

UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

Multithreaded Multiway Constraint Systems with Rust and WebAssembly

Author: Rudi Blaha Svartveit
Supervisor: Jaakko Järvi



UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

August, 2021

Abstract

User interfaces are difficult to get right, and implementing and maintaining them takes up a significant portion of development time. Ensuring that all dependencies between Graphical User Interface (GUI) widgets are maintained, such as the value of one being computed from another, can be challenging and prone to bugs with a standard callback-based approach. The dependency graph formed from relations and constraints between variables quickly becomes unwieldy for humans, especially with multi-directional dataflow and transitive dependencies.

HotDrink is a library for declaratively modeling constraints between widgets as a *constraint system*. This model includes information about how to enforce the constraints, which the library can use to automatically enforce them when values are changed, a process called *solving*. The programmer can thus focus on individual constraints without being distracted by their effect on the rest of the system. Previous implementations of HotDrink have been written in TypeScript and Flow, but they sometimes suffer from poor performance in larger constraint systems.

In this project, we have explored the design space of constraint-based GUI programming for web applications, with a focus on static typing and multithreading. We have developed the library `hotdrink-rs`, a version of HotDrink implemented in Rust. To improve the performance of the *planning* step of solving, we have used an optimization technique called *pruning* that can speed up planning by several orders of magnitude. This enables use of the library for modeling larger systems, and for more performance-sensitive tasks. Our implementation falls short in systems where this optimization is not effective, which suggests that experiments with further optimizations, e.g., *incremental planning algorithms*, should be done. The library also supports multithreaded execution of plans, which both speeds up solving and guarantees GUI responsiveness in the face of long-running computations. The GUI is thus also more resilient to programmer mistakes that cause long-running or non-terminating computations.

We have also developed `hotdrink-wasm`, a library that wraps data structures from `hotdrink-rs` to allow the library to be compiled to WebAssembly. `hotdrink-wasm` supports the use of Web Worker-based threads for multithreaded constraint system solving with cancelable computations in web applications.

Finally, we present more memory-efficient data structures for constraint systems by representing variable indices with individual bits. In addition to saving memory, it may also provide performance benefits by being more cache-friendly.

Acknowledgments

First and foremost, I would like to thank my supervisor, Jaakko Järvi, for his exceptional guidance. The knowledge and feedback he has provided me with in our discussions has been invaluable for my work. I am very glad to have been introduced to the topic of this thesis, and to have had a chance to work more with Rust and WebAssembly.

I would also like to thank Magne Haveraaen and Knut Anders Stokke for the discussions we have had about examples of constraint systems, and about features such as automatic testing of constraint system definitions. Finally, I would like to thank my partner, Marianne Luengo Fuglestad, as well as my family, who have been incredibly supportive throughout my entire degree.

Rudi Blaha Svartveit
August, 2021

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Questions and Expected Results	3
1.3	Areas of Improvement	3
1.3.1	Planner Improvements	4
1.3.2	Multithreaded Solving	4
1.3.3	Minimizing the Memory Footprint	4
1.3.4	Component DSL	4
1.3.5	Generalizing the Core Library	5
1.4	Terminology	5
1.5	Thesis Outline	6
2	Background	7
2.1	Multiway Dataflow Constraint Systems	7
2.2	Planning Algorithms	8
2.3	Related Work	12
3	Implementation of hotdrink-rs	15
3.1	Overview	15
3.1.1	Creating Components	16
3.1.2	Editing Variables	18
3.1.3	Solving	18
3.1.4	Subscribing	19
3.1.5	Creating Constraint Systems	20
3.2	Model Module	20
3.2.1	Constraint System	20
3.2.2	Component	22
3.2.3	Constraint	28
3.2.4	Method	29

3.2.5	Error Propagation	29
3.3	Planner Module	31
3.3.1	Simple Planner	31
3.3.2	Hierarchical Planner	32
3.4	Method Executor Module	33
3.5	Solver Module	34
3.5.1	Activation	35
3.6	Macros Module	39
3.6.1	component!	39
3.6.2	component_type!	40
3.6.3	ret!	42
3.6.4	fail!	42
4	Implementation of hotdrink-wasm	44
4.1	Overview	44
4.2	Heterogeneous Constraint Systems	46
4.2.1	The JsValue Type	46
4.2.2	The Any Trait	47
4.2.3	Rust Enums	48
4.3	Generating a WebAssembly-Compatible Constraint System	50
4.4	Image Scaling Example	51
4.4.1	Defining the Constraint System	52
4.4.2	Wrapping the Constraint System	53
4.4.3	Compilation to WebAssembly	54
4.4.4	Importing WebAssembly from JavaScript	54
4.4.5	Usage from JavaScript	55
4.5	Benefits of Multithreading	57
4.5.1	Parallel Execution	59
4.5.2	Guaranteed Responsiveness	59
4.6	Multithreading with Rust and WebAssembly	60
4.6.1	Limitations	60
4.6.2	Web Worker-Based Threads	60
4.6.3	Web Worker-Based Thread Pools	64
4.6.4	Termination Strategies	65
4.7	Data Flow in a Multithreaded Constraint System	66
4.8	Generating a Multithreaded Constraint System	67
4.9	Pitfalls	69
4.9.1	Use after Move	69

4.9.2	Breaking the Borrowing Rules	69
5	C/C++ Bindings	71
5.1	Creating a Dynamic Library	71
5.2	Creating a C-Compatible Constraint System	71
5.2.1	Construction and Destruction	72
5.2.2	Subscribing	72
5.2.3	Editing	73
5.2.4	Solving	73
5.3	Using the API	74
6	Performance Analysis	75
6.1	Constraint Systems Used in Benchmarks	75
6.1.1	Linear-oneway	75
6.1.2	Linear-twoway	75
6.1.3	Ladder	76
6.1.4	Unprunable	77
6.1.5	Random	77
6.2	Optimization Methodology	79
6.3	Simple Planner Benchmarks	79
6.4	Hierarchical Planner Benchmarks	81
6.5	Solver Benchmarks	82
6.6	Comparison to Other Implementations	84
7	Memory-Efficient Data Structures	88
7.1	Naive Implementation	89
7.2	Representing Method Inputs and Outputs with Individual Bits	90
7.3	Representing Constraint Variables with Individual Bits	91
7.4	Comparison	92
7.5	Drawbacks	94
8	Discussion	98
8.1	Rust	98
8.1.1	Strict Type System	98
8.1.2	Methods With an Arbitrary Number of Arguments	99
8.1.3	Methods With Arbitrary Return Types	99
8.1.4	Variable Access	100
8.1.5	Multithreading in Rust	100
8.2	WebAssembly	100

8.2.1	Multithreading with Web Workers	100
8.2.2	Cancellation	101
8.3	Implementation and Results	102
8.3.1	Features and API	102
8.3.2	Performance	102
8.3.3	Responsiveness	102
8.3.4	Memory-Efficient Data Structures	103
9	Future Work	104
9.1	Planner Optimization	104
9.1.1	Making the Planner Fully Incremental	104
9.1.2	Minimizing Allocation	104
9.1.3	Reusing Variable Reference Counts	105
9.2	Improved Scheduling	105
9.2.1	Breadth-First Scheduling	106
9.2.2	Multi-Method Tasks	107
9.2.3	Deferred Scheduling	108
9.3	Using Procedural Macros for the Component DSL	109
9.4	Undo and Redo in Mutable Constraint Systems	111
9.5	Dynamic Constraint System Construction	111
9.6	Pre- and Postconditions	111
9.7	Enabling and Disabling Components	112
9.8	Improvements to Subscribing from JavaScript	113
10	Conclusion	115
Bibliography		117

List of Figures

1.1	A simple constraint system	2
1.2	A simple constraint system with explicit methods	2
1.3	Image scaling example	3
2.1	Simple planner example.	10
2.2	The constraint graph after adding a stay constraint to a	11
2.3	The solution graph after adding a stay constraint to a and solving the system.	11
3.1	The <code>hotdrink-rs</code> module hierarchy.	21
3.2	Variable generations	26
3.3	Full propagation versus reusing old values	30
3.4	Eager cancellation	38
3.5	Missing Default implementation.	40
4.1	Type conversion in the constraint system wrapper	47
4.2	A simple image scaling example made with <code>hotdrink-wasm</code> and JavaScript.	52
4.3	Scaling an image while preserving its aspect ratio.	57
4.4	Scaling an image without preserving its aspect ratio.	58
4.5	Web Worker message passing.	67
4.6	Data flow	67
4.7	Use after move in Rust.	69
6.1	Linear-oneway.	75
6.2	Linear-twoway.	76
6.3	Ladder.	76
6.4	Unprunable.	77
6.5	A randomly generated constraint system.	78
6.6	Flamegraph of a call to the <code>solve</code> method.	80
6.7	Simple planner performance on different constraint systems.	81
6.8	Hierarchical planner performance on different constraint systems.	82

6.9	Solver comparison.	83
6.10	Time to solve unprunable systems.	87
7.1	Variable lookup	91
7.2	Memory usage per technique.	95
7.3	Comparison of naive strategy and using BitVec in methods.	96
9.1	Flamegraph of the simple planner on the unprunable system.	105
9.2	Chain scheduling	106
9.3	Parallel Directed Acyclic Graph (DAG)	106

List of Tables

3.1	Values per step	35
6.1	Simple planner benchmarks	79
6.2	Hierarchical planner benchmarks	81
6.3	Solver benchmarks	83
6.4	Planner feature comparison between HotDrink-implementations.	85
6.5	Constraint system solve benchmark comparison	86
7.1	Symbols and their meanings.	88
7.2	Sizes of types on a 64-bit architecture.	89
7.3	Relative bits per component	94
7.4	Relative bits per component	95

List of Listings

1	Defining a constraint system wrapper with <code>hotdrink-wasm</code>	45
2	Wrapping a constraint system made with <code>hotdrink-rs</code>	46
3	Using the constraint system from JavaScript.	46
4	Use-after-move error in JavaScript.	69

Chapter 1

Introduction

1.1 Motivation

User interfaces often have a multitude of constraints that must be enforced while the user interacts with them. Buttons may, for instance, be disabled until all text fields have been filled with appropriate values, and some values may be computed from other ones, forming a kind of *dependency graph*. Once a value changes, other values that depend on it must also be recomputed to re-enforce the constraint. As the number of related elements in the GUI increases, it can become difficult for the programmer to ensure that all the constraints are enforced at all times, especially in the following cases:

1. There are transitive constraints: a is needed to compute b , b is needed to compute c .
2. Changes may propagate in multiple directions: changing a changes b , while changing b changes a .
3. There are potential cycles: enforcing a new constraint invalidates one that was already enforced.

The related difficulties lead to a significant number of code defects [24], and the choice of values to modify upon handling a change may be disruptive to the user if implemented poorly [16, p. 177].

This thesis builds on *HotDrink*, a library that uses *multiway dataflow constraint systems* to model GUIs. A *constraint system* is a set of variables and constraints between them; *dataflow* refers to the changes that propagate through the system when values are edited, and *multiway* means that constraint may be enforced in multiple ways, giving multiple possible dataflows. Constraints have a set of associated *methods*, each of which represents one way to enforce the constraint. A more detailed description of multiway dataflow constraint systems can be found in Chapter 2.

Given two direct constraints, e.g., $a = 2b$ and $b = 3c$, there is also an indirect constraint $a = 6c$. The number of indirect constraints can grow exponentially with respect to the variables in the system, and can become difficult to handle explicitly. HotDrink instead lets programmers describe how to satisfy each constraint individually and declaratively. An edit to a single variable will re-enforce the constraints it is bound by, which may trigger other constraints to be re-enforced, until all indirect constraints are satisfied.

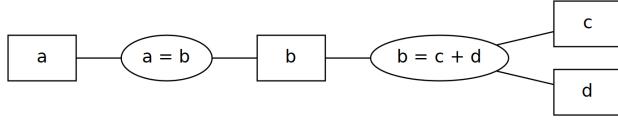


Figure 1.1: A simple constraint system. a , b , c , and d are variables, while $a = b$ and $b = c + d$ are constraints between them. Modifying a would require changing b to maintain $a = b$, and this change to b requires changing c or d to maintain $b = c + d$.

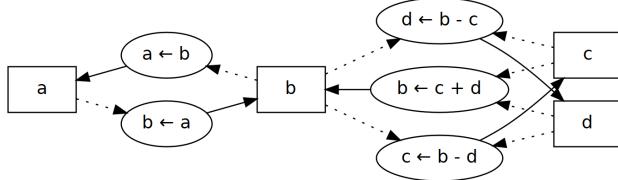


Figure 1.2: A simple constraint system with explicit methods. Dotted lines are *reads* and leave variables unchanged, while solid lines are *writes* and modify variables. There may be multiple possible data flows: A change to a will update b with $a \leftarrow b$, followed by a update of c with $c \leftarrow b - d$, or an update of d with $d \leftarrow b - c$. A change to d would propagate towards a through $b \leftarrow c + d$ and $a \leftarrow b$. Using HotDrink lets the programmer focus on the explicit constraints, instead of having to think about how the implicit constraint between a and d should be enforced.

The library also aims to update values in the least surprising way by preferring to change variables that the user has not interacted with in a while.

A small constraint system can be seen in Figure 1.1, where we represent one constraint between a and b , and another between b , c , and d . For this system to be satisfied, we must enforce both $a = b$ and $b = c + d$ at the same time. Since all variables are connected, a manual implementation would have to be careful to update variables in the right order. A graph with the different ways of enforcing the constraints is seen in Figure 1.2. If we first assign a new value to a with $a \leftarrow b$, and then assign a new value to b with $b \leftarrow c + d$, then there is no guarantee that the first constraint is still enforced.

The simple GUI in Figure 1.3 provides a more practical example of what constraint systems can be used to model. Constraints can be found between the various input fields and sliders; the relative height should, for instance, be equal to $\frac{\text{absolute_height}}{\text{initial_height}}$ at all times, and an additional constraint between the height and width is activated when the “Preserve ratio” checkbox is selected.

As the number of constraints in the system increases, the performance of the *planner* (described further in Section 1.3.1) can become a problem. For instance, 50 constraints can lead to a planning time of 30 milliseconds, and 100 constraints can lead to a planning time of over 100 milliseconds [13]. This can lead to the GUI being unresponsive for a substantial amount of time [40, p. 135]. Taking custom code written by a library user into account, solving a constraint system may become arbitrarily slow, and even cause the GUI to become unresponsive indefinitely.

There are two earlier implementations of HotDrink, written in [TypeScript](#) [13, 14, 12] and [Flow](#) [22, 12] respectively, both of which compile to JavaScript. While JavaScript is a de facto standard for GUI

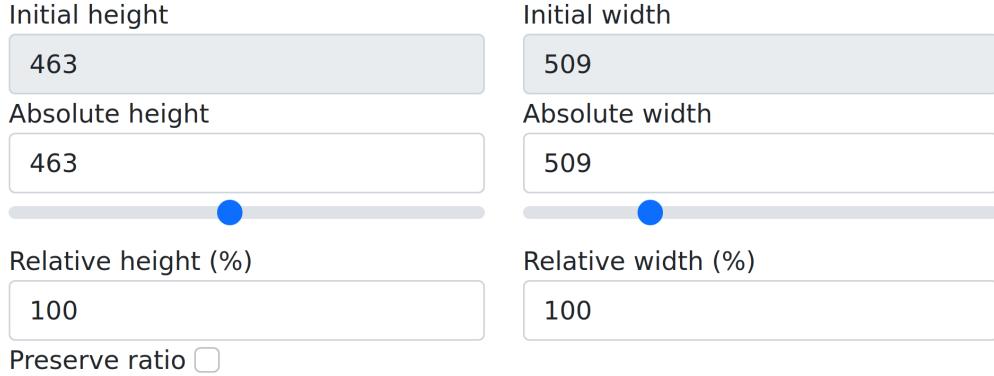


Figure 1.3: An image scaling example.

programming in browsers, it is not the most performant language [19] and is inherently single-threaded [28]. Potential improvements to the library’s performance and responsiveness may be achieved by (1) using a different implementation language, (2) adding new optimizations to the planning algorithm, and (3) by employing multithreading. By running method computations on different threads, we can ensure that the main thread is always available to keep the GUI responsive to user input.

1.2 Research Questions and Expected Results

This paper describes the work on a new implementation of HotDrink written in Rust and WebAssembly, and how rewriting the library has affected its speed, memory efficiency, and responsiveness.

Rust [50] is a relatively young systems programming language with a focus on both performance and safety. It can also be compiled to WebAssembly [81], which is a binary instruction format that can be executed in browsers for near-native performance [81, 82]. The combination of these two traits gives us the opportunity to rewrite libraries in Rust and compile them to WebAssembly for improved performance in web applications. Even compiling existing JavaScript code to WebAssembly can already result in code that is many times faster [49]: we thus hope for similar (or even better) performance improvements for HotDrink. From this, the following research questions arise:

1. How much can Rust and WebAssembly improve HotDrink’s performance and memory efficiency?
2. Can we guarantee GUI responsiveness during solving by employing multithreading?

