b) is a function from $\texttt{mset}C$ to $\texttt{mset}C$, and RULE(k, r, g, b, S) evaluates to an $S' \in \texttt{mset}C$. Also, COMPOSE will accept an arbitrary number of subprograms (but always an input state).

Finally, we need to implement the *run* function.

*Definition 16 (Pure CHR-programs in **Set** (cont.)).*
Algorithm 2 implements application until exhaustion for FreeCHR programs defined using Algorithm 1.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

RUN(f, S) applies the function f to the argument state S repeatedly. After each iteration, it is compared to the last state. As soon as both are equal it is assumed that no more rules could be applied. Then, the state is returned as the final state.

Note, that this algorithm would also terminate if a propagation rule with an empty body is applied. As such rules are nonsensical, we will ignore this case.

5.3 Correctness

We will now prove the correctness of $\mathbf{Exec}_a$. With $\mathbf{exec}_a = [\text{RULE}, \text{COMPOSE}]$, let

$$(\!|\mathbf{exec}_a|\!) : \text{CHR}_C^\star \to \mathbf{Exec}_a$$

be the CHR_C-catamorphism to $\mathbf{Exec}_a$, this is, the interpretation of programs in μCHR_C that respects the semantics of Algorithm 1.

Theorem 3.
The very abstract operational semantics $\omega_a^\star$ is $(\!|\mathbf{exec}_a|\!)$-complete with respect to $\mathbf{Exec}_a$ (Algorithm 1), this is,

$$S \overset{(\!|\mathbf{exec}_a|\!)(p)}{\longmapsto} S' \implies S \overset{p}{\mapsto} S'$$

Proof.
To prove the theorem, we will first discern two basic cases.

Case 3 $(S = S')$.
This case is trivial by rule PASS/FINAL.

Case 4 $(S \neq S')$.
We prove this case via induction over the structure of $p \in \text{CHR}_C$.

Induction Base Case $(p = rule(k, r, g, b))$.
The transition

$$S \overset{(\!|\mathbf{exec}_a|\!)(rule(k,r,g,b))}{\longmapsto} S'$$

implies that

$$\begin{aligned} &(\!|\mathbf{exec}_a|\!)(rule(k, r, g, b))(S) \\ =\ &\text{RULE}(k, r, g, b, S) \\ \equiv\ &S' \end{aligned}$$

From Algorithm 1 follows that the condition in line 2 must be true, since there were changes to the state. This implies that there is a subset $\{c_1, \ldots, c_{n+m}\} \subseteq S$ *for which*

$$k_1(c_1) \wedge \ldots \wedge k_n(c_n) \wedge r_1(c_{n+1}) \wedge \ldots \wedge r_m(c_{n+m}) \wedge g(c_1, \ldots, c_{n+m}) \equiv_2 true$$

Hence

$$S = \{c_1, \ldots, c_{n+m}\} \uplus \Delta S$$

and

$$(S \cup \{c_{n+1}, \ldots, c_{n+m}\}) \uplus b(c_1, \ldots, c_{n+m}) = \{c_1, \ldots, c_n\} \biguplus b(c_1, \ldots, c_n) \biguplus \Delta S$$

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

From this, we can construct a proof

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

Induction Step ($p = p_1 \odot \ldots \odot p_n$).
As the induction hypothesis, we assume

$$\forall p \in \mathrm{CHR}_C . S \overset{(\!|\mathbf{exec}_a|\!)(p)}{\longmapsto} S' \Longrightarrow S \overset{p}{\mapsto} S'$$

The transition

$$S \overset{(\!|\mathbf{exec}_a|\!)(p_1 \odot \ldots \odot p_n)}{\longmapsto} S'$$

implies that

$$(\!|\mathbf{exec}_a|\!)(p_1 \odot \ldots \odot p_n)(S) = \mathrm{COMPOSE}(p_1, \ldots, p_n, S) \equiv_{\mathtt{mset}C} S'$$

From Algorithm 1, especially line 10, we know that there must hence be a p_i with $1 \leq i \leq n$ such that $(\!|\mathbf{exec}_a|\!)(p_i)(S) \equiv_{\mathtt{mset}C} S'$, or in terms of the transition relation

$$S \overset{(\!|\mathbf{exec}_a|\!)(p_i)}{\longmapsto} S'$$

By induction hypothesis, we then have a proof for $S \overset{p_i}{\longmapsto} S'$. We can thus construct a proof

$$\frac{S \overset{p_i}{\longmapsto} S' \qquad \dfrac{}{S' \overset{p_1 \odot \ldots \odot p_n}{\longmapsto} S'}\ \text{PASS/FINAL}}{S \overset{p_1 \odot \ldots \odot p_n}{\longmapsto} S'}\ \text{STEP}_i$$

□

Theorem 4.
The very abstract operational semantics $\omega_a^\star$ is $(\!|\mathbf{exec}_a|\!)$-sound up to repeated transition with respect to $\mathbf{Exec}_a$ (Algorithm 1), this is,

$$S \overset{(\!|\mathbf{exec}_a|\!)(p)}{\longmapsto}{}^{*} S' \Longleftarrow S \overset{p}{\mapsto} S'$$

Proof.
We prove the theorem by induction over the inference rules of $\omega_a^\star$.

