Constrained Zig

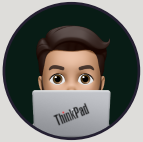
Exploring properties of constraint programs

Lukas Pietzschmann
lukas.pietzschmann@uni-ulm.de

Zigtoberfest
Hochschule München

October 19th, 2024

1 About Me



Hi, I'm Lukas!

Studies

I'm studying computer science for my master's degree at Ulm University. Currently, I'm working on my thesis.

Interests

I enjoy various things, but compilers, typesetting, and functional programming spark the most joy inside me.

2 Motivation

A first small example

> *Constraint Programming represents one of the closest approaches computer science has yet made to the holy grail of programming: the user states the problem, the computer solves it.* (Eugene C. Freuder [Fre97])

$$\underbrace{\texttt{min(N)}}_{\text{First min constraint}} \ \underbrace{\backslash}_{\text{Operation}} \ \underbrace{\texttt{min(M)}}_{\text{Second min constraint}} \iff \underbrace{\texttt{N} \leqslant \texttt{M}}_{\text{Guard}} \mid \underbrace{\texttt{true}}_{\text{Result}}$$

Introduction to CHR

CHR knows about three kinds of rules:

Propagation

`C ⇒ G | CN`

`CN` is inferred if `G` holds

Simplification

`C ⟺ G | CN`

`C0` is simplified to `CN` if `G` holds

Simpagation

$C_1 \setminus C_2 \Longleftrightarrow G \mid CN$

C_2 is simplified to `CN` if `G` holds

C	Head	CHR Constraints
G	Guard	Conjunction of built-ins
CN	Body	CHR Constraints and conjunction of built-ins

Zig code

CHR knows about three kinds of rules:

Propaga…

C ⇒

CN is inferr…

| CN

…o CN if

Generalized Simpagation Rule

Behind the curtain, the embedding represents every rule as a 4-tuple:

$$\langle KH, RH, G, B \rangle$$

KH: Kept Head RH: Removed Head G: Guard B: Body

A propagation rule would then look like this: $\langle KH, \emptyset, G, B \rangle$

C	Head	CHR Constraints
G	Guard	Conjunction of built-ins
CN	Body	CHR Constraints and conjunction of built-ins

Zig code

> Query ...

`min(N) \ min(M)` $\iff$ `N ⩽ M | true`

- Add constraints to solve for

Constraint Store

Introduction to CHR

```
> min(3)
```

```
min(N) \ min(M) ⟺ N ⩽ M | true
```

- Add constraints to solve for

Constraint Store

Introduction to CHR

```
> min(3), min(1)
```

min(N) \ min(M) ⟺ N ⩽ M | **true**

- Add constraints to solve for

Constraint Store

Introduction to CHR


```
> min(3), min(1), min(5)
```

```
min(N) \ min(M) ⟺ N ⩽ M | true
```

- Add constraints to solve for

Constraint Store

```
> min(3), min(1), min(5)
```

$\texttt{min(N)} \setminus \texttt{min(M)} \Longleftrightarrow \texttt{N} \leqslant \texttt{M} \mid \textbf{true}$

- Check if we can find a matching for the currently active constraint

Constraint Store

Introduction to CHR

```
> min(3), min(1), min(5)
```

```
min(N) \ min(M) ⟺ N ≤ M | true
min(3) \ min(M) ⟺ 3 ≤ M | true
```

- Check if we can find a matching for the currently active constraint

Constraint Store

```
> min(3), min(1), min(5)
```

min(N) \ ~~min(M)~~ min(3) ⟺ N ⩽ ~~M~~ 3 | **true**

- Check if we can find a matching for the currently active constraint

Constraint Store

```
> min(3), min(1), min(5)
```

min(N) \ min(M) ⟺ N ⩽ M | **true**

- Fire the rule that matched

min(3)

Constraint Store

Introduction to CHR

```
> min(3), min(1), min(5)
```

```
min(N) \ min(M) ⟺ N ≤ M | true
```