1.3 Areas of Improvement

The following sections describe the concrete parts of HotDrink that we improved, and how this was done. Additionally, we present an overview of a more memory-efficient representation of constraint systems that can be used to minimize the library’s memory footprint. This representation is not implemented in the current version of the library, but is specified and analyzed in Chapter 7.

1.3.1 Planner Improvements

When values in the constraint system are updated, it is the task of the *planner* to find out how to solve the constraint system. The planner computes a *plan*, which determines the flow of data when the constraint system is solved. In non-trivial constraint systems, planning can take a long time, and may benefit a great deal from being rewritten in Rust to be compiled to WebAssembly. For this, we implemented a variation of the planning algorithm QuickPlan [73].

The planner implementation also involved the creation of the appropriate data structures in the strongly, statically typed language Rust, while keeping as much of the flexibility as possible that writing it in JavaScript gave (such as allowing variables of different types).

1.3.2 Multithreaded Solving

After the planner has found a valid plan, the plan must be executed. Since this involves executing user code, it may take an arbitrarily long time – there could even be defects that cause, say, an infinite loop. By offloading these computations to other threads, we improve the general performance, and also guarantee responsiveness since the main thread is free to handle new user input and update the GUI.

1.3.3 Minimizing the Memory Footprint

Constraints are special instances of graphs; instead of using a general graph data structure to represent them, we can use specialized, more efficient data structures to improve not only memory consumption, but also performance by increasing cache-friendliness. For instance, if we use five of a total of eight variables, we can either have a list of five indices (e.g., 0, 2, 3, 5, 6) which would take up 40 bytes of memory¹, or we can use a single byte and set the corresponding bits (to get 10110110). While not implemented, this optimization could be done for multiple data structures used in HotDrink, such as for method inputs and outputs, and variables involved in constraints. The change would also enable us to use bitwise operations on method inputs and outputs, which could allow further performance optimizations.

1.3.4 Component DSL

The Flow version of HotDrink implements a Domain-Specific Language (DSL) using template literals that enables programmers to define constraint systems more succinctly. An example of a constraint system with a constraint representing $a + b = c$ can be seen below.

```
component C {
    var a, b, c;
    constraint Sum {
        m1(a, b -> c) => ${(a, b) => a + b};
```

¹Assuming we use 64-bit integers.

```

    m2(b, c -> a) => ${(b, c) => b - a};
    m3(a, c -> b) => ${(a, c) => c - a};
}
}

```

This feature has been replicated using Rust macros², which can be used to define DSLs³ embedded in Rust.

1.3.5 Generalizing the Core Library

In order to make the library useful in other contexts than just web programming, we kept the core of the implementation-independent from WebAssembly. This makes it possible to use the library as a normal dependency from Rust, and allows for further extensions to be developed. Supporting more languages and frameworks may help increase adaptation of the library, and streamlining this process for potential users is a major goal.

1.4 Terminology

Throughout this thesis, *the programmer* refers to a user of HotDrink, or to a developer of user interfaces in general. In addition, *the user* will refer to the end-user of software written by the programmer, whether they used HotDrink or not.

Multiway dataflow constraint systems will often be shortened to just *constraint systems*. An *edit* is a change done to the values of a constraint system by the user, while *updates*, *modifications* and *changes* are done by the constraint system itself. For more terminology surrounding *constraint systems*, refer to Chapter 2.

Some words such as *constraint* can have multiple meanings even in the same context. It can be used like in everyday language, in the mathematical sense like in Chapter 2, or like the name of a data structure in Chapter 3. The two first must be deduced from context, but the latter is written with a different font, like `Constraint`, and is often clarified with words such as “struct” or “type”. This applies to all data structures used in `hotdrink-rs`.

In the sections discussing Rust and WebAssembly, all the Rust code is generally compiled to WebAssembly before being executed in a browser. We still write about Rust code snippets as if they were executed normally since they have the same semantics either way. This simplifies the writing and lets us write about what happens when the code at hand is executed, and not the compiled WebAssembly output. The main distinction is then between JavaScript and Rust, with WebAssembly simply being a translation layer.

²<https://doc.rust-lang.org/book/ch19-06-macros.html>

³<https://doc.rust-lang.org/rust-by-example/macros/dsl.html>

1.5 Thesis Outline

An overview of the different chapters may help readers orient themselves while reading through the thesis. The following gives a short description of the content in each chapter.

Chapter 1 An overview of the problem, existing work, and how our implementation improves upon it.

Chapter 2 The mathematical background for constraint systems and property models, an overview of relevant literature and past work, and some useful algorithms. This chapter should give the user a better idea of what a constraint system is, and how it can be (and is) used.

Chapter 3 The implementation and rationale behind `hotdrink-rs`, the core of the Rust implementation of HotDrink. This should give the reader an idea of how the library is used, and the implemented features.

Chapter 4 The implementation and rationale behind `hotdrink-wasm`, a library for generating WebAssembly bindings for constraint systems made with `hotdrink-rs`.

Chapter 5 The implementation and rationale behind `hotdrink-c`, an example of how to generate C bindings for constraint systems made with `hotdrink-rs`.

Chapter 6 Benchmarks for planning and solving in `hotdrink-rs`, and a comparison with the performance of previous implementations of HotDrink.

Chapter 7 A specification of specialized, more memory-efficient data structures for representing constraint systems, and an analysis of their memory consumption compared to naive implementations.

Chapter 8 A discussion about our experiences with using Rust and WebAssembly for implementing HotDrink, as well the features, API, and performance of our implementations.

Chapter 9 Features that were not implemented, either due to time constraints or due to requiring more research.

Chapter 10 A summary of our work.

Chapter 2

Background

This chapter provides a technical description of multiway dataflow constraint systems and some useful algorithms that operate on them. It also reviews relevant literature.

2.1 Multiway Dataflow Constraint Systems

A constraint system can be modeled as a tuple $\langle V, C \rangle$ with *variables* V and *constraints* C [24], where each constraint C is a tuple $\langle R, r, M \rangle$. $R \subseteq V$ is the set of variables involved in the constraint, r is some n -ary relation between variables in R where $n = |R|$, and M is a set of *constraint satisfaction methods*. If the variables in R satisfy r , then the constraint is *satisfied*. Constraint satisfaction methods describe how to satisfy the constraint.

In HotDrink, a *component* is just a self-contained collection of variables and constraints that forms an independent constraint system. Small, reusable components can be defined separately to model GUI elements, and then added to (or removed from) a constraint system that models the entire GUI.

A constraint represents a relation that must be maintained among a subset of the component's variables. It can, for instance, represent an equation such as $E_k = \frac{1}{2}mv^2$. Each variable in the equation corresponds to a variable in the constraint system, and the constraint must be re-enforced when a value is changed.

Constraints have an associated set of *constraint satisfaction methods* — or just methods — that describe the ways that the constraint can be enforced. Each method has a set of *input variables* that it reads from and a set of *output variables* that it writes to. Invoking any of the methods of a constraint will satisfy it. In the example below, methods are generated by solving the equation for different variables. We could, for instance, have one method that changes the value of E_k to match the right-hand side of the equation, written as $E_k \leftarrow \frac{1}{2}mv^2$. Alternatively, we can solve for m in order to get a second method $m \leftarrow \frac{2E_k}{v^2}$, or solve for v to get a third method $v \leftarrow \sqrt{\frac{2E_k}{m}}$. This gives us three different ways to re-enforce the constraint once a variable has been modified.

Instead of a tuple $\langle V, C \rangle$, a constraint system can be viewed as an *undirected bipartite graph* with vertices for each constraint and variable, where the edges are between variables and the constraints they

are involved in. A more detailed directed variant, called a *constraint graph*, replaces the constraints by their associated constraint satisfaction methods. In this graph, the edges are directed edges from method inputs to methods, or from methods to their outputs. The constraint graph view is commonly used by algorithms that operate on constraint systems.

The constraint graph can be analyzed to *solve* the constraint system, that is, to find a valuation for the variables of the constraint system such that all the constraints are satisfied. The first task in solving the system is to identify a *solution graph*, a subgraph of the constraint graph that

1. contains one method from each constraint,
2. is acyclic, and
3. has at most one incoming edge to each variable.

Once we have a solution graph, we can find an order to execute its methods in to create a *plan*. A plan must order the methods such that once a variable has been read from, no other method writes to it. Otherwise, we would risk breaking constraints that were already enforced by an earlier method. This can be summarized as valid plans being topological orderings of the method vertices of the solution graph.

There may be no valid solution graphs, in which case the system is *overconstrained*. This can happen if there are two conflicting constraints such as $a = b$ and $a \neq b$.¹ Conversely, there may be multiple valid solution graphs, in which case the system is *underconstrained*.

Having an underconstrained system is not an issue if we just want *any* solution, but this is often not the case in GUIs. We want to find the solution that is the least surprising to the user. To rank different solutions, we use *hierarchical multiway dataflow constraint systems* that include a priority for each variable. This priority gives us an indication of how important it is to leave the variable unchanged. With these priorities, we select the *lexicographically greatest* solution graph. That is, we select the graph and a plan that avoids writing to the highest priority variables, and use the plan to compute a new valuation.

2.2 Planning Algorithms

To find a valid plan for a constraint system, we require a *planning algorithm*. There are many to choose from, such as *DeltaBlue* [17], *SkyBlue* [54], and *QuickPlan* [73]. *SkyBlue* improves upon the work done on *DeltaBlue*, and *QuickPlan* improves upon *SkyBlue* by guaranteeing a solution in more cases, as well as finding them more quickly. All three solvers are *incremental*; they reuse previous solutions to find new ones faster when the constraint system changes.

Brad Vander Zanden’s paper “An Incremental Algorithm for Satisfying Hierarchies of Multiway Dataflow Constraints” cites concerns about predictability and efficiency as two of the reasons for why one-way

¹The method bodies are not analyzed, so even two compatible constraints such as $3 < x$ and $x < 5$ will make the constraint system overconstrained if they use the same set of variables. In this case, the two constraints should be merged into one.

dataflow constraints often are preferred to multiway dataflow constraints [73, p. 30]. The paper argues that while *constraint hierarchies* can be used to solve the predictability problem, and incremental algorithms can help with the efficiency issue, there were still problems with existing solutions. The first issue was that the existing algorithms could not guarantee an acyclic solution if there are any potential cyclic solutions, and the second was that they require worst-case exponential time in systems with multioutput constraints. The paper describes the implementation of *QuickPlan*, an algorithm that is developed to address these problems.

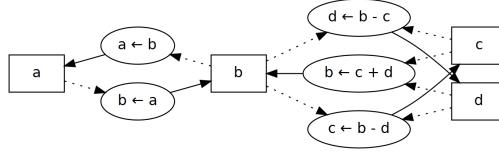
In a multiway, multioutput constraint system with N constraints and at least one acyclic solution, QuickPlan has a worst-case complexity of $O(N^2)$ for satisfying the system, while often finding a solution in $O(N)$ time or less [73, p. 70]. This performance guarantee makes QuickPlan a good candidate for HotDrink, as the guarantees apply to the constraint systems used in HotDrink (hierarchical, multiway, and multioutput). Variations of QuickPlan has been used in the TypeScript version of HotDrink [14], the Flow version [22], and now also our library, `hotdrink-rs`. This gives us the opportunity to compare both the implementation language differences and various augmentations done to the algorithm.

The three algorithms that we will be using are the *multi_output_planner*, *constraint_hierarchy_planner*, and *constraint_hierarchy_solver*. Throughout this thesis, the first will be referred to as the *simple planner*, while the second and third are combined and referred to as the *hierarchical planner*. Zanden also presents modified versions of these algorithms to support incremental solving, which speeds them up by exploiting information from the previous solutions. A summary of the algorithms can be found below, while the full definitions and associated proofs can be found in the original paper [73, p. 36].

Simple planner The simple planner is described as a *propagate-degrees-of-freedom* algorithm, meaning it uses the number of constraints a variable is involved in to select methods. It begins by finding a *free variable*, a variable that is attached to only one constraint and is an output of one of its methods. We can then select this method for enforcing the constraint, and then remove the constraint from the graph. Since the variable is not involved in any other constraints, writing to it will not affect the solutions for the remainder of the graph. The algorithm then repeats the process until (1) all constraints are removed from the graph, or (2) there are no more free variables. In the first case, we have found an acyclic solution and can return it, and in the second there are no acyclic solutions and we can abort [73, p. 40].

Note that in order to support multioutput methods, we must look for a constraint with a method that writes to a set of free variables, and if multiple ones exist we select the one that writes to the fewest number of variables to not limit our choices later.

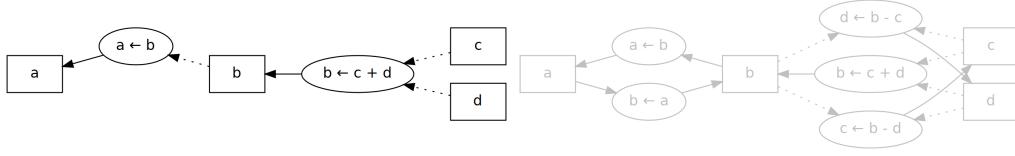
An illustration of the steps of the algorithm can be seen in Figure 2.1. We see that a is free since it is only involved in the constraint between a and b , and is written to by $a \leftarrow b$. The variables c and d are also free since they are only involved in the constraint with b , and are written to by $c \leftarrow b - d$ and $d \leftarrow b - c$ respectively. Which of the free variables (and subsequently, which constraint) is chosen does not affect whether or not a solution is found, but does affect which solution is found. For this example, we choose to enforce the constraint between a and b with the method $a \leftarrow b$. The



(a) The initial constraint graph before selecting any methods.



(b) The solution graph (left) and the remaining constraint graph (right) after selecting $a \leftarrow b$ to enforce the constraint between a and b . The gray part of the constraint graph is the newly enforced constraint and its corresponding methods.



(c) The final solution graph (left) after selecting $b \leftarrow c + d$ to enforce the constraint between b , c and d , and the remaining empty constraint graph (right).

Figure 2.1: Simple planner example.

constraint between a and b and its methods are removed from the constraint graph, and the selected method is added to the solution graph.

Repeating the process for the remaining subgraph, we find that all remaining variables are free: only one constraint remains, and each variable is written to by one of its methods. Selecting any method is thus valid, and we choose $b \leftarrow c + d$. This removes the last constraint from the constraint graph and gives us the final solution graph with the two methods $a \leftarrow b$ and $b \leftarrow c + d$. Sorting the methods of the solution graph topologically gives us the plan $[b \leftarrow c + d, a \leftarrow b]$, which we can execute to enforce all constraints.

Hierarchical planner The hierarchical planner essentially re-runs the simple planner with different additional constraints called *stay constraints*. Each variable can be given a stay constraint, which has just one method. This method's sole output is the variable, and it keeps the value unchanged. If a variable has a stay constraint, the simple planner cannot select other methods that write to the variable.

If the user edits some variable, then the simple planner is free to find a plan that overwrites the change the user just did, which is something we would like to avoid. We therefore try to add different combinations of stay constraints to the constraint graph, and run the simple planner to figure out which combinations work. In particular, adding stay constraints to variables that the user recently modified ensures that they are not overridden.

The variant of QuickPlan that hotdrink-rs uses starts off by adding a stay constraint for the highest priority variable before attempting to solve the system. If the initial solve succeeds, it keeps the constraint in the constraint graph and continues with the second-highest priority variable. If it fails, however, it removes the newly added stay constraint before continuing with the second-highest priority variable. This process is repeated until the algorithm has tried to add a stay constraint for each of the variables in descending priority order. If it has not found a solution by this point, it attempts to solve the system without any stay constraints. Note that the implementation also performs a few additional optimizations that are described in Chapter 3.

For the following example, assume the latest value the user edited was a . The algorithm adds a constraint with a single method $a \leftarrow a$ to the graph, as seen in Figure 2.2.

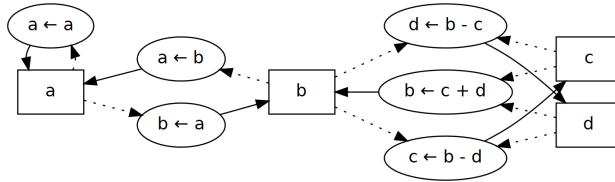


Figure 2.2: The constraint graph after adding a stay constraint to a .

This means that a is not a free variable like it was in Figure 2.1, forcing the algorithm to start with c or d instead. One choice is to write to c by selecting $c \leftarrow b - d$, which makes b a free variable. Following that, the only choice is to write to b by selecting $b \leftarrow a$, after which only the stay constraint remains, where we have to select $a \leftarrow a$ since there are not alternative methods. This gives us the plan $[a \leftarrow a, b \leftarrow a, c \leftarrow b - d]$, or alternatively $[a \leftarrow a, b \leftarrow a, d \leftarrow b - c]$ if we had chosen to write to d instead. One of the possible solutions is shown in Figure 2.3.