*Induction Base Case (*PASS/FINAL*).*
Trivial, since $S \overset{(\!|\mathbf{exec}_a|\!)(p)}{\longmapsto}{}^{*} S$ is always true for zero transition steps

*Induction Base Case (*APPLY*).*
Given a proof

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

We want to show that

$$\{c_1, \ldots, c_{n+m}\} \uplus \Delta S \overset{(\!|\mathbf{exec}_a|\!)(rule(k,r,g,b))}{\longmapsto} \{c_1, \ldots, c_n\} \uplus b\,(c_1, \ldots, c_{n+m}) \uplus \Delta S$$

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

or respectively

$$\begin{aligned}&(\!|\mathbf{exec}_a|\!)(\mathit{rule}(k,r,g,b))(\{c_1,\ldots,c_{n+m}\}\uplus\Delta S)\\ &=\textsc{rule}(k,r,g,b,\{c_1,\ldots,c_{n+m}\}\uplus\Delta S)\\ &=\{c_1,\ldots,c_n\}\uplus b\,(c_1,\ldots,c_{n+m})\uplus\Delta S\end{aligned}$$

Since by assumption the condition in line 2 will be *true*. From line 3 we then know that

$$\begin{aligned}&\textsc{rule}(k,r,g,b,\{c_1,\ldots,c_{n+m}\}\biguplus\Delta S)\\ &\equiv\Big(\{c_1,\ldots,c_{n+m}\}\biguplus\Delta S\cup\{c_{n+1},\ldots,c_{n+m}\}\Big)\biguplus b(c_1,\ldots,c_{n+m})\end{aligned}$$

which can be simplified to $\{c_1,\ldots,c_n\}\uplus b\,(c_1,\ldots,c_{n+m})\uplus\Delta S$.

*Induction Step (*STEP $_i$*)*

Given a proof

$$\frac{S\overset{p_i}{\mapsto}S'\quad S'\overset{p_1\odot\ldots\odot p_l}{\mapsto}S''}{S\overset{p_1\odot\ldots\odot p_l}{\mapsto}S''}\textsc{step}_i$$

with $1\le i\le l$. We assume

$$S\xmapsto{(\!|\mathbf{exec}_a|\!)(p_i)}{}^{*}\,S'\Longleftarrow S\overset{p_i}{\mapsto}S' \tag{6}$$

$$S'\xmapsto{(\!|\mathbf{exec}_a|\!)(p_1\odot\ldots\odot p_l)}{}^{*}\,S''\Longleftarrow S'\overset{p_1\odot\ldots\odot p_l}{\mapsto}S'' \tag{7}$$

as induction hypotheses. We want to show that

$$S\xmapsto{(\!|\mathbf{exec}_a|\!)(p_1\odot\ldots\odot p_l)}{}^{*}\,S''$$

From the premises of the given proof we can infer $S\xmapsto{(\!|\mathbf{exec}_a|\!)(p_i)}{}^{*}\,S'$ and $S'\xmapsto{(\!|\mathbf{exec}_a|\!)(p_1\odot\ldots\odot p_l)}{}^{*}\,S''$ by induction hypotheses (6) and (7), respectively. By the definition of COMPOSE we know that, since $(\!|\mathbf{exec}_a|\!)(p_i)(S)\equiv S'$, S' is also a possible outcome of $(\!|\mathbf{exec}_a|\!)(p_1\odot\ldots\odot p_l)(S)$. From $S\xmapsto{(\!|\mathbf{exec}_a|\!)(p_i)}{}^{*}\,S'$, we can hence assume $S\xmapsto{(\!|\mathbf{exec}_a|\!)(p_1\odot\ldots\odot p_l)}{}^{*}\,S'$. Therefrom $S\xmapsto{(\!|\mathbf{exec}_a|\!)(p_1\odot\ldots\odot p_l)}{}^{*}\,S''$ follows directly. □

With Theorem 3 and Theorem 4 we established that Algorithm 1 correctly implements the *very abstract operational semantics* $\omega_a^\star$. The theorems also imply that $\mathbf{Exec}_a$ is an equivalent representation of $\omega_a^\star$.

