A Web Based Command Line Interface for Specifying Multiway Dataflow Constraint Systems

Bo Victor Isak Aanes

INF319 Project
Department of Informatics
UNIVERSITY OF BERGEN
NORWAY

June 20, 2023

Contents

1 Introduction **2**

2 Background **2**

2.1 Multiway Dataflow Constraint Systems 2
2.2 HotDrink 3
2.3 Structure Manipulation 3
2.4 React 4
2.5 Web Application Programming Interfaces 4
2.6 Haskell 5

3 Frontend Implementation **6**

3.1 Component View and State 6
3.2 Terminal and User Interaction 7

4 API Implementation **8**

4.1 Haskell API 8
4.2 Sending Requests and Receiving Responses 11

5 Usage **12**

5.1 Project Structure 13

6 Conclusion **14**

1 Introduction

Graphical user interfaces (GUIs) are prevalent in most of today's technology that involves user interaction. However, developing and maintaining GUIs can still be difficult. Much of this is because as a GUI involves a larger set of state, the business logic neede in order to keep track of this state quickly becomes complex. This problem grows even bigger when we introduce dependencies between elements of the GUI. Multiway dataflow constraint systems (MDCS) is a programming model designed to let a developer specify dependencies between elements of the GUI using constraints [6]. This is usually done declaratively and MDCS will handle all business logic related to these constraints.

2 Background

2.1 Multiway Dataflow Constraint Systems

A multiway dataflow constraint system consists of a set of variables and a set of constraints. A variable holds a value which can be modified, and a constraint consists of a set of methods which contain expressions that can modify a variable.

Libraries that implement MDCSs usually also handles enforcing of constraints, such that whenever a value gets updated by a user the constraint system automatically satisfies all its constraints. The main part of *solving* a constraint system is achieved through what is called *planning* [8]. When a variable is updated by a user, the constraints involving this variable may no longer be satisfied, that is, that the relations specified in the methods no longer hold. As such, we have to select the appropriate methods to invoke in order to enforce the given constraints, such that the constraint system is satisfied. This is what the planner does.

When a user updates a variable, they will expect that the constraint system does not overwrite that variable instantly. Because of this we do not want the planner to select a method writing to this variable, but rather a method that reads from it. I.e., we want to select the methods from the constraints that result in the least surprising outcome to the user. This is what is known as a *hierarchical* planning algorithm [8].

The hierarchical planner assigns a strength to each variable, which gets updated when a user modifies a variable. The variable with the highest strength is the one that was updated most recently, and the variable with the lowest strength was the one updated least recently. By assigning a strength to the variables, the planner can make selections on which variables to preserve by selecting the appropriate methods.

When the appropriate methods are selected, they are invoked in the topological order of the dataflow graph. This is done to ensure that all variables are updated in the correct order, such that no variable is updated before its dependencies are updated.

2.2 HotDrink

HotDrink is a JavaScript library for specifying multiway dataflow constraint systems [5]. It provides both an API and a DSL for this purpose. Variables can be declared along with constraints between these, which in turn can be bound to specific HTML-elements. The constraints themselves contain *methods* which can hold expressions or JavaScript functions that return values which are written to specified variables. In Listing 1, we can see how we can use HotDrink's API and DSL to define a constraint system to perform currency conversion between EUR and NOK.

Listing 1: Currency conversion in HotDrink.

```
const constraintSystem = new ConstraintSystem();

const component = component`
    var exchangeRate = 10;
    var eur = 0;
    var nok = 0;

    constraint {
        eurToNok(eur, exchangeRate -> nok) = eur *
            ↪ exchangeRate;
        nokToEur(nok, exchangeRate -> eur) = nok /
            ↪ exchangeRate;
    }
`;

constraintSystem.addComponent(component);
constraintSystem.update();
```

We first define a constraint system using the `ConstraintSystem` class. We then define a component using the `component` template literal, which contains specifications for a currency conversion using HotDrink's DSL. Here we define variables and a constraint containing two methods: one for converting EUR to NOK, and one for converting NOK to EUR. After specifying our component, we add this component to the constraint system and call `constraintSystem.update()` to enforce the constraint.

