University of Bergen
Department of Informatics

A Functional Implementation of a Multiway Dataflow Constraint System Library

Author: Bo Victor Isak Aanes
Supervisors: Mikhail Barash, Jaakko Järvi, Knut Anders Stokke

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

Developing graphical user interfaces (GUIs) is a time consuming and error prone task. The complexity and difficulity of implementation tends to arise as more and more elements are introduced into the GUI, along with dependencies and constraints between these elements. This will often lead to poor maintainability and technical debt, which in turn slows down development rates of projects. Libraries like HotDrink (Freeman et al., *SIGPLAN Notices 48*, 2012) and domain specific languages like WarmDrink (Stokke et al., *J.Comput.Lang. 74*, 2023) aim to eliminate errors, while also improving the developer experience. These systems provide a declarative way of defining constraints between elements in a user interface and manage all business logic behind the scenes. This thesis introduces a purely functional implementation of both HotDrink and WarmDrink and a way to interact with these engines through a command line interface. We show that implementing multiway dataflow constraint systems and an engine for structural manipulation in a functional language, such as Haskell, can deliver a codebase with clear and concise function definitions that highlight the ideas of constraint systems themselves. Such clarity helps in further development of GUI frameworks and libraries. It also helps new programmers to learn and adapt this new technology. We desribe how our Haskell implementation was used by the students of a programming language class.

Acknowledgements

I would like to thank my supervisors Mikhail Barash, Jaakko Järvi, and Knut Anders Stokke for their invaluable advice, expertise, and persistent encouragement.

My heartfelt thanks are extended to my family, whose belief in me has been a great support throughout this challenging journey. I am grateful for my friends, whose companionship and support have transformed countless hours of study into an enjoyable and unforgettable experience.

Finally, my acknowledgement wouldn't be complete without thanking everyone at Jafu. The collaborative spirit, endless hours of procrastination, and warmth of everyone have been a lot of fun.

Thank you.

Bo Victor Isak Aanes
Thursday 15th June, 2023

Contents

List of Figures

Listings

Chapter 1

Introduction

Developing graphical user interfaces (GUIs) can be a complex task [23, 24], especially with large amounts of elements and dependencies, as well as interactions between them. Without a proper architecture, difficulty in both maintaining and further developing GUIs quickly rise with the size of the GUI. Inconsistencies in user experience or unexpected behavior can occur if these interactions are not properly handled. *HotDrink* [9, 8], a *multiway dataflow constraint system* (MDCS) library written in JavaScript, introduces a way of specifying constraints based on relations between GUI elements. It also handles all business logic to ensure that all relations between elements hold. This programming model (MDCS) is a powerful way for developers to manage constraints between elements, not only in GUIs, but in other applications as well, such as spreadsheets [27], and collaborative systems [19].

Difficulties regarding user interfaces are also prevalent in the context of structures, such as lists [28]. *WarmDrink* [28] is a *domain specific language* (DSL) that generates JavaScript code for specifying relations between elements in GUI structures. WarmDrink assists the developer in specifying operations for manipulating a GUI strucutre such that relations between all elements hold.

There exists a variety of implementations of MDCSs which either use HotDrink directly or implements HotDrink in some other way. Examples of these are: HotDrink used in spreadsheet applications [27], visual specifications used to define constraint systems with HotDrink [2], as well as an implementation of HotDrink's functionality in Rust and WebAssembly [29].

In this thesis we combine an implementation of a multiway dataflow constraint system and a system for specifying constraints between elements in structures. Furthermore we provide the end-user or developer with a way of interacting with these implementations through a command line interface (CLI). This allows the user to define constraint systems within isolated components as well as manipulate lists of these components, where components can exhibit relations to each other. The CLI provides a way of interacting with our system, which in turn utilizes both of the engines for constraints and structure manipulation to simplify this usually tedious work, and deliver something the user can trust and confidentially rely on.

Our implementation is written in Haskell, which is a purely functional programming language. This benefits us in several ways, as we can use the language to express our ideas in a concise and elegant way, while being able to trust that our constraint models always remain pure. Using a pure and declarative language like Haskell allows the focus during development of these systems to be on the actual ideas and semantics of the systems, rather than getting lost in all the implementation clutter that may arise when using an imperative language.

This thesis is organized as follows. Chapter 2 provides the reader with the necessary background information regarding multiway dataflow constraint systems and structural manipulation, mainly within the context of HotDrink and WarmDrink. Functional programming and Haskell is also introduced in order to provide necessary information about this programming paradigm, and some of Haskell's semantics. Chapter 3 goes in depth about HDFun, a functional version of HotDrink's business logic implemented in Haskell. We explain how the user interacts with the engine and how constraint systems can be constructed. Furthermore, we go into details about the algorithmic parts and data types. In Chapter 4 we introduce how we have tackled structural manipulation in our implementation. We discuss the data types and algorithms related to our interpretation of WarmDrink, as well as how the user can specify relations between elements in lists. Chapter 5 is split into two parts where we present the actual Haskell implementation of HDFun and WDFun, respectively. In Chapter 6, we discuss how our implementation has been utilized in a course at the University of Bergen. Chapter 7 presents related work, and Chapter 8 offers a conclusion as well as discussions about future work.

Chapter 2

Background

In this chapter we will give an overview of multiway dataflow constraint systems [15] and HotDrink library [9, 8], which provides a way to define multiway dataflow constraint systems in JavaScript. We will also provide an overview of structural manipulation of components in such constraint systems, where we will discuss WarmDrink [28] — a specification language for this purpose. We will touch upon benefits of approaching these domains using functional programming.

2.1 Multiway Dataflow Constraint Systems

A multiway dataflow constraint system S is defined as a tuple of variables and constraints denoted as $\langle V, C \rangle$, where V is the set of variables in S, and C is the set of constraints in S [29]. A constraint is defined as a tuple $\langle R, r, M \rangle$. $R \subseteq V$ is the set of variables in the constraint, r is the set of n-ary relations among the variables in R, and M is the set of methods in the constraint. If a method m in M is executed, the constraint is *enforced* by taking some subset of R as input, and evaluating and storing the results in another subset of R. If all values in V satisfy the relations in r, we say that the constraint is *satisfied*.

2.1.1 HotDrink

HotDrink is a JavaScript library that provides both an API, as well as a DSL to build constraint systems and bind elements of the *document object model* (DOM) to variables

of the constraint system. To build a view-model for a working GUI, the developer only needs to define the different *components* of the constraint systems, and the methods of these components. This is done declaratively via an API or a DSL. A user updating the value of a DOM-element that causes a constraint system's variable to change, might lead to the constraint system not being satisfied. HotDrink will then need to recompute the values of the variables which do not satisfy their constraint. This is done through *planning* and *solving*.

An alternative way to represent a constriant system is through a graph, more accurately an *oriented bipartite graph*. The graph $G = \langle V + M, E \rangle$ consists of vertices V and M, which contain the variables and methods of the constraint system, respectively. E represents the directed edges between the vertices and expresses the inputs and outputs of methods in M [14].

Let us consider an example of how we can define constraints of a rectangle that a user can manipulate in a GUI. The rectangle has a width and a height, and we want to be able to manipulate these values. However, we also want to be able to manipulate the area of the rectangle, and we alaways want the values to uphold their relations. This means that if we change the width or the height, the area should change accordingly, and vice versa. Furthermore, we also want to take the circumfrence, or rather the perimeter, of the rectangle into consideration as well. We can express these relations using mathematical equations, which we will use later to define our constraints in HotDrink. The constraint system consists of the values for *width*, *height*, *area* and *perimeter* of a rectangle, respectively denoted as w, h, a, and p. By observing the properties of a rectangle we see that we have relations between area, and width and height, as well as relations between perimeter, and width and height. In other words, we can compute a based on w and h, as well as the other way around. We can also compute p based on the w and h, as well as w and h individually using p and either w or h.

These properties lets us define two constraints, one between a, w and h, and one between p, w and h. We will refer to these namely as C_1 and C_2. C_1 contains three relations, one for computing a, one for computing w and one for computing h. These relations can be defined using the following equations:

$$a = w \cdot h$$
$$w = \sqrt{a}$$
$$h = \sqrt{a}$$

For C_2 we have tree relations as well: one for computing p, one for computing w and one for computing h. These relations can be defined using the following equations:

$$p = 2 \cdot (w + h)$$
$$w = \frac{p}{2} - h$$
$$h = \frac{p}{2} - w$$

To define a constraint system consisting of these relations using the HotDrink API and DSL in JavaScript, we can do the following:

Listing 2.1: Defining a constraint system using HotDrink's API.

```
1  const constraintSystem = new ConstraintSystem();
2
3  const component = component`
4      var w, h, a, p;
5
6      constraint {
7          area(w, h -> a) = w * h;
8          widthHeight(a -> w, h) = [Math.sqrt(a), Math.sqrt(a)];
9      }
10
11     constraint {
12         perimeter(w, h -> p) = 2 * (w + h);
13         height(p, w -> h) = p/2 - w;
14         width(p, h -> w) = p/2 - h;
15     }
16 `;
17
18 constraintSystem.addComponent(component);
19 constraintSystem.update();
```

In Listing 2.1, we first instantiate a `ConstraintSystem` object using the API, then we construct a component using JavaScripts template literal syntax[1]. Within this template literal string, we use the HotDrink DSL to define our constraint system. First we define our variables w, h, a and p. We then define two constraints, namely C_1 and C_2 (not named in the DSL-example). The constraints both consists of a set of methods that correspond to our equations defined above. Notice how *width* and *height* are computed by one method alone with the method returning a list list of two expressions that write to their corresponding variables in the order defined by the method signature, namely: `widthHeight(a -> w, h)`.

Finally, we add our component to the constraint system, and invoke any methods needed to satisfy the constraint system with the `update` function.

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals`

2.2 Structure Manipulation

Manipulating structures such as lists or trees in GUIs may seem like a trivial task, and in many cases it is. However, when elements of such structures are related to each other, things might not be so trivial after all.

As an example [28], consider a GUI that represents an agenda of talks at a conference. The structure of the GUI would be a list of talks. Each talk would have a start time and a duration, among other less important properties, such as a title and the name of the speaker. The first talk may start at, e.g., `09:00` and have a duration of `30` minutes. This implies that the second talk must start no earlier than `09:30`. If the second talk has a duration of `20` minutes, then the third talk cannot start before `09:50` and so on.

At first glance, it may seem like all we need to in the example above is to calculate the start time based on the previous talk's duration and start time. However, since we want to be able to manipulate the list, there are several other things we need to be able to compute. Say we want to swap talks 1 and 2. This means that talk 1 will now last `20 minutes`, and that talk 2 will start at `09:20`. Furthermore, we may want to support other operations, such as inserting a talk wherever we would like, and of course removing a talk.

In addition to this, we want to be able to perform operations on the talks themselves, such as changing the duration, which will in turn affect the start time of the following talks. The complexity of what may seem like a simple implementation grows quickly.

2.2.1 WarmDrink

WarmDrink [28] is a *domain specific langauge* (DSL) designed to relieve the developer from having to implement the logic for manipulating structures such as lists and trees. The developer only needs to define the structure of the GUI and the relations between the elements of the structure. The developer can then use the DSL to define the operations that can be performed on the structure. The DSL will then generate the logic for manipulating the structure.

Formally, a WarmDrink specification can be represented as a tuple $\langle S, F, R, T \rangle$, where $S = \langle V, E \rangle$ is a GUI *structure* specification, more specifically a rooted acyclic graph. V

represents a set of vertices which in turn represents *WarmDrink components* [28]. E is a set of labeled edges between components that represents *features* inside structural elements [28]. F is a set of *subroutines* which can be either procedures or predicates defined in JavaScript code. R is the set of finitary relations defined on elements of the structure. A relation r in R is in turn its own triple $\langle r^{test}, r^{establish}, r^{unestablish} \rangle$ where r^{test} checks whether a predicate holds, $r^{establish}$ establishes a predicate, and $r^{unestablish}$ unestablishes a predicate. The two former elemets are optional, and are JavaScript functions. T is the set of *transformation rules* to define transformations on relations.

2.3 Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids mutating the state of data. It is a declarative programming paradigm in the sense that the program logic is expressed without describing its control flow.

Unlike imperative programming, where the developer writes specific instructions for what the computer should do, functional programming is more about specifying the result we want from the computer. This is done by defining functions that take input and return output. The output of one function can be used as input to another function, and so on. This is called *function composition*.

A well-known example of a functional programming language is Haskell [4]. Haskell is a purely functional programming language, which means that it does not have any side effects. Functions in Haskell are also mathematical functions: a function will always return the same value for the same input. Furthermore, a Haskell function will never perform updates to values or other expressions outside of itself. This is what defines Haskell as a *pure* language.

