University of Bergen
Department of Informatics

Multiway Dataflow Constraint System and UI Programming

Author: Mathias Skallerud Jacobsen
Supervisors: Mikhail Barash and Jaakko Järvi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2022

Contents

1 Introduction **1**

2 Background **2**

2.1 Svelte 2

2.2 Constraint systems 2

2.3 HotDrink 3

3 Implementation **5**

3.1 Early implementation 5

3.2 ThreeJS 6

3.3 Posten 7

3.4 Current state of HotDrink 8

List of Acronyms and Abbreviations **10**

Bibliography **11**

List of Figures

List of Tables

3.1 Pricing model for domestic packages from Posten 8

Listings

2.1 Example of the HotDrink Domain Specific Language (DSL) 3
2.2 Function for binding HotDrink and Svelte variable 3
2.3 Using the reactive statement, in Svelte [2], to update HotDrink coresopnding value, and trigger HotDrink to enforce the constraint system . . . 3
2.4 Using the onMount callback, in Svelte [2], to update the frontend value that correspont to the same value in HotDrink, when the HotDrink value changes. 4
3.1 Example of the constraint calculating the values of the box 5
3.2 Example of a SveltThree setup 6
3.3 HotDrink logic for determining the price 7
3.4 Example of how to use the HotDrink Application Programming Interface (API) to simulate the corranltion between fahrenheit and celsius 9

Chapter 1

Introduction

This report is part of my INF319 project at the University of Bergen (UiB). The goal of the project was to understand event-based Graphical User Interface (GUI) programming and the limitation to such programming. Specify dataflow constraints and understand how they connect to GUI widgets. Understand the possibilities and limitations of constraint systems based on GUIs. The multiway dataflow constraint system used in the development of this project is HotDrink. Before developing this project I had no prior experience with HotDrink.
TODO: Skriv hva rapporten inneholder.

Chapter 2

Background

In this chapter…

2.1 Svelte

Svelte is a JavaScript framework used for building user interfaces, just like React, Angular, Vue. Where some of these frameworks bulk there work in the browser, Svelte shifts that work onto a compile step [3]. Svelte is written using TypeScript. Instead of using virtual Document Object Model (DOM) svelte uses build time to covert the code into JavaScript [2].

2.2 Constraint systems

A constraint system can be seen as a tuple $\langle V, C \rangle$, where V is a set of *variables* and C a set of *constraints*. Each variable in V has a associated value of a given type (string, integer, boolean, object, etc.). Each constraint in C is a tuple $\langle R, r, M \rangle$. The variables involved in the constraint is given by $R \subseteq V$, r is some n-ary relation among variables in R, where $n = |R|$. M is a set of non-empty set of *constraint system methods*. Executing any method m in M enforces the constraint by computing values for some subset og R, using another disjoint subset of R as inputs, such that the relation R is satisfied [7].

2.3 HotDrink

In this project I used the multiway dataflow constraint system library HotDrink [6], witch is a JavaScript-based library for multiway dataflow constraint systems in GUIs. Instead of writing explicit event handlers, the programmer writes declarative specification of data dependencies, from which the library derives the GUI behavior. This library features a DSL for defining constraint systems. The DSL allows one to specify *components*, *constraints*, *methods* and *variables*. A component in HotDrink holds a set of constraints and variables, as described in section 2.2. Variables often depend on each other, in that case it gets into a setting of multi-way dataflow. The HotDrink DSL is implemented JavaScript tagged template literals [1], which can be seen in Listing 2.1. This lets the programmer integrate the library with frontend JavaScript frameworks such as React[2] and Svelte[3].

Listing 2.1: Example of the HotDrink DSL

```
import { component } from 'hot-drink';

const comp = component`
    var f=1337, c;

    constraint c1 {
        m1(c -> f) => c * (9/5) + 32;
        m2(f -> c) => (f -32) * 5/9;
    }
`;
```

Currently there are no integrated methods in HotDrink to bind the constraint system to frontend frameworks. So the programmer have to decide on the best way to integrate the HotDrink to the frontend web application. Listings 2.2, 2.3, 2.4 shows one way of binding HotDrink with Svelte.