The variables defined in the constraint system in Listing 1 can be bound to elements in the document object model (DOM), e.g., two input fields. When the user changes one of these input fields, the other one's value will be updated to the correct value determined by the constraint system.

2.3 Structure Manipulation

WarmDrink is a domain specific language (DSL) for defining relation between components in constraint systems [11]. This allows for defining what we call *intercalating constraints* which can manipulate values between elements in structures such as lists or trees. An example of this could be a multi-city flight itinerary [11]. Let us say the first flight is at 15th of June, then the next flight can be no earlier than 15th of June as well. This relation can be introduced as an intercalating constraint between all flights in flight itinerary. The date of each flight must be equal to or greater than the preceding flight's date. If one of the flight's date gets updated, then so must all the succeeding flights as well.

2.4 React

React is a JavaScript library for building user interfaces [3]. It is based on the concept of components, which are reusable pieces of code that can be composed together to build complex user interfaces. React components are written in JavaScript, and can be written using either a class-based or a functional approach. In Listing 2, we can see an example of a React component written using the functional approach. Notice how the `return`-function contains HTML-like syntax. This is known as JSX (or JavaScript XML), which is an extension of JavaScript's syntax that allows for using HTML directly in you JavaScript files. React then transpiles the code into regular JavaScript, which can be run in a web browser.

Listing 2: A React component written using the functional approach.

```
function MyComponent(props) {
    return (
        <div>
            <h1>{props.title}</h1>
            <p>{props.text}</p>
        </div>
    );
}
```

React components can be rendered to the DOM using the `ReactDOM.render` method. This method takes a React component and a DOM element as arguments, and renders the component to the DOM element. In Listing 3, we can see how we can render the component from Listing 2 to the DOM. Notice how we pass *props* from the parent component down to the child component within the JSX.

Listing 3: Rendering a React component to the DOM.

```
ReactDOM.render(
    <MyComponent title="Hello, world!" text="This is my first
        ↪ React component." />,
    document.getElementById('root')
);
```

2.5 Web Application Programming Interfaces

An application programming interface (API) is a set of rules and protocols for building software applications. It defines how different software components can interact with each other. This can for example be between a frontend web application and a backend database. This is known as a Web API, or an HTTP API.

A web server can provide *endpoints* for a client to send requests to. These endpoints are URLs that the client can send HTTP requests to, and the server will respond with a HTTP response. The client can send different types of requests, such as GET, POST, PUT, and DELETE. These requests are used to retrieve, create, update, and delete data from the server, respectively. The server can then respond with different types of responses, such as 200 OK, 201 Created, 400 Bad Request, and 404 Not Found. These responses are used to indicate whether the request was successful or not, and can also contain data.

2.6 Haskell

Haskell is a purely functional programming language first released in 1990 [7]. It is a statically typed general purpose language which uses lazy evaluation and type inference. Lazy evaluation is a technique where the compiler does not evaluate expressions until they are needed. And type inference is a feature in the compiler which can infer the types of expressions without them having to be explicitly annotated in the actual code..

Haskell also provide a powerful type system, which allows for the creation of custom types and type classes. Type classes are similar to interfaces in object-oriented programming languages, and can be used to define functions that can be used on different types. For example, the `Show` type class defines a function `show` which can be used to convert a value to a string. While the `Eq` type class is used for determining equality between values.

Data types in Haskell can be considered algebraic data types. They can be either *product types* or *sum types*. Product types are types that contain multiple values, such as tuples and records. Sum types are types that can be one of multiple types, such as the `Maybe` type. The `Maybe` type can either be `Just a` or `Nothing`, where `a` is a type variable. This type is used to represent values that may or may not be present, such as the result of a database query.

3 Frontend Implementation

3.1 Component View and State

The frontend part of our CLI tool is written in TypeScript [2] with React as our frontend library of choice.The main component `App` is the root component of our frontend and embodies everyhting the user sees and interacts with. The user is presented with a split screen consisting of a two main components: the view of the constraint system, and the terminal, see Figure 1.