6 Implementation case studies

We now present two implementations that are based on the execution algorithm described in Section 5 as case studies. The source code of the implementations is also available on GitHub.[15]

[15] https://gist.github.com/SRechenberger/d5e1eb875ae72ce5cafe6ea1c5b3ee38

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

The purpose of this section is twofold: first, we show how the abstract execution algorithm can be used as a guide to implement FreeCHR. Second, we show how the chosen host language flavors the implementation of FreeCHR both in the back end, as well as in the front end.

The presented implementations have two limitations to be noted: first, they solve any source of non-determinism in the original algorithm. Whenever the original algorithm *chooses* an element from a (multi) set, the implementations will sequentially search for them. Especially is the state, in both implementations, a list instead of a multiset. Also, the rules of a program will be traversed sequentially to find an applicable one. It is, though, easy to see that they will only perform transitions Algorithm 1 would perform as well. They are, in other words, concretizations of $\mathbf{Exec}_a$.

Second, as they are based on the *very abstract* operational semantics, they do not yet account for trivial non-termination due to repeated application of propagation rules (this is, rules with an empty removed head) (Frühwirth (2009)). Hence, a program with such rules will not terminate if any of them is at some point applicable. This problem is solved with the *abstract* operational semantics by recording a *propagation history*. It saves applied constraints (by a unique identifier) as well as the rule applied to them. If a propagation rule is to be applied, it is first checked if the combination was already executed. If this is the case, the rule will not fire again.

The consequence for the presented implementations is that they only work on a subset of CHR/FreeCHR without propagation rules.

6.1 Host languages

To sufficiently demonstrate how FreeCHR can be implemented in different programing languages and how the respective host language flavors the implementation, we chose two programing languages of very different paradigms: *Haskell* and *Python*.

Haskell is a statically typed functional programing language with a powerful type system. Python is a dynamically typed imperative language with plenty of practical applications.

One might consider both languages to be on the opposite of a static-dynamic/functional-imperative spectrum as well. They will hence serve well for this case study, as most languages are featurewise somewhere in between. Languages like Java or C# successively adopt functional features but stay imperative in their core. They also have a static type system like Haskell, but not as powerful and strict. Languages like Kotlin move further towards the functional end of the spectrum but still inherit many characteristics from Java. On the other hand, the whole LISP family of programing languages is functional but dynamically typed.

6.2 Execution algorithm

We will now give a detailed overview of the implementations, connect them to the abstract algorithm and compare them to each other.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

```
1  newtype SolverStep c = SolverStep { runSolverStep :: [c] -> [c] }
```

Fig. 3. Wrapped solver function type.

6.2.1 Carrier

We first want to discuss the implementation of the carrier of the implemented CHR_C-algebra.

In Haskell, the carrier is explicitly defined as the type `SolverStep c` shown in Figure 3. The type is essentially equal to functions of type `[c] -> [c]` which is the Haskell representation of the set $(\texttt{mset}C)^{\texttt{mset}C}$. As mentioned above, we will use lists instead of multisets. This eliminates one source of non-determinism.

Wrapping the functions by defining a new type allows the implementation of type class instances (which is used for COMPOSE) and to discern programs that are "in construction" from such that were already passed to `run`.

The Python implementation directly models functions mapping lists to lists. Since Python is (by default) dynamically typed, there are no constraints on the type of the elements of the lists. This, on the one hand, gives more freedom to the user of the FreeCHR instance since it is easier to define rules on different types of constraints. This is also possible when using Haskell, but needs additional work like defining and matching on a sum type. On the other hand, in Python, one should always explicitly check if the constraints have the correct type in the head of the rule.

For example, the program in Figure 4b allows values of any type in its state and will apply the rules only if integers are encountered. Not checking the type of constraints beforehand is a serious source of runtime errors. If a rule of the `gcd` program is applied to something other than two integers, the comparison operations (l. 3 and 6) in the guard or the subtraction (l. 7) in the body will raise an exception. Such errors are impossible in the Haskell version in Figure 4a since the input type needs to be some sort of integer (`Integral`; see l. 1) which guarantees support for comparison (`Ord`) and subtraction (`Num`).[16]

6.2.2 RULE

The implementations of RULE can be seen in Figure 5. First, either function lazily computes the sequence of possible matchings (l. 5 in Figure 5a and l. 3–5 in Figure 5b). This is done using the `match` functions which are shown in Figure 6. Both generate a lazy sequence of all permutations of the state that have the same length as the head of the rule (l. 3–4 and l. 3, respectively) and select those for which the conjunction of the elementwise application (done by `zipWith` and `zip`) of the elements of the head evaluates to true, this is, those for which $k_1(c_1) \wedge \ldots \wedge k_n(c_n) \wedge r_1(c_{n+1}) \wedge \ldots \wedge r_m(c_{n+m}) \equiv_2 \mathit{true}$, for a head $[k_1, \ldots, k_n, r_1, \ldots, r_m]$ and a permutation $[c_1, \ldots, c_{n+m}]$ of the state. Then, both implementations of RULE take the first of those sequences for which the guard g holds, this is, $g(c_1, \ldots, c_{n+m}) \equiv_2 \mathit{true}$. In total, this corresponds to l. 2 in Algorithm 1. If such a matching is found, the new state is computed and returned as in l. 3 (l. 7 and 8–11). In the Python implementation, an explicit copy of the state is created to not modify the original state. If no matching is found, the state is returned unchanged (l. 8 and 6–7).