Another advantage many functional programming languages provide is algebraic data types. An algebraic data type is a data structures that provides one or more constructors, each of which can contain zero or more arguments. These algebraic data structures may inhabit certain properties a developer can exploit in order to write more concise and legible code. In Haskell, for instance, a developer can instantiate certain *type classes* [11] for a custom data type. A type class is an interface consisting of a set of functions that must be implemented for the given data type. A developer can define functions that

work for any type that instantiates a specific type class. This enables the developer to write *polymorphic functions* [11] that work for any type that instantiates a specific type class. An example of a type class in Haskell is `Eq`, which defines the behavior of equality. Any type that instantiates `Eq` can be compared for equality using the polymorphic `==` function.

Chapter 3

Constraint Systems in Haskell

In the previous chapter, we explained how multiway dataflow constraint systems work, and how they can be specified in JavaScript using the HotDrink library. We also discussed structure manipulation and the use of WarmDrink. In addition to this we explained benefits of using functional programming to approach different problems and domains.

In this chapter we will discuss an implementation of the core of HotDrink in Haskell, which we call *HDFun*. By this core we mean the ability to specify constraints between values, i.e., to specify constraint systems, plan and solve them.

As discussed in the previous chapter, HotDrink is a JavaScript library that provides the developer with both an API and a DSL to define constraints between variables. It is implemented in JavaScript and also uses a graph representation of the constraint system the user builds. The variables defined with HotDrink can be subscribed to by DOM-elements in HTML and will be updated automatically when constraints are enforced by HotDrink. HotDrink follows the Model-View-ViewModel architecture [9]. In short, the Model embodies business logic of the application. The View represents the GUI, which binds values to the ViewModel. The ViewModel serves as an intermediary, managing data transmission between the Model and the user. In HotDrink, the ViewModel is a constraint system. Its connections to the view are through variable bindings.

While our Haskell implementation of HotDrink contains all the necessary logic needed to represent, plan and enforce constraint systems, we do not provide an immediate way of binding values of variables in a given constraint system to some View. Instead, our implementation can be considered an *engine*, which allows it to be used as a tool for building other applications on top of.

3.1 User Interface

The user interface of our implementation is a *command line interface* (CLI) that allows the user to define constraint systems. In our implementation, a constraint system consists of a collection of variables, a collection of constraints, as well as a strength for each variable, which will be further elaborated later in the thesis. The user can define variables of different types, such as numbers and booleans. These variables can be updated whenever the user wants to, and the constraint system will satisfy all its constraints according to the current values and strengths.

When a user runs the CLI, they are prompted with a welcome message and a prompt (see Figure 3.1). The user can list all the available commands with the `help` command.

```
Type 'help' for a list of commands
$ help
Available commands are:
new var <var> <val>        add a variable to the constraint system
new ctrn <n>               add a constraint with n methods

upd var <var> <val>   update a variable
del var <var>              delete a variable

show var                   show all variables
show ctrn                  show the constraints
show str                   show the strength of the variables
show pln                   show the current plan

run                        enforce the plan

help                       show this message
exit                       exit the program
$
```

Figure 3.1: The prompt of the CLI.

3.1.1 Constructing a Constraint System Interactively

Fiugre 3.1 shows the available commands in the CLI. The user can define variables with the `new var` command, which takes a variable name and a value as arguments. The user can update the value of a variable with the `upd var` command.

The process of defining a constraint in the CLI involves specifying the relationships between methods and their input and output variables. This is achieved by using the `new ctrn` command, which takes a single argument representing the number of methods involved in the constraint.

For each method, the user is prompted to provide the following information:

- `Method name:` The user is prompted with "Enter name of method:" and is expected to input a unique identifier for the method.
- `Input variables:` The user is prompted with "Enter space-separated input names to method:" and is expected to input the names of the input variables, separated by spaces.
- `Output variables:` The user is prompted with "Enter output variables to method:" and is expected to input the names of the output variables, separated by spaces.
- `Expressions for output variables:` For each output variable, the user is prompted with "Enter expression for [variable]:" and is expected to input the expression for that variable in terms of the input variables. The system will then attempt to parse the expression and return either "Parse success" or an error message.

Upon successful parsing of expressions for all methods in the constraint, the system constructs a graph representing the possible data flows within the constraints, that is, the methods with their input and output variables, as well as the expressions of their methods. Figure 3.2 outlines an example of how a graph representation of a constraint might look. The squared nodes represent variables and the circular nodes represent methods. The dashed edges represent which variables a given method reads from, and the solid edges represents which variables a given methods writes to.

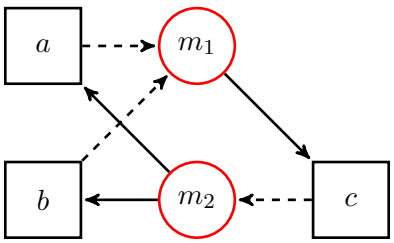

Figure 3.2: Graph representation of an example constraint system.

Once all methods have been defined and their expressions successfully parsed, the CLI confirms the addition of the constraint to the system by displaying the message "Added constraint with [number of methods] methods." This signifies the completion of the constraint definition process.

3.2 Planning

Whenever the value of a variable in the constraint system is updated, constraints that contain that variable may need to be re-enforced as the required relation between variables may no longer hold. Invoking any of the defined methods in the corresponding constraint will enforce the constraint and make all variables satisfy their relations. However, when a method is invoked, it may also update variables that are part of other constraints, meaning that these constraint now need to be enforced again too. To satisfy all relations in the constraint system, at least one method from each constraint need to be executed, and this needs to happen in a topological order according to the flow of data in the graph [14].

A graph containing all variables and a single method from each constraint is called a *solution graph*. A solution graph G consists of variables V, where each variable have a maximum in-degree of 1, methods M where each method is part of a unique constraint, and contains no cycles.

In order to satisfy all constraints, we must determine which methods need to be invoked. Specifically, this involves computing a solution graph, an action we refer to as *planning*. A planning algortihm will select one method from each constraint such that the selections results in a valid solution graph. There are several approaches to this, and different planning algorithms we can use to solve a constraint system, to ensure that all relations hold. Below we discuss two types of planning algorithms, the *simple planner* and the *hierarchical planner*.

3.2.1 Simple Planner

The simple planner is what is known as a *propagate degrees of freedom* planner [30]. A variable that is only part of one constraint is called a *free variable*, and a *free method* is a method that only writes to free variables. These properties let us observe that as long as we only pick methods that are free, we can safely invoke these methods without writing values to variables part of other constraints. As such we can satisfy the constraint system by picking one free method from each constraint.

The pseudocode for the algorithm can be found in a previously published work, which also contains a detailed explanation. This work is referenced in the provided citation

Algorithm 1 Simple Planner($G\langle V + M, E\rangle$)[14]

$M_s \leftarrow \emptyset$
$M_u \leftarrow M$
while $M_u \neq \emptyset$ **do**
    if no free methods in $G[V + M_u]$ **then**
        return *no solution*
    end if
    $m \leftarrow$ a free method in $G[V + M_u]$
    $M_u \leftarrow M_u \setminus \{m\}$
    $M_s \leftarrow M_s \cup \{m\}$
end while
return $G[V + M_s]$

[14] and the algorithm is outlined as Algorithm 1. The algorithm for the simple planner carries a state, consisting of: M_s, the set of methods that are part of the solution graph, and M_u, the set of methods for the constraints which are not yet satisfied [14].

Let us consider the example from Section 2.1.1 and apply the simple planner to it. The graph representation of the constraint system is shown in Figure 3.3. The squared nodes represent variables, and the circular nodes represent the methods. The dashed edges represent the inputs to the methods, or which variables a method reads from. The solid edges represents the output of the methods, or which variables a method writes to. The constraint C_1 represents the relations between a, w and h and contains the methods m_1 and m_2 (colored red). The constraint C_2 represents the relations between p, w and h and contains the methods m_3, m_4 and m_5 (colored green).

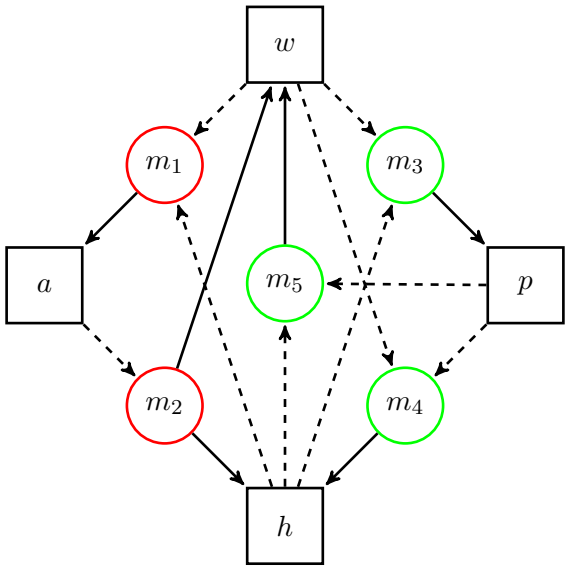

Figure 3.3: Rectangle constraint system.

To satisfy the constraint system using the simple planner, we first need to pick a free method from each constraint. We can pick any free method, but for simplicity we will pick the first free method we encounter. In this case we pick m_1 from C_1 and m_3 from C_2 (note that these are also the only free methods in the constraint system). We then add these methods to the set M_s, the set of methods we want to invoke. We then remove these methods from the set M_u, the set of methods we have not yet invoked. We then repeat this process until M_u is empty. In this case we pick m_1 from C_1 and m_3 from C_2, as these are the only free methods in each constraint. We now end up with a solution graph containing m_1 and m_2 along with all their inbound and outbound edges, as well as all the variables (see Figure 3.4 below). We can now invoke the methods of m_1 and m_3 in any order we like and the constraint system will satisfy all its relations.

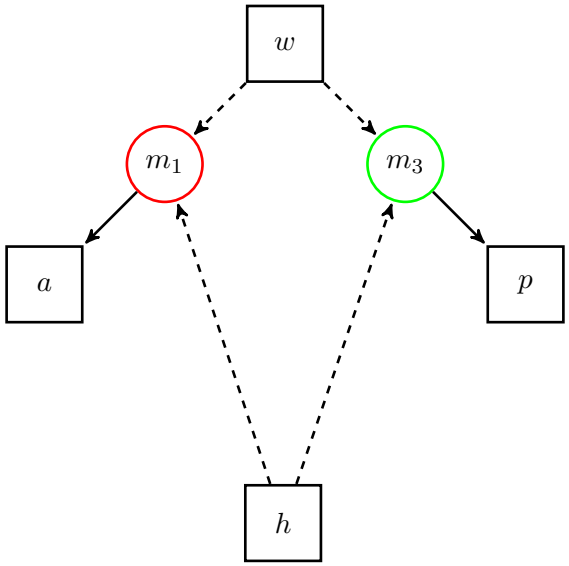

Figure 3.4: Solution graph for the rectangle constraint system using the simple planner.

3.2.2 Hierarchical Planner

Although the basic planner outlined above is capable of satisfying all constraints, it may inadvertently overwrite variables that have just been updated by the user. There is no control over how the constraint system should be satisfied, if there is more than one way to do it. For instance, in the aforementioned example, variables a and p were overwritten, but this is not the only choice. With different choices of free methods, the algorithm could also write to a, and w, a and h, or w, h and p. Ideally, to make sure the variables are updated in an expected manner, we should update variables in the constraint system in a way that avoids overwriting variables recently modified by the user as much as possible.

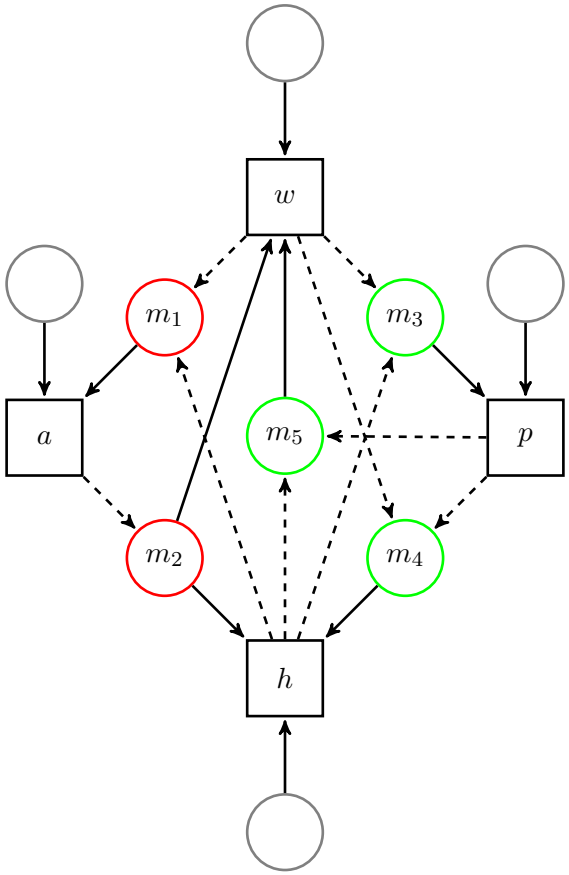

Figure 3.5: Rectangle constraint system with stay constraints.