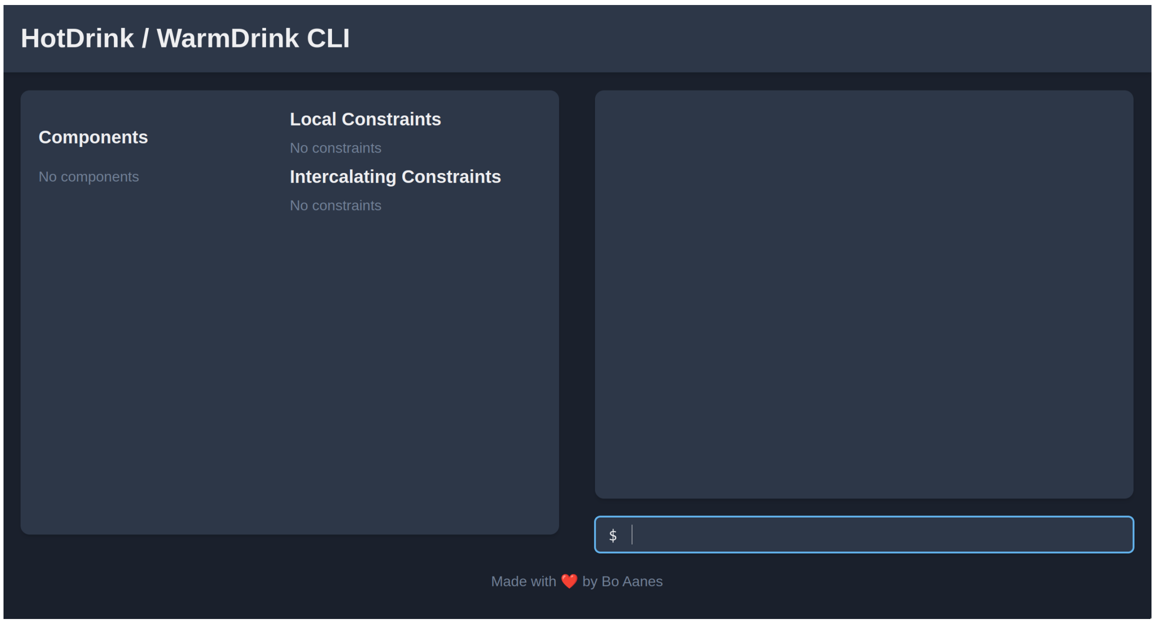


Figure 1: The view of the constraint system and the terminal.

In addition to the view of the program, the `App`-component also manages the state of the constraint system. We have defined types for components, constraints, methods, variables and component lists as can be seen in Listing 4.

To manage state in React we use a *hook* [1] called `useState`. `useState` is a function that takes an optional default value state and returns an array consisting of two elements. The first being the current state, and the second begin a function to update the state. In Listing 5, we show how the state for the application is handled in the root component. We manage state for the components, the local constraints, as well as the intercalating constraints. In addition to this, we also have a state `highlighted`, and a state `enforceFromIndex`. `highlighted` contains a list of strings indicating elements in the GUI that are highlighted in a unique color, this is explained later in this Section. `enforceFromIndex` indicates the index of the com-

Listing 4: Types for constraint systems.

```
export interface Method {
  methodName: string;
  inputs: string[];
  expressions: Array<[string, string | undefined]>;
}

export interface Constraint {
  constraintName: string;
  methods: Method[];
}

export interface Variable {
  varID: string;
  varValue: number | boolean;
}

export interface IComponent {
  compID: number;
  variables: Variable[];
  strength: string[];
}

export interface ComponentList {
  components: IComponent[];
}
```

Listing 5: Using `useState` to manage state.

```
const [components, setComponents] = useState<IComponent[]>([]);
const [constraints, setConstraints] =
    ↪ useState<Constraint[]>([]);
const [intercalatingConstraints, setIntercalatingConstraints]
    ↪ = useState<
  Constraint[]
>([]);
const [highlighted, setHighlighted] = useState<string[]>([]);
const [enforceFromIndex, setEnforceFromIndex] =
    ↪ useState<number>(0);
```

ponent where the enforcing of constraints should be triggered. E.g., if a user updates a value in `component 2`, then this is where the constraints should be enforced from, as no values needs to be changed in the preceding components in order to satisfy the entire constraint system.