[16] Both constraints are inherited via Haskell's type class `Real`.1.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

```haskell
gcd :: Integral a => SolverStep a
gcd = rule [] [(<= 0)]
          (const True)
          (const []) <>
      rule [(> 0)] [(> 0)]
          (\[n, m] -> n <= m)
          (\[n, m] -> [m - n])
```

(a) Haskell

```python
gcd = compose(
    rule([], [lambda n: isinstance(n, int)],
        lambda n: n <= 0,
        lambda _: []),
    rule([lambda n: isinstance(n, int)], [lambda m: isinstance(m, int)],
        lambda n, m: 0 < n <= m,
        lambda n, m: [m-n])
)
```

(b) Python

Fig. 4. Greatest common divisor.

Note that both variants explicitly create and return a function after getting the rule elements passed. The implementations of COMPOSE (Figure 7) work in the same way.

Also, the Python implementation makes heavy use of the *spread operator* (∗ before function arguments) (l. 4 and 11). In consequence, we can write the guard and body of a rule as functions that accept as many arguments as the head has patterns. In Haskell, guard and body accept a list as their single argument (compare lines 6 and 7 in either implementation in Figure 4).

An issue specific to Python can be seen in Figure 8b. Python does only allow expressions in `lambda` functions. If any function of a rule needs to use statements, it must be defined separately.

6.2.3 COMPOSE

Figure 7 shows the implementations of COMPOSE. Figure 7a implements the algorithm as the binary operator `(<>)` of the `Semigroup` type class. First, the left subprogram will be applied to the state (l. 5). If this has no effect, the second subprogram is applied (l. 7). Otherwise, the result of the first application which must have altered the state, is returned (l. 8). One can easily see that the implementation is associative, as required by `Semigroup`. In nested expressions like `p1 <> p2 <> ... <> pn`, the algorithm will always perform a depth-first search, beginning from the leftmost subprogram, effectively traversing the program from left to right.

The same is true for the Python variant in Figure 7b. Here, we loop through the subprograms, passed to the function `compose` (l. 4–7) until a program alters the state (l. 6 & 7). Finally, `result` is returned which is either the input state (see l. 3) if the program made it through the loop without finding an applicable subprogram, or the first result of an effective application (see l. 5).

Contrary to the Haskell variant, COMPOSE is implemented as a variadic function by use of the spread operator in Python. The consequence for the user can be seen in Figure 4, where we use `(<>)` to compose the rules (l. 4) in Haskell and `compose` in Python.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

```haskell
rule :: Eq c => [c -> Bool] -> [c -> Bool] -> ([c] -> Bool) -> ([c] -> [c])
     -> SolverStep c
rule kept removed guard body = SolverStep
    { runSolverStep = \state ->
        let matching = find guard (match (kept ++ removed) state)
        in case matching of
            Just ms -> (state \\ drop (length kept) ms) ++ body ms
            _       -> state
    }
```

(a) Haskell

```python
def rule(k, r, g, b):
    def solver(state):
        matching = next(
            (matching for matching in match(k+r, state) if g(*matching)),
            None)
        if not matching:
            return state
        state_copy = state.copy()
        for c in matching[len(k):]:
            state_copy.remove(c)
        return b(*matching) + state_copy
    return solver
```

(b) Python

Fig. 5. Implementations of RULE.

```haskell
match :: [a -> Bool] -> [a] -> [[a]]
match ps as = [ as''
    | as' <- subsequences as, length as' == length ps
    , as'' <- permutations as'
    , and (zipWith ($) ps as'')
    ]
```

(a) Haskell

```python
def match(pattern, cs):
    return (perm
        for perm in permutations(cs, len(pattern))
        if all(p(c) for p, c in zip(pattern, perm))
    )
```

(b) Python

Fig. 6. Lazy matching.