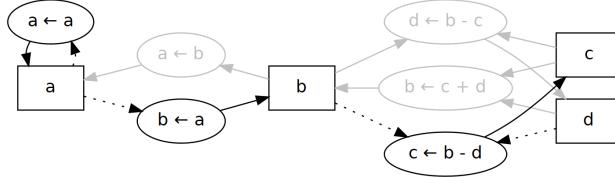


Figure 2.3: The solution graph after adding a stay constraint to a and solving the system.

Note that the method of a stay constraint both reads from and writes to the same variable (and is an identity function). This is not, however, considered a cycle. For planning purposes, we only consider stay constraint methods to write to their associated variable.

Multiway dataflow constraint systems are often underconstrained, giving us many possible solutions. Adding stay constraints guides the algorithm towards our preferred solution, possibly leading to some overconstrained systems along the way, which are dealt with by ignoring some of the stay constraints. At the cost of re-running the simple planner multiple times, we can thus direct the data

flow away from the variables that the user recently modified.

Incremental hierarchical planner The main idea of incremental solving is that adding and removing constraints often only affects a small part of the graph, which means that re-examining all of it is not necessary. One of the *incremental techniques* [73, p. 45] called *The Upstream Constraint Technique* changes what happens upon adding new constraints. Adding new constraints causes the non-incremental version to pass the entire graph to the simple planner again, which takes a significant amount of time when done for each variable that gets a stay constraint. The incremental version instead only has to re-enforce the *upstream* constraints, constraints from which there exists a path to the newly added constraint in the existing solution graph.

Zanden justifies this by looking at the reversed solution graph, which represents what he calls “elimination” dependencies [73, p. 46]: an edge from constraint c_1 to c_2 shows that c_1 was eliminated before c_2 during solving, and that c_2 ’s elimination depended on c_1 ’s elimination. All these dependencies will flow towards the upstream subgraph, meaning that the downstream constraints do not depend on upstream constraints, and are thus not affected by the change.

2.3 Related Work

The research on how to design user interfaces, and how to ensure that they are correct and responsive, stretches back decades [37]. For instance, *Garnet* and the *Gilt Interface Builder* aims to reduce the “spaghetti” caused by relying on too many callbacks in user interfaces [39, 38]. Constraint systems and property models [26] have also been proposed as remedies for these problems of GUI programming. There are many GUI features that constraint systems essentially “give us for free” after a declarative description has been created, such as automatically maintaining relations between variables, disabling of components that are not in use, and *pinning* of variables that a user does not want to change [24, 16, 15].

One major source of complexity in GUI programming is dealing with asynchronous events and orchestrating responses to them in a consistent manner. Several works that fall under the general paradigm of reactive programming target this source of complexity. “Tackling the Awkward Squad for Reactive Programming: the actor-reactor model” [72] specifies a number of issues to look out for in *reactive systems* such as GUIs, such as long-lasting computations, and why they can be a problem. The paper also proposes a new programming model that is used to handle these issues.

“Guaranteeing Responsiveness and Consistency in Dynamic, Asynchronous Graphical User Interfaces” [13] proposes another programming model, and explains the rationale behind the TypeScript implementation of HotDrink. It describes how one can solve a constraint system *asynchronously*, and what effects this has on the usual operations in a constraint system. For instance, the usual values are replaced by *promises* to allow interacting with them before method computations have finished, and methods do not perform work directly but have the computations *scheduled* to run at a later point.

The Gilt Interface Builder is a part of the research project *Garnet* [39], which provides many tools to

simplify GUI development. The *Amulet Environment* [36] extends this work even further. Amulet allows for values of any type to be computed by arbitrary code, and even supports multiple constraint solvers. It also enables the GUI to automatically refresh upon changes to the constraint system, supports ways to undo operations, and includes built-in editing commands such as cut, copy and paste.

There also exists newer work, such as the *Adobe Software Libraries* [2], that aims to develop the technology to create commercial applications with declarative descriptions. Two such technologies are *Adam*, which is a property model library, and *Eve*, a layout library. Adam consists of a solver and a declarative language for describing constraints between variables. Upon making a change to the system, the dependent values will be recalculated, much like in HotDrink, at least on a high level. Eve also has a solver and its own language for describing the layout of user interfaces. With Adam and Eve, both the visual aspects of a GUI and constraints between variables in the underlying property model can be specified declaratively.

HotDrink is not the only JavaScript-compatible library that provides a way to describe constraints declaratively. For instance, ConstraintJS [41] has a pitch similar to HotDrink: “ConstraintJS enables *constraints* – relationships that are declared once & automatically maintained” [7]. It does, however, not provide multiway constraints, as the authors of ConstraintJS appear to see their inclusion as an unnecessary complication [41, p. 237].

In addition to the various libraries and frameworks that use constraint systems, there is also more theoretical research on the properties of the constraint systems themselves. For instance, by modeling dataflow constraint systems as a monoid clarifies their properties, such as how constraint strength assignment affects the solution [25, p. 30], and that “sets of constraints can be composed to new constraints in any combinations and composed in any order with no impact on the final result” [25, p. 31]. Modeling them as a monoid also allows for reuse of all monoid operations, and allows for more concise algorithms, e.g., solving by folding the monoid’s binary operation over the system’s constraints [25, p. 25].

While planning speed can be improved by using a different planning algorithm, it is also possible to create *specialized planners* when the constraints are known in advance [23, 13]. These specialized planners are Deterministic Finite Automaton (DFA)s, and can be more than an order of magnitude faster than the general-purpose planners described in Section 2.2, such as QuickPlan [23, p. 9]. This can be very useful for performance-sensitive applications that have static constraint systems, i.e., constraint systems with non-changing constraints.

There are also theoretical frameworks for working with constraint systems and analyzing their properties. “Semantics of multiway dataflow constraint systems” includes, among other things, a description of how specifications can be added to constraint systems to allow for verification and testing [20]. In the implementations of HotDrink, the relation that a constraint represents is never explicitly shown in code; it is only implicitly shown via the computations performed in methods. Adding the relation explicitly to constraints would allow for automatic testing of the constraint system specification upon solving to ensure that the methods always enforce the constraint. For instance, for a constraint that represents the equality

$a = b + c$, we could have this property explicitly tested after a method is executed. The paper also aims to improve the reusability of components by introducing a module system, one that allows renaming of variables [20, p. 3]. Each component then becomes a building block that can be extended further with additional combinators, such as the logical connectives $\&$, $|$, and \Rightarrow .

Separate but relevant issues, such as how to handle bidirectional control flow, can provide useful results and insights that shape the development of newer libraries such as HotDrink. For instance, one way is to use *algebraic effects* to guarantee that all effects are handled and that none are handled accidentally [89].

In much of the research, two things appear to be constant: GUIs are difficult to get right and they take a significant portion of development time to complete.

Chapter 3

Implementation of hotdrink-rs

While the target of HotDrink has been previously web programming, implementing the core library in Rust makes HotDrink available in other contexts too. For instance, the library could be used as a normal dependency from Rust, compiled to WebAssembly, or even compiled to a library to be used from C. The current chapter will provide examples of how to use the library from Rust, while usage for web programming with JavaScript can be found in Chapter 4. Lastly, an example of how to use the library from C can be found in Chapter 5.

The core library is open-source and available on GitHub [65]. It is also published under the name `hotdrink-rs` at crates.io [63], and its documentation can be found on docs.rs [64]. Since it is published as a Rust crate, using it in a Rust project is as simple as adding it as a dependency.

This chapter discusses some data structures in `hotdrink-rs` that are not exposed in the public Application Programming Interface (API). Documentation for everything, including private items, can be generated by running `cargo doc --document-private-items` from the root directory of the project repository.

3.1 Overview

In order to get an idea of how `hotdrink-rs` can be used, we start with an example concept we want to model as a constraint system. Given a rectangle with a height h , width w , area a , and perimeter p , the equations for its area (Eq. 3.1) and perimeter (Eq. 3.2) must hold at all times. If any of the values are changed, then the others must change accordingly.

$$a = h \cdot w \tag{3.1}$$

$$p = 2h + 2w \tag{3.2}$$

In the following sections, we explain how to model a rectangle with these constraints using `hotdrink-rs`. This includes creating a constraint system, modifying variables, and re-enforcing the constraints, as well as observing variables to be notified of changes to them.

3.1.1 Creating Components

A *component* is a *constraint system fragment*; an independent set of variables and constraints. Components can be used to modularize the constraint system; they can be created, added, and removed separately. Each component contains a set of variables along with their values, as well as constraints with methods to enforce them. This means that a component can be used as an independent constraint system if only one component is needed. One could for instance create one component that represents a rectangle by including the variables and constraints discussed in Section 3.1, and another that represents a separate concept.

Multiple Components can later be combined into a single `ConstraintSystem`, which exposes an API for interacting with all components at once instead of handling them one by one, but for this example, a single component has all the required functionality.

The easiest way to declaratively construct a `Component` is with the `component!` macro, which implements an Embedded Domain-Specific Language (eDSL) for specifying constraint system fragments.

```
let mut rectangle: Component<i32> = component! {
    component Rectangle {

        // Define four variables of type i32 with initial value 0.
        let height: i32 = 0, width: i32 = 0,
            area: i32 = 0, peri: i32 = 0;

        // Define a constraint representing `height * width = area`.
        constraint HeightTimesWidthEqualsArea {
            // Define three ways to enforce it.
            hwa(height: &i32, width: &i32) -> [area] = ret![*height * *width];
            haw(height: &i32, area: &i32) -> [width] = ret![*area / *height];
            wah(width: &i32, area: &i32) -> [height] = ret![*area / *width];
        }

        // Define a constraint representing `2 * height + 2 * width = peri`.
        constraint TwoHeightPlusTwoWidthEqualsPerimeter {
            // Define three ways to enforce it.
            hwp(height: &i32, width: &i32) -> [peri] = ret![2 * *height + 2 * *width];
            hpw(height: &i32, peri: &i32) -> [width] = ret![*peri - 2 * *height];
            wph(width: &i32, peri: &i32) -> [height] = ret![*peri - 2 * *width];
        }
    }
};
```

In the example above, we construct a `Component<i32>`, a component with variables of type `i32`. In this case, all variables must have the same type; how to have constraint systems with variables of different types is described later. The example also shows how to define the variables used by the component, their types, and their initial values. The initial values may be elided, in which case the `Default trait` will be used to generate a value instead.

We then define two constraints, each with three methods made from rearranging their respective equations. For the first constraint, we update the area with $a \leftarrow h \cdot w$, the width with $w \leftarrow \frac{a}{h}$, or the height with $h \leftarrow \frac{a}{w}$.¹ For the second constraint we can rearrange Eq. 3.2 to get the three different methods: we can update the perimeter with $p \leftarrow 2h + 2w$, the width by $w \leftarrow p - 2h$, or the height by $h \leftarrow p - 2w$.

Typing in components and methods

Components can contain values of different types by using sum types, created with the `enum` keyword in Rust. We could for instance create a type that allows values to be integers or strings as follows:

```
// A type with values that can be either `i32` or `String`.
enum IntOrString {
    Int(i32),
    String(String)
}
```

The inputs of a method are typed references, declared in the method's parameter list. Specifying the type in the parameter list makes it possible to convert the values of the constraint system to what the programmer wants automatically, e.g., from `IntOrString` to `i32` or `String` by writing a method `m(a: &i32, b: &String)`. Note that this will give an error if the value does not have the specified type. A limitation of the current implementation is that the outputs are not typed, which means that any value that can be converted to the `Component<T>`'s value type `T` can be returned. The programmers must thus be careful not to change the type of a variable accidentally.

The `ret!` macro is used in method bodies to return a successful result with the outputs stored in a `Vec`, but it will also wrap each value back into the appropriate enum variant (if an enum is used as the component's value type). For instance the two expressions below are equivalent:

```
1. ret![5, String::from("hello")]
2. Ok(vec![IntOrString::Int(5), IntOrString::String(String::from("hello"))])
```

The latter is a lot more cumbersome to type manually. Note that each of the variants (`i32`, `String`) must implement `TryFrom<Component<IntOrString>>` to be converted to the type specified in the method parameter list, and `Into<Component<IntOrString>>` to be converted back to `IntOrString` with the `ret!` macro.

¹If we want to be pedantic, the methods using integer division would not always enforce the constraints since the result is truncated. This, however, is beyond the point of the example.

3.1.2 Editing Variables

We need an API for notifying a component that a variable has a new value, e.g., if a user has modified an input field. This can be done with `edit`, which takes the name of a variable and its new value as arguments. Continuing with the rectangle component defined in Section 3.1.1, the `edit` method can be used as follows:

```
rectangle.edit("height", 3);
rectangle.edit("width", 5);
```

3.1.3 Solving

Simply editing variables in the component will not trigger the enforcement of the constraints. Instead, this can be requested with either the sequential `solve` method, or the parallel variant `par_solve`. This is most commonly done immediately after updating values with `edit` to make the system consistent again. Both variants attempt to find a plan to satisfy the active constraints and then begin its execution.

Finding the plan always happens sequentially, but executing the methods can happen in parallel, as long as the data dependencies are respected. We use parallel execution for two main reasons, both of which help counteract that the user code in method bodies can be arbitrarily slow.

1. Method execution can be sped up with a thread pool.
2. Running methods outside of the main thread guarantees that the GUI remains responsive while methods are being computed (unless planning is the bottleneck).

When these reasons are not a concern, the `solve` method is there to provide a simple way of just solving the system sequentially. The `solve` method can be used as follows:

```
rectangle.solve();
```

If planning is successful, each method is executed on the main thread, and the function call will not be done until all methods have finished.

In order to call the parallel solver function `par_solve`, we must provide a `MethodExecutor` for the methods to be executed by. This trait is explained further in Section 3.4, but for now, it can be viewed as something that can run computations (specifically, functions or closures). `hotdrink-rs` comes with three predefined implementations: `DummyExecutor`, as well as two based on `rayon`, a data parallelism library for Rust [45]. The former simply runs the computation on the main thread, while the two latter let us seamlessly use `rayon`'s thread pools.

To use the `DummyExecutor`, simply create an instance of it and pass it in to `par_solve`. This is equivalent to calling `solve`.

```
use hotdrink_rs::executor::DummyExecutor;
let de = DummyExecutor::new();
rectangle.par_solve(&de);
```

Using rayon requires some more setup. Begin by adding the rayon feature flag to hotdrink-rs, and add rayon as a dependency. The dependencies in Cargo.toml should then look something like the following:

```
[dependencies]
hotdrink = { version = "0.1.3", features = ["rayon"] }
rayon = "1.5.1"
```

Then all that remains is to create a [rayon thread pool](#), and then call par_solve with it as an argument. See rayon's documentation for more information on how to customize the thread pool.

```
let pool = rayon::ThreadPoolBuilder::new().build().unwrap();
rectangle.par_solve(&pool);
```

If the planning step succeeds, the method computations will be scheduled to execute in parallel on the thread pool. This allows par_solve to return almost immediately after planning, which returns control flow to the caller. See Section 3.5 for more information about how methods are scheduled to run on the executor, and Section 9.2 for potential future improvements to scheduling.

3.1.4 Subscribing

Since par_solve may be used with an executor that executes methods on a separate thread, the values may not be ready in time for the function to return. In hotdrink-rs, we instead register callbacks in order to observe the changes once they are ready.

```
use hotdrink_rs::event::Event;
rectangle.subscribe("area", |event| match event {
    Event::Pending => println!("area is being computed"),
    Event::Ready(v) => println!("area's new value is {:?}", v),
    Event::Error(e) => println!("area's computation failed: {:?}", e),
});
```

The closure above will be called whenever a new Event is sent from the component. There are three possible events:

Pending when the current variable value becomes old, and will later be overwritten by a new value.

Ready when a computation succeeds, and a new value is available.

Error when a computation fails, and an error is available.

The Ready event does not necessarily contain a new value (it contains an Option<T>), as it can also be used to notify the GUI that the current value of a variable is no longer considered erroneous.

3.1.5 Creating Constraint Systems

Even though each Component is technically a constraint system on its own, `hotdrink-rs` provides an abstraction called `ConstraintSystem`. This lets the programmer store multiple components in one place, and perform actions (like solving) on all of them at once. With only one component, creating a `ConstraintSystem` is not strictly necessary. We therefore introduce another component called `circle` for this example, created in the same way as `rectangle`.

```
use hotdrink::model::ConstraintSystem;
let mut cs: ConstraintSystem<i32> = ConstraintSystem::new();
cs.add_component(rectangle);
cs.add_component(circle);
cs.edit("Rectangle", "height", 3);
cs.edit("Circle", "radius", 5);
cs.solve(); // Solve all modified components
```

3.2 Model Module

Rust crates are organized into *modules*, similar to *packages* in Java. To understand how the library is connected, we will go through the most important ones from the top down. Not every implementation detail is included, as that is what the library documentation is for. The following sections will serve as an overview of the implementation and the rationale behind some of the design choices that were made. A graph of the most important modules can be seen in Figure 3.1.