3.2 Terminal and User Interaction

For the terminal emulator of our frontend implementation we have put together two visual components: one box where the user can see their command history, as well as an input field at the bottom (see Figure 1). While this is actually just a simple form with a display of everything previously entered into the input field, the idea is that it should still feel like a regular terminal emulator. The `Terminal` component contains two main states: one for the current state of the input field, as well as one for the command history. In addition to these states, we have a function `submit`, which takes a *FormEvent* as input and handles this input as a command. This is our main function for the business logic of our CLI, it handles all commands and is responsible for all user interaction.

The function `submit` is triggered when the form containing the input field is submitted. We use HTMLs native `onSubmit` event handler to trigger this function, like so:

```
<form onSubmit={submit}>
```

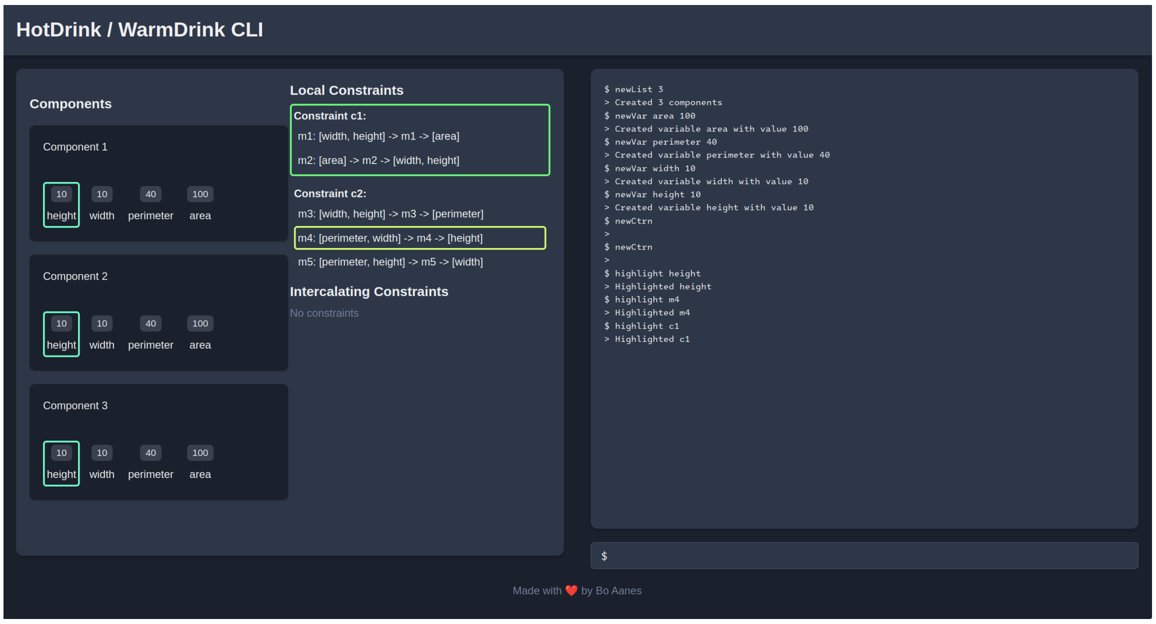
The function then prevents the default behavior of the form, which is to reload the page, and instead uses the current state of the input field as a command. The input state is then split into words, where the first word is treated as the command, and the following words are treated as arguments. First we check if the command is a valid command. If it is not a valid command, we clear the input state and append a message "Invalid command" to the command history and return void. Proceeding, we have a *switch* statements with cases that match the individual commands, where each case handles the business logic related to that command. This may be to insert a new component, swap the positions of two components, adding or updating a variable, or highlighting a given variable or constraint. Figure 2 shows an example state of the application where the user has defined some components with variables and constraints and highlighted both a variable, a method, and a constraint. Notice how the components and constraints are displayed, as well as the command history on the right side.