6.2.4 RUN

Figure 9 shows the implementations of RUN. The implementation in Figure 9a first applies the passed program to the query (also called the initial state) (l. 6). If this has no effect, the state is returned (l. 3). Otherwise, `run` is recursively called with the new state (l. 4), repeatedly applying the program until exhaustion.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

```haskell
instance Eq c => Semigroup (SolverStep c) where
    (<>) :: Eq c => SolverStep c -> SolverStep c -> SolverStep c
    solver <> solver' = SolverStep
        { runSolverStep = \state ->
            let state' = runSolverStep solver state
            in if state == state'
                then runSolverStep solver' state
                else state'
        }
```

(a) Haskell

```python
def compose(*solvers):
    def solver(constraints):
        result = constraints
        for s in solvers:
            result = s(constraints)
            if result != constraints:
                break
        return result
    return solver
```

(b) Python

Fig. 7. Implementations of COMPOSE.

```haskell
alldifferent :: Eq a => SolverStep a
alldifferent = rule [const True, const True] []
    (\[x, y] -> x == y) (const $ error "False")
```

(a) Haskell

```python
def false(*args):
    raise Exception(False)

all_different = rule(
    [lambda a: True, lambda b: True], [],
    lambda a, b: a == b,
    false
)
```

(b) Python

Fig. 8. Implementation of `all_diff` constraint.

The algorithm in Figure 9b accomplishes the same by the loop in lines 5 to 9, which is interrupted if the state remains unchanged after applying the program (l. 6 & 7).

Again, the use of the spread operator in Python causes a minor difference in usage. While in Haskell, we need to pass a list to `run`, in Python `run` it is a variadic function. Hence, to calculate the greatest common divisor of 6, 9 and 12, we call `run(gcd)(6, 9, 12)` in Haskell and `run gcd [6, 9, 12]` in Python and get in both cases `[3]` as the result.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

```haskell
run :: Eq c => SolverStep c -> [c] -> [c]
run solver query
    | result == query = result
    | otherwise       = run solver result
    where
        result = runSolverStep solver query
```

(a) Haskell

```python
def run(solver):
    def solve(*query):
        state = list(query)
        result = solver(state)
        while True:
            if result == state:
                break
            state = result
            result = solver(result)
        return result
    return solve
```

(b) Python

Fig. 9. Implementations of RUN.

6.3 Side effects

A major difference between the two presented implementations is the handling of side effects. Python allows for arbitrary side effects like I/O, assigning variables or raising exceptions, as can be seen in Figure 8b. Although this is also possible in Haskell (see Figure 8a), it is highly discouraged by the language. The programs raise an error if they encounter the same value in the query twice. Although the implementations allow it, this behavior is not defined by the operational semantics and domain of future work.

7 Limitations

We want to briefly discuss the current limitations of FreeCHR as it is presented in this paper.

The very abstract operational semantics ω_a for CHR is meant as a tool for formal reasoning and an abstract baseline. It does not describe the behavior of an actual implementation, since it is heavily non-deterministic. As discussed above, it does also not account for trivial non-termination due to repeated application of a propagation rule on the same constraints. Since the very abstract operational semantics for FreeCHR $\omega_a^\star$ is a direct translation of ω_a, it inherits these issues.

The presented implementations are hence only defined on programs without propagation rules as they would inevitably get stuck on the repeated application of such a rule. This issue is solved by the *refined* operational semantics ω_r as presented by Duck *et al.* (2004). Translating ω_r to FreeCHR is currently ongoing work. For now, the implementations hence only serve as a proof of concept of the applicability of our framework and are

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

not meant for actual use.[17] We therefore did also not concern ourselves with applying any optimization techniques and do hence not yet expect any reasonable performance.

The approach presented herein is strictly to be understood as the formal foundation upon which we will build our future work, as it connects our framework to the original definition of CHR on the most fundamental level.

8 Related work

8.1 Constraint handling rules

8.1.1 Embeddings

The first approach to embed CHR into a host language was via source-to-source transformation. Holzbaur and Frühwirth (2000) and Schrijvers and Demoen (2004), for example, translate CHR via Prolog's macro system. Similarily, Abdennadher *et al.* (2002) and Van Weert *et al.* (2005) use a precompiler to translate CHR programs into Java, Wuille *et al.* (2007) into C, Nogatz *et al.* (2018) into Javascript, and Barichard (2024) into C++.

Van Weert *et al.* (2008) introduces compilation schemes for imperative languages, upon which, for example, Nogatz *et al.* (2018) builds. The work of van Weert (2010) also provides compilation schemes for imperative languages, but optimizes the matching process by performing it lazily. The algorithms do not first compute a full matching and check it against the patterns and the guard but collect it sequentially.

The inherent disadvantage of an approach using source-to-source compilation is the need for a precompiler in the build chain if the host language does not have a sufficiently expressive macro system like Prolog or LISP. It comes with the cost of more sources of errors and an additional dependency that is often rather tedious to fulfill. Hanus (2015) approaches this problem by extending the Curry compiler to be able to compile CHR code. The obvious problem with this approach is that it is in most cases not viable to extend the compiler of the host language.