To this end, a *hierarchical* planning algorithm [16] can be employed to determine the optimal set of methods to be invoked. This algorithm assigns a *strength* value to each variable, which is used to order the variables based on the order they were last updated. Each variable is given a *stay constraint* that is comprised of only one method, which (if selected as part of the solution graph) ensures that the value of the corresponding variable remains unchanged. If the planner selects a stay constraint, it cannot select any other methods that overwrite the value of that variable. The hierarchical palnner favors stronger stay constraints over weaker ones when searching for a plan.

Again, consider the rectangle as an illustrative example. Suppose that the user has updated the variables in a descending order, namely, a, w, h, and p, whereby a was the most recently updated variable and p was the least recently updated. Accordingly, the priority of the variables in terms of retaining their values should be such that a has the highest priority, followed by w, h, and p, respectively. The addition of the stay constraints to the constraint system can be visualized in Figure 3.5, where the gray circles represent the method in each stay constraint. It is noteworthy that the variables which were previously free, are no longer free upon the inclusion of stay constraints in the

graph. Consequently, if we intend to maintain the value of variable a, we need to select the corresponding stay constraint that is associated with a. This, in turn, precludes us from selecting the method m_1 to a valid solution graph, as variable a would then have two incoming edges, which is not permissible. I.e., the stay constraints we select (the values we would like to preserve) directly influences the selection of the remaining methods from the constraints.

To determine which stay constraints we should select, we first start with none of the stay constraints selected, and then add constraints one by one in the order of their strength. If adding a stay constriant results in an invalid solution graph, we skip this stay constraint an move on to the next. We continue this way until a valid solution graph is found, i.e. a graph with one method from each constraint (excluding the stay constraitns that were not included of course). This algorithm will be described in more detail in Section 3.3.

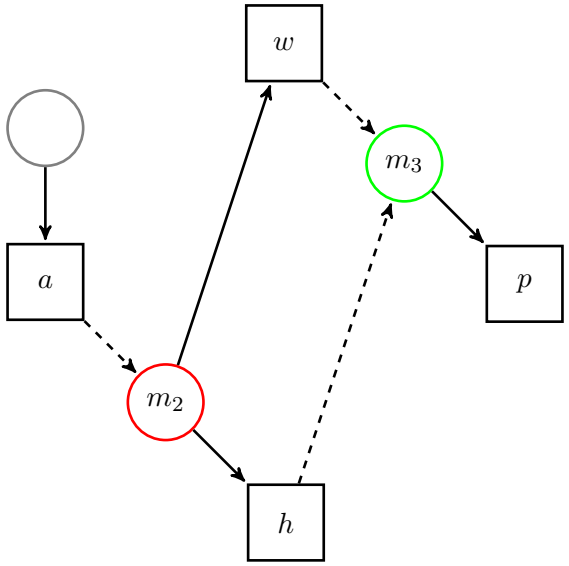

Figure 3.6: Solution graph using hierarchical planner with descending variable strength ordering of a, w, h, and p.

Let us now walk through the process of selecting the appropriate methods using the variable strength mechanism described earlier. We commence by selecting the stay constraint associated with variable a. As a result, we can observe that method m_1 is no longer a viable option for our solution graph. Instead, we choose method m_2 from constraint C_1.

Next, we consider variables w and h, which are also part of our solution graph. Since each of these variables only have one incoming edge, we cannot select their corresponding stay constraints. In the case of constraint C_2, we note that selecting either m_4 or m_5 would

result in an invalid solution graph, as it would introduce an additional incoming edge for either w or h. Consequently, the only viable option is to choose m_3 from constraint C_2. Finally, variable p acquires an incoming edge from m_3, implying that its stay constraint cannot be included in our solution graph. We are now left with a valid solution graph (see Figure 3.6) containing the stay constraint of a, as well as the methods m_2 from C_1 and m_3 from C_1. In order to satisfy the constraint system we need to sort the method vertices topologically in order to ensure that the data flows downstream. Once this is accomplished, we invoke each method in this topological order to satisfy the constraint system.

3.3 Constraint System as a Commutative Monoid

In languages like Haskell, and functional languages in general, designing algorithms is approached quite differently as compared to imperative languages. In functional programming, we tend to think of algorithms as a series of transformations on data, rather than a series of instructions to be executed (recall Section 2.3). As such, concepts like loops do not exist and we rather handle looping through recursion or other techniques. This makes implementing an algorithm like the simple planner less trivial, let alone the hierarchical planner.

Many functional programming languages, including Haskell, has support for algebraic data types. Often we can *prove* that these algebraic data types we define can be algebraic structures which have certain properties we can exploit in order to write simple and concise code.

One example of such an algebraic structure is a *monoid* [10]. A monoid is a set M with a binary operation $\circ$ and an identity element e, such that the following properties hold:

- **Closure**: $\forall a, b \in M : a \circ b \in M$
- **Associativity**: $\forall a, b, c \in M : (a \circ b) \circ c = a \circ (b \circ c)$
- **Identity**: $\exists e \in M : \forall a \in M : a \circ e = e \circ a = a$

A very simple example of a monoid is the set of all non-negative integers as the set M, along the binary operation + (addition) with 0 as its identity element. We can easily observe that the above laws hold for this example.

When studying multiway dataflow constraint systems, we make the observation that they can be expressed as monoids [14]. We define the monoid as follows:

- **Set**: A constraint (which is a list of method graphs, explained below)
- **Binary operation**: The cartesian product of two methods graphs with a custom graph union function
- **Identity element**: A constraint with a list consisting of an empty graph as a single element

As stated above, we can express methods as graphs (called *method graphs*). A method graph is denoted as m^g and contains both its vertex m as well as all its inbound and outbound vertices along with their respective edges. In turn, we can express the graph of a constraint as the graph union of all method graphs in a constraint. Consider the earlier example of the constraint system for the rectangle. The method graph for m_1 consists of the method vertex m_1 itself, as well as both its inbound and outbound vertices, w, h and a (see Figure 3.7 below).

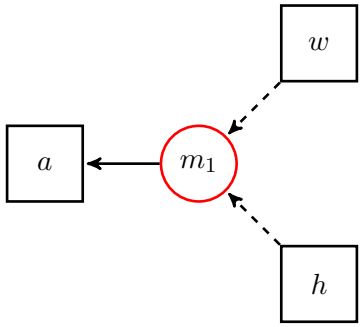

Figure 3.7: Method graph of m_1.

Now, if we consider the constraint C_1, we can express its graph as the graph union of the method graphs of m_1 and m_2, where the method graph of m_2 consists of m_2 itself as well as w, h and a, just like m_1. We then end up with the graph representation of C_1 as can be seen below in Figure 3.8. The same process clearly applies for C_2 as well.

As mentioned previously, a solution graph is a subset of the constraint graph that comprises a method graph from each constraint, optionally excluding stay constraints that may invalidate the solution graph. Furthermore, the solution graph should be acyclic and each variable can have a maximum in-degree of one.

In previous work [14], it is demonstrated that the possible solution graphs of a constraint system can be obtained by folding[1] the cartesian product over the constraint

[1] `https://wiki.haskell.org/index.php?title=Foldr_Foldl_Foldl%27&oldid=62842`

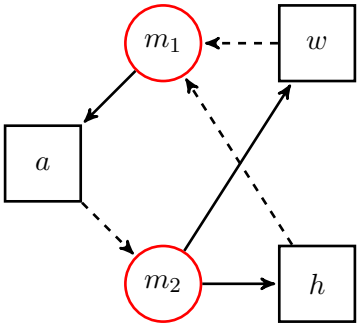

Figure 3.8: Graph representation of C_1.

system, where a constraint is represented as a list of the method graphs within that constraint. Subsequently, any invalid solution graphs are removed from the obtained set [14]. The cartesian product is our binary monoid operation $\circ$. This fold gives us a list of all the possible solution graphs of the constraint system. The function applied on the pairs of method graphs in the cartesian product is a custom graph union operation. We call this operation `methodUnion` [14]. If the union of the two graphs is acyclic and contains no variable vertices with an in degree of more than one, we return the union. If the operation fails we discard it. This function will be adressed in detail in Chapter 5.

When we have computed the possible plans (or rather all possible flows of data), we can take stay constraints into consideration. As discussed earlier, stay constraints are constraints that only include one method graph which only retains the value of its corresponding variable. We use stay constraints as a technique to avoid enforcing a constraint system in a way that could be surprising to the user. Recall that the strength of all variables is maintained in accordance with the sequence in which they were last updated. Although the variables are not explicitly assigned a literal value, a list is maintained to track the variables in descending order based on their most recent update.

To find the best possible solution, that is the plan that does not overwrite the variables most recently edited by the user, we can fold the binary operator over the constraints along with the stay constraints. We first start by including all stay constraints, meaning that we presevere all the variables values. If this does not result in any valid solution graph, we remove the stay constraint associated with least recently updated variable, and perform the fold again. We continue by removing stay constraints in lexicographically descending order, while performing the fold for each iteration until a valid solution graph is found. By removing the stay constraints in lexicographically order, we preserve the variable's values in the best way possible. The fold we perform for each iteration is shown in Listing 3.1.

Listing 3.1: Fold operation.

```
1 foldl (\a b -> if a <> b == Constraint [] then a else a <> b) mempty
```

While this technique certainly works, it is not very efficient. The fold is performed for each combination of stay constraints, which is exponential in the number of variables within a constraint system. Instead of discarding stay constraints one by one in lexicographically descending order, we can precompute the possible solution graphs of the regular constraints and attempt to add stay constraints one by one starting with the highest prioritized stay constraint. If adding a stay constraint results in no valid solution graphs, we simply discard the stay constraint and move on to the next.

Let us revisit the example discussed previously and use the priority of the variabels in the descending order, now denoting each stay constraint as S_{id}, with id representing the variable identifier. We will denote our constraint consisting of possible solution graphs as G_{sol}. We have, in descending order of priority, G_{Sa}, G_{Sw}, G_{Sh}, and G_{Sp}. In sequence we apply the monoid operation on each stay constraint in the list. First, we compute all possible solution graphs including S_a by doing $S_a \circ G_{sol}$. This operation applies the union of S_a to all the possible solution graphs and discards the invalid ones. In this example we would introduce an incoming edge to a resulting in any solution graph containing m_1 to be invalid (see Figure 3.5). Unless the result of $S_a \circ G_{sol}$ results is an empty list (i.e., there are no possible solution graphs containing S_a), we continue with the next stay constraint by doing $S_b \circ S_a \circ G_{sol}$ and so on. If we cannot add the stay constraint to any of the solution graphs, we discard the result and proceed to the next stay constraint by doing $S_b \circ G_{sol}$.

The technique described above highlights how a planner for a multiway dataflow constraint system can be expressed in short and concise code, when leveraging its monoidal properties, which in turn allows the developer to work with a concise implementation, and thus overall improving the developer experience.

Chapter 4

Structure Manipulation in Haskell

As discussed in Section 2.2, WarmDrink is a DSL for manipulating structures in constraint system-based GUIs [28]. In this chapter we introduce our way of manipulating GUI structures in Haskell, called *WDFun*. While WarmDrink supports strucutre manipulation of rooted directed acyclic graphs [28], the work in this thesis is limited to manipulation of lists; extending support for more kinds of structures will be discussed in Chapter 8. Furthermore, our WarmDrink interpretation differs from the original specification in terms of defining, manipulating and enforcing relations between components.

This chapter presents an implementation that expands on our implementation of HDFun. Similar to HDFun, WDFun functions as an engine, rather than a full implementation tied to a GUI. This engine facilitates structure manipulation, while distinguishing itself from the DSL nature of WarmDrink. The co-utilization of WDFun and HDFun permits the user to define *components* containing isolated constraint systems, as well as defining relations between these components. In this process, we take advantage of the graph representation of constraint systems elaborated in Sections 2.1 and 3.2.

We also explain how we have extended the CLI introduced in Section 3.1 to support the new features of WDFun. This will be further elaborated in Section 4.2.

4.1 WDFun

In our implementation of structure manipulation, we have defined a data type that represents components and relations between these. The data type, called `ComponentList`,

consists of a set of *components* and a set of what we call *intercalating constraints*. A component is, as mentioned above, a complete constraint system itself that utilizes the HDFun engine. A component contains its own isolated constraints that we refer to as *local constraints*. An intercalating constraint is defined in the same way as a local constraint, and actually uses the exact same data type and graph representation explained in Section 3.3. However, due to the nature of structures in WarmDrink being acyclic, we can skip the planning step for these constraints.