The `model` module contains the core data structures and types for representing constraint systems. We start at the top with `ConstraintSystem` in order to preserve the context as we delve deeper into the implementation.

3.2.1 Constraint System

A `ConstraintSystem<T>` is a container for `Component<T>`s where the variable values of the system are of type `T`. Each component is technically an independent constraint system on its own, but having these two layers provides a number of advantages.

1. `ConstraintSystem` provides an API for interacting with all Components at once.
2. It stops a single Component from becoming too large and unwieldy.
3. It improves modularity, as Components can be added and removed to modify the constraint system.

In `hotdrink-rs`, variables in different components are entirely independent, unless they are manually connected by observing changes in one component then editing another. This is not implemented in quite

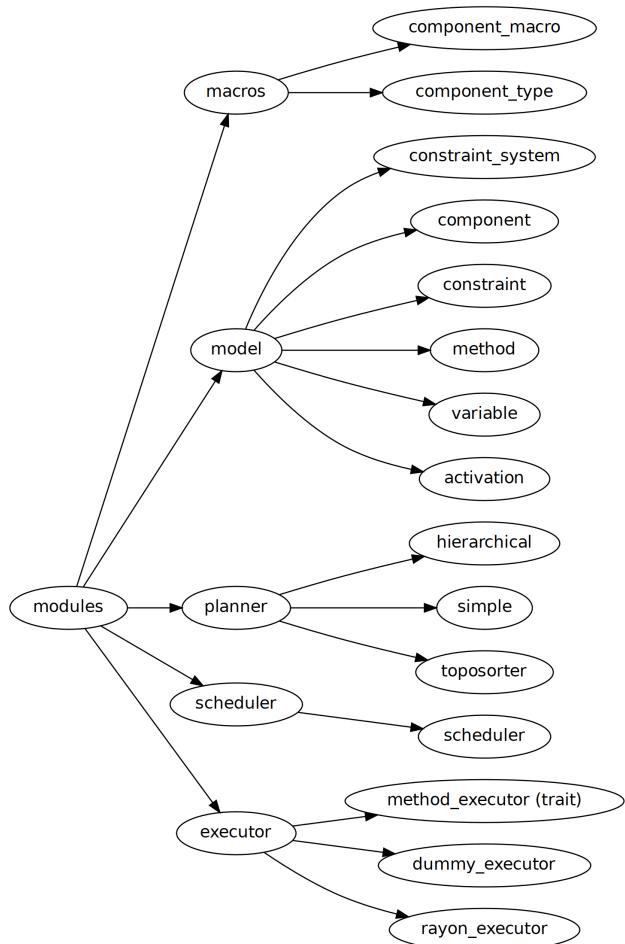


Figure 3.1: The `hotdrink-rs` module hierarchy.

the same way as in the Flow implementation of the library, where variables can be bound together across component boundaries.

The constraint system provides access to the different components stored within it, the API is very similar to the one found for Component in Section 3.2.2. The biggest difference is that in order to interact with variables, the component must also be specified. For instance, for editing operations, this can either be done by calling `edit` directly on the `ConstraintSystem`, or by getting a mutable reference to a specific Component and calling `edit` on that.

```
component.edit("a", 3);
constraint_system.edit("Component", "a", 3);
// Or alternatively
constraint_system.component_mut("Component").edit("a", 3)
```

The `solve` method behaves slightly differently for `ConstraintSystems` than it does for `Component`. Instead of always re-enforcing all constraints for all components, the former will only update ones that have actually been modified since the last time the constraint system was solved. This lets us avoid running expensive computations again when they would not produce any new results.

Undo and redo

To implement undo and redo for constraint systems, we maintain two separate stacks for operations done to the components (`edit`, `undo` and `redo`), one for “forward changes” and one for “backward changes”. An `edit` will place the edited component’s id on the “forward change” stack, while `undo` will place the component id on the “backward change” stack. The `redo` operation will pop from the “backward change” stack, and push it to the “forward change” stack to make it possible to undo the redo itself. For instance, with a component edit order of A, B, C , the first call to `undo` will undo the latest change to C , and another call will undo the latest change to B . Calling `redo` at this point would redo the latest change to B , and another call would redo the change to C .

3.2.2 Component

A `Component<T>` exposes much of the same API as `ConstraintSystem<T>`, but acts as an independent set of variables with constraints between them. The following sections will not cover the entire API, but should give an overview of the most important parts.

The edit method

The API allows giving new values to variables with `edit`. This will cause all constraints related to the variable, both directly and indirectly, to be re-enforced upon the next call to `solve`. Upon editing, the callbacks for that variable are also notified of the new value to synchronize multiple views of the same variable, such as an input field and a slider.

```
pub fn edit<'s>(
    &mut self,
    variable: &'s str,
    value: impl Into<T>
) -> Result<(), NoSuchVariable<'s>>
```

The value does not necessarily have to have type T: it can have any type that implements Into<T>. That is, any value that can be converted into a value of type T will do. This can be useful when using an enum as the value type of the component to avoid having to wrap values manually:

```
let c: Component<IntOrString> = component! { ... };
c.edit("a", 3);           // edit a as i32
c.edit("b", "foo".to_string()); // edit b as String
```

The solve and par_solve methods

As seen in the overview in Section 3.1, either solve or par_solve must be called to enforce the constraints of the system. For the Component type, their signatures look like the following:

```
pub fn solve(&mut self) -> Result<(), PlanError>
where
    T: Send + Sync + 'static + Debug

pub fn par_solve(&mut self, pool: &impl MethodExecutor) -> Result<(), PlanError>
where
    T: Send + Sync + 'static + Debug
```

The first, solve, is rather simple, and only requires a component to call the method on. It is implemented by calling par_solve and passing in a DummyExecutor as the MethodExecutor to solve the system on the main thread. For both of them, some additional constraints have been added to the type T: the Send, Sync, and 'static constraints are there to ensure that the values can safely be sent to another thread during the execution of the plan. The constraints being on solve as well is a consequence of it calling par_solve internally.

The system is not solved until one of the solve variants is called; splitting editing and solving provides two benefits:

1. Unnecessary work is avoided if the programmer wants to perform multiple edits before solving.
2. It allows for a custom MethodExecutor to be used if par_solve is chosen.

On the other hand, it also has at least one drawback: after performing an edit, the system is not necessarily consistent until the system is solved again. An alternative to this is to have edit call it automatically,

which would ensure that the system's constraints always are satisfied (if possible). We could also provide an alternative that does not, i.e., `set_variable`. Then the more "dangerous" option (the one that does not automatically make the system consistent) has to be explicitly selected, rather than being the default. However, this would either require an additional `MethodExecutor` parameter for `edit`, or that one is stored within the component itself. Since the `MethodExecutor` is unlikely to change for a given constraint system or component, it may be beneficial to set it once, and then have a single `solve` method that uses the internal `MethodExecutor`.

The subscribe method

The type signature for `subscribe` implemented for the `Component` type looks like the following:

```
pub fn subscribe<'s>(
    &mut self,
    variable: &'s str,
    callback: impl Fn(Event<'_, T, SolveError>) + Send + Sync + 'static
) -> Result<(), NoSuchVariable<'s>>
where
    T: 'static,
```

The most interesting part here is the callback itself. The programmer must provide a function that receives an event with a value type `T` and error type `SolveError`. This function will then be called each time a variable becomes pending, ready, or fails to be computed.

```
component.subscribe("a", |event| match event {
    Event::Pending => ...,
    Event::Ready(v) => ...,
    Event::Error(e) => ...,
});
```

As seen with the `solve`-methods, this callback also has a few additional constraints that are needed to ensure that it can be called from the threads executing the plan. Specifically, the `Send` trait ensures that all values stored in the closure can be sent to other threads, `Sync` ensures that they can be shared between threads without race conditions, and `'static` ensures that the values are not, and do not contain, references that may become invalid before the thread exits.

The exact events to include is not very clear cut: For instance, after a new edit has been made previous errors may no longer apply. We thus need some way of telling the GUI that the last value that was sent is not erroneous anymore (unless new errors appear after the new edit). Some alternatives are:

1. Create a new event type, `ClearError`.
2. Send a new Ready event with the old value.

3. Make Ready contain an Option<T> instead of T, and let a None value mean that the value currently in the GUI is not erroneous.

hotdrink-rs uses the third solution. This makes it difficult for the programmer to forget to handle the clearing of errors, like having the ClearError event type could do. The second alternative may seem good at first, until we start to consider which value to send. The previous value may still be in the process of being computed, have been canceled, or failed in another way. We could add a callback to the latest value that is being computed, but this value may not be computed successfully. Selecting the latest successfully computed value could be a viable alternative, but at this point, it is a lot more complex than just sending the GUI an event that tells it to keep the current value and clear the errors.

The variable and value methods

Each Variable in the component contains information about its different Activations: a [Future](#) of its value in a *generation* (See Figure 3.2). Activations are produced from method activations (the scheduling of a method), and work as slots to place values in once they have been computed. This lets the rest of the code interact with them (e.g., store the newest generation of variables in a component) without having to wait for their computation to finish. The values that are produced for one generation are then used to produce the next generation, and will be waited for once they are required as an input to another method.

The Component API has a method `variable` that returns a specific `Variable`, and `value` that returns the current Activation of a variable. The latter allows programs to get a Future of the most recent value of the variable instead of exclusively having to rely on callbacks. This is, for instance, useful in the [iced](#) (a GUI library inspired by [Elm](#)) example found in the repository under `examples/hotdrink-rs-iced`.

```
pub fn variable<'a>(
    &self,
    variable: &'a str
) -> Result<&Variable<Activation<T>>, NoSuchVariable<'a>>

pub fn value<'a>(
    &self,
    variable: &'a str
) -> Result<Activation<T>, NoSuchVariable<'a>>
```

The pin and unpin methods

One of the more minor (but useful) features is *pinning* of variables. This adds a *stay constraint* for the specific variable, which means that any successful plan does not involve modifying that variable's value. This lets the user ensure that some certain values do not change, though overuse can easily overconstrain the system.

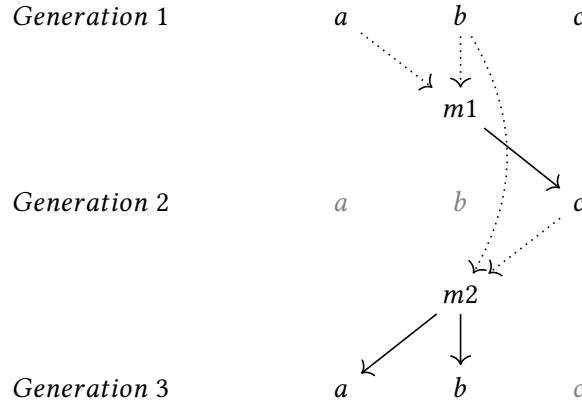


Figure 3.2: Variable generations. Dotted lines are reads, solid lines are writes. Gray values don't have to exist in the new generation, since we can reuse the earlier ones. This means that in this example we start with the 3 initial values, and then only add 3 new values instead of 3 per generation.

```
pub fn pin<'s>(&mut self, variable: &'s str) -> Result<(), NoSuchVariable<'s>>
pub fn unpin<'s>(&mut self, variable: &'s str) -> Result<(), NoSuchVariable<'s>>
```

The undo and redo methods

Lastly, Component's API also provides a way to undo and redo changes in the component. This works by maintaining a list of the variables that were changed between each generation: undo will switch to the previous variable activation for each variable modified between the previous and current generation, and redo will switch to the next variable activation for each variable modified between the current and next generation. If a previous or next value does not exist, then an error will be returned instead, indicating that there is nothing more to undo (or redo).

```
pub fn undo(&mut self) -> Result<(), NoMoreUndo>
pub fn redo(&mut self) -> Result<(), NoMoreRedo>
```

The values of all variables in the component can be visualized as a list of stacks, where each starts off having one element: the initial value for that variable. The asterisk indicates the current value, and is where the generation index of the variable points.

0*	0*	0*	0*
a	b	c	d

If we then perform an update that gives new values to *a* and *c*, it will look like this:

3*	5*		
0	0*	0	0*
a	b	c	d

The difference between these two generations (0 and 1) can then be recorded as a list $[a, c]$, and we can easily undo the change by decrementing the index of the current value for the two variables (visualized as moving the asterisks down one level). Redoing the change is just as simple, and requires us to increment the index of the two variables (visualized as moving the asterisks up one level).

We can continue doing new changes in the same way. If we now make a change to a and b , our difference list between generation 1 and 2 becomes $[a, b]$, and we end up in the following state:

```
7*
3   9*  5*
0   0   0   0*
a   b   c   d
```

Now, what if we would like to limit the number of generations that are kept? We can easily delete the first generation (given that there are at least 2) by deleting the first value of each variable modified between generations 0 and 1. They are guaranteed to have an additional value, otherwise, they would not be in the difference list. We do this for a and c since they were modified first.

```
7*  9*
3   0   5*  0*
a   b   c   d
```

We can repeat it for a and b to only maintain a single generation, and no undo- or redo-history.

```
7*  9*  5*  0*
a   b   c   d
```

There is currently no way to specify the number of generations to keep via the API of `Component` or `ConstraintSystem`, but, the internal data structure `Variables` supports it through its constructor `new_with_limit` and a method `set_limit`. Adding a limit to `ConstraintSystem`'s undo history does not have very clear semantics when each `Component` has its own limit. Here are two candidates for how to handle this:

1. A `ConstraintSystem` with limit l simply sets the limit of all its n `Components` to l .
2. Editing a `Component` such that we go past the `ConstraintSystem`'s undo limit will delete the earliest generation in the earliest modified component.

The first alternative may lead to up to $l \cdot n$ stored generations, and will increase further as new components are added. The second alternative would limit the number of stored generations to l , but also make the implementation more complex. The latter, however, may at least provide a starting point for future improvements.

For another potential issue, let us consider the following scenario: The user starts some expensive computation, but provides a new input before the computation is complete. What should then happen to the first computation? There are two main choices:

1. If we allow it to complete, then it may keep multiple threads busy when they are needed to perform the new computation. For the user, this would manifest as values not being updated without any indication of why.
2. If we cancel the computation, then the threads will be available for the new computation, but a new potential issue arises. If the user now undoes their latest action, they will be in a generation that was never fully computed. Unless this is handled in some special way, the user will just see the value being pending forever.

`hotdrink-rs` uses the second approach and gives the activation a special “canceled” error state. Undoing an operation to move back to a partially computed generation will send an event this error state to subscribers. This allows the programmer to annotate the affected fields in some way to notify the user that the value is currently wrong, and will not be updated until a new edit and solve is done. The main reason for choosing this solution is to guarantee that newer computations are run instead of continuing with old ones that are unlikely to be used; cancellation as a feature is prioritized over undo and redo.

3.2.3 Constraint

Constraint stores the name of the constraint, the indices of variables that it uses, and methods that are used to enforce it. Information about the relation that the methods enforce is not stored explicitly unless something like post-conditions in Section 9.6 is implemented. This means that the constraint is only guaranteed to become enforced if all methods are correctly implemented, and that the relation it enforces is decided by its methods, not the intended meaning of the constraint. To enable future implementations of post-conditions, the `assert` field with an optional assert statement for verifying that the Constraint is satisfied is included, but this feature is still experimental.

```
struct Constraint<T> {
    name: String,
    variables: Vec<usize>,
    methods: Vec<Method<T>>,
    assert: Option<Assert<T>>,
    active: bool,
}
```

A constraint can also be deactivated, which will just cause it to be filtered out upon reaching the planner. This allows the programmer to change the constraint system dynamically without having to add and remove the constraint, and can, for instance, enable an additional constraint when some checkbox is selected. Activation and deactivation can be done by using the `set_active` method.

3.2.4 Method

The Method implementation is split into two parts: an `enum` for representing stay-methods and normal methods, as well as a `struct` wrapper to make it easier to change the inner representation later.

```
enum MethodInner<T> {
    /// A stay method.
    Stay(usize),
    /// A normal method.
    Normal {
        name: String,
        inputs: Vec<usize>,
        outputs: Vec<usize>,
        apply: MethodFunction<T>,
    },
}

struct Method<T> {
    inner: MethodInner<T>,
}
```

An `enum` is used for `MethodInner` to make the representation of stay constraints more efficient. Instead of storing a name, two vectors that contain the same variable, and an identity function for the variable, we only store the variable index. This is useful during planning since we can potentially add as many stay constraints as there are variables.