A relatively new approach by Ivanović (2013) was to implement CHR as an internal language in Java. This has a major advantage: CHR can now simply be imported as a library, similarly as if implemented by a macro system like by Schrijvers and Demoen (2004). Wibiral (2022) further builds upon this idea and introduces the idea of explicitly composing CHR programs of singular rules by an abstract and modular execution strategy and describing rules through anonymous functions. This was our main inspiration and FreeCHR aims to improve and generalize this idea for arbitrary languages.

8.1.2 Operational semantics

Duck *et al.* (2004) first formalized the behavior of existing implementations which were mostly derived from but not exactly true to existing formal definitions of operational semantics. The techniques used in the implementations were also generalized and improved upon by Van Weert *et al.* (2008) and van Weert (2010). Duck (2005) standardized call-based semantics for CHR, especially in logical programing languages.

[17] Aside maybe from very small applications.

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

8.2 Algebraic language embeddings

The idea of algebraic embeddings of a domain specific language (DSL) into a host language was first introduced by Hudak (1998). The style in which the languages are embedded was later called *tagless*, as it does not use an algebraic data type, to construct an abstract syntax tree (see, e.g., Carette *et al.* (2007)). The *tags* are the constructors of the data type which defines the syntax of the language. Instead, the embedding directly defines the functions which implement the semantics of the language. This is, defining the free F-algebra versus defining the concrete F-algebras, for a functor F which defines the syntax of the language. The advantage of embedding a DSL in this way is that it does not rely on external build tools, but can instead be easily implemented and used as a library. It also enables the use of any features the host language offers, without any additional work. With an external code-to-code compiler, it would be necessary to re-implement at least the syntax of any desired host language features.

Hofer *et al.* (2008) and Hofer and Ostermann (2010) then extended this idea by using type families in order to provide more flexibility concerning the semantics of the language.

8.3 Logic and constraint based languages and formalisms

CHR was initially designed as a tool to implement constraint solvers for user-defined constraints. Hence, its domain intersects with those of *Answer Set Programming* (ASP) and *Constraint Logic Programming* (CLP). However, on the one hand, CHR can rather be understood as a tool in combination with ASP or especially CLP, as it provides an efficient language to implement solvers, than as an alternative approach. On the other hand, CHR has already exceeded its original purpose and developed more towards a general purpose language.

Another relevant logic based language is *miniKanren* (see Byrd (2009)). *miniKanren* is a family of EDSLs for relational and logic programing. There exists a myriad of implementations for plenty of different host languages as well as formal descriptions of operational semantics (see e.g., Rozplokhas *et al.* (2020)). The *miniKanren* language family, though, seems to suffer from similar issues as traditional *Constraint Handling Rules* does. As there is no unified embedding scheme, there is an inherent disconnect between any implementation and the formally defined semantics. Applying an approach similar to ours might be beneficial to the *miniKanren* project as well, especially as it generally is implemented in a way similar to the methods described by Hudak (1998) or Carette *et al.* (2007).

9 Future work

With this foundational introduction of FreeCHR in place, our future work will focus on three main aspects.

First, since the execution algorithm presented here is only of limited practical use, we plan to introduce an algorithm that implements the *refined* operational semantics ω_r of

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

CHR. The ω_r semantics of CHR is the basis for most existing implementations. Since we already have the *very abstract* operational semantics as a baseline, we can use it as a basic anchor to verify any execution algorithm. The algorithm we intend to formalize is already implemented in Haskell[18] and Python.[19]

Second, we currently work on translating the *refined* operational semantics and verifying them against $\omega_a^\star$, as well as the aforementioned execution algorithm. This will provide a base for theory concerning practically relevant aspects of FreeCHR.

We plan to verify both the execution algorithm implementing $\omega_r^\star$ and $\omega_r^\star$ itself by showing that they are valid concretization of $\omega_a^\star$.

Finally, we plan to tackle the issue of side effects that we encountered in the case study in Section 6. This can be done by generalizing the functor CHR_C to use monads to encapsulate arbitrary side effects (see Jones and Wadler (1993)). This was already done in implementations in Haskell but is not yet formalized. This is an important aspect, as it further formalizes practical aspects of everyday programing.

10 Conclusion

In this paper, we introduced the framework FreeCHR that 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 with respect to the original *very abstract* operational semantics ω_a of CHR. We also provided a first abstract instance $\mathbf{Exec}_a$ of FreeCHR, proved its correctness with respect to to $\omega_a^\star$ and discussed deterministic implementations of the algorithms in Haskell and Python.

Hereby, we established FreeCHR as a valid representation of the original definition of CHR and a usable framework for the implementation of the embedded language in a wide range of host languages.

Moreover, are the presented execution algorithm and the implementations of it, although very simple, to our knowledge the first implementations of CHR for which there is an actual prove of correctness with respect to to a formal definition of their operational semantics.