All components in a `ComponentList` must have the same set of variables. This means that if a variable a is defined in component x, it must also be defined in component y, if x and y are in the same `ComponentList`. The same is true for constraints. If a component x has a constraint C, so must component y, and the converse is also true. The only difference between components is that the values of variables can be different. This may seem restrictive, but it is actually very beneficial as each component is well-formed and eliminates the possibilities for heterogeneity in user-defined components.

An intercalating constraint is, as mentioned above, an instance of the same data type as a local constraint, and is defined in the same way as a constraint in a component. The only difference to a regular constraint that it is not tied to a component, but rather exists on its own in the set of intercalating constraints.

We will consider the example from Section 2.2 regarding the conference agenda. Let us say we want to have a sequence of talks, where each talk has three variables: *start*, *duration* and *end*. Observing the properties of a talk's variables, we can define a relation between these variables, such that a talk's end time is equal to its start time plus its duration. We have the following equation defining the relation between a talk's variables:

$$end = start + duration \tag{4.1}$$

With this relation we can define a constraint C_1 containing one method called m_1. The graph representation of this constraint is shown in Figure 4.1.

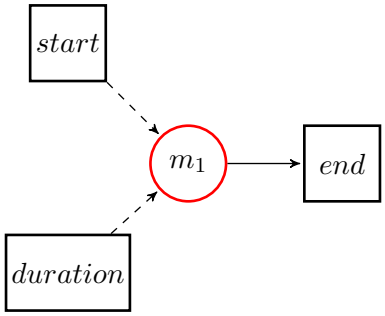

Figure 4.1: Graph representation of constraint C_1 containing one method m_1.

In addition to this, we want to have a small break of five minutes between each talk so every presenter has time to prepare. Thus, for every following talk, the start time is equal to the previous talk's end time plus five minutes. We introduce this relation defined by the equation below using subscripts to refer to components in the list. The unit used here are minutes, but for simplicity we will regular integers to represent minutes.

$$\text{start}_{next} = \text{end}_{prev} + 5 \tag{4.2}$$

The references *next* and *prev* are only used to indicate which component the variable belongs to. The actual constraint containing the relation does not have references to the given components. Because the data between components only propagates in one direction, we know always to read from the preceding component, and write to the succeeding one. The way this is handled is explained in Section 5.2. Using the above relation, we can define an intercalating constraint I_1 containing one method called im_1. The graph representation of this constraint is shown in Figure 4.2.

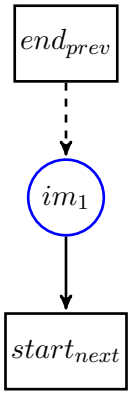

Figure 4.2: Method graph of im_1.

While this constraint exists in its own collection of intercalating constraints, we can visualize how the data would propagate through the entire system. The graph representation of the entire conference agenda, containing three talks, is shown in Figure 4.3.

Every dotted rectangle represents a component in our structure. Each component is a talk in the agenda containing the same set of variables and constraints as every other talk. We can see how the intercalating constraint I_1 propagates data throughout the list of talks.

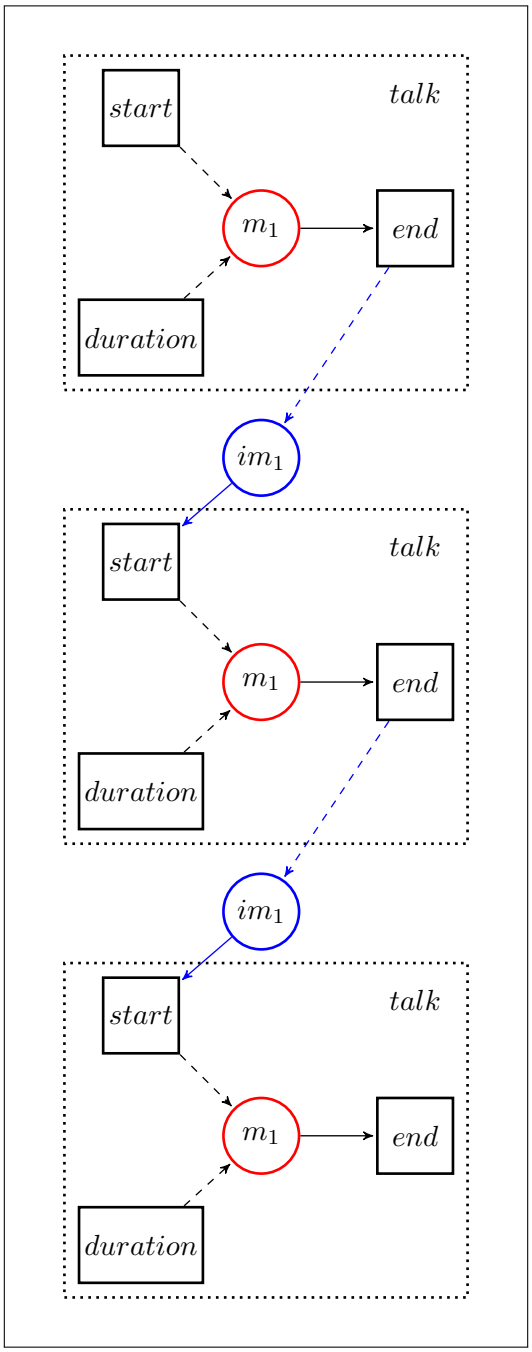

Figure 4.3: Graph representation of the component list.

To give an illustrative example of how the data propagation would work in this example, we can start at talk number one. We will set its start time to 1 and its duration to 10. Invoking m_1 in C_1 would then set the end time of talk number one to 11. Because

we have now changed the values of a component, we need to apply the intercalating constraint I_1 on the next component. I_1 will then read the value of *end* in talk one, and write this value (plus five), to *start* of talk two. The value of *start* of talk two, would then be set to $11 + 5 = 16$. This chain of data would then flow down stream until the last component.

The example in Figure 4.3 gives an example of how data propagates through the list of talks. While the dataflow is correctly visualized, the way the intercalating constraint is applied to the list of components is by traversing accross the list of components and computing the correct values while writing to the desired components' variables. As such, the intercalating constraints do not "exist" between each component as visualized in Figure 4.3. This is explained in Section 5.2.

4.2 User Interface

In order to provide the user with a way to define components and lists of these along with intercalating constraints, we have extended the CLI described in Section 3.1.

4.2.1 Constructing a List of Components Interactively

Figure 4.4 shows the output of the command `help`. This command lists all the available commands in the CLI. The user can define a component with the command `new comp`, which adds a component to the component list. If there are no items in the list, the created component will not contain any variables or constrains. If the list contains one or more components, however, it will duplicate the last component and add it to the end of the list, before enforcing the constraints (both local, and intercalating). The user can also create a list of components using the `new list` command which takes a single integer as an argument for how many components to add.

When the component list has a set of components, the user can define variables and constraints on a component level using the commands `new var` and `new ctrn`, respectively, as described in Section 3.1. These commands add variables and constraints to each component in order to preserve structural equality between all components. After defining a variable, the user can update the actual values on an individual level for each

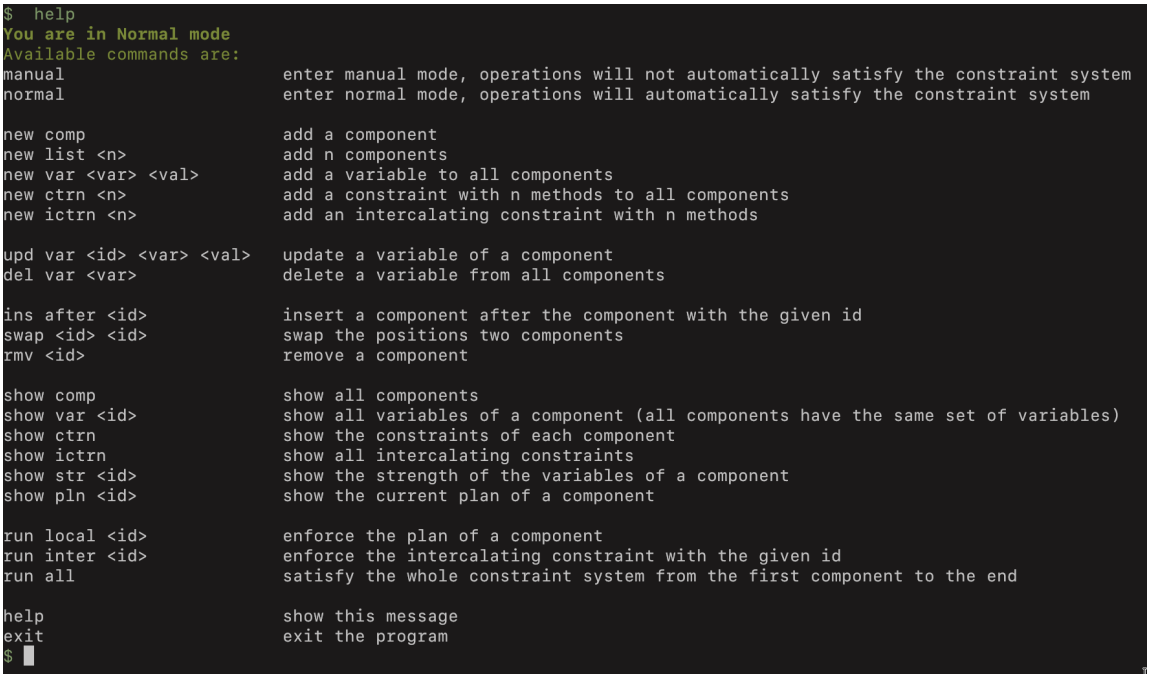

Figure 4.4: WarmDrink CLI.

component by using the command `upd var`, which takes three arguments: the identifier of the component, the variable's identifier, as well as the actual value.

We can define the list of talks used in the conference agenda example with the following commands used in succession:

Listing 4.1: Defining the list of talks in the CLI.

```
$ new list 3
$ new var start 1
$ new var duration 10
$ new var end 1
$ new ctrn 1
```

This will create a list of three components, each containing the variables `start`, `duration` and `end`. We also define the constraint C_1 with one method, which is then applied to each component. This sequence of commands results in the system containing three components (talks), all with the same variables, constraints and values. To define an intercalating constraint, we can use the command `new ictrn`. Similarily to defining a regular constraint, this command also takes an integer as its only argument specifying the number of methods the constraint contains. In this example, we add an intercalating constraint containing one method. The user is prompted for the method name, input variables, output variables as well as expressions for each output variable (see Listing 4.2).

Listing 4.2: Defining an intercalating constraint in the CLI.

```
$ new ictrn 1
Enter name of method:
$ im1
Enter space separated input names to method:
$ end
Enter output variables to method:
$ start
Enter expression for s:
$ end + 5
Parse success
Added intercalating constraint with 1 methods
```

The process of adding an intercalating constraint is exactly the same as adding local constraints, except the constraints are added to the set of intercalating constraints instead. In fact, the command invokes the same set of IO actions described in Section 3.1.1. The implementation of this is explained in Section 5.2.

Once the intercalating constraint has been added, it is applied to the list of components in order to satisfy each local constraint as well as the relations between all the components. We can display the components' values using the command `show comp`. In our case we are left with three components, or rather talks, with the following values.

Listing 4.3: Values of the components after applying the intercalating constraint.

```
$ show comp
Component 0:
start = 1.0
duration = 10.0
end = 11.0

Component 1:
start = 16.0
duration = 10.0
end = 26.0

Component 2:
start = 31.0
duration = 10.0
end = 41.0
```

The example illustrated in Listing 4.3 show the three components we defined, with their respective variables and values. We can verify that the local constraint, C_1, in each component is satisfied as each component's end time is the sum of its start time and its duration. Furthermore, we observe that the start time of component `2` and `3`, respectively, is equal to its preceding component's start time plus five. Hence, our intercalating constraints are satisified across the list of components.

4.2.2 Manipulating the List of Components

While defining components, variables and constraints lays the foundation for constructing a list of arbitrary components, we want to be able to manipulate this list, continually making sure that each constraint (both local, and intercalating) is satisfied. In a list, particularly, we want to be able to both remove and insert elements at any given position, as well as swap any two given elements. With our implementation we can provide this functionality without having to update values manually. This is achieved by invoking the methods of constraints downstream from where values have been modified.

Let us consider an example using the list of talks from Listing 4.3. Suppose we want to insert a talk after the first talk. We can use the command `ins after`. The command takes an integer as its only argument, and it inserts a component into the list after the component with the given identifier. As with the commands used in the example above, the added component will be added along with the same set of variables (with values duplicated from the preceding component) and local constraints as the other components. After inserting this component, we invoke the method in the intercalating constraint from its preceding component, which in turn will invoke all other methods downstream from the preceding component, consequently satisfying all the constraints in the component list.