4 API Implementation

4.1 Haskell API

To handle the algorithmic parts, mainly planning and solving of the constraint system, we have built an API in Haskell using a library called *Scotty* [10, 4]. Scotty provides a minimal way of defining endpoints for HTTP requests. In Listing 6 we can see how a simple endpoint for a GET-request reponding with "Hello World" can be created. The function `main` calls the function `scotty` with two arguments, a port number for where to listen for requests, as well as a `ScottyM ()`[1]. The `ScottyM ()` is scotty's own monad for performing sequential operations. Within the outermost do-notation in Listing 6 we can list endpoints with their respective HTTP request types. In the example, we use the function `get` which takes two arguments, a route pattern ("/hello") and a `ActionM ()`. The `ActionM ()` is an action we

[1]`https://hackage.haskell.org/package/scotty-0.12.1/docs/Web-Scotty.html#t:ScottyM`


HotDrink / WarmDrink CLI

Components

Component 1

10	10	40	100
height	width	perimeter	area

Component 2

10	10	40	100
height	width	perimeter	area

Component 3

10	10	40	100
height	width	perimeter	area

Local Constraints

Constraint c1:

m1: [width, height] -> m1 -> [area]

m2: [area] -> m2 -> [width, height]

Constraint c2:

m3: [width, height] -> m3 -> [perimeter]

m4: [perimeter, width] -> m4 -> [height]

m5: [perimeter, height] -> m5 -> [width]

Intercalating Constraints

No constraints

```
$ newList 3
> Created 3 components
$ newVar area 100
> Created variable area with value 100
$ newVar perimeter 40
> Created variable perimeter with value 40
$ newVar width 10
> Created variable width with value 10
$ newVar height 10
> Created variable height with value 10
$ newCtrn
>
$ newCtrn
>
$ highlight height
> Highlighted height
$ highlight m4
> Highlighted m4
$ highlight c1
> Highlighted c1
```

```
$
```

Made with ❤️ by Bo Aanes

Figure 2: Example state of the application with some highlighted elements.

want the server to perform, in this case return a raw text displaying "Hello World!".

Listing 6: Example of a simple endpoint in Scotty.

```haskell
main = scotty 3000 $ do
  get "/hello" $ do
    text "Hello World!"
```

As stated previously, we use our API to handle the algorithmic parts of the constraint system. The main idea here, is to use our Haskell API as a black box. We send a request to the server with our current state of the constraint system, and the server responds with our updated variable values, which we assign to our frontend state. Becuase of this, we need to be able to accept requests with a body that contains our state. HTTP requests can carry this information structured as JSON[2], but because Haskell is statically typed, we need to define data types for how this JSON-data is structured, along with parsers and encoders.

To achieve this we use a library called *Aeson* [9]. Aeson provides a way to encode and decode Haskell data types to and from JSON. In Listing 7 we show an example of how we can define a data type `JSONVariable` with two fields, `varID` and `varValue`.

The language extensions `DeriveAnyClass` and `DeriveGeneric` are used to derive the `FromJSON` and `ToJSON` type classes. These type classes are

[2]`https://www.json.org/json-en.html`

Listing 7: Example of a data type for a JSON-variable.

```haskell
{-# LANGUAGE DeriveAnyClass     #-}
{-# LANGUAGE DeriveGeneric      #-}

data JSONVariable
  = JSONVariable
      { varID :: String
      , varValue :: String
      }
  deriving (FromJSON, Generic, ToJSON)
```

derived automatically, which provide a parser and encoder without having to write these manually.

Furthermore, we have defined data type for the API's input and output JSON types. As stated earler, the API recieves the state of the whole constraint system, and solves this before outputing the new values of each variable. Listing 8 shows how these data types are defined.