Listing 2.2: Function for binding HotDrink and Svelte variable

```
function setHDValue<T>(HDvariable: Variable<T>, n: T) {
    if (n !== HDvariable.value) {
        HDvariable.set(n);
    };
};
```

Listing 2.3: Using the reactive statement, in Svelte [2], to update HotDrink corresopnding value, and trigger HotDrink to enforce the constraint system

```
$: {
    setHDValue(HotDrinkValue, frontendValue);
}
```

[1] Template literals

[2] For more information about the framework can be found at reactjs.org

[3] For more information about the framework can be found at svelte.dev

Listing 2.4: Using the onMount callback, in Svelte [2], to update the frontend value that correspont to the same value in HotDrink, when the HotDrink value changes.

```
1 onMount(() => {
2     HotDrinkValue.subscribeValue((value: number) => frontendValue = value);
3 });
```

Chapter 3

Implementation

In this chapter…

3.1 Early implementation

In the early stages of implementation the idea for the application was to learn how to use the constraint system library HotDrink in a web application. Thus the application in mind was a simple GUI for a box in 3D. Her we wanted to have few constraints and keep the complexity low.

The box had three dimensions: *depth*, *width* and *height*. From these three values we calculate the value for the *volume* of the box. See `calculateVolume` method in listing 3.1. `calculateHeight`, `calculateDepth`, `calculateWidth` methods calculate the values for `height`, `depth` and `width` individually, whenever the value of `volume` is updated. Then HotDrink chooses the value lowest on the priority list, which is the one that was updated last.

Listing 3.1: Example of the constraint calculating the values of the box

```
1 var width=1, depth=1, height=1, volume;
2
3 constraint metrics {
4     calculateVolume(width, depth, height -> volume) => width * depth *
      ↪ height;
5     calculateHeight(volume, depth, width -> height) => volume / (depth *
      ↪ width);
6     calculateDepth(volume, width, height -> depth) => volume / (width *
      ↪ height);
7     calculateWidth(volume, depth, height -> width) => volume / (depth *
      ↪ height);
8 }
```

HotDrink previously been tested out with React. Thus we decided to test it on Svelte. From listings 2.2, 2.3, 2.4 as seen in section 2.3 we can see how to bind HotDrink to Svelte. This becomes discombobulating when more values are added. Thus when we add one variable in HotDrink we have to make the same corresponding variable in Svelte. Then we have to bind the HotDrink variable to update the Svelte variable when HotDrink enforces constraints (listing 2.4). Use the Svelte reactive statement to listen to changes in the Svelte variable (listing 2.3) and call the `setHDValue` (2.2) on when it is updated.

3.2 ThreeJS

We decided that it would be good for the end user to see the box in 3D. Thus we decided on adding a 3D library to the application. We ended up using SveltThree [4] which is a component library for Svelte for using ThreeJS [5].

SveltThree works by adding *Scene*s to an *canvas* component. A scene could include a *camera*, *lights* and *meshes*. Meshes are used to represent a object in the scene — the box in our case — and the camera where the view of the canvas is placed. The different types of light are there to make the object feel more natural, in the application we use two types of lights: *ambient-* and *directional-light*. The canvas also takes a *WebGLRender* component which is used to add configurations to the renderer.

Listing 3.2: Example of a SveltThree setup

```
<Canvas let:sti w={canvasWidth*.6} h={canvasHeight * .6} interactive>

    <Scene {sti} let:scene id="scene1" props={{ background: 0x00000 } } >

        <PerspectiveCamera {scene} id="cam1" pos={[0, 0, 0]} lookAt={[0,
            ↪ 0, 0]} />
        <AmbientLight {scene} intensity={1.25} />
        <DirectionalLight {scene} pos={[3, 3, 3]} />
        <Mesh
        {scene}
        geometry={cubeGeometry}
        material={cubeMaterial}
        mat={{ roughness: 0.5, metalness: 0.5, color: 0xFF8001, }}
        pos={[-1, 0, 0]}
        rot={[.3, .4, 0]}
        scale={[1, 1, 1]}
        />

        <Mesh
            {scene}
            geometry={sphereGeometry}
            material={sphereMaterial}
            mat={{ roughness: 0.5, metalness: 0.5, color: 0xF6E05E, }}
            pos={[$widthS*(3/4)+ ($depthS/5), 0, 0]}
            rot={[.2, .2, 0]}
            scale={[1, 1, 1]}
```

```
        />

    </Scene>

    <WebGLRenderer
    {sti}
    sceneId="scene1"
    camId="cam1"
    config={{ antialias: true, alpha: true }}
    />

</Canvas>
```

3.3 Posten

The Norwegian post-service packages domesticity and internationally from Norway. A package can in many cases be seen as a box. We decided to make a web application that would give you the price for sending a package with Posten. To still keep the application simple we decided to only support packages sent domesticity. Posten measures the height, depth, width and weight of the package. Thus we had to add a new variable to the constraint system, see listing 3.3, and with it the logic for calculating the price. Posten uses the following pricing model [1][1] for domestic packages:

Listing 3.3: HotDrink logic for determining the price

```
constraint price {
    m1(w, d, h, kg -> price) => {
        if (kg <= 5) {
            if (h <= 35 && w <= 25 && d <=12) {
                return 70;
            } else if (h <= 120 && w <= 60 && d <=60) {
                return 129;
            } else {
                let spesialgodstillegg = 149;
                return 129 + spesialgodstillegg;
            }
        } else if (kg <=10) {
            if (h <= 120 && w <= 60 && d <=60) {
                return 129;
            } else {
                let spesialgodstillegg = 149;
                return 129 + spesialgodstillegg;
            }
        } else if (kg <= 25) {
            if (h <= 120 && w <= 60 && d <=60) {
                return 229;
            } else {
                let spesialgodstillegg = 149;
                return 229 + spesialgodstillegg;
            }
        } else if (kg <= 35) {
            if (h <= 120 && w <= 60 && d <=60) {
```

[1]If any of the measures — height, depth, width — of the package is larger than the maximum allowed there is a special package fee of 149 kr.

```
28                  return 299;
29              } else {
30                  let spesialgodstillegg = 149;
31                  return 299 + spesialgodstillegg;
32              }
33          }
34      }
35  }
```

Table 3.1: Pricing model for domestic packages from Posten

	<35 x 25 x 12 cm	<120 x 60 x 60 cm
0-5 kg	70 kr	129 kr
0-10 kg	129 kr	
10-25 kg	229 kr	
25-35 kg	299 kr	

3.4 Current state of HotDrink

As mentioned in section 2.3 HotDrink is a JavaScript-based library for multiway dataflow constraint systems in GUIs. In the current sate of the library there is no easy documentation for the library. There are two ways writing HotDrink, the developer can either access the HotDrink API seen in listing 3.4 or the developer can use the HotDrink DSL seen in listing 2.1. Both with there own disadvantages.

Using the API can easily lead to errors and is not recommended. On line 9 and 10 of listing 3.4 the developer have to specify the index of the input-variables and the output-variables. In the example there are only two variables — `c` and `f` — thus it is not complicated to keep track of the index for the variables. However, if there are more variables it becomes cumbersome to keep track of all the indexes and there corresponding to the variable.

Using the DSL is the most used way to write HotDrink code. This is also easier to understand and more compact then the API, as seen in the listing 2.1 and 3.4 where both have the *celsius to fahrenheit* example. In this project we used the HotDrink DSL. The use of the tagged template literals, as discussed in section 2.3, makes it error prone to implement the DSL for the developer. This together with the fact that the error messages from HotDrink when using the DSL are not helpful to the developer. There are no way of differentiating between a missing semi-colon or a missing closing bracket. They are generic error stacks.

Listing 3.4: Example of how to use the HotDrink API to simulate the corranltion between fahrenheit and celsius

```
import { Component, Method, ConstraintSpec, MaskNone } from 'hotdrink';

// create a component and emplace some variables
export const Temp = new Component("Temp");
const c = Temp.emplaceVariable("c", 1337);
const f = Temp.emplaceVariable("f");

// create a constraint spec
const toFahrenheit = new Method(2, [0], [1], [MaskNone], (c) => (c * 9 / 5
    ↪ + 32));
const toCelsius = new Method(2, [1], [0], [MaskNone], (f) => (f - 32) * 5
    ↪ / 9);

const graderSpec = new ConstraintSpec([toFahrenheit, toCelsius]);

// emplace a constraint built from the constraint spec
const grader = Temp.emplaceConstraint("grader", graderSpec, [c, f]);
```

List of Acronyms and Abbreviations

API Application Programming Interface.

DOM Document Object Model.

DSL Domain Specific Language.

GUI Graphical User Interface.

UiB University of Bergen.

Bibliography

[1] Norgespakken. https://sending.posten.no/bestill/pakke?_ga=2.21151167.568534240.1651753510-2089638552.1651753510, May 2022.

[2] Svelte api documentation. http://svelte.dev/docs, May 2022.

[3] Svelte webpage. http://svelte.dev/, May 2022.

[4] Svelttthree webpage. https://svelthree.dev, May 2022.

[5] Threejs webpage. https://threejs.org, May 2022.

[6] Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, page 121–130, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336871. doi: 10.1145/2814204.2814207.
URL: https://doi.org/10.1145/2814204.2814207.

[7] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob Smith. Algorithms for user interfaces. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, page 147–156, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584942. doi: 10.1145/1621607.1621630.
URL: https://doi.org/10.1145/1621607.1621630.