Listing 4.4: The state of the list after inserting a component and updating its value of *duration* to 50.

```
$ ins after 1
$ show comp
Component 1:
start = 1.0
duration = 10.0
end = 11.0

Component 4:
start = 16.0
duration = 50.0
end = 66.0

Component 2:
start = 71.0
duration = 10.0
end = 81.0

Component 3:
start = 86.0
duration = 10.0
end = 96.0
```

Listing 4.5: The state of the list after swapping components 3 and 4.

```
$ swap 3 4
$ show comp
Component 1:
start = 1.0
duration = 10.0
end = 11.0

Component 3:
start = 16.0
duration = 10.0
end = 26.0

Component 2:
start = 31.0
duration = 10.0
end = 41.0

Component 4:
start = 46.0
duration = 50.0
end = 96.0
```

Moreover, suppose that the newly inserted talk's duration is 50, rather than 10 (the value set when the component was inserted). We can use the command `upd var` with

the component's identifier, the variable's identifier, as well as the value to set. We set the value to 50 like this: `upd var 4 duration 50`.

The `swap` command allows for the swapping of two components. It requires two integer arguments, which correspond to the identifiers of the components that need to be switched. When two components are swapped, it triggers the intercalating constraint. This invocation occurs from the component preceding the first component selected for swapping. However, if the very first component in the list is the one chosen for swapping, the invocation will occur from there.

Chapter 5

Implementation

5.1 HDFun

As previously stated, HotDrink is a JavaScript library that has been specifically developed for the purpose of applying multiway dataflow constraint systems within graphical user interfaces on web-based platforms. HotDrink allows the developer to handle dataflow and constraints between variables in the constraints system. These variables are typically bound to specific DOM-elements. Our implementation introduces a multiway dataflow constraint system engine in Haskell that enables the user to define variables and constraints via interacting with a CLI. The planning algorithms and data structures all take advantage of common patterns in functional programming to allow for an implementation that focuses more on the overall idea, instead of the artefacts of the implementation itself. This chapter explains how we have implemented a HotDrink-engine in Haskell[1], and how we take advantage of functional programming patterns to accomplish this. We also discuss how the CLI is implemented and what techniques are used here. We have also implemented a custom lightweight language to define the expressions contained in methods; we will discuss how this is implemented and what techniques we use to parse these expressions from the CLI.

5.1.1 Graph Representation of Constraint Systems

As explained in Section 2.1.1, we represent any given constraint system as an oriented bipartite graph, denoted as $G = \langle V + M, E \rangle$, where V is the variable vertices, M is

[1]The implementation is found at `https://github.com/boaanes/hdwdfun`

the method vertices, and E is the edges between the vertices. To represent a graph in Haskell, we use the Algebraic Graphs library (`alga`) [21]. This library provides several ways to construct graphs as algebraic data structures. At the core of the library lies the polymorphic type class `Graph g`, which defines four basic polymorphic constructors that all type class instances must satisfy, namely: `empty`, `vertex`, `overlay`, and `connect` [21]. With these operations, we can construct any valid graph. The library comes with several different data types for representing graphs, one of which is `AdjacencyMap`[2]. It is a parameterized data type that allows constructing algebraic graphs and comes with different functions to facilitate different types of graph operations.

Because the vertices in a constraint system's graph can be one of two types (a variable or a method), we have defined a data type that lets us differentiate between these [14]. The data type `VertexType` is defined in Listing 5.1:

Listing 5.1: VertexType data type.

```
1 data VertexType
2   = VertexVar String
3   | VertexMet Method
4   deriving (Eq, Ord, Show)
```

The `VertexVar` constructor only holds a `String` as its parameter, which refers to the name of the variable. This is because we store the actual values themselves in a state, which is discussed in Section 5.1.6. The `VertexMet` constructor holds a parameter called `Method`, which is defined as follows:

```
type Method = (String, [(String, Expr)])
```

The `Method` type is a tuple that holds a `String` and a list of tuples. The `String` refers to the name of the method. In the list of tuples, each tuple connects a method to its output variable. The `String` is the variable identifier and `Expr` is the expression from our own AST discussed in Section 5.1.3. As an example, consider method M_2 from Section 3.3. It reads from a, and writes to w and h. The value of a constant defining this method would then be:

```
m2 = ("M_2", [("w", UnOp "sqrt" (Var a)), ("h", UnOp "sqrt" (Var a))])
```

[2] `https://hackage.haskell.org/package/algebraic-graphs-0.7/docs/Algebra-Graph-AdjacencyMap.html`

To define a method graph, we have implemented another type alias for the `AdjacencyMap` data type from the `alga` library. We call this type alias `MethodGraph` and define it as follows:

```
type MethodGraph = AdjacencyMap VertexType
```

Finally, we can represent methods as method graphs (described in Section 3.3). For example, constraints containing the values corresponding to the method graphs for M_1 and M_2 can be constructed as follows:

Listing 5.2: MethodGraph for M_1 and M_2.

```
1  m1 :: Method
2  m1 = ("m1", [("w", BinOp "*" (Var "w") (Var "h"))])
3
4  m2 :: Method
5  m2 = ("m2", [("w", UnOp "sqrt" (Var "a")), ("h", UnOp "sqrt" (Var
       ↪ "a"))])
6
7  m1Graph :: MethodGraph
8  m1Graph =
9      edges [
10         (VertexVar "w", VertexVar m1),
11         (VertexVar "h", VertexMet m1),
12         (VertexMet m1, VertexVar "a")
13     ]
14
15 m2Graph :: MethodGraph
16 m2Graph =
17     edges [
18         (VertexVar "a", VertexMet m2),
19         (VertexMet m2, VertexVar "w"),
20         (VertexMet m2, VertexVar "h")
21     ]
```

In this example, we use the function `edges` from the `alga` library to construct the graphs. The function takes a list of tuples, where each tuple represents an edge in the graph. The first element of the tuple is the source vertex, and the second element is the target vertex.

Finally, to represent an actual constraint, we have defined a *newtype*[3] `Constraint`, which contains a list of method graphs. A `newtype` in Haskell, is a type declaration that works the same way as a normal data type, except it only has one constructor that only holds one field. We use Haskell's record syntax to wrap the type in a single function `unConstraint`, which allows us to easily access the inner value of the data type. We define a constraint as shown in Listing 5.3. The value of the constraint corresponding to C_1, containing M_1 and M_2 is shown in Listing 5.4.

[3] `https://wiki.haskell.org/Newtype`

Listing 5.3: Constraint data type.

```
newtype Constraint = Constraint { unConstraint :: [MethodGraph] }
```

Listing 5.4: Constraint containing M_1 and M_2.

```
constraintOne :: Constraint
constraintOne = Constraint [m1Graph, m2Graph]
```

5.1.2 Commutative Monoid Representation

Because we represent `Constraint` as a newtype rather than a type alias, Haskell allows us to instantiate type classes. To prove that our constraint data type indeed is a monoid we also have to prove that it is a *semigroup*. A semigroup is the same as a monoid, but without an identity element [10]. We have to provide an implementation of the binary operator `<>`. This can be achieved by instantiating the `Semigroup` type class on our constraint type, as seen in Listing 5.5. As discussed in Section 3.3, the binary operator is a concatenation of the custom union function applied on each pair of the Cartesian product of two constraints, which in turn results in a new constraint. To complete the monoid definition, we have to provide a definition for the monoid identity element, which is a constraint with the empty graph as its only element (see Listing 5.6).

Listing 5.5: Semigroup instantiation.

```
instance Semigroup Constraint where
  Constraint as <> Constraint bs =
    Constraint $ catMaybes [methodUnion a b | a <- as, b <- bs]
```

Listing 5.6: Monoid instantiation.

```
instance Monoid Constraint where
  mempty = Constraint [empty]
```

5.1.3 Method Expressions

In the JavaScript implementation of HotDrink, the variables in the constraint system can be of any type. Methods can be represented by regular JavaScript functions as they are dynamically typed. However, due to the strict type system of Haskell, storing actual functions in the method type is not feasible. While technically possible, doing so would require that each function has the same type, which would consequently limit the implementation's functionality, as every function would need to accept a fixed number of

input arguments of a certain type and return only a specific type of output. Therefore, we have implemented a simple AST, along with an evaluator and a parser (discussed in 5.1.9), which can handle arithmetic and Boolean expressions (see Listing 5.7). This language can be expanded as needed; for this thesis it is considered as a placeholder for a more expressive DSL. The capabilites of this expression language are an orthogonal issue to planning, solving and dynamically managing the structure of constraint systems — which are the main emphases of this thesis.

Listing 5.7: AST data type for method expressions.

```
data Value
  = DoubleVal Double
  | BoolVal Bool
  deriving (Eq, Ord, Show)

data Expr
  = BinOp String Expr Expr
  | UnOp String Expr
  | Var String
  | Lit Value
  deriving (Eq, Ord, Show)
```

5.1.4 Planning

The instance declarations that make the constraint type a monoid give us leverage. By using the properties of the monoid, it becomes possible to implement the algorithm in a succinct way that finds the best plan given the strength of variables. We have implemented a function `plan`, in a module called `Algs.hs`, that takes two arguments: a list of stay constraints in descending order by strength, and a constraint containing all possible solution graphs (obtained by folding the binary operator over the user defined constraints), as shown in Listing 5.8. The code for this planner differs drastically from the fold we introduced in Section 3.3, but conforms with the more efficient algorithm described at the end of Section 3.3. While both this planner and the fold accomplish the same task, this implementation is more efficient. This is because the fold in Section 3.3 will have to be run for each combination of stay constraints, in lexicographically descending order until a valid solution graph is found. Because of this, all (method graphs of) constraints in a given constraint system will be concatenated for each combination of stay constraints. This may lead to bad performance for larger constraint systems. Because of this, we have designed the function in Listing 5.8 to receive an already concatenated list of constraints as its second argument. This function is only called once in order to find the best plan.

In Section 3.3, it is established that the most optimal solution graph can be determined, given the current strengths of the variables, by applying the monoid operation

Listing 5.8: Plan function.

```
plan :: [Constraint] -> Constraint -> Constraint
plan [] c = c
plan (x:xs) c =
    let cAndX@(Constraint cx) = x <> c
    in if null cx
        then plan xs c
        else plan xs cAndX
```

to each stay constraint in descending order of strength. The `plan` function (Listing 5.8) conducts this operation recursively according to the order of the list that contain the constraints. If applying the binary operator results in an empty list, a recursive call is made with the solution graphs computed in the preceding function call, and discarding the current stay constraint from the solution graph. The returned value is either a constraint containing a valid solution graph, or an empty constraint (in which no valid plan is found).

5.1.5 Extracting the Methods

While the solution graph contains both vertices for variables and methods along with their respective edges, we are only really interested in the methods and in which order they must be executed. The method vertices themselves contain the expressions which need to be invoked, along with the variables they write to. As mentioned in Section 3.2 the methods contained in the solution graph need to be invoked in such an order that data flows downstream. This can be achieved by performing a topoloical sort of the solution graph and filtering out the variable vertices. To achieve this, we use a function called `methodsToEnforce`, which takes a constraint as input and returns a value of the type `Maybe [VertexType]`. `Maybe` is a Haskell way to handle null-like values[4]. In Listing 5.9, we perform a pattern matching[5] on the input constraint. If the constraint does not contain any method graphs, we return `Nothing` as there are no valid solution graphs found. If it contains a method graph, we sort the graph topologically using a function `topSort` from Alga [22]. This function returns a value of the type `Either (Cycle a) [a]`, which means we either get a cycle, meaning the graph cannot be sorted topologically, or we get a list of vertices in topologial order.

In instances where a cycle occurs, the function returns the value `Nothing`. However, the presence of cycles is impossible, given that the planner is incapable of computing

[4] https://wiki.Haskell.org/index.php?title=Maybe&oldid=64585
[5] https://en.wikibooks.org/w/index.php?title=Haskell/Pattern_matching&oldid=4276286

Listing 5.9: Function to find methods to enforce.

```
methodsToEnforce :: Constraint -> Maybe [VertexType]
methodsToEnforce (Constraint [x]) = case topSort x of
    Right es -> Just $ filter (\case VertexVar _ -> False; _ -> True)
      ↪ es
    Left _   -> Nothing
methodsToEnforce _ = Nothing
```

a plan containing cycles, as no valid solution graph with cycles exists. Therefore, this serves as a safeguard to ensure exhaustive pattern matching. Upon obtaining the list of vertices organized in topological order, variable vertices are filtered out, as the focus lies on extracting the methods in order to invoke them.

5.1.6 Command Line Interface and State Management

While the data structures, the planner, and other algorithms make up the engine for multiway dataflow constraint systems we need a way for an end user to interact with the system itself. To enable this process, we have developed a command line interface (as described in Section 3.1) to allow the user to define variables and constraints. The values of the variables are stored in a state using Haskell's state monad[6]. While mutability is not directly possible in Haskell, the state monad allows us to emulate state by consuming some current state and producing some new state. To combine the state monad with input/output (IO) operations, we use the monad transformer[7] `StateT`. The `StateT` functionality permits the execution of operations on a user-defined state in a sequential manner, while at the same time having the ability to *lift* out of this monad and perform actions in another monad.