3.2.5 Error Propagation

This section describes alternatives for how to handle propagation of errors through the constraint system upon solving, and answers two important questions: What should happen when a computation fails, and how should this affect other variables? For `hotdrink-rs`, we considered three main alternatives for the propagation of errors.

Full propagation Once a method fails and its outputs are considered erroneous, all outputs of any other method that uses these inputs also become erroneous. So if there are three variables a , b and c , as well as two methods $m_1 : a \rightarrow b$, $m_2 : b \rightarrow c$, then the failure of m_1 causes b to be erroneous, which then propagates to c through m_2 . This is the simplest solution, but can put the constraint system in a state with erroneous variables that is difficult to get out of. Consider having a constraint C between a , b and c with methods $m_1 : a \rightarrow (b, c)$, $m_2 : (b, c) \rightarrow a$. If m_1 fails after a was modified, then both b and c are marked as erroneous. If either b is modified to attempt to fix the values, then c being

erroneous will make m_2 fail and set a to erroneous. The same would happen if c was modified, and it is clear that we are stuck. This is visualized (though a little simplified) in Figure 3.3a.



(a) A visualization of how full propagation can get the user stuck in a bad state. Even if b_1 receives a new value from an edit, c_1 is in an erroneous state, and does not have a value. This makes m_2 fail and propagate the error to a_2 .

(b) A visualization of how reusing old values of variables when newer ones failed to be computed can bring the system back to a good state. As opposed to full propagation like in Figure 3.3a, both b_1 and c_1 now get good values; b_1 from an edit and c_1 from the reusing its old value, which means that m_2 can successfully compute a new value for a_2 .

Figure 3.3: A comparison between full propagation and reusing old values.

Reusing old values Instead of having variables that *either* have a value or an error, we may keep the old value in addition to the error. When a variable failed to get a new value, we can keep computing using its previous one (which matches the one currently in the GUI, since no new value has been sent from the constraint system). In the previous example, let $m_1(a) = (\frac{a}{2}, \frac{a}{2})$, and $m_2(b, c) = b + c$, with initial values $a = 4$, $b = 2$, and $c = 2$. If we then do a modification to a which makes m_1 fail, then b and c are still marked as erroneous, but their values remain as 2 both in the constraint system and GUI. Setting b to 3 clears its erroneous state, and m_2 adds that to the old value of c , namely 2, to form 5 as the new value for a . The constraint system is always consistent with the published values, and fixing the broken state is trivial.

How this approach fixes the issues of full propagation can be seen in Figure 3.3b.

Split solve This is another solution that involves maintaining the old values of variables even if they are erroneous. Unlike the two previous solutions, methods with failed inputs will not automatically fail their outputs, but use the old values of the inputs instead. This would let some parts of the system update with new values, even when some method inputs failed. Only variables that are outputs of

the original method that failed will be marked as erroneous, while the rest do not have any errors since the constraints are still enforced. However, this could be very confusing for the user, as it is not clear which variables are actually affected by the error, as they do not know which variables are connected by the underlying constraint system.

For `hotdrink-rs`, the second option was chosen. This makes it easier for users to get out of bad states by supplying new values to variables involved in the relevant constraint, instead of having the constraint system “forget” the values currently displayed in the GUI. This should also be much less surprising to the user, as they will likely base their expected behavior on the state that they can see.

3.3 Planner Module

The planners in this section are based on Brad Vander Zanden’s QuickPlan [73], as described in Chapter 2. The following subsections describe some details of how each planner variant is implemented, and how they deviate from Zanden’s algorithm.

3.3.1 Simple Planner

The simple planner implementation is based on Zanden’s *multi-output-planner* [73, p. 39]. Implementing the pseudocode description from the paper in Rust involved many design choices such as selecting appropriate data structures.

The implementation in `hotdrink-rs` starts by constructing a map from variables to constraints they are referenced by. A `Vec<Vec<usize>>` is used to store this information,² and accessing it with a variable index provides the indices of the constraints that reference that variable. Constructing this map involves iterating through every constraint c , and for each variable v that c refers to, adding an entry $v \rightarrow c$. Once the map is constructed, we can obtain the indices of constraints that reference the variable with `refs[v]`, and the number of references to the variable with `refs[v].len()`.

One iteration over the newly constructed map is then done to find all free variables (variables with one reference). Each one is added to a queue of “potentially free variables” (“potential” since later variables may have zero references left by the time we inspect them).

Variables are then popped from the queue until all constraints have been enforced, or no more free variables are found. Any variables with zero references left are simply ignored, as all of their constraints are already enforced. For variables with one reference left, we inspect the methods of the referencing constraint. Methods that (1) write to the variable and (2) to only other free variables are valid candidates, and the one that has the fewest outputs is selected. The constraint’s references to variables are then removed, and the method becomes part of the solution graph. Any variables of the removed constraint that now become free are added to the potentially free variables vector for processing in the next iteration, always giving us constant-time access to new candidates.

²Technically `Vec<VariableRefCounter>`, each of which contains a `Vec<usize>`.

When there are no more constraints to enforce, we have a set of methods that form the solution graph. The method vertices can then be topologically sorted to form a completed plan.

3.3.2 Hierarchical Planner

The hierarchical planner is based on the *constraint hierarchy planner and solver* in Zanden's paper [73, p. 44]. It attempts to solve the system with different combinations of stay constraints to guide the solution towards the one that is the least surprising to user. Solutions are compared by which variables they modify, and we prefer ones that avoid changing values that the user recently modified.

The order of attempting to add stay constraints is different from QuickPlan, where all stay constraints are added in the beginning. If planning fails, the weakest constraint is retracted before attempting to solve again [73, p. 41].

We start by attempting to solve the system with a stay constraint for the highest ranked variable. If it succeeds, we keep the stay constraint, and continue with the second-highest ranked variable. If it fails, we discard the constraint, then continue as we did in the first case. In the case that we never manage to add any stay constraints, we try solving without any of them: if this fails, then the system is overconstrained, and no solution exists.

Attempting to solve without any stay constraints first would let us detect overconstrained systems early, at the cost of having to call the simple planner an extra time for systems that can be solved with a stay constraint added. This may not sound like much, but when combined with *pruning*, an optimization explained later, this may actually double the number of simple planner calls in some cases. We would rather accept a performance penalty on overconstrained systems than in ones that can be satisfied. There are multiple optimizations that we can do to avoid so many calls to the simple planner.

Adding stay constraints directly

The first optimization is relatively simple. If the variable we are trying to add a stay constraint for is a source in the current solution graph, then adding it would still form a valid solution. We can see this by looking at the two criteria for solution graphs:

1. We must not introduce any cycles. If the variable is a source then it is only read from, so adding a stay constraint to it cannot introduce a cycle.
2. No variable must be written to twice. If the variable is a source then no other method writes to it, so the only write is the one we add.

This optimization allows us to skip calling the simple planner in some cases.

Pruning

Upon successfully solving with a new stay constraint added for a variable, we can remove all other methods that write to it. This is justified by the fact that this variable has a higher priority than any other ones we

would write to later. This will simplify the constraint system for subsequent calls to the simple planner, making it faster for each iteration.

We can also do some pruning when a stay constraint cannot be added, because this means that some other method must have been forced to write to the stay constraint's variable instead. If this method is unique, it can be selected to enforce its constraint, and we can remove that constraint from the system.

Cascading pruning

Pruning once can in some cases allow us to prune further, possibly propagating through the entire constraint system. In a chain of variables with two-way constraints between them like $a \xleftrightarrow{c_1} b \xleftrightarrow{c_2} c$, successfully adding a stay constraint to a means that c_1 cannot be enforced by a method that writes to a . We can thus remove one of c_1 's methods to get $a \xrightarrow{c_1} b \xleftrightarrow{c_2} c$. This may then propagate through the system as follows:

$\rightarrow a \xleftrightarrow{c_1} b \xleftrightarrow{c_2} c$	(a stay constraint is added to a)
$\rightarrow a \xrightarrow{c_1} b \xleftrightarrow{c_2} c$	(a is written to by stay, c_1 cannot write to it)
$\rightarrow a \xrightarrow{c_1} b \xleftrightarrow{c_2} c$	(only one method left to enforce c_1 , must be selected)
$\rightarrow a \xrightarrow{c_1} b \xrightarrow{c_2} c$	(b is written to by c_1 , c_2 cannot write to it)
$\rightarrow a \xrightarrow{c_1} b \xrightarrow{c_2} c$	(only one method left to enforce c_2 , must be selected)
$\rightarrow a \xrightarrow{c_1} b \xrightarrow{c_2} c$	(c is written to by c_2)

After just adding a stay constraint to a , we have determined that c_1 must be enforced by writing to b , and that c_2 must be enforced by writing to c . In fully “prunable” systems with n variables, this can let us go from n calls to the simple planner to only a single one.

Unfortunately, this strategy does not work for all configurations, e.g., a constraint with the two methods $m_1 : (a, b) \rightarrow c$ and $m_2 : (a, c) \rightarrow b$. If a gets a stay constraint we are unable to choose between m_1 and m_2 , since either may be selected later. See Section 6.1.4 for a more detailed description of an “unprunable” system.

3.4 Method Executor Module

To make the library as general as possible, the programmer may use any *method executor* implementation they wish when solving the system. For instance, Component::par_solve's type signature looks like this:

```
pub fn par_solve(&mut self, me: &mut impl MethodExecutor) -> Result<(), PlanError>
where
    T: Send + Sync + 'static + Debug,
```

The most important part now is me: `&mut impl MethodExecutor`, which just means that the function accepts any type that implements the `MethodExecutor` trait:

```
pub trait MethodExecutor {  
    type ExecError: Debug;  
  
    fn schedule(  
        &mut self,  
        f: impl FnOnce() + Send + 'static  
    ) -> Result<TerminationHandle, Self::ExecError>;  
}
```

The trait essentially just requires that closures can be scheduled to be run on the implementor, which can then decide how and when this happens. This could for instance be a thread pool, which would then result in parallel execution of methods.

The simplest implementation is the `DummyExecutor`. This does not actually use any additional threads at all, and simply executes the closures on the main thread as they are scheduled. If compiled with the `rayon` feature flag, `hotdrink-rs` also provides `MethodExecutor` implementations for `rayon` thread pool types, making parallel execution very simple; simply construct your thread pool and pass it in as an argument to `par_solve`.

```
let pool = rayon::ThreadPoolBuilder::new().build().unwrap();  
component.par_solve(&pool);
```

The `TerminationHandle` allows the method executor to know when results are no longer required. When all `TerminationHandle`s have gone out of scope, a flag for its associated result will be set. This is for instance used to decide when to cancel threads in `hotdrink-wasm`.

3.5 Solver Module

After creating a valid plan, we must schedule each method to be executed. The state starts off with the initial values of each variable in an array, which will then be modified as we execute new methods. For a variable v , we will call its initial value v_0 , followed by v_1 for its next value, and so on.

If we start by executing the method $m_1 : (a, b) \rightarrow c$, its input values will be a_0 and b_0 , and this will produce a new value for c , which we will call c_1 . We can then run another method $m_2 : (c, d) \rightarrow e$, which will use c_1 and d_0 as inputs, and produce e_1 as its output. The values after each step can be seen in Table 3.1.

While the above gives a decent idea of how this works during single-threaded execution, the picture is more complicated when methods are executed in separate threads. This module does not actually execute

Table 3.1: Values per step

Step	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
Initial values	a_0	b_0	c_0	d_0	e_0
Values after m_1	a_0	b_0	c_1	d_0	e_0
Values after m_2	a_0	b_0	c_1	d_0	e_1

the methods, but simply sends the computations to a supplied `MethodExecutor`. Instead of having simple values of type `T`, each variable's current value is actually an `Activation<T>`, which works similarly to a `Future` or `Promise` that will eventually resolve to a `T`. This means that it works as a placeholder that can be passed around regardless of whether the value has been computed or not. *Scheduling* the methods can be done by sending `closures` that do the following to the `MethodExecutor`: (1) wait for the appropriate input activations to complete, (2) execute the method with the resolved inputs, and (3) place the computed values in appropriate output-activations.

While executing a method has a type similar to `Vec<T> -> Result<Vec<T>, MethodFailure>`, activating a method cannot return a result immediately. It will instead return a `Vec<Activation<T>>`, which will contain the final values when the method has been executed, something which could happen on a separate thread.

3.5.1 Activation

This section takes closer look at `Activation` and `ActivationInner`. First of all, we need some shared state that both the computing thread and the main thread can access. In `hotdrink-rs`, this data is called `ActivationInner`.

```
pub struct ActivationInner<T> {
    state: State<T>,
    waker: Option<Waker>,
}
```

It contains a `Waker` [74] to call when its state changes, which will make asynchronous code check the value again. In addition, it has a `State` that can either be `Pending`, `Ready` or `Error`. Changing its state from `Pending` will (1) call the waker to notify anyone awaiting it, and (2) clear the dependency list to decrease the number of interested parties.

```
enum State<T> {
    Pending(PendingData<T>),
    Ready(Arc<T>),
    Error(ErrorData<T>)
}
```

The Pending state is the state an activation is in until the method that produces its result has finished or failed. It also has the previous activation of the current variable, as well as a list of the current activation's dependencies.

```
struct PendingData<T> {
    previous: Activation<T>,
    dependencies: Vec<Activation<T>>,
}
```

The previous field is the previous value of the variable this activation is for. If the current activation fails, then the old value will be used to solve the system instead. This will appear consistent to the user since the previous value either (1) was sent to the constraint system by the user, (2) was produced by the constraint system and sent to the GUI, or (3) is still pending. In any case, the value will be consistent with the one that the user currently sees. The dependencies field maintains references to the input activations to keep them alive until their values have been computed.

The Ready state simply contains the resulting value from a successful computation. It is behind a reference counted pointer to allow sharing the value between all methods that require it; more specifically, an atomically reference counted pointer to allow sharing the value between threads. An alternative to this would be to clone the value each time, but this places a [Clone](#) bound on the value type, which may be undesired. When a value is in this state, the previous activation and the dependencies have been dropped.

```
struct ErrorData<T> {
    previous: Activation<T>,
    errors: Vec<SolveError>,
}
```

The final state, Error, keeps the previous activation in order to produce a result. It also maintains a set of errors that happened during its computation. This can be any of the following:

1. The computation was canceled.
2. A method used a non-existent variable.
3. A method received the wrong number of inputs.
4. A method produced the wrong number of outputs.
5. There was an attempt to downcast a value to the wrong type.
6. A custom error message from the programmer, for instance with the `fail!` macro.

All the errors will be propagated from method inputs to outputs, which makes it possible to mark all erroneous values in the GUI, and even provide a reason for the failure.

An Activation contains a shared ActivationInner (Arc to let us share across threads, Mutex to synchronize access), as well as a TerminationHandle. The former allows us to pass references to the future value of a variable to other threads, while the latter ensures that the computation is kept alive until there are no clones left.

```
struct Activation<T> {
    inner: Arc<Mutex<ActivationInner<T>>>,
    producer: Option<TerminationHandle>,
}
```

Before scheduling a new method, we drop the TerminationHandles of its outputs. This is to ensure that the number of references goes down *before* a MethodExecutor like StaticPool from hotdrink-wasm attempts to cancel any computations. Doing it after would make StaticPool think that the result is still needed.

Eager cancellation

The ability to add and remove constraints complicates when it is fine to cancel activations, i.e., cancel the method computation producing it. In the case where the constraint graph cannot change, writing to a variable means that all other variables that in the same generation that depend on will read from the most recent activation, guaranteed by the topological ordering of the plan; the old value is thus not required in the new generation. Any method computation that depends on the previous activation will be replaced by a new selected method, since the same constraints must be enforced in the new generation too. Thus, when a variable is an output in the new solution graph, we can cancel the computation of its previous activation. We will call canceling a variable's old computation with only the criterion that it is an output in the solution graph *eager cancellation*.

However, the same arguments for the correctness of eager cancellation do not apply in the case where the constraint system can change. Consider the following example, with one constraint between a and b with the method $ab : a \rightarrow b$, and another constraint between b and c with a method $bc : b \rightarrow c$.

$$edit \rightarrow a \xrightarrow{ab} b \xrightarrow{bc} c$$

Say we edit a , which schedules two computations: ab and bc . If we now remove the constraint between b and c , there is no longer a connection between the two. Editing a again before the previous value of b is computed will result in the following: First, since a is written to, the previous value of a is marked as not needed, but since its value was from an edit, no computation is canceled. Second, b receives a new activation that is being computed from the new value of a , and the computation of b 's previous activation is canceled. The problem now is that since the constraint between b and c is gone, c still depends on b 's old

activation, the one we canceled. Thus, since the computation of b 's activation was not complete before we canceled it, neither was the computation of c 's activation, leaving it in a failed state in the last generation. This is visualized in Figure 3.4.