Listing 8: Input and output data types for the API.

```haskell
data Input
  = Input
      { components                :: [JSONComponent]
      , constraints               :: [JSONConstraint]
      , intercalatingConstraints  :: [JSONConstraint]
      , enforceFromIndex          :: Int
      }
  deriving (FromJSON, Generic, ToJSON)

data Output where
  Output :: {vars :: Map String [JSONVariable]} -> Output
  deriving (FromJSON, Generic, ToJSON)
```

The `Input` data type contains the set of components, constraints, and intercalatingConstraints, in addition to the index of which component to enforce constraints from, as described in the state of the frontend application from Section 3.1 and Listing 5. The `Output` data type is a map from component identifiers to values of JSONVariables.

Listing 9: Example of an endpoint for the API.

```haskell
routes :: ScottyM ()
routes = do
  post "/" $ do
    input <- jsonData :: ActionM Input
    let componentList = solve input
    case satisfied of
      Just newVars -> do
        json $ Output newVars
      Nothing -> do
        status status500
```

With these defined data types we can now use functions for capturing the JSON request body as well as respond with a JSON response of the desired structure. In Listing 9, we decode the JSON request body using the function `jsonData`, we then solve the constraint system using a MDCS library in Haskell, before returning the formatted response with `json`. In the case that there is an error in solving the constraint system, we return a HTTP 500 response, which indicates an internal server error. This is a simplified version of the actual function in our implementation, but provides an overview of how we handle requests and responses.

4.2 Sending Requests and Receiving Responses

When one of the states regarding components, constraints, or intercalating constraints changes (which we discussed in Section 3.1 and outlined in 5), the constraint system may be un-satisfied. Things that could lead to this can be e.g., the user updating a variables value, introducing a new constraint or simply inserting a new component, among other operations. When a state changes in React, we can use the *useEffect*-hook[3] to trigger side-effects. The useEffect-hook is a function that takes another function as input (the effects we would like to perform), as well as a dependency array — an array containing the states which when changed, should trigger the effects. Listing 10 outlines how we utilize the useEffect-hook to send a request to the Haskell API and update the state using the response.

Listing 10: useEffect-hook to send a request to the Haskell API and update the state.

```
useEffect(() => {
  const enforceConstraints = async () => {
    const response = await fetch('http://localhost:8000/', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        components: components,
        constraints: constraints,
        intercalatingConstraints: intercalatingConstraints,
        enforceFromIndex: enforceFromIndex
      });
    });
    const data = await response.json();
    setComponents(data);
  }
  enforceConstraints();
}, [components, constraints, intercalatingConstraints,
   ↪ enforceFromIndex]);
```

As with the function outline in Listing 9, the function outlined in Listing 10 is a simplified version of the actual implementation. The actual implementation uses a helper function for the request to the server, which restructures the state to fit the data types defined in Haskell. In addition to this, the actual implementation contains error handling, as well as a check to see if the response is different from the current state, before updating the state. This is to avoid an infinite loop of requests and responses. The call

[3]`https://react.dev/reference/react/useEffect`

to setComponents is also simplified in the example, as data is not in its correct format, but still illustrates the functionality of setting the new state of components.

5 Usage

The user can interact with the application entirely throught the terminal. The terminal has a set of commands that can be listed with the help-command. Listing 11 shows the available commands.