The state of our constraint system consists of a set of variables, a set of constraints, and a strength assignment to each variable. The methods are contained within each constraint themselves. Listing 5.10 shows how we define the data type of our state using Haskell record syntax.

Listing 5.10: State data type.

```
data ConstraintSystem
  = ConstraintSystem
      { variables   :: Map String (Maybe Value)
      , constraints :: [Constraint]
      , strength    :: [String]
      }
```

[6] https://wiki.Haskell.org/index.php?title=State_Monad&oldid=62675
[7] https://en.wikibooks.org/w/index.php?title=Haskell/Monad_transformers&oldid=4055108

Using the `ConstraintSystem` datastrucutre, we construct a state monad transformer of the type `StateT ConstraintSystem IO ()`. This enables us to modify the state of a constraint system while letting the user interact with our program. As can be seen in Listing 5.10, variables are stored in a `Map`[8] with variable identifiers as keys and nullable doubles as values.

5.1.7 User Input

The user input is handled by a recursive looping function called `userInputLoop` (see Listing 5.11), which is of type `StateT ConstraintSystem IO ()`. This function prompts the user for a command, using a custom prompt function (see Listing 5.12), and then processes this command in a function called `processInput`. This is the CLI function that handles most of the state management.

Listing 5.11: User input loop.

```
1 userInputLoop :: StateT ConstraintSystem IO ()
2 userInputLoop = do
3     input <- liftIO prompt
4     processInput input
5     unless (input == "exit") userInputLoop
```

Listing 5.12: Prompt function.

```
1 prompt :: IO String
2 prompt = do
3     putStr "\ESC[32m$ "
4     hFlush stdout
5     input <- getLine
6     putStr "\ESC[0m"
7     return input
```

The above-mentioned prompt function displays a green dollar sign to the user, flushes the standard output, and subsequently reads a line from standard input. ANSI escape codes are employed to achieve the green dollar sign [31]. Utilizing the `hFlush` function ensures that the standard output buffer is flushed, thereby guaranteeing the prompt's display prior to reading user input. Meanwhile, the `getLine` function is responsible for reading a line from standard input, while the ANSI escape code `\ESC[0m` resets the terminal's color to its default setting. As a result, the prompt function returns the user's input in the form of a string. This function is employed not only within the user input loop function (Listing 5.11), but also throughout the CLI, as specific commands necessitate additional prompts.

[8] `https://hackage.Haskell.org/package/containers-0.4.0.0/docs/Data-Map.html`

5.1.8 Processing User Input

As mentioned in Section 5.1.7, the looping user input function (Listing 5.11) calls a function `processInput` with the user's command. This function splits the input string into words and pattern matches the input using Haskells `case of` syntax [20]. In the example below (Listing 5.13), we can see how the function is defined, and how we can add varialbes to the constraint system state. The function pattern matches the keywords "new var" along with two arguments, namely `var` and `val` and inserts the variable `var` with the value `val` into the `variables` map mentioned above in Section 5.1.6. We use the function `modify` to update the constraint system. `modify` takes a function of the type `s -> s` that receives the current state as an argument and outputs the new state. In the example below (Listing 5.13), we insert a variable with the key `var` that points to the value `val`. Furthermore, we also prepend the variable identifier to the list of strengths. Finally a message is displayed to the user indicating the variable insertion.

Listing 5.13: Processing user input.

```
1  processInput :: String -> StateT ConstraintSystem IO ()
2  processInput input = do
3    case words input of
4      ["new", "var", var, val] -> do
5        case parseValue val of
6          Nothing -> liftIO $ putStrLn "Couldnt parse the value"
7          Just v -> do
8            modify $ \s -> s { variables = Map.insert var (Just v)
                 ↪ (variables s) }
9            modify $ \s -> s { strength = var : strength s }
10           liftIO $ putStrLn $ "Added variable: " ++ var ++ " = " ++ val
11 ...
```

Updating a variable in the constraint system is much like adding one, in fact we can use the `insert` function exaclty like in the example of adding one as it replaces the exisitng entry. Instead of prepending the variable to the list of strength, we rather move the corresponding variable identifier to the front of the list.

In the process of adding and updating variables, the implementation is generally straightforward. However, when it comes to introducing constraints, certain challenges arise. First, it is necessary to determine the number of methods involved in the constraint. Additionally, each method may have varying inputs. It is also important to note that a single method can write to multiple output variables, with different expressions associated with each output.

To add a constriant to the constraint system the user inputs the command "new ctrn" followed by a positive integer n, the number of methods involved in the constraint

(as explained in Section 3.1). We then proceed by adding an empty constraint to the constraint system by prepending the value `Constraint []` to `constraints`. Continuing, we traverse[9] a function `inputMethod` n times to add each method. `inputMethod` is a function of the type `StateT ConstraintSystem IO ()` that prompts the user name of the method, along with its input and output variables. Furthermore we parse an expression for each output varible by traversing a function `inputExpr`, which we will discuss in Section 5.1.9. For each method we construct a method graph using a function `methodToGraph` and add the method graph to the empty constraint that we initally added to the constraint system. See Listing 5.14 and Listing 5.15 for the implementation for `inputMethod` and `methodToGraph`, respectively.

Listing 5.14: Input method.

```
1  inputMethod :: StateT ConstraintSystem IO ()
2  inputMethod = do
3      liftIO $ putStrLn "Enter name of method:"
4      name <- liftIO prompt
5      liftIO $ putStrLn "Enter space separated input names to method:"
6      inputsStr <- liftIO prompt
7      liftIO $ putStrLn "Enter output variables to method:"
8      outputsStr <- liftIO prompt
9      let inputs = words inputsStr
10         outputs = words outputsStr
11     exprs <- liftIO $ traverse inputExpr outputs
12     let method = (name, exprs)
13         methodGraph = methodToGraph inputs method
14     modify $ \s -> s { constraints = Constraint (methodGraph :
       ↪ unConstraint (head $ constraints s)) : drop 1 (constraints
       ↪ s) }
```

In the function in Listing 5.14, the user is prompted to provide the essential inputs to a method. Subsequently, the obtained input is utilized to construct a `Method` by parsing expressions, followed by the creation of a `MethodGraph`, which is then integrated into the constraint system. This incorporation is carried out using the previously explained `modify` function. The supplied function operates by obtaining the current state of the constraints, discarding the first element (which corresponds to the constraint being modified or to which a method is being added), accessing the inner value through the `unConstraint` function, and subsequently appending the newly created method graph to this value (list of method graphs). Finally, the constraint is put back into the `Constraint` type and reintroduced to the list of constraints.

To construct a method graph based on the user's input in the previous step, we use the function `methodToGraph`. This function takes two arguments, namely a list of input variables and a `Method`. The method is a tuple consisting of a name and a list of expressions, as explained in Subsection 5.1.1. The function first creates a list of vertices

[9] `https://en.wikibooks.org/w/index.php?title=Haskell/Traversable&oldid=3850129`

Listing 5.15: Function for converting a method to a graph.

```
methodToGraph :: [String] -> Method -> MethodGraph
methodToGraph inputs method =
    let inputVertices = map VertexVar inputs
        methodVertex = VertexMet method
        inputEdges = map (, methodVertex) inputVertices
        outputEdges = map ((methodVertex,) . VertexVar . fst) (snd
            ↪ method)
    in overlay (edges inputEdges) (edges outputEdges)
```

corresponding to the input variables using the `VertexVar` constructor, followed by a vertex corresponding to the method, using the `VertexMet` constructor. Subsequently, edges are created between the input vertices and the method vertex, and between the method vertex and the output variables. Finally, the edges are combined into a single graph using the `overlay` function from alga.

In order to satisfy the constraint sytem, we use the function `satisfy` (see Listing 5.16), which will retrieve the current strengths and the constraints from the state, compute a plan, and invoke the methods of the computed solution graph. `computePlan` is a function which maps strengths to stay constraints, and returns the method vertices of the methods to enforce. The function `enforceMethods` will then take a list of tuples containing variable identifiers and expressions, evaluate these and modify the state with the evaluated expressions.

Listing 5.16: Enforcing constraints.

```
satisfy :: StateT ConstraintSystem IO ()
satisfy = do
    st <- gets strength
    cs <- gets constraints
    maybe (putLnIO "No plan found") (enforceMethods .
        ↪ concatExprsInMethodList) (computePlan st cs)

enforceMethods :: [(String, Expr)] -> StateT ConstraintSystem IO ()
enforceMethods = traverse_ (\(name, e) -> do
    vars <- gets variables
    let newVal = eval e vars
    modify $ \s -> s { variables = Map.insert name newVal (variables
        ↪ s) }
    )
```

5.1.9 Parsing Expressions

As discussed in Section 3.1, each method contains an expression for each output variable. These expressions are represented as abstract syntax using the AST data types explained in Subsection 5.1.3. However, the user does not input these expressions as abstract

syntax, hence we need to parse them. As such we have built a parser using a parsing technique called parser combinators, which is a technique that involves combining simple parsers using higher order functions to form more complex parsers [13].

We have utilized two parsing libraries called *megaparsec* [17] and *parser-combinators* [18] to build several parsers and combine these to fit to our abstract syntax tree. Consider Listing 5.7, We can see that we need to be able to parse literal values (booleans, and doubles), as well as expressions which include binary operators, unary operators, variable identifiers, as well as literals which reference values. With the technique of combining parsers, we can build smaller parsers to parse each of these elements on their own. A parser in this context is essentially a constant function that can be supplied as an argument to the function `parse` from megaparsec. `parse` will then run the given parser on the given string.

As an example we can consider the parser for parsing values. This parser first tries to parse the input string as a `DoubleVal` (which in turn will try to parse the value as a double or an integer). If that does not work, it will try to parse the input string as a `BoolVal`. We use Haskell's alternative operator `<|>`, which first attempts to parse the input using the expression on the left. If this is successful, it returns its result. If the left expression fails, it then tries the expression on the right. If both expressions fail, it results in a failed parse, typically yielding a Nothing (in the case of the Maybe monad) or a Left value (in the case of the Either monad), which would encapsulate an error message or detail.

Listing 5.17: Parser for values.

```
1 parseValue :: Parser Value
2 parseValue = try (DoubleVal <$> (parseInt <|> parseDouble)) <|>
     ↪ (BoolVal <$> parseBool)
3   where
4     parseDouble = signed ws $ lexeme ws float
5     parseInt = signed ws $ lexeme ws (fromIntegral @Integer @Double
        ↪ <$> decimal)
6     parseBool = lexeme ws ((True <$ string "true") <|> (False <$
        ↪ string "false"))
```

The function `try`[10] (line 2, Listing 5.17) is a function from megaparsec which will try to parse with the given parser. If it fails, it will backtrack and try the next parser without consuming the input. If we were to only use `<|>` in this example, the parser would consume the input, thus there would be nothing left to parse for the other alternative.

[10] `https://hackage.haskell.org/package/megaparsec-9.3.1/docs/Text-Megaparsec.html#v:try`

Listing 5.18: Parser for factors

```
parseFactor :: Parser Expr
parseFactor = parseParen <|> parseLit <|> try parseKeyword <|> parseVar
  where
    parseParen = between (symbol "(") (symbol ")") parseExpr
    parseLit = Lit <$> parseValue
```

To parse factors in our AST we can introduce a couple of more parsers and combine these with the parser for values shown in Listing 5.17. The parser shown in Listing 5.18 combines parsers for parentheses, literal values, keywords, and variable identifiers. The parser for literal values is just a wrapper around our parser from Listing 5.17, and the parser for parentheses just consumes the symbols for parentheses and parses the middle content. `parseKeyword` parses keywords for unary operators and booleans, while `parseExpr` is the top-most parser, that combines all the other parsers.

Because different operators and keywords (or functions), can have different order of precendece, we need to be able to parse expressions in a way that respects this. For example, we want to parse the expression `1 + 2 * 3` as `1 + (2 * 3)`, and not `(1 + 2) * 3`. The megaparsec library allows us to specify parsers that decides this precedence using a table as input. As such we have defined the parser for expressions as shown in Listing 5.19.

Listing 5.19: Parser for expressions

```
parseExpr :: Parser Expr
parseExpr = makeExprParser parseFactor operatorTable
  where
    operatorTable =
        [ [ InfixL (BinOp <$> symbol "*")
          , InfixL (BinOp <$> symbol "/")
          ]
        , [ InfixL (BinOp <$> symbol "+")
          , InfixL (BinOp <$> symbol "-")
          ]
        , [ InfixL (BinOp <$> symbol "==")
          , InfixL (BinOp <$> symbol "!=")
          ]
        ]
```

In Listing 5.19, we can see how the function `makeExprParser`, allows us to specify a list of lists of operators, where each list represents a level of precedence. The first list in the list of lists will have the highest precedence, and the last list will have the lowest precedence. In each list, we can specify the associativity of the operators, as well as the parser for the operator. In this case we have specified that the operators `*` and `/` have the highest precedence, followed by `+` and `-`, and lastly `==` and `!=`. We have also specified that all operators are left associative (by using `InfixL`), meaning that `1 + 2 + 3` is parsed as `(1 + 2) + 3`.