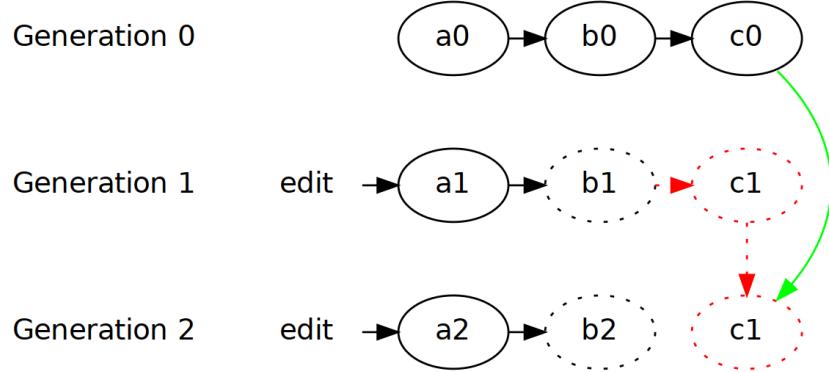


Figure 3.4: Eager cancellation may lead to failed computations in the new generation due to canceling required dependencies. Activations that are not complete have dotted borders. The value c_2 in generation 2 can be seen as a “virtual” value, as it is the same value as in the previous generation. This means that it still depends on b_1 . The green arrow indicates that we can reuse the last computed value since c_1 is no longer bound by any constraints.

We do not want to end up with erroneous values due to canceling too early; we present two alternatives for how to handle this.

1. The first alternative is to not use eager cancellation. Instead, use reference counting to know when an activation is no longer required: in the example above, the computation of b_1 would not have been canceled, as it is still required by c_1 . This produces some overhead, with each activation having to keep track of its dependencies, but always keeps the computations alive until they are no longer needed.
2. The second alternative is to not consider failure due to cancellation in this case to be an error. Since c_1 is no longer bound by any constraints, any value would be correct. The user also made further edits and performed an action that removed the constraint, which suggests that they do not care about the value of the variable's old activation. Simply reusing the latest published value is then an option, and it may even be more predictable since this is likely the same value as is currently shown in the GUI. For the user, the variable is just no longer pending, instead of suddenly getting a new value computed from values from the previous generation.

While the current implementation uses the first approach, we have considered using the second. The first may be more “correct” according to the constraint system, but the new value of c_1 would be based on values from a generation that the user can no longer see in the GUI. The second approach would let the library

cancel computations that the user likely does not care about (since they made further edits, and likely want new, updated values) earlier, which would allow work to be done on newer computations instead.

3.6 Macros Module

The library relies on Rust [macros](#) to make an ergonomic DSL for declaratively describing constraint systems. For the same task, the Flow- and TypeScript-implementation of HotDrink use [template literals](#) and “method chaining” respectively.

The macros are intended to be used for smaller constraint systems (up to a few dozen constraints) that a programmer can write by hand, or an initial system if it will be expanded upon programmatically. For instance, many of the examples in `hotdrink-rs/examples` use the macro, while the constraint systems generated for benchmarks are constructed programmatically since they contain thousands of constraints.

3.6.1 component!

The `component!` macro is the primary way to define small constraint systems. A simple example can be seen below, followed by an explanation of what the different parts do.

```
// Create a component with values of type i32
let component: Component<i32> = component! {
    component MyComponent {
        // Define three variables of type i32.
        let a: i32 = 0, b: i32 = 0, c: i32;
        constraint MyConstraint {
            // Define a method (a, b) -> c that returns a normal expression
            my_method(a: &i32, b: &i32) -> [c] = ret![a + b];
            // Define a method (a, b) -> c that returns a block-expression
            my_method_block(a: &i32, b: &i32) -> [c] = {
                let x = a;
                let y = b;
                ret![x + y]
            };
        }
    }
};
```

The macro starts off by specifying the name of the component so that it can be referred to later, such as if it is added to a `ConstraintSystem`, or for location information if an error happens within it.

Following that, the variables of the components are defined following the `let`-keyword. Each variable must have a name and a type, followed by an optional initial value. If no such value is specified, the macro

will attempt to use the type's `Default` implementation, and if that fails then a compile-time error will occur, as seen in Figure 3.5. After having defined the variables, an arbitrary number of constraints may be defined.

```
Compiling hotdrink-rs v0.1.1 (/home/rudi/git/hotdrink-rs/hotdrink-rs)
error[E0599]: no function or associated item named `default` found for struct `NoDefault` in the current scope
--> hotdrink-rs/examples/misc.rs:84:5
82 |     struct NoDefault;
83 |     ----- function or associated item `default` not found for this
84 | /      hotdrink_rs::component! {
85 | /         component Comp {
86 | /             let a: NoDefault;
87 | /         }
88 | /     };
88 | /     ^ function or associated item not found in `NoDefault`
= help: items from traits can only be used if the trait is implemented and in scope
= note: the following trait defines an item `default`, perhaps you need to implement it:
        candidate #1: `Default`
= note: this error originates in a macro (in Nightly builds, run with -Z macro-backtrace for more info)
```

Figure 3.5: Missing Default implementation.

Each constraint requires a name and a non-empty set of methods. If we allowed an empty set of methods, then the constraint could never be enforced, and would thus not be very useful.

As shown in the component macro example above, methods start with an identifier, followed by a parameter list defining the variables the method reads from (immutable references) and their types, then a list of outputs, and finally the method body.

The macro first constructs a `RawMethod` from each method in the DSL. The `RawMethod` is then post-processed and converted into a `Method`. A method can be constructed manually as follows.

```
let m1: Method<i32> = Method::new("m1", vec![0,1], vec![2], Arc::new(|inputs| {
    let a = *inputs[0];
    let b = *inputs[1];
    Ok(vec![Arc::new(a + b)])
}))
```

The macro takes care of many of the conversions for us, such as converting the `Arc<T>` to `&T`. The reason that the types must be specified in the parameter list is related to how values of multiple types are stored in the constraint system. For instance, if we use an enum `NumOrString` for the values, where `&NumOrString` implements `TryInto<&i32>` and `TryInto<&String>`, then this conversion can happen automatically by specifying the desired type in the parameter list. That is, if the component has type `Component<NumOrString>`, we can still write a method `m(x: &i32) -> [y] = { ... }`, which stops us from always having to manually unwrap the values. The `component_type!` macro from Section 3.6.2 simplifies defining types that automatically implement the required conversion traits.

3.6.2 component_type!

Since the way to allow multiple types in a component is to use an `enum`, which can then be automatically converted into its variants by the component macro if it implements the appropriate traits, it would be

beneficial to have a simple way of defining such a type. This is exactly what the `component_type!` macro does: The programmer specifies which types they would like to use in their constraint system, and the enum is automatically constructed together with the required `From` and `TryFrom` implementations. Manually implementing `From` to wrap the variants and `TryFrom` to unwrap them would easily reach 100 lines with only a few variants, so this reduces a lot of boilerplate. The automatic conversion allowed by using these traits improves ergonomics significantly, as shown in the example below.

```
component_type! {
    #[derive(Clone, Debug)]
    enum NumOrString {
        i32,
        String
    }
}

// Without automatic conversion (`Arc<T>` is still turned into `&T`)
m1(a: &NumOrString, b: &NumOrString) -> [c] = {
    match (a, b) {
        (NumOrString::i32(a), NumOrString::i32(b)) => Ok(vec![NumOrString::i32(a + b)]),
        (a, b) => fail!("a and b are not numbers: {:?}", (a, b)),
    }
};

// With automatic conversion
m2(a: &i32, b: &i32) -> [c] = ret![a + b];
```

In either case, calling the method with values that are not the correct variant will fail, but this is done automatically by the macro in `m2`. In this example, the `component_type!` macro is used to automatically generate the `enum`, along with the following trait implementations:

1. `From<i32>` for `NumOrString` to wrap numbers.
2. `From<String>` for `NumOrString` to wrap strings.
3. `TryFrom<&NumOrString>` for `&i32` to unwrap numbers.
4. `TryFrom<&NumOrString>` for `&String` to unwrap strings.

The two last implementations are fallible since the value may not be of the correct variant. Note that implementing `From` and `TryFrom` traits automatically provide implementations for `Into` and `TryInto` respectively.

Another option is to use the `derive_more` crate, in which case one can write the following to generate the same type. Using this removes the need to have the `component_type!` macro at all, and may be the suggested way to generate such types in the future.

```
use derive_more::{From, TryInto};
#[derive(Clone, Debug, From, TryInto)]
enum NumOrString {
    i32(i32),
    String(String),
}
```

3.6.3 ret!

Since methods can perform conversions from the actual type of their inputs, we must also make sure to convert them back after running the method body and computing the results. Method bodies should return a value of type `Result<Vec<Arc<T>>, MethodFailure>`, e.g. `Ok(vec![Arc::new(2), Arc::new(3)])` if our component contains values of type `i32`.

The issue arises when the type is an `enum` with multiple variants: all values must be wrapped before being stored in the `Vec`, which means that the `component!` macro cannot post-process the method body to fix it. For the `NumOrString` type, we may have to return an expression like

```
Ok(Arc::new(NumOrString::String(s)), Arc::new(NumOrString::i32(i)))
```

The conversion must happen before storing the values in a homogeneous `Vec`, and can thus not happen in a post-processing step. With `ret!` the expression above is simplified to

```
ret![s, i]
```

The macro has multiple functions: it uses the `Into` trait to wrap the values in the appropriate enum-variants, wraps each value in an `Arc`, combines them in a `Vec`, and finally wraps the `Vec` in the `Ok` variant of `Result`. This is all captured in the macro's definition below.

```
#[macro_export]
macro_rules! ret {
    ($($e:expr),*) => {{ Ok(vec![ $($e.into()),* ]) };}
}
```

3.6.4 fail!

The `fail!` macro serves as a dual to returning a successful result with `ret!`, and represents a failed method execution. It can contain a custom error message that specifies what went wrong by using `format!:> fail!("Method failure: {}", msg)`. The macro returns the `Err`-variant of `Result`, and contains a `MethodFailure` with the specified error message. It is defined as follows:

```
#[_macro_export]
macro_rules! fail {
    ($($arg:tt)*) => {{ Err($crate::planner::MethodFailure::Custom(format!($($arg)*))) }};
}
```

Chapter 4

Implementation of hotdrink-wasm

This chapter describes the implementation of `hotdrink-wasm`, a library that simplifies the process of generating WebAssembly-compatible data structures to enable usage of `hotdrink-rs` from JavaScript. It also includes information on how to generate and use these wrappers, and why they are required. For more specific implementation details, take a look at the published documentation,¹ or generate more up-to-date documentation with the following command in the `hotdrink-wasm` directory: `cargo doc --no-deps --document-private-items --features thread`.

Complete examples of how to use the library can be found in the `examples/` directory in the same repository as `hotdrink-rs` [65].

4.1 Overview

Creating the constraint system in `hotdrink-wasm` is no different from that in `hotdrink-rs`. However, while it may be implemented in the future, WebAssembly does not yet support generics [18]. This means that all generic data structures and functions must be *monomorphized* [35] upon being compiled to WebAssembly. For instance, if one constraint system has a value type of `i32`, and another has a value type of `String`, we end up with two separate copies of the same code; one for `ConstraintSystem<i32>` and another for `ConstraintSystem<String>`. When compiling the library to WebAssembly, types such as `Component<T>` and `ConstraintSystem<T>` thus no longer exist, and cannot be exposed in the public API. The types can still be used in Rust, but must be hidden behind types or functions that do not use generics.

We can, for instance, create a new type `IntConstraintSystem` that contains a `ConstraintSystem<i32>`, and generate WebAssembly-bindings with `wasm-bindgen`. We will refer to the contained constraint system as the *inner constraint system*.

```
#[wasm_bindgen]
pub struct IntConstraintSystem {
```

¹<https://docs.rs/hotdrink-wasm/>

```

inner: ConstraintSystem<i32>,
}

```

We can also implement methods on this struct that can be used from JavaScript:

```

impl IntConstraintSystem {
    #[wasm_bindgen]
    pub fn edit(&mut self, component: String, variable: String, value: i32) {
        self.inner.edit(component, variable, value);
    }
}

```

Though this struct only works for values of type `i32`, it demonstrates how the generic type can be hidden behind a WebAssembly-compatible wrapper. A wrapper struct must be written for each value type one wants to use in a constraint system. In order to automatically generate such a constraint system wrapper, we can use the `constraint_system_wrapper!` macro, which has three main purposes.

1. Generate an enum that represents the possible values in a constraint system.
2. Generate a WebAssembly-compatible wrapper around a value type.
3. Generate a WebAssembly-compatible wrapper around a constraint system.

After generating the wrapper types for a given constraint system as seen in Listing 1, we can define a system to store it with `hotdrink-rs`, and store it within the wrapper as seen in Listing 2. The code can then be compiled to WebAssembly, and the constraint system can be constructed by calling the function `my_constraint_system`. As demonstrated in Listing 3, the constraint system can then be used from JavaScript with an API similar to the one found for the `ConstraintSystem` type in `hotdrink-rs`.

```

hotdrink_wasm::constraint_system_wrapper! {
    pub struct ConstraintSystemWrapper {
        pub struct IntOrStringWrapper {
            #[derive(Clone, Debug)]
            pub enum IntOrString {
                i32,
                String
            }
        }
    }
};

```

Listing 1: Defining a constraint system wrapper with `hotdrink-wasm`.

```

#[wasm_bindgen]
pub fn my_constraint_system() -> ConstraintSystemWrapper {
    let component = ...;
    let mut cs = ConstraintSystem::new();
    cs.add_component(component);
    ConstraintSystemWrapper::wrap(cs)
}

```

Listing 2: Wrapping a constraint system made with hotdrink-rs.

```

let cs = wasm.my_constraint_system();
let wrapper = wasm.IntOrStringWrapper;

// Send changes from the GUI to the constraint system.
let variable = document.getElementById("s");
variable.addEventListener("input", () => {
    cs.edit("MyComponent", "s", wrapper.String(variable.value));
    cs.solve();
});

// Send changes from the constraint system to the GUI.
cs.subscribe("MyComponent", "i", v => { variable.value = v; });

```

Listing 3: Using the constraint system from JavaScript.

4.2 Heterogeneous Constraint Systems

As seen in Chapter 3, hotdrink-rs uses [enums](#) to allow values of multiple types in constraint systems. There are other ways of doing this, with their own advantages and disadvantages. This section will describe two other alternatives, and how the limitations of hotdrink-wasm affected the solution chosen for hotdrink-rs. The solutions that were considered are

- `wasm_bindgen::JsValue`, a wrapper around values received from JavaScript,
- `std::any::Any`, a trait to emulate dynamic typing in Rust,
- and Rust [enums](#).

Of these solutions, the first two were rejected, for reasons explained below.

4.2.1 The JsValue Type

The most “obvious” solution is to use `wasm_bindgen::JsValue`, which simply represents values received from JavaScript. It does, however, have a number of flaws.

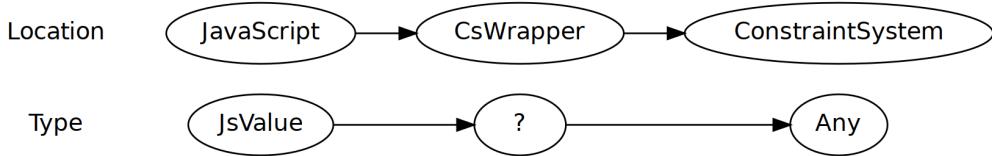


Figure 4.1: Type conversion in the constraint system wrapper. The value can be received by the wrapper from JavaScript as a `JsValue`, but must then be converted to the type of the variables of the inner constraint system.

1. As described in 4.6, a `JsValue` cannot be passed between threads since it contains a pointer to memory that Web Workers cannot access. Consequently, it does not implement the `Send` trait, and attempting to send it across the thread boundary in Rust results in a compile-time error. Using this strategy would thus entirely eliminate the possibility of executing methods in parallel.
2. Using `JsValue` as the value type of `hotdrink-rs` would limit the usability of the library. By making the value type a generic type parameter instead, we place nearly no restrictions on it.

These limitations are too severe to ignore, especially since multithreading is such a large part of our work. `JsValue` was thus eliminated as a candidate for enabling heterogeneous constraint systems.

4.2.2 The Any Trait

The second alternative is to use the `Any` trait, which allows us to store pointers to any Rust value, with the ability to safely attempt to downcast it to our desired type. By adding another constraint, `Send`, we can also ensure that the types can be sent across threads safely.

This approach is not without its issues either. While it would work for `hotdrink-rs`, passing data between JavaScript and Rust is more problematic. At some point, we must decide which concrete Rust types to convert values received from JavaScript to, otherwise we run into the same limitations as in Section 4.2.1.