- Make next query constraint active

Constraint Store

```
min(3)
```

```
> min(3), min(1), min(5)
```

$$\texttt{min(N)}\ \backslash\ \texttt{min(M)} \iff \texttt{N} \leqslant \texttt{M}\ |\ \texttt{true}$$

$$\texttt{min(1)}\ \backslash\ \texttt{min(M)} \iff \texttt{1} \leqslant \texttt{M}\ |\ \texttt{true}$$

- Check if we can find a matching for the currently active constraint

min(3)

Constraint Store

```
> min(3), min(1), min(5)
```

```
min(N) \ min(M) ⟺ N ≤ M | true
min(1) \ min(3) ⟺ 1 ≤ 3 | true
```

- Check if we can find a matching for the currently active constraint

```
min(3)
```

Constraint Store

Introduction to CHR

```
> min(3), min(1), min(5)
```

```
min(1) \ min(3) <=> 1 ≤ 3 | true
```

- Fire the rule that matched

Constraint Store

3.2 Introduction to CHR

Constraint Store

```
> min(3), min(1), min(5)
```

min(N) \ min(M) ⟺ N ⩽ M | **true**

- Fire the rule that matched

```
min(1)
```

Constraint Store

```
> min(3), min(1), min(5)
```

`min(N) \ min(M) ⟺ N ≤ M | true`

- Make next query constraint active

```
min(1)
```

Constraint Store

```
> min(3), min(1), min(5)
```

min(N) \ ~~min(M)~~ **min(5)** $\Longleftrightarrow$ N $\leqslant$ ~~M~~ **5** | **true**

- Check if we can find a matching for the currently active constraint

```
min(1)
```

Constraint Store

```
> min(3), min(1), min(5)
```

```
min(1) \ min(5) <=> 1 =< 5 | true
```

- Check if we can find a matching for the currently active constraint

Constraint Store:

```
min(1)
```

```
> min(3), min(1), min(5)
```

```
min(N) \ min(M) ⟺ N ⩽ M | true
```

- Make next query constraint active

```
min(1)
```

Constraint Store

Introduction to CHR


```
> min(3), min(1), min(5)
```

`min(N) \ min(M) ⟺ N ⩽ M | true`

- The remaining constraints in the store are the solution

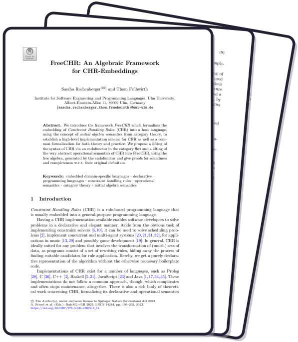
```
min(1)
```

Constraint Store

4.1 Inner workings of the Embedding

- Based on the concepts established in “FreeCHR: An Algebraic Framework for CHR-Embeddings” [RF23]

- Every rule is represented as a 4-tuple (see slide 3)
- We then compose more complex programs from subprograms — or ultimately from single rules
- Lastly, we apply the composition to a state until a fixpoint is reached



FreeCHR: An Algebraic Framework for CHR-Embeddings

Sascha Rechenberger and Thom Frühwirth

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

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

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

1 Introduction

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

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

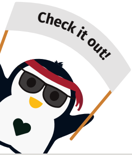
Implementations of CHR exist for a number of languages, such as Prolog [28], C [36], C++ [3], Haskell [5,21], JavaScript [23] and Java [1,17,34,35]. These implementations do not follow a common approach, though, which complicates and often stops maintenance, altogether. There is also a rich body of theoretical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

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
A. Fensel et al. (Eds.): RuleML+RR 2023, LNCS 14244, pp. 190–205, 2023.
https://doi.org/10.1007/978-3-031-45072-3_14

ZigCHR online

- The embedding's source code is available on GitHub
- Clone it and try it out yourself!
- Also, the repo contains some of the examples we're discussing today

https://lukas.pietzschmann.org/talks/zigtoberfest



5.1 Anytime Algorithms

- We can interrupt the execution at anytime
- The intermediate state will be an approximation of the final result
- After that, the program will continue like nothing happened
- This is especially useful if we need to guarantee a certain response time