5.1.10 Evaluating Expressions

In Section 5.1.5, we desbribed how we can retrieve the identifiers of the methods we need to invoke to run a given plan, in order to satisfy all constraints. As mentioned in Section 5.1.1, these methods contain the identifiers of variables to write the new values to, along with the respective abstract syntax of our AST to evaluate. We have written a simple evaluator which takes an expression and a variable store as an input and outputs a value of the `Maybe Value` type.

Listing 5.20: Evaluation of expressions.

```
eval ::  Expr -> Data.Map.Map String (Maybe Value) -> Maybe Value
eval (BinOp op e1 e2) env = do
  v1 <- eval e1 env
  v2 <- eval e2 env
  case op of
    "+"  -> liftBinOp (+) v1 v2
    "-"  -> liftBinOp (-) v1 v2
    "*"  -> liftBinOp (*) v1 v2
    "/"  -> liftBinOp (/) v1 v2
    "==" -> liftBoolBinOp (==) v1 v2
    "!=" -> liftBoolBinOp (/=) v1 v2
    _    -> Nothing
eval (UnOp op e) env = do
  v <- eval e env
  case op of
    "!"    -> liftBoolUnOp not v
    "sqrt" -> liftDoubleUnOp sqrt v
    "log"  -> liftDoubleUnOp log v
    _      -> Nothing
eval (Var name) env = do join $ Data.Map.lookup name env
eval (Lit v) _ = Just v
```

The evaluator is shown in Figure 5.20. We pattern match on the different constructors of the `Expr` type, and recursively evaluate the expressions. For binary operators, we use the `liftBinOp` function to lift the binary operator into the `Value` type, and apply it to the two inner values before returning a new `Maybe Value`. If both values passed to the function are of the type `DoubleVal`, we return the result of the binary operation wrapped in `Just`, otherwise we return `Nothing`. The `liftBoolBinOp` function is similar, but for boolean binary operators. Furthermore, the functions `liftDoubleUnOp` and `liftBoolUnOp` are also similar, but for unary operators. For variables, we use the `Data.Map.lookup` function to retrieve the value from the variable store. If the variable is present in the environment, the function returns the value wrapped in `Just`, otherwise it returns `Nothing`. For literals, we simply return the value wrapped in `Just`.

5.2 WDFun

As discussed in Section 2.2.1, a specification of a WarmDrink structure can be represented as a tuple, $\langle S, F, R, T \rangle$ consisting of its GUI structure specification, a set of subroutines, a set of relations on the elements in the structure as well as a set of transformation rules [28]. As highlighted in Chapter 4, our methodology for manipulating structure varies significantly from WarmDrink's specification. In our implementation we can view a WarmDrink specification as an ordered list of components, and a set of constraints which define the relations between components. These constraints, called intercalating constraints, are not directly related, or tied to any components. But are, when needed, traversed accross the ordered list of components, enforcing their methods to ensure all constraints are met.

5.2.1 Data Structure

To extend the functionality of the implementation to support structural manipulation of lists, we first introduce a new data type. We call this data type `ComponentList`. It contains the ordered list of components, along with any intercalating constraints defined by the user.

Listing 5.21: ComponentList data type.

```
1 data ComponentList
2   = ComponentList
3       { components                :: [Component]
4       , intercalatingConstraints  :: [Constraint]
5       }
```

The type `Component` is a version of the type `ConstraintSystem` introduced in Section 5.1.6, extended with an identifier. Hence, all the components in our component list are also their own isolated constraint systems.

Listing 5.22: Component data type.

```
1 data Component
2   = Component
3       { identifier  :: Int
4       , variables   :: Map String (Maybe Value)
5       , constraints :: [Constraint]
6       , strength    :: [String]
7       }
```

As seen in Listing 5.21, an intercalating constraint is a list of elements of the type `Constraint`. This is the same type as the constraints defined for components, however they are more "constrained" in terms of data flow direction. This is becuase in a WarmDrink strucutre, data propagation is acyclic between components.

Listing 5.23: Agenda data structure.

```
c1 :: Constraint
c1 = Constraint [graphOfM1]

i1 :: Constraint
i1 = Constraint [graphOfI1]

talk1 :: Component
talk1 = Component
  { identifier = 1
  , variables = Map.fromList [("start", Just (DoubleVal 1.0)),
    ↪ ("duration", Just (DoubleVal 10.0), ("end", Nothing))]
  , constraints = [c1]
  , strength = ["start", "duration", "end"]
  }

-- talk2 and talk3 are defined similarly to talk1, with different
  ↪ identifiers

agenda :: ComponentList
agenda = ComponentList
  { components = [talk1, talk2, talk3]
  , intercalatingConstraints = [i1]
  }
```

Using the data types defined in Listing 5.21 and 5.22, we can define the agenda of talks from Section 5.2.1. Listing 5.23 shows how the values of the component list of agendas would be represented in code.

5.2.2 Expanding the State and CLI

To expand the functionality of the CLI, we need to expand the state of the program in order to work on a component list, rather than a single constraint system or component. As such we introduce a new state type, which contains the component list, as well as a list of all the methods defined by the user. We use the monad transformer `StateT` again, in combination with `ComponentList` and `IO`, to enable a user to manipulate components. Our IO functions (`processInput`, `inputExpr`, etc.) discussed in Section 5.1, now operate on this state instead and allows the user to define an ordered list of components along with intercalating constraints, as introduced in Section 4.2.

Furthermore, we've incorporated a special *mode* that the user has the option to engage. When this mode is activated, the system won't automatically satisfy the constraints. This

allows for debugging, as well as observing the properties of the component list and the constraint system itself. This mode is a simple data type consisting two data constructors.

```
data Mode = Normal | Manual
```

The CLI contains commands for switching between these modes, namely `normal` and `manual`, as can be seen in Figure 4.4. The function `processInput` introduced in Section 5.1.8 is extended to take a `Mode` as an argument and produce a result of `Mode`. Consider Listing 5.24. Here we can observe how `processInput` now returns a `Mode`, as well as the functionality for switching between these. Notice how when switching to normal mode, we satisfy the constraint system using the function `satisfyInter` to ensure that all constraints are satisfied. This function is discussed in Section 5.2.3.

Listing 5.24: CLI mode.

```
1  processInput :: Mode -> String -> StateT ComponentList IO Mode
2  processInput mode input = do
3      case words input of
4          ["manual"] -> putLnIO "Entering manual mode" >> return Manual
5          ["normal"] -> do
6              comps <- gets components
7              maybe
8                  (putLnIO "Entering normal mode" >> return Normal)
9                  (\c -> satisfyInter ((show . identifier) c) >> putLnIO
                       ↪ "Entering normal mode" >> return Normal)
10                 (safeHead comps)
```

In order to avoid enforcing the constraint system when the user is in manual mode we use the function `unless`, where required. `unless` accepts two parameters: a boolean expression and a monadic action. It executes the monadic action under the condition that the provided boolean expression evaluates to `False`. In other words, the action is carried out *unless* the condition is `True`.

```
monadicFunction :: Mode -> StateT ComponentList IO Mode
monadicFunction mode = do
    -- perform any actions here
    unless (mode == Manual) $ satisfyInter ident
    return mode
```

5.2.3 Enforcing Intercalating Constraints

In Section 5.2.1, we examined the concept of intercalating constraints, which are not inherently tied to any individual component. The challenge is, therefore, to properly traverse our list of components and correctly apply these intercalating constraints to each one. This is achieved through the function `satisfyInter`, as delineated in Listing 5.25.

The `satisfyInter` function, specifically designed to manage this task, cleverly combines two additional functions — `satisfy` and `enforceIntercalatingConstraint`. It accepts a single input, the identifier of a component, and then systematically progresses through each component starting from the given identifier, executing `satisfy >> enforceIntercalatingConstraint` in sequence. The function `satisfy`, which has been modified from its original definition in Listing 5.16, now takes a component as its input. Despite the modification, it performs the same function for the specified component. Its companion function, `enforceIntercalatingConstraint`, mirrors the operational flow of `satisfy`. The primary distinction lies in its role — it applies intercalating constraints rather than local ones. For this function, inbound variable values are pulled directly from the given constraint. These values are then used as the variable store for the evaluation of the methods in the intercalating constraint, which are written to the next component in the list.

Listing 5.25: Satisfying intercalating constraints.

```
1 satisfyInter :: String -> StateT ComponentList IO ()
2 satisfyInter ident = do
3     case readMaybe @Int ident of
4         (Just n) -> do
5             comps <- gets components
6             traverse_ (\c -> satisfy c >>
                  ↪ enforceIntercalatingConstraint (identifier c)) $
                  ↪ dropWhile (\c -> identifier c /= n) comps
7         _       -> putLnIO "Couldnt parse id"
```

When the user manipulates the component list in such an order that the intercalating constraints are no longer satisfied, that is, runs a command that invalidates either local or intercalating constraints, the function `satisfyInter` is called at the correct component in order to enforce the constraints. Thus, if the user updates a local variable of a component, we run `satisfyInter` from this given component, which will first enforce its local constraints, and then the intercalating constraints before traversing downstream until the end of the list. Another example is if the user swaps two components. The function `satisfyInter` must then be run from the preceding component of the first component the user wants to swap.

`satisfyInter` lies at the core of the functionality of our implementation. Given an identifier of a constraint, it will satisfy all constraints downstream from the the given constraint until the end of the list.

Chapter 6

Evaluation

As a part of the course *Programming Languages*, or *INF222*, taught during the spring semester of 2023 at the University of Bergen[1], the students were given a mandatory assignment with the objective of defining a multiway dataflow constraint system using *procedures* in a custom language called *PIPL*. The main objective of the task was to define these procedures correctly in accordance with the example constraint system given with the assignment. The constraint system used in the task consisted of relations between properties of triangles, rectangles, parallelograms and circles.

The first task in the assignment consisted of defining a constraint system for a triangle's properties such as ensuring that the Pythagorean theorem is satisfied, as well as trigonometrical properties and relations between the length of the sides in accordance with perimeter and height. The subsequent task expanded with the students having to define a constraint system for a parallelogram, with its corresponding properties. Finally, the last task involved specifying a constraint system consisting of the properties of a parallelogram, two rectangles, two triangles and two circles. The students were to define constraints locally within each geometric figure, as well as constraints involving relations between the figures. Among these were a relation which specified that one of the rectangles should have the same area as the parallelogram, and that the two triangles and the other rectangle should be a decomposition of the parallelogram. That is, the sum of the areas of the three figures should equal the area of the parallelogram etc.

The assignment was given to the students in order to give them a better understanding of how multiway dataflow constraint systems work, and how constraints can be implemented. During the course, which 153 students were signed up for, our implementation of

[1] `http://www.uib.no/en/course/INF222`

HDFun was used in order to facilitate planning of constraint systems defined by the user. The students were given an introduction into multiway dataflow constraint systems, and they designed their own constraint system, but did not undertake any tasks regarding the algorithms of planning or other business logic of MDCS. Our implementation was used as a black box for the constraint systems the students defined in PIPL.

Using our implementation in this mandatory assignment gave us the opportunity to evaluate our implementation in a real world scenario, as well as contribute to the course's future development.

Chapter 7

Related Work

In this chapter we will discuss different approaches to constraint systems and functional frontend web frameworks.

7.1 Constraint Systems

7.1.1 ConstraintJS

ConstraintJS [25] is a JavaScript library that allows the developer to define one-way dataflow constraint systems [3]. Unlike a multiway dataflow constraint, where two values can change each other depending on which one is updated, a one-way dataflow constraint only has the ability to update variables in a single direction. An example of this can be the sides of a rectangle. We can have a slider which determines the length of one side, and the other side will always be, e.g., two times the length. This can be represented by a simple equation `sideA = 2*sideB`. Here the right side is reevaluated whenever the left side's value gets updated, but not the other way around.

ConstraintJS aims to provide an API that allows the user to specify bindings between elements in the DOM with variables in the constraint system using handlebars-like syntax.

7.1.2 Babelsberg

Babelsberg [6] is an *object constraint langauge* that allows the developer to specify constraints between variables. The langauge itself uses Ruby [7] as its base language and introduces some semantic extensions to support constraints. Consider Figure 7.1 for how we can define a currency conversion between NOK and USD.

```
1 class CurrencyConversion
2   attr_accessor :usd, :nok
3   def initialize
4     @usd = 10.0
5     @nok = 107.59
6     always { @usd == @nok * 10.759 }
7   end
8 end
```

Figure 7.1: Currency conversion in Babelsberg.