Each value we send to the constraint system wrapper must be converted before being passed to the inner constraint system. Figure 4.1 shows the conversion steps. The concrete type selection problem can also be shown with the code below. The question mark indicates the point where we must perform the conversion, but do not know which type to convert the value to.²

```
impl ConstraintSystemWrapper {
    #[wasm_bindgen]
    fn edit(&mut self, variable: String, value: JsValue) {
        let converted: ? = convert(value);
        self.inner_cs.edit(variable, Box::new(converted));
    }
}
```

²Note that the value must be stored behind a pointer (`Box` is similar to `unique_ptr` in C++) so that its size can be determined at compile-time. See <https://doc.rust-lang.org/book/ch17-02-trait-objects.html>.

```

    }
}

impl<T> ConstraintSystem<T> {
    fn edit(&mut self, variable: String, value: Box<dyn Any>) { ... }
}

```

Whether we let `wasm-bindgen` take care of the conversion, or receive a `JsValue` and convert it ourselves, some type must be selected. The type can be different for each constraint system, and each variable within it. We can thus not automatically generate a wrapper unless the programmer provides information about which types they would like to use.

If the programmer wants to, e.g., use values of the types `i32` and `String`, there must be some way to specify which type to convert the value to. One way of doing this is to have two different functions `edit_i32` and `edit_string`. However, new identifiers cannot be generated automatically with declarative macros due to hygiene (though procedural macros do not have this limitation) [6], and it requires that the programmer selects the correct type. Passing in an argument to `edit` that specifies the type would also suffer from conversion errors potentially happening in `edit`. A third way is for `edit` to take some kind of enum as an argument, where the caller passes in a value of the appropriate variant. In this case, any conversion errors will happen before the value is passed in to `edit`. The last option leads us to the third solution on how to store values of multiple types in the constraint system.

4.2.3 Rust Enums

Not every type can be compiled to WebAssembly: An overview of the available types can be found in the `wasm-bindgen` guide [62]. Rust enums, for instance, do not have a direct translation to WebAssembly or JavaScript. Since this is our main way of describing value types for components and constraint systems, we require some kind of workaround to support WebAssembly. The macro `wrap_enum!` simplifies the process by creating a WebAssembly-compatible wrapper for such types.

```

hotdrink_wasm::wrap_enum! {
    pub struct IntOrStringWrapper {
        #[derive(Debug, Clone)]
        pub enum IntOrString {
            i32,
            String
        }
    }
}

```

We call `IntOrStringWrapper` the *wrapper type*, and `IntOrString` the *inner type*.

`wrap_enum!` implements a superset of the functionality of the `component_type!` macro from `hotdrink-rs`, and requires the same information as input. `wrap_enum!` therefore starts off by invoking `component_type!` to generate the `From`- and `Into`-implementations, followed by it generating the WebAssembly-compatible constructors and code for conversion to a `JsValue`. Generating new constructors is required since the enum (`IntOrString`) cannot be exposed directly to WebAssembly, and thus neither can its constructors. To replace them, we can generate associated functions for `IntOrStringWrapper` for creating the variants instead, as shown below.

```
#[wasm_bindgen]
pub struct IntOrStringWrapper {
    inner: IntOrString
}

#[wasm_bindgen]
impl IntOrStringWrapper {
    #[wasm_bindgen]
    pub fn i32(value: i32) -> Self {
        IntOrString::i32(value)
    }

    #[wasm_bindgen]
    pub fn String(value: String) -> Self {
        IntOrString::String(value)
    }
}
```

These functions can then be used to create values of type `IntOrStringWrapper` from JavaScript, e.g., `IntOrStringWrapper.i32(3)` and `IntOrStringWrapper.String("foo")`. When this value is received by Rust, the inner `enum` can be extracted from the wrapper.

The wrapper type `IntOrStringWrapper` can be used as the argument type for `edit` as follows, given that the inner constraint system's value type is the `enum` `IntOrString` defined earlier:

```
fn edit(&mut self, variable: String, wrapper: IntOrStringWrapper) {
    let ios: IntOrString = wrapper.unwrap();
    self.inner_cs.edit(variable, ios);
}
```

The macro will also automatically generate code for converting an `IntOrString` to `JsValue`:

```
impl From<IntOrString> for wasm_bindgen::JsValue {
    fn from(value: IntOrString) -> Self {
```

```

        match value {
            IntOrString::i32(i) => i.into(),
            IntOrString::String(s) => s.into(),
        }
    }
}

```

This lets the library return a `IntOrString` directly to JavaScript, and it will appear there as the appropriate type: an `IntOrString` of the `i32` variant will result in a number, while a one of the `String` variant will become a string. Naturally, each variant of the `enum` must already be compatible with WebAssembly, for instance by generating the appropriate bindings with `wasm-bindgen`.

4.3 Generating a WebAssembly-Compatible Constraint System

In order to interact with a constraint system from WebAssembly, we must generate a WebAssembly-compatible wrapper that replicates its API. The `constraint_system_wrapper!` macro creates a wrapper type for single-threaded constraint systems. The following sections will analyze the most important parts of the macro's output to understand what it has to do to replicate the `ConstraintSystem` API.

We start with an example of an invocation of the single-threaded version of the macro:

```

hotdrink_wasm::constraint_system_wrapper! {
    pub struct ConstraintSystemWrapper {
        pub struct IntOrStringWrapper {
            #[derive(Clone, Debug)]
            pub enum IntOrString {
                i32,
                String
            }
        }
    }
};

```

`ConstraintSystemWrapper` is the name of the generated constraint system wrapper type, and it will contain an inner constraint system of type `ConstraintSystem<IntOrString>`. Everything from `pub struct IntOrStringWrapper` and within is first passed on to the `wrap_enum!` macro to generate the appropriate value type wrappers. The `constraint_system_wrapper!` macro thus gathers all code for generating the appropriate wrappers in one place.

The constraint system wrapper contains functions that replicate each part of the `ConstraintSystem` type's API, such as the `subscribe` function. The `subscribe` function is relatively simple; the callbacks for

a given variable are first stored (as `js_sys::Function` objects) in a dedicated data structure within the constraint system wrapper called an `EventHandler`. The callback sent to the inner constraint system adds all received events to a queue, and after the `subscribe` has completed, the events are sent to the actual callbacks provided from JavaScript. We cannot call the callbacks from JavaScript directly since passing it to `subscribe` would require it to satisfy a `Send` constraint, which the callbacks from JavaScript do not. The code below shows a simplified version of the relevant part of `subscribe`.

```
...
inner.subscribe(component, variable, |event| { queue.add(event) });
for event in queue {
    event_handler.handle_event(event);
}
```

Note that a call to `inner.subscribe` will immediately call the callback with the most recent value of the variable to provide the initial values of the system, and since we are currently only using a single thread, all relevant events are sent before the call completes.

The constraint system wrapper also has an associated function for solving the system, but unlike the `ConstraintSystem` type from `hotdrink-rs`, the wrapper only has one solve variant named `solve`. This method calls `solve` on the inner constraint system, which solves it on the main thread. If solving the system succeeds, all events should be in the event queue, ready to be handled. Otherwise, an error with the failure information is logged to the JavaScript console. This is shown in the (slightly simplified) code below.

```
pub fn solve(&self) {
    match self.inner.solve() {
        Ok(_) => self.handle_events(),
        Err(e) => {
            log::error!("Update failed: {}", e);
        }
    }
}
```

The `edit` function does nearly no additional work, but must unwrap the value wrapper to get the value within, as shown in the `edit` implementation in Section 4.2.3. The same applies to `undo`, `redo`, `enable_constraint` and `disable_constraint`: the operation is simply propagated to the inner constraint system.

4.4 Image Scaling Example

The easiest way to get started with `hotdrink-wasm` is to use a template such as the [rust-webpack-template](#) [52], as this makes it easy to write Rust code, compile it to WebAssembly, and import it from

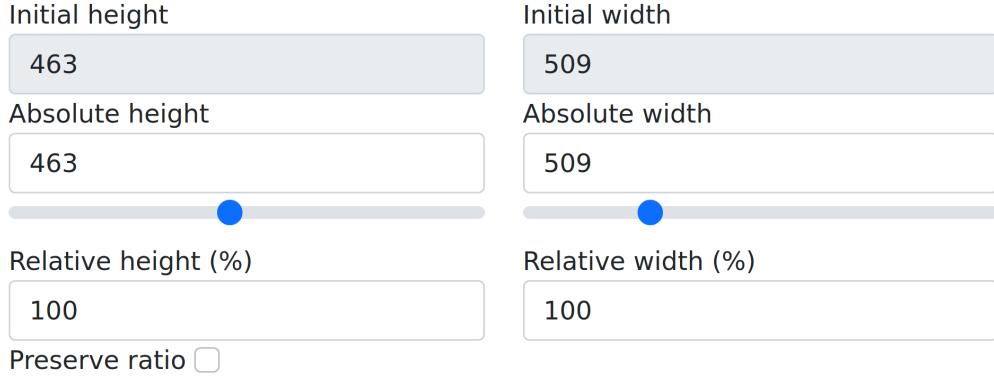


Figure 4.2: A simple image scaling example made with hotdrink-wasm and JavaScript.

JavaScript. To start using hotdrink-rs, one just has to use the template to generate a project, add hotdrink-rs and hotdrink-wasm to the dependencies in `Cargo.toml`. The example uses the Rust nightly channel (nightly-2021-03-01) and `wasm-pack` version 0.9.1; breaking changes may happen in later versions. For more up-to-date information, take a look at the Rust and WebAssembly book, the `wasm-pack` book, and the `wasm-bindgen` book [51, 78, 75].

This section describes how to build a more complex hotdrink-wasm example: image scaling. The complete implementation can be found in the `hotdrink-rs` repository under `examples/hotdrink-wasm-simple`.

4.4.1 Defining the Constraint System

To define the constraint system, we must figure out which constraints are actually required. We want the relative height to be the absolute height divided by the initial height, the relative width to be the absolute width divided by the initial width, and finally a constraint that ensures that the aspect ratio is equal to the absolute width divided by the absolute height.

$$\text{relative_height} = \frac{\text{absolute_height}}{\text{initial_height}} \quad (4.1)$$

$$\text{relative_width} = \frac{\text{absolute_width}}{\text{initial_width}} \quad (4.2)$$

$$\text{aspect_ratio} = \frac{\text{absolute_width}}{\text{absolute_height}} \quad (4.3)$$

We can then translate this to a hotdrink-rs Component.

```
// lib.rs
let component: Component<Number> = hotdrink_rs::component! {
    component ImageScaling {
        let initial_height: i32 = 400, initial_width: i32 = 400,
        relative_height: i32 = 100, relative_width: i32 = 100,
    }
}
```

```

absolute_height: i32, absolute_width: i32,
aspect_ratio: f64 = 1.0;

// relative_height = absolute_height / initial_height
constraint RelativeHeight {
    a(initial_height: &i32, absolute_height: &i32) -> [relative_height]
    = ret![100 * absolute_height / initial_height];
    b(initial_height: &i32, relative_height: &i32) -> [absolute_height]
    = ret![initial_height * relative_height / 100];
}

// relative_width = absolute_width / initial_width
constraint RelativeWidth {
    a(initial_width: &i32, absolute_width: &i32) -> [relative_width]
    = ret![100 * absolute_width / initial_width];
    b(initial_width: &i32, relative_width: &i32) -> [absolute_width]
    = ret![initial_width * relative_width / 100];
}

// aspect_ratio = absolute_width / absolute_height
constraint AspectRatio {
    c(absolute_height: &i32, absolute_width: &i32) -> [aspect_ratio]
    = ret![*absolute_width as f64 / *absolute_height as f64];
    a(aspect_ratio: &f64, absolute_height: &i32) -> [absolute_width]
    = ret![(*aspect_ratio * *absolute_height as f64) as i32];
    b(aspect_ratio: &f64, absolute_width: &i32) -> [absolute_height]
    = ret![(*absolute_width as f64 / *aspect_ratio) as i32];
}
}
};


```

4.4.2 Wrapping the Constraint System

We require both integers and floats, and must thus use a sum type as the value type in the constraint system. Again, we can use `hotdrink_wasm::constraint_system_wrapper!` to generate the required types. This will generate an `enum Number` to represent the sum of our two required types (`i32` and `f64`), a wrapper `struct NumberWrapper` to let us construct the variants from JavaScript, and the constraint system wrapper `struct NumberCs`.

```
// lib.rs
hotdrink_wasm::constraint_system_wrapper! {
    pub struct NumberCs {
        pub struct NumberWrapper {
            #[derive(Debug, Clone)]
            pub enum Number {
                i32,
                f64
            }
        }
    }
}
```

NumberCs can then be used from JavaScript with a very similar API to ConstraintSystem, but we first require an instance of the type.

```
// lib.rs
#[wasm_bindgen]
pub fn create_image_scaling_cs() -> NumberCs {
    let component = ...;
    let mut cs = ConstraintSystem::new();
    cs.add_component(component);
    NumberCs::wrap(cs)
}
```

We can then call `create_image_scaling_cs` from JavaScript to create the constraint system.

4.4.3 Compilation to WebAssembly

Compiling the library to WebAssembly should be as simple as running `wasm-pack build` (if the template was used). The result is an [npm package](#) placed in the `./pkg` directory that can be imported from JavaScript.

4.4.4 Importing WebAssembly from JavaScript

How exactly the compiled WebAssembly is imported depends largely on the project setup; the instructions here do not apply to all cases.

First of all, WebAssembly must be imported asynchronously. One can for instance use `import`, and then add a callback in which the loaded WebAssembly can be used.

```
import("./pkg").then(wasm => { /* Use the WebAssembly module here */ })
```

Alternatively one can have a [webpack](#) entry point called `bootstrap.js` with the following contents:

```
import("./index.js")
```

And then use the import statement in index.js:

```
import * as wasm from "./pkg";
```

A more general way of loading WebAssembly with WebAssembly.instantiateStreaming is explained in Mozilla's documentation [32], and does not depend on the WebAssembly being compiled to an npm package.

4.4.5 Usage from JavaScript

After importing the WebAssembly code, we can construct our constraint system by using the function create_image_scaling_cs that we defined earlier.

```
// Construct constraint system
let cs = wasm.create_image_scaling_cs();

// Send information about GUI interaction to the constraint system
let absoluteWidth = document.getElementById("absoluteWidth");
absoluteWidth.addEventListener("input", v => {
    // Turn the string into a number
    let parsed = parseInt(v);
    // Turn the number into a Number (the Rust type we defined)
    let wrapped = wasm.NumberWrapper.i32(parsed);
    // Send the value to the constraint system
    cs.edit("ImageScaling", "absolute_width", wrapped);
    // Solve the system
    cs.solve();
});
// ...

// Receive updates from the constraint system
cs.subscribe("ImageScaling", "absoluteWidth", v => {
    absoluteWidth.value = v;
});
```

This binding has to be done for all variables. It can quickly become very repetitive. This, binding variables from the GUI to the constraint system can be abstracted away in a function bind.

```
// Create a two-way binding between the GUI and the constraint system
function bind(component, variable, convert) {
```

```

let box = document.getElementById(variable);
box.addEventListener("input", () => {
    cs.edit(component, variable, convert(box.value));
    cs.solve();
});
cs.subscribe(component, variable, v => { box.value = v; })
}

// If the value type has an `i32` variant
function bindInt(component, variable) {
    return bind(component, variable, s => wasm.ValueWrapper.i32(parseInt(s)));
}

// If the value type has a `String` variant
function bindText(component, variable) {
    return bind(component, variable, s => wasm.ValueWrapper.String(s));
}

```

Since the implementation of such binders can vary a lot depending on the program's needs, they are not included in the library. For instance, for this particular example, only `bindInt` is required.

Given the above binders, and that we have defined the HTML elements with names corresponding to the constraint system variables, we can bind the variables of the GUI to the ones in the constraint system.

```

bindInt("initial_height");
bindInt("initial_width");

bindInt("absolute_height");
bindInt("absolute_height_range");

bindInt("absolute_width");
bindInt("absolute_width_range");

bindInt("relative_height");
bindInt("relative_width");

```

To enable optional preservation of the aspect ratio, we can bind a checkbox to the `aspect_ratio` variable. This is done by listening for changes to the checkbox, and either pinning or unpinning the variable as appropriate.

```
let aspectRatio = document.getElementById("aspect_ratio_checkbox");
```

```

aspectRatio.addEventListener("change", () => {
  if (aspectRatio.checked) {
    cs.pin(component, "aspect_ratio");
  } else {
    cs.unpin(component, "aspect_ratio");
  }
});

```

The complete image scaling example can be found in the project repository. The GUI allows for interaction with any of its widgets, and it then automatically updates the other widgets. An image with a size that corresponds to the specified values is also shown and updated in real-time. Using the “preserve ratio” checkbox ensures that the ratio between the height and width does not change when editing values, and leads to a normal-looking image as seen in Figure 4.3. Unchecking the “preserve ratio” checkbox can give results like the one seen in Figure 4.4.