Acknowledgements

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

Competing interests

The authors declare none.

[18] https://github.com/SRechenberger/free-chr-hs
[19] https://github.com/SRechenberger/freechr-python

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

References

Abdennadher, S., Krämer, E., Saft, M. and Schmauss, M. 2002. JACK: a java constraint kit. In *Electronic Notes in Theoretical Computer Science,* vol. 64, Elsevier, pp. 1–17. https://doi.org/10.1016/S1571-0661(04)80344-X

Abdennadher, S. and Marte, M. 2000. University course timetabling using constraint handling rules. In *Applied Artificial Intelligence*, vol. 14, issue 4, Taylor & Francis, pp. 311–325. https://doi.org/10.1080/088395100117016

Barichard, V. 2024. CHR++: An efficient CHR system in C++ with don't know non-determinism. In *Expert Systems with Applications Part B*, vol. 238, Elsevier. https://doi.org/10.1016/j.eswa.2023.121810

Betz, H. 2007. Relating coloured petri nets to constraint handling rules. In *CHR 2007*, Ulm University. https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.170/home/betz/betz_petri_nets_chr07.pdf

Byrd, W. E. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph.D. thesis, Indiana University, USA. https://scholarworks.iu.edu/dspace/items/450e1b65-70da-4a38-8e73-c182818de110

Carette, J., Kiselyov, O. and Shan, C.-c. 2007. Finally tagless, partially evaluated. In *APLAS 2007*, Z. Shao (Ed.), Lecture Notes in Computer Science, Berlin, Heidelberg, Springer, pp. 222–238. https://doi.org/10.1007/978-3-540-76637-7_15

Chin, W., Sulzmann, M. and Wang, M. 2003. A type-safe embedding of constraint handling rules into haskell. Technical Report. https://www.academia.edu/download/40187865/A_type-safe_embedding_of_constraint_hand20151119-21055-1osq67f.pdf

Christiansen, H. and Kirkeby, M. H. 2015. Confluence modulo equivalence in constraint handling rules. In *Logic-Based Program Synthesis and Transformation*, M. Proietti and H. Seki (Eds.), Lecture Notes in Computer Science, Cham, Springer International Publishing, pp. 41–58. https://doi.org/10.1007/978-3-319-17822-6_3

De Koninck, L., Schrijvers, T., Demoen, B., Fink, M., Tompits, H. and Woltran, S. 2006. INCLP(R) - interval-based nonlinear constraint logic programming over the reals. In *WLP 2006*, vol. 1843-06-02, Technische Universität Wien, Austria. https://lirias.kuleuven.be/1653773

Duck, G. J. 2005. *Compilation of Constraint Handling Rules*. Ph.D. thesis, University of Melbourne, Victoria, Australia. https://www.comp.nus.edu.sg/~gregory/papers/thesis.pdf

Duck, G. J., Stuckey, P. J., De La Banda, M. G. and Holzbaur, C. 2004. The refined operational semantics of constraint handling rules. In *ICLP 2004*, B. Demoen and V. Lifschitz (Eds.), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 90–104. https://doi.org/10.1007/978-3-540-27775-0_7

Fowler, M. and Parsons, R. 2011. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ, ISBN 978-0-321-71294-3.

Frühwirth, T. 2006. Complete propagation rules for lexicographic order constraints over arbitrary domains. In *CSCLP 2005*, B. Hnich, M. Carlsson, F. Fages and F. Rossi (Eds.), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 14–28. https://doi.org/10.1007/11754602_2

Frühwirth, T. 2009. *Constraint Handling Rules*. Cambridge University Press, Cambridge, U.K. New York, ISBN 978-0-521-87776-3

Frühwirth, T. 2015. Constraint handling rules - what else?. In *RuleML 2015*, N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke and R. D. Roman (Eds.), Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 13–34. https://doi.org/10.1007/978-3-319-21542-6_2

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

Gall, D. and Frühwirth, T. 2018. An operational semantics for the cognitive architecture ACT-R and its translation to constraint handling rules. In *ACM Transactions on Computational Logic*, vol. 19, issue 3, Association for Computing Machinery (ACM), pp. 1–42. https://doi.org/10.1145/3218818

Hanus, M. 2015. CHR(Curry): interpretation and compilation of constraint handling rules in curry. In *PADL 2015*, E. Pontelli and T. C. Son (Eds.), Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 74–89. https://doi.org/10.1007/978-3-319-19686-2_6

Hofer, C. and Ostermann, K. 2010. Modular domain-specific language components in scala. In *GPCE '10*. ACM Press. https://doi.org/10.1145/1942788.1868307

Hofer, C., Ostermann, K., Rendel, T. and Moors, A. (2008) Polymorphic embedding of dsls. In *GPCE '08*. ACM Press. https://doi.org/10.1145/1449913.1449935