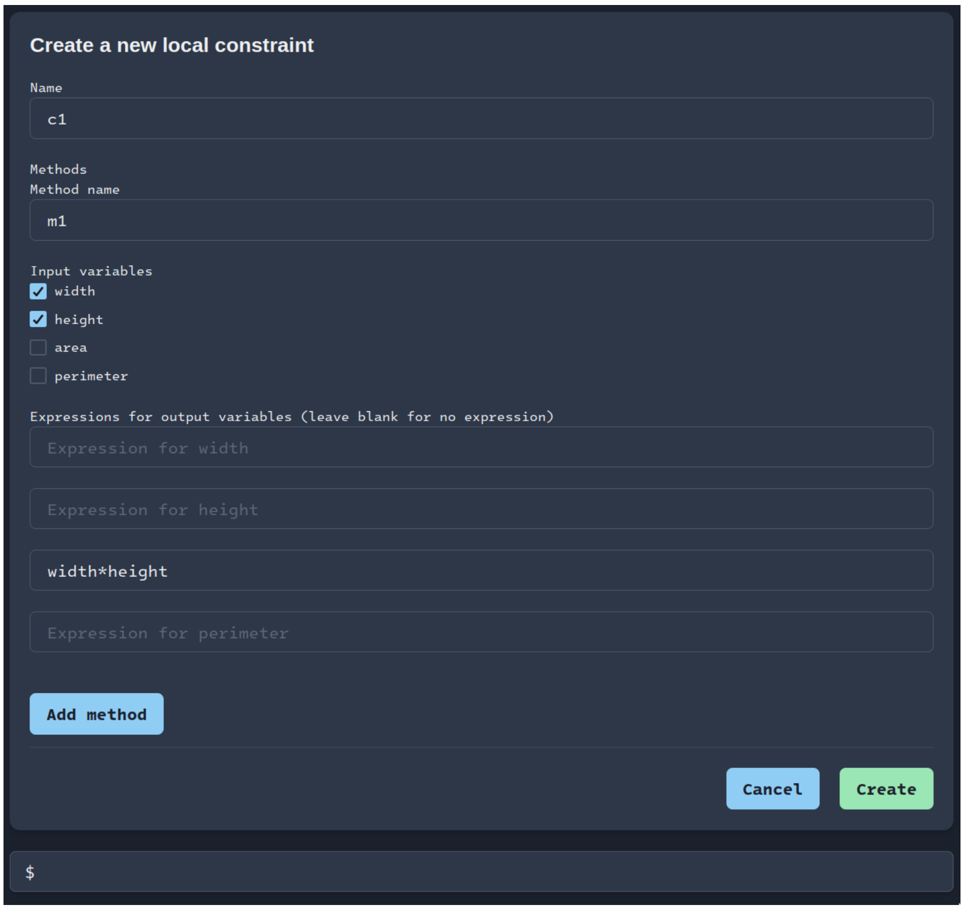
Listing 11: Available commands in the terminal.

```
$ help
> Available commands:
> newComp - Add a new component
> newList <n> - Add a new list of size n
> newVar <varID> <value> - Add a new variable to all components
> newCtrn - Add a new constraint to all components
> newIctrn - Add a new intercalating constraint
> updVar <compID> <varID> <value> - Update the given variable
> delVar <varID> - Delete the given variable
> insAfter <compID> - Insert a new component after the given
    ↪ component
> swap <compID> <compID> - Swap the given components
> rmv <compID> - Remove the given component
> highlight <identifier> - Highlight the element for the given
    ↪ identifier
> reset - Reset highlights
> help - Display this help message
```

Most of these commands are self explanatory, as they manipulate the constraint system in a rather simple way. However some are worth elaborating further on, mainly highlight, newCtrn and newIctrn. Both of these commands adds a new constraint to the set of local constraints, and the set of intercalating constraints, respectively. Their functionality is the same, except for the state they update.

The command highlight is designed to highlight any given part of the constraint system using its identifier, regardless of whether the identifier points to a constraint, a method or a variable. It is important, however, to keep all identifiers unique across the these different types. When the user highlights a variable, e.g., by doing highlight width, the variable with the identifier width will be highlighted (see Figure 2) by surrounding it with a border of a color generated using the hash of its identifier.

When a user invokes the command newCtrn, the command history is replaced by a dialog, with a form. This form contains the necessary fields for adding a new constraint. Adding a constraint involves specifying its name, as well as its method. A method consists of a name, a set of input variables, as well as a set of expressions for the method's output variables. Figure 3 outlines how the constraint dialog looks. In this example we have named the constraint c1 and added one method m1 taking the variables width and height as input. And writing the expression width*height to the variable area. Notice how the placeholders of the input fields for the expressions indicate the corresponding output variable. The button Add method will extend the form with another method form. The button Cancel will cancel the dialog and return the user to the regular terminal with the command history, while the Create button will add the constraint to the application's state. The expressions are parsed and evaluated by the Haskell server.


Create a new local constraint

Name

c1

Methods

Method name

m1

Input variables

- [x] width
- [x] height
- [ ] area
- [ ] perimeter

Expressions for output variables (leave blank for no expression)

Expression for width

Expression for height

width*height

Expression for perimeter

Figure 3: The constraint dialog.