Easy Memorization

- Quizz: Change three symbols to make this have linear complexity

```
fib(0, M) ⟺ M = 1
fib(1, M) ⟺ M = 1
fib(N, M) ⟺ N ⩾ 2 |
    fib(N-1,M1), fib(N-2,M2), M is M1 + M2

fib(0, M) ⟹ M = 1
fib(1, M) ⟹ M = 1
fib(N, M) ⟹ N ⩾ 2 |
    fib(N-1,M1), fib(N-2,M2), M is M1 + M2
```

- We have to change the simplification rules to propagations
- This way, we do not remove calculated values from the store
- But we remember them for future use

Rule order matters

```
throw(Coin) ⟺ head
throw(Coin) ⟺ tail
```

- Depending on which rule we check first, we get a different result
- That's great for coin tossing
- Not so much for determinism

Constraint order matters

```
set(K,V), c(K,V') ⟺ c(K,V)
```

- Imagine the following query: `c(x,0),set(x,1),set(x,2)`
- Depending on what constraint we select first, `x` will be set differently

› In confluent programs, the relation between the initial and final state is a function

Confluence

$$\begin{tikzcd} & \mathtt{a} \arrow[dl] \arrow[dr] & \\ \mathtt{false} \arrow[dr] & & \mathtt{b} \arrow[dl] \\ & \mathtt{false} \neq \mathtt{b} & \end{tikzcd}$$

- Both rules are applicable to a state containing only a
- From this, we can derive the two new states
- Confluence requires then to be joinable

$$\begin{array}{lcl} \mathtt{a} & \Rightarrow & \mathtt{b} \\ \mathtt{a} & \Rightarrow & \mathtt{false} \end{array}$$

- We can make the two states joinable by adding a single rule

$$\begin{array}{lcl} \mathtt{a} & \Rightarrow & \mathtt{b} \\ \mathtt{a} & \Rightarrow & \mathtt{false} \\ \mathtt{b} & \Rightarrow & \mathtt{false} \end{array}$$

Confluence

- ☑ Not every program can be made confluent

```
a ⇒ true
a ⇒ false
```

- ☑ Be aware of the semantics of your program

```
set(K, V), c(K, V') ⟺ c(K, V)
```

▲ does not equal ▼

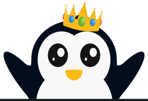
```
set(K, V), c(K, V') ⟺ V = V' | c(K, V)
```

Confluence

- If a program terminates, we can check if it's confluent
- If it is, we get more cool properties for free:

Incrementality

- We can add constraints during the program's execution
- They will eventually participate in the computation
- The result will be the same as if was there from the beginning



Parallelism

- The algorithm can be executed in parallel
- Without any modifications

```
> map(square), reduce(add), v(2), v(3), v(4), v(5)
```

```
map(OP) \ v(X) ⟺
    C =.. [OP, X, R],
    call(C),
    r(R)

reduce(OP) \ r(X), r(Y) ⟺
    C =.. [OP, X, Y, R],
    call(C),
    r(R)
```

```
reduce(add)
map(square)
r(54)
```

Constraint Store

6 WHAT'S LEFT TO SAY

Constraint programming, too 😉

7

References (I)

[Fre97] Eugene C. Freuder. In: *Constraints* 2.1 (1997), pp. 57–61. ISSN: 1383-7133. DOI: 10.1023/a:1009749006768. URL: http://dx.doi.org/10.1023/A:1009749006768.

[Frü09] Thom Frühwirth. *Constraint handling rules*. Cambridge University Press, 2009.

[RF23] Sascha Rechenberger and Thom Frühwirth. "FreeCHR: An Algebraic Framework for CHR-Embeddings". In: *International Joint Conference on Rules and Reasoning*. Springer. 2023, pp. 190–205.

Lukas Pietzschmann

Munich, October 19th, 2024

lukas.pietzschmann@uni-ulm.de