Holzbaur, C. and Frühwirth, T. 2000. A prolog constraint handling rules compiler and runtime system. In *Applied Artificial Intelligence*, vol. 14, issue 4, Taylor & Francis, pp. 369–388. https://doi.org/10.1080/088395100117043

Hudak, P. 1998. Modular domain specific languages and tools. In *ICSR 1998*, IEEE Computer Society, USA, pp. 134–142. https://doi.org/10.1109/ICSR.1998.685738

Ivanović, D. 2013. Implementing constraint handling rules as a domain-specific language embedded in Java, arXiv preprint arXiv:1308:3939. https://doi.org/10.48550/arXiv.1308.3939

Johann, P. and Ghani, N. 2007. Initial algebra semantics is enough! In *TLCA 2007*, S. R. Della Rocca (Ed.), Lecture Notes in Computer Science, Berlin, Heidelberg, Springer, pp. 207–222. https://doi.org/10.1007/978-3-540-73228-0_16

Jones, S. P. and Wadler, P. 1993. Imperative functional programming. In *POPL '93*, ACM Press, New York, NY, USA, pp. 71–84. http://doi.acm.org/10.1145/158511.158524

Lam, E. S. L. and Sulzmann, M. 2007. A concurrent constraint handling rules implementation in Haskell with software transactional memory. In *DAMP '07*. ACM Press, Nice, France, 19–24. https://doi.org/10.1145/1248648.1248653

Lam, E. and Sulzmann, M. 2006. Towards agent programming in CHR. In *CHR 2006*, Department of Computer Science, KU Leuven, pp. 17–31. https://www.cs.kuleuven.be/publicaties/rapporten/cw/CW452.pdf

Milewski, B. 2019. *Category Theory for Programmers*. Lightning Source UK, Milton Keynes. ISBN 978-0-464-24387-8

Nogatz, F., Frühwirth, T. and Seipel, D. 2018. CHR.js: a CHR implementation in JavaScript. In *RuleML 2018*, C. Benzmüller, F. Ricca, X. Parent and D. Roman (Eds.), Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 131–146. https://doi.org/10.1007/978-3-319-99906-7_9

Rozplokhas, D., Vyatkin, A. and Boulytchev, D. 2020. Certified semantics for relational programming. In *Programming Languages and Systems*, B. C. d. S. Oliveira (Ed.), Springer International Publishing, Cham, pp. 167–185. https://doi.org/10.1007/978-3-030-64437-6_9

Schrijvers, T. and Demoen, B. 2004. The K.U. Leuven CHR system: implementation and application. In *CHR 2004*, Department of Computer Science, KU Leuven, pp. 1–5. https://lirias.kuleuven.be/1653784&lang=en

Sneyers, J., Weert, P. V., Schrijvers, T. and Koninck, L. D. 2010. As time goes by: Constraint handling rules: A survey of CHR research from 1998 to 2007. In *Theory and Practice of Logic Programming*, vol. 10, issue 1, Cambridge University Press, pp. 1–47. https://doi.org/10.1017/S1471068409990123

Thielscher, M. 2002. Reasoning about actions with CHRs and finite domain constraints. In *LP 2002*, P. J. Stuckey (Ed.), *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, pp. 70–84. https://doi.org/10.1007/3-540-45619-8

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press

Thielscher, M. 2005. FLUX: a logic programming method for reasoning agents. In Theory and Practice of Logic Programming, vol. 5, issue 4–5, Cambridge University Press, pp. 533–565. https://doi.org/10.1017/S1471068405002358

Van weert, P. 2010. Efficient lazy evaluation of rule-based programs. In *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, issue 11, pp. 1521–1534. https://doi.org/10.1109/TKDE.2009.208

Van weert, P., Schrijvers, T., Demoen, B., Schrijvers, T. and FRÜHWIRTH, T. 2005. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for java. In *CHR 2005*. Department of Computer Science, KU Leuven. https://lirias.kuleuven.be/1654703

Van weert, P., Wuille, P., Schrijvers, T. and Demoen, B. 2008. CHR for imperative host languages. In *Constraint Handling Rules: Current Research Topics*, T. Schrijvers and T. Frühwirth (Eds.), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 161–212. https://doi.org/10.1007/978-3-540-92243-8

Wibiral, T. 2022. *JavaCHR – A Modern CHR-Embedding in Java*. Bachelor thesis, Ulm University. http://dx.doi.org/10.18725/OPARU-43430

Wuille, P., Schrijvers, T. and Demoen, B. 2007. CCHR: The fastest CHR implementation. In *CHR 2007*, Department of Computer Science, KU Leuven, pp. 123–137. https://lirias.kuleuven.be/1655099

https://doi.org/10.1017/S1471068425000043 Published online by Cambridge University Press