5.1 Project Structure

The project[4] is structured in two parts, the frontend Web-CLI (found in the directory web-cli) as a React project, and the backend Haskell API (found in the directory server) as a Stack project.

To run the entire application locally, a user can either use Stack[5] and

[4]Found at https://github.com/boaanes/319

[5]https://docs.haskellstack.org/en/stable/

Yarn[6], or Docker[7].

To run using Stack and Yarn, open two terminals, one in the `web-cli` directory and one in the `server` directory. In the `web-cli` directory, run the command `yarn start`, and in the `server` directory, run the command `stack run`. This will start the frontend and backend respectively, and the frontend application can be accessed at `http://localhost:3000`.

To run the entire application using docker, Docker must be installed. This repository can either be cloned and run using *docker compose*[8] by running the command `docker compose up` in the root directory. The frontend application will then be available at `http://localhost`

The easiest and simplest solution to runnig the application however, is to use the Docker images for both the frontend and backend hosted at Docker Hub[9]. By running the application this way, the only prerequisite is to have Docker installed, as there is no need to clone the repository. This can be achieved by running the following commands in sequence:

```
docker run --name haskell-server -p 8000:8000 -d
    ↪ boaanes/319-haskell-server
```

```
docker run --name react-app --link
    ↪ haskell-server:haskell-server -p 80:80 -d
    ↪ boaanes/319-react-app
```

In addition to the directories for the application itself, there is a directory `report` containing the LaTeX source code for this report, as well as additional resources (such as figuers and generators) used in this report.

6 Conclusion

In this report we have presented a tool for visualizing constraint systems. The tool is designed to be used by anyone who want to experiment with MDCSs and structure manipulation of lists. It is intended to be used as a tool for debugging and understanding constraint systems in general. The tool is implemented as a web application, with a Haskell backend and a React frontend, and is designed to be used entirely through the terminal, where the user can interact with the application by issuing commands.

In addition to this, the application itself is fit for demoing MDCS to students, as it should be farily easy to use and understand. Features such as highlighting variables and constraints also help in this context of teaching.

[6] `https://yarnpkg.com/`

[7] `https://www.docker.com/`

[8] `https://docs.docker.com/compose/`

[9] `https://hub.docker.com/`

References

[1] Introducing Hooks. `https://legacy.reactjs.org/docs/hooks-intro.html`. Accessed: 2023-06-19.

[2] TypeScript: JavaScript With Syntax For Types. `https://www.typescriptlang.org/`, 2023. Accessed: 19.06.2023.

[3] Alex Banks and Eve Porcello. *Learning React: modern patterns for developing React apps*. O'Reilly Media, 2020.

[4] Andrew Farmer. scotty: Haskell web framework inspired by Ruby's Sinatra, using WAI and Warp, 2022.

[5] John Freeman, Jaakko Järvi, and Gabriel Foust. HotDrink: A Library for Web User Interfaces. *SIGPLAN Not.*, 48(3):80–83, sep 2012.

[6] Magne Haveraaen and Jaakko Järvi. Semantics of multiway dataflow constraint systems. *Journal of Logical and Algebraic Methods in Programming*, 121:100634, 2021.

[7] Paul Hudak and Joseph H Fasel. A gentle introduction to Haskell. *ACM Sigplan Notices*, 27(5):1–52, 1992.

[8] Jaakko Järvi, Gabriel Foust, and Magne Haveraaen. Specializing Planners for Hierarchical Multi-Way Dataflow Constraint Systems. *SIGPLAN Not.*, 50(3):1–10, sep 2014.

[9] Bryan O'Sullivan. aeson: Fast JSON parsing and encoding. `https://hackage.haskell.org/package/aeson`, 2023.

[10] Ecky Putrady and Ecky Putrady. RESTful APIs. *Practical Web Development with Haskell: Master the Essential Skills to Build Fast and Scalable Web Applications*, pages 135–164, 2018.

[11] Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. A domain-specific language for structure manipulation in constraint system-based GUIs. *Journal of Computer Languages*, 74:101175, 2023.