To instantiate an instance of the class `CurrencyConversion` we can do what is shown in Figure 7.2. When we set the value of one of the variables of the object to a certain value, the solver will be triggered and the constraint will be satisfed. Notice that the keyword `always` will make sure that the constraint should hold indefinitely [6].

```
1 cc = CurrencyConversion.new
2 cc.usd = 20.0
3 # this will trigger the solver and set cc.nok to 215.18
```

Figure 7.2: Instantiating and using the CurrencyConversion class.

7.1.3 SolidJS

SolidJS is a frontend web framework intended for building graphical user interfaces [1]. It uses a *primitive* known as *signals* for tracking and updating values. To run a side effect when the value of a signal changes, we can use a primitive known as *effects*. By leveraging these two primitives, we can define the currency constraint system we defined for Babelsberg in Section 7.1.2 as shown in Listing 7.1.

We can compare the signals in Listing 7.1 to variables in HotDrink, and the bodies of the `createEffect` functions to the methods in HotDrink.

Listing 7.1: Currency conversion in SolidJS.

```
function CurrencyConversion() {
  const exchangeRate = 10.759;
  const [usd, setUsd] = createSignal(1);
  const [nok, setNok] = createSignal(10.759);

  createEffect(() => {
    setUsd(nok() * exchangeRate)
  });

  createEffect(() => {
    setNok(usd() / exchangeRate)
  });


  return (
    /* display the values here */
  );
}
```

7.2 Functional Web Frameworks

7.2.1 Elm

Elm is a strongly typed purely functional programming language used for frontend web development [5]. It compiles to Javascript that runs on the client side of a website, similar to other web frameworks/libraries like e.g. React[1].

Elm's architecture consists of three parts: the model, the view and update. The model is the datastructure that represents the state of an application. It can be structured in any way the user wants to fit to their needs. The view is Elm's way to convert the state into HTML. The view can be considered one or several functions that takes the Model as input and outputs HTML. Update is Elm's way of handling user interactions. A user may click a button which then triggers the update function that takes the state and an action (the button click) as input and outputs some new state (Model). Because of Elm's purely functional nature, the state of the application can only be updated through the Update-function and Elm applications rely completely on this "loop" of user interaction.

Elm has many similarities with Haskell, like its syntax and type system. However Elm's type system is simpler than Haskell's. While both lanugages' type systems are strong, static and inferred, Elm does not support type classes like Haskell.

[1] `https://react.dev/`

7.2.2 PureScript

Another purely functional frontend web framework is *PureScript* [26]. Like Elm, PureScript is also syntactically similar to Haskell, but it offers more flexibility and functionality than Elm. One of the advantages to PureScript is that it offers interoperability between languages. As a developer you can for instance, set up PureScript to handle all logic and state, and then use JavaScript to write the user interface. Another example is writing everything in some other web framework, but have tests written in PureScript.

In addition to language interoperability, PureScript offers a more complex type system than Elm. While Elm keeps things simple and more constrained, PureScript allows for the usage of type classes, like Haskell does, for instance.

7.2.3 Clojure, ClojureScript and Reagent

Clojure is a general purpose language that is *mostly* functional [12]. Clojure by default uses immutable data structures, but one can also take advatage of mutable state if needed. Hence, Clojure is not *purely* functional, like Elm or PureScript, but still offers exclusive immutability if one desires. While Elm and PureScript are strongly influenced by Haskell, and as such are similar syntactially, Clojure is completely different (it is actually a dialect of Lisp [12]).

As mentioned, Clojure is a general purpose language, and therefore not solely used for frontend web development. *ClojureScript*, however, is a compiler for Clojure which compiles (or transpiles) Clojure code into JavaScript, which can be run in web browsers. To provide an easier way of creating a website using Clojure, *Reagent*[2] offers an interface between ClojureScript and React. This allows the developer to take advantage of React's design pattern, and build their website block by block using small reusable components.

[2] `https://reagent-project.github.io/`

Chapter 8

Conclusion and Future Work

Multiway dataflow constraint systems is a powerful programming model for defining relations between variables, especially in the context of user interfaces. Libraries like HotDrink and DSLs like WarmDrink allow for defining constraint systems and specifications manipulating GUI-structures. HotDrink exists as a library in the JavaScript NPM ecosystem, and while WarmDrink is not directly a library, it generates JavaScript code to use alongside HotDrink. This is especially useful in the context of web development, as JavaScript is the main language supported by most web browsers. However, JavaScript is an imperative langauge, which can often be cumbersome to work with.

In this thesis we have presented a functional implementation of a library for multiway dataflow constraint systems and structure manipulation on lists with relations between elements. This is done entierly in Haskell, a purely declarative functional programming language. We demonstrate that implementing both the HotDrink library and the WarmDrink specification in a purely functional language is valuable. A key reason for this is because we can ensure that the constraints, along with its methods and variables always remain pure and without side effects. Another argument for developing a functional implementation is that we express all the logic, data structures and alogrithms declaratively. This enables the development of constraint systems to focus on features and semantics, as it keeps the focus on the essence of the ideas by expressing the implementation declaratively, instead of the clutter that can be found in an imperative implementation.

In the course *Programming Languages* (INF222) at the University of Bergen, our Haskell-based implementation of multiway dataflow constraint systems was used as a practical teaching tool. Students used our implementation to define procedures and

constraints in the context of geometric figures using PIPL. The exercise helped to deepen their understanding of multiway dataflow constraint systems and provided us with a valuable opportunity to observe our system in use, highlighting both its utility and areas that could benefit from further refinement. This application in a real-world educational setting underlines the potential of a functional approach to developing multiway dataflow constraint systems.

We have identified a range of opportunities for future work. First of all, our implementation covers manpipulating lists. However, expanding the implementation to support other types of structures like trees and grids would be beneficial as these are common structures that are prevalent in GUIs.

In addition to this we have have exploited the monoidal properties of constraints in order to make the implementation concise and easier to work with. This mainly applies to the HDFun part of our implementation. However, discovering a purely algebraic formulation and generic specification for structures of WDFun would certainly be advantageous.

Furthermore, we could use the implementation of our library in order to generate graphical user interfaces for the end-user. A user-study could then be conducted on these as well as being tested using some end-to-end testing framework like Cypress[1]. As for extending the implementation itself, we would like to include other types of constraint system libraries as well, and integrate these with HDFun and WDFun. Another technical extension that would greatly improve the usability of our implmentation, is extending the AST covered in Section 5.1.3.

[1] `https://www.cypress.io/`

Bibliography

[1] The New Solid Docs. Online, 2023.
URL: https://docs.solidjs.com/. [Online; accessed 13.06.2023].

[2] Daniel Berge. Visual specification of multi-way data-flow constraint systems. Master's thesis, The University of Bergen, 2022.
URL: https://hdl.handle.net/11250/3001154.

[3] Borning, Alan and Duisberg, Robert. Constraint-Based Tools for Building User Interfaces. *ACM Trans. Graph.*, 5(4):345–374, oct 1986. ISSN 0730-0301. doi: 10.1145/27623.29354.
URL: https://doi.org/10.1145/27623.29354.

[4] Haskell.org Committee. Haskell documentation, 2023.
URL: https://www.haskell.org/documentation/. [Online; accessed 13.02.2023].

[5] Evan Czaplicki. Elm language, 2021.
URL: https://elm-lang.org/. [Online; accessed 13.06.2023].

[6] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. *Babelsberg: Specifying and solving constraints on object behavior*, volume 81. Universitätsverlag Potsdam, 2014.

[7] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language: Everything You Need to Know.* " O'Reilly Media, Inc.", 2008.

[8] Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating Reactive Programs for Graphical User Interfaces from Multi-Way Dataflow Constraint Systems. *SIGPLAN Not.*, 51(3):121–130, oct 2015. ISSN 0362-1340. doi: 10.1145/2936314.2814207.
URL: https://doi.org/10.1145/2936314.2814207.

[9] John Freeman, Jaakko Järvi, and Gabriel Foust. HotDrink: A library for web user interfaces. *SIGPLAN Not.*, 48(3):80–83, sep 2012. ISSN 0362-1340. doi: 10.1145/2480361.2371413.
URL: https://doi.org/10.1145/2480361.2371413.

[10] Pierre A. Grillet. *Abstract Algebra*. Springer, 2 edition, 2007. Chapter 1, Section 1: Semigroups.

[11] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, mar 1996. ISSN 0164-0925. doi: 10.1145/227699.227700.
URL: https://doi.org/10.1145/227699.227700.

[12] Rich Hickey. Clojure, 2023.
URL: https://clojure.org/. [Online; accessed 13.06.2023].

[13] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998. doi: 10.1017/s0956796898003050.
URL: https://doi.org/10.1017/s0956796898003050.

[14] Jaakko Järvi, Magne Haveraaen, John Freeman, and Mat Marcus. Expressing multi-way data-flow constraint systems as a commutative monoid makes many of their properties obvious. In *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming*, WGP '12, page 25–32, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315760. doi: 10.1145/2364394.2364399.
URL: https://doi.org/10.1145/2364394.2364399.

[15] Jaakko Järvi, Gabriel Foust, and Magne Haveraaen. Specializing planners for hierarchical multi-way dataflow constraint systems. *SIGPLAN Not.*, 50(3):1–10, sep 2014. ISSN 0362-1340. doi: 10.1145/2775053.2658762.
URL: https://doi.org/10.1145/2775053.2658762.

[16] Jaakko Järvi, Gabriel Foust, and Magne Haveraaen. Specializing Planners for Hierarchical Multi-Way Dataflow Constraint Systems. *SIGPLAN Not.*, 50(3):1–10, sep 2014. ISSN 0362-1340. doi: 10.1145/2775053.2658762.
URL: https://doi.org/10.1145/2775053.2658762.

[17] Mark Karpov. Megaparsec: Monadic parser combinators. Hackage, 5 2023.
URL: https://hackage.haskell.org/package/megaparsec. [Online; accessed 06.06.2023].

[18] Mark Karpov and Alex Washburn. Parser-combinators: Lightweight package providing commonly useful parser combinators. Hackage, 2 2021.
URL: https://hackage.haskell.org/package/parser-combinators. Available at https://hackage.haskell.org/package/parser-combinators.

[19] Kai Lin, David Chen, Geoff Dromey, and Chengzheng Sun. Maintaining constraints expressed as formulas in collaborative systems. In *2007 International Conference on*

Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007), pages 318–327, 2007. doi: 10.1109/COLCOM.2007.4553850.

[20] Miran Lipovača. Learn You a Haskell for Great Good!, 2011.
URL: http://learnyouahaskell.com/. [Online; accessed 11.05.2023].

[21] Andrey Mokhov. Algebraic graphs, 2022.
URL: https://hackage.haskell.org/package/algebraic-graphs. [Online; accessed 06.05.2023].

[22] Andrey Mokhov. Algebra-graph-adjacencymap-algorithm: Algebraic graphs and algorithm implementation, topological sort, 2022.
URL: https://hackage.haskell.org/package/algebraic-graphs-0.7/docs/Algebra-Graph-AdjacencyMap-Algorithm.html#v:topSort. [Online; accessed 10.05.2023].

[23] Brad Myers. Why are Human-Computer Interfaces Difficult to Design and Implement? 09 1993.

[24] Brad Myers and Mary Beth Rosson. Survey on User Interface Programming. pages 195–202, 01 1992. doi: 10.1145/142750.142789.

[25] Oney, Stephen. ConstraintJS, 2018.
URL: https://soney.github.io/constraintjs/. [Online; accessed 12.06.2023].

[26] PureScript. Purescript, 2023.
URL: https://www.purescript.org/. [Online; accessed 13.06.2023].

[27] Torjus Fitje Schaathun. Integrating multi-way data-flow constraint systems in spreadsheets. Master's thesis, The University of Bergen, 2022.
URL: https://hdl.handle.net/11250/3001141.

[28] 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. ISSN 2590-1184. doi: https://doi.org/10.1016/j.cola.2022.101175.
URL: https://www.sciencedirect.com/science/article/pii/S2590118422000727.

[29] Rudi Blaha Svartveit. Multithreaded Multiway Constraint Systems with Rust and WebAssembly. Master's thesis, The University of Bergen, 2021.
URL: https://bora.uib.no/bora-xmlui/handle/11250/2770614.

[30] Brad Vander Zanden. An Incremental Algorithm for Satisfying Hierarchies of Multiway Dataflow Constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, jan 1996. ISSN 0164-0925. doi: 10.1145/225540.225543.
URL: https://doi.org/10.1145/225540.225543.

[31] Wikipedia contributors. ANSI escape code — Wikipedia, the free encyclopedia, 2023.
URL: https://en.wikipedia.org/w/index.php?title=ANSI_escape_code&oldid=1153083102. [Online; accessed 10.05.2023].