Figure 4.3: Scaling an image while preserving its aspect ratio.

4.5 Benefits of Multithreading

Multithreaded constraint satisfaction method execution simplifies the process of making a *reactive system* [47], as some of the properties required for a reactive system show up automatically. The system is

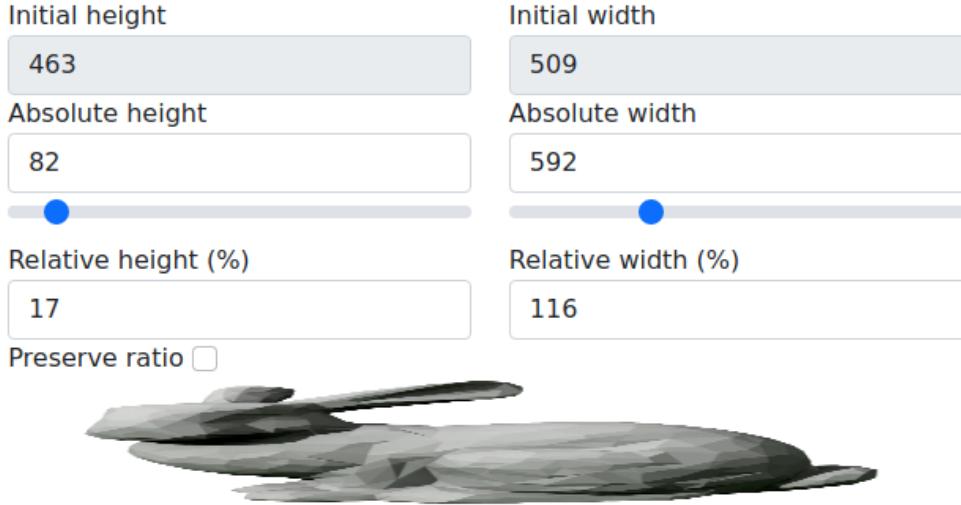


Figure 4.4: Scaling an image without preserving its aspect ratio.

automatically responsive: while individual results of computations can take an arbitrary amount of time to complete, they do not block the entire system. The status of the system can at all times be shown to the user, who is free to make other edits while waiting. The system is also more resilient: bugs that would usually make the system unresponsive, e.g., non-terminating operations, will only block a single thread, which is eventually replaced. Errors also only propagate to relevant outputs, and a good state can be restored by further edits from the user.

“Tackling the Awkward Squad for Reactive Programming” brings up some relevant concerns for reactive programming [72]. The first is called the *Reactive Thread Hijacking Problem*: long-lasting computations in a single-threaded environment can prevent the reactive program from responding to new input. This issue is solved by executing methods outside of the main thread, which is implemented as a form of *hidden concurrency* [72, p. 7] in hotdrink-rs and hotdrink-wasm. The second issue brought up is computations with side effects. Strange behavior due to this is not prevented in hotdrink-rs or hotdrink-wasm, and depends on a correct specification of the constraint graph. If a variable of the constraint system is used within a method without declaring it, the system provides no guarantee that it is updated in time. The plan can change each time the system is solved, which can cause the variable to sometimes be computed before it is used, and other times after it is used, potentially breaking constraints. However, only declared variables will be available in method bodies. The programmer would have to specifically extract the variable from the constraint system and arrange access to it by other means, so it should be difficult to accidentally run into this error.

4.5.1 Parallel Execution

Heavy computations in constraint satisfaction methods in a single-threaded constraint system can quickly lead to delays that are noticeable to the user. Multithreaded execution provides two main benefits in such instances:

1. The main thread is not blocked, as all computations happen on a different thread.
2. Independent computations can be executed in parallel, which provides huge potential speedups.

The former allows the programmer to focus less on the performance,³ and enables implementation of features that would otherwise block the GUI for too long. The latter may be useful for slower operations, such as processing of large images or graphs, or for any other code that requires the processing of a lot of data.

4.5.2 Guaranteed Responsiveness

Small mistakes or unexpected edge cases in algorithms can easily lead to infinite loops. The example below is rather simple, but has one major mistake: the domain of the function does not match the domain of the factorial operation. In this example, the signed integer type `i64` is used. This allows negative numbers to be passed in as arguments, which is not handled by the function and results in an infinite loop.

```
fn factorial(mut n: i64) -> i64 {
    let mut factorial = 1;
    while n != 0 {
        factorial *= n;
        n -= 1;
    }
    factorial
}
```

If the developer of the function only tests for correct inputs (natural numbers), then the implementation appears to work. However, if the input is provided by a user, and the function is run on the main thread, then control will never be returned to JavaScript and the event loop. The GUI will thus be completely unresponsive, and the user has no way of undoing their change.

However, if we run the function on another thread, such as when a constraint satisfaction method is executed in a multithreaded constraint system, this issue will be non-existent. Naturally, the result of the operation will never be computed, but the user is free to change any other part of the GUI, even the input to the factorial function. The old computation may then be canceled, and replaced with the new one.

³However, this is not necessarily always a good thing [86].

While this example is trivial, the issue can easily arise in more complex algorithms; many graph algorithms must explicitly handle cycles to guarantee termination. For instance, if we run the Bellman-Ford shortest path algorithm until it finds no further improvements, it will never terminate for negative cycles. Even something as simple as a depth-first search might not terminate if visited nodes are not marked correctly.

Multithreaded method execution can thus ensure that a GUI remains responsive and partially usable in the presence of a bug that leads to non-termination, or even just long computations.

4.6 Multithreading with Rust and WebAssembly

Unfortunately, using multithreading with Rust and WebAssembly is not as simple as using the standard library or thread pools from `rayon` [45]. The standard library threads cannot be compiled to WebAssembly, and the WebAssembly threads proposal is still a work in progress [70]. There is, however, a way to get around this [8] by using Web Workers, but this has a number of limitations.

4.6.1 Limitations

Web Workers are intended to communicate with the main thread through message passing, and while most JavaScript types can be sent without issue [68], there are some exceptions. For instance, when receiving a JavaScript function object in Rust, it is exposed as an object of the `js_sys::Function` type, which allows for storing and calling the function. This type, however, is not thread-safe for the same reason as `wasm_bindgen::JsValue` is, namely that the values are managed by `wasm-bindgen` in a way that does not allow them to cross thread-boundaries [29, 10]. The types `JsValue` and `Function` thus do not implement `Send`, which is a *marker trait* that marks types that are safe to send across threads. Attempting to do so with values of a type that does not implement `Send` will fail at compile-time.

Web Workers are also capable of communicating via memory sharing by using `SharedArrayBuffer` [59], though this type has gone through multiple changes from 2018 until 2020 [59] due to Meltdown and Spectre [60, 30], and was even disabled for a time [34]. `SharedArrayBuffer` can be used in the implementation of Web Worker-based threads that can execute Rust closures similarly to `std::thread::spawn`.

While we *can* terminate Web Workers, thread termination causes some issues when running Rust code. When terminating a worker, allocated memory may not be deallocated properly for values that have not gone out of scope [46, 8]. This means that the terminated workers leak memory, and that terminating a thread at the wrong time may leave locks permanently locked, which may lead to deadlocks.

4.6.2 Web Worker-Based Threads

Alex Crichton's article about multithreading with Rust and WebAssembly [8] and his parallel raytracing demo [46] have been very useful in creating an appropriate Rust abstraction over Web Worker-based

threads. Ingvar Stepanyan’s article “Using WebAssembly threads from C, C++ and Rust” also provides a valuable, up-to-date overview of the current state of using WebAssembly threads in other languages [61].

There are already a few crates for using Web Worker-based threads, such as `wasm-mt`, `web_worker`, `wasm_thread`, and `wasm-bindgen-rayon`. These do not, however, expose a way to terminate workers, a feature we are interested in experimenting with [77, 80, 79, 76]. Creating our own implementation allows us to customize it to fit our needs.

We begin by creating a data structure that represents the work we would like to execute in another thread, and store a closure within it. The work — or more specifically, a pointer to it — can then be sent to a Web Worker, which calls `generic_worker_entry_point` to send it back to Rust. Any code executed within `generic_worker_entry_point` is then executed in the Web Worker’s associated thread.

```
pub struct Work {  
    func: Box<dyn FnOnce() + Send + 'static>,  
}
```

The `generic_worker_entry_point` implementation is rather simple; it casts the pointer (a `u32`) back to a `Work`-pointer (`*mut Work`), and then executes the stored closure.

```
#[wasm_bindgen]  
pub fn generic_worker_entry_point(ptr: u32) -> Result<(), JsValue> {  
    let ptr = unsafe { Box::from_raw(ptr as *mut Work) };  
    (ptr.func)();  
    Ok(())  
}
```

To give the Web Worker access to the WebAssembly module [85] (the code), the shared memory [84] (which is implemented using `SharedArrayBuffer`), and the work to perform, we make it execute the script below [87]. It starts by importing the *WebAssembly shim*, a JavaScript file generated by `wasm-bindgen` that defines the `wasm_bindgen` global, which simplifies compilation and instantiation of the WebAssembly module. The Web Worker will then wait until it receives the WebAssembly module and the shared WebAssembly memory. The module is then *instantiated* [83], and the Web Worker thus has access to the Rust code and can share memory with other instances of the module (such as the one on the main thread). Once that is done, each message received by the Web Worker should be a pointer to `Work` defined in Rust. Each of these pointers are sent to `generic_worker_entry_point`, which will execute the `Work`’s inner closure.

```
importScripts('WASM_BINDGEN_SHIM_URL');  
  
self.onmessage = msg => {  
    let [module, memory] = msg.data;
```

```

let instantiated = wasm_bindgen(module, memory);
self.onmessage = async msg => {
    await instantiated;
    wasm_bindgen.generic_worker_entry_point(msg.data);
};
};

```

We can then create a new Web Worker that executes the generic script we created, and send it the current WebAssembly module and memory. The Web Worker is then ready to work; we can send it Work that we would like it to perform.

```

let worker = Worker::new("worker_script.js");

// Send module and memory
let array = js_sys::Array::new();
array.push(&wasm_bindgen::module());
array.push(&wasm_bindgen::memory());
worker.post_message(&array)?;

// Send work
let work = Work { func: Box::new(|| { ... }) };
let ptr: *mut Work = Box::into_raw(work);
worker.post_message(&JsValue::from(ptr as u32))

```

To include `worker_script.js` in the library so that developers do not have to do it themselves, we can include the script as a URL encoded blob as done in `wasm_thread`⁴. That is, we include the script itself in the code as a `String`, replace the `WASM_BINDGEN_SHIM_URL` with the actual path, and get a URL to a version of the file stored in memory.

```

use wasm_bindgen::JsValue;
use web_sys::{Blob, Url};

/// Extracts path of the `wasm_bindgen` generated .js shim script
fn get_wasm_bindgen_shim_script_path() -> String {
    js_sys::eval(include_str!("./script_path.js"))
        .unwrap()
        .as_string()
        .unwrap()
}

```

⁴https://github.com/chemistry/wasm_thread/blob/2dfcf37ef1fc9f2392272185a014280464fd028/src/lib.rs

```

}

/// Generates worker entry script as URL encoded blob
pub fn create() -> String {
    let wasm_bindgen_shim_url = get_wasm_bindgen_shim_script_path();

    // Generate script from template
    let template = include_str!("./generic_worker.js");
    let script = template.replace("WASM_BINDGEN_SHIM_URL", &wasm_bindgen_shim_url);

    // Create url encoded blob
    let arr = js_sys::Array::new();
    arr.set(0, JsValue::from_str(&script));
    let blob = Blob::new_with_str_sequence(&arr).unwrap();
    Url::create_object_url_with_blob(&blob).unwrap()
}

```

As mentioned in Section 4.6.1, cancellation of threads may lead to deadlocks if we are not careful. We must ensure that no worker can be terminated while holding a shared lock, and that a worker cannot acquire a shared lock after the termination signal has been sent. In the implementation `PoolWorker`, each worker will begin by locking access to synchronized meta-information about its current state. The worker will then check if the meta-information says that it has been terminated, in which case it will not do any further work. This data is specific to this worker, so if it stops executing while holding the lock, no other workers are affected. The only other party that has access to this lock is the main thread, but if the main thread has terminated the worker, it has also discarded the shared state. If the worker manages to lock the shared state and finds that it has not been terminated, it cannot be terminated until the lock is released. The worker will then wait until it receives more work before it can finally be terminated again. Pseudocode can be seen below, and the full implementation can be seen in the repository in the `spawn` method in `hotdrink-wasm/thread/pool/pool_worker.rs`.

```

// Worker thread
loop {
    let work = {
        shared.outer_lock.lock();
        if shared.terminated {
            break;
        }
        let work = shared_queue.lock().pop();
        shared.ready = false;
    }
}

```

```

    work
};

work.run();

outer_lock.lock();
shared.ready = true;
}

// Main thread
if shared.outer_lock.try_lock() {
    if shared.unused_result && !shared.ready {
        shared.terminated = true;
        worker.terminate();
        worker = Worker::new();
    }
}

```

If the main thread does not get the lock, we know that the thread is busy getting more work, and should not be terminated. If the main thread does get the lock, and decides to terminate the worker, then the worker will observe that the termination flag is set before trying to acquire the task queue lock.

4.6.3 Web Worker-Based Thread Pools

One of the main goals of the library is to make it impossible for poorly implemented constraint satisfaction methods to make the GUI unresponsive. This means that we must execute them in separate threads, but spawning a new Web Worker for each method can become slow and memory intensive. A thread pool that spawns an appropriate amount of workers for the tasks at hand is thus necessary to reach a good compromise. We call the work to be executed in the thread pool a *task*.

Static thread pool

The first alternative is to maintain a static number of workers at all times, as seen in the implementation of the `StaticPool` type in `hotdrink-wasm`. This means that the overhead of starting a new Web Worker, which is approximately 40ms [33] (or around 250ms in our tests, but this includes instantiating the WebAssembly module as well), is avoided when processing new tasks.

The main idea is that executing a task on the thread pool puts the task in a shared queue, from which the workers take tasks when they are ready. However, an issue with this approach is that having n workers only allows up to n parallel tasks, and if none of them terminate, no progress will be made for any other

tasks. The GUI will, however, continue to accept new user input, which may potentially terminate the workers that are “stuck”, at which point new work can be done.

When executing a new task in the thread pool, the meta-information about each worker is accessed to determine if it should be restarted. The conditions for restarting may change depending on which termination strategy has been selected (See Section 4.6.4). Workers that no longer produce a useful result, or have worked for too long, are then sent the termination signal, and the shared state with that worker is discarded. An entirely new worker is then started, replacing the previous one.

Dynamic thread pool

The DynamicPool type reverses the control, making the main thread give the workers tasks instead of having them take tasks when ready. A task may be passed to workers that are considered *ready* (awaiting a task). A worker is considered busy until it completes the task. We cannot send tasks to workers that are not ready, as a task is not guaranteed to terminate; doing so could result in a task never getting to execute.

In the case that there are no ready workers, we must spawn new ones; waiting within the call to Rust would block the main thread, and thus the GUI. Instead, a new worker is spawned for each task that requires it. This guarantees that each task that *can* be executed, actually will be, as opposed to the static pool which may stop if there are sufficiently many tasks that do not terminate.

An issue with the dynamic pool is that if there are no ready workers, then having to spawn a new one may increase the time that computation takes by orders of magnitude. Assuming a task takes 1ms to execute, its execution time may suddenly take over 200 times as long as one would expect if we have to wait for a worker to get started. The cost of the overhead becomes worse when dozens – if not hundreds – of workers are spawned when all methods need to be executed. The memory usage will also increase, especially since the stacks of canceled threads are not cleaned up properly [8].

Our library currently spawns new workers once they are required, and makes no attempt to prepare them ahead of time. A possible optimization is to use a *worker buffer* with a few idle workers; a new task can then be assigned to a worker immediately, and a new worker to take its place is spawned in the background. The buffer could be improved further by attempting to predict how many workers are required.

The programmer can select a thread pool implementation that satisfies their needs. The StaticPool is less resource-intensive, but with n workers, there must not be more than $n - 1$ long-running tasks at once. Otherwise, new tasks must wait for one of them to be completed. To guarantee that all tasks that can run in parallel actually are, the DynamicPool must be used, at the cost of more memory usage and computational overhead when spawning many workers.

4.6.4 Termination Strategies

There are many different ways of deciding when a worker should be terminated. Terminating too often, and we may have to replace them more often than we should. Terminating too infrequently, and we may end up with many unnecessary workers and computations wasting processing power and memory.

Never With this strategy, workers are never terminated. Long-running tasks will continue until completion, and non-terminating tasks will continue indefinitely.

Unused result and not done For a worker to be terminated with this strategy, its result must no longer be needed, and the worker must still be working on its task. The first requirement guarantees that stopping the computation does not lead to missing results, and the second avoids the additional overhead of starting a new worker when it can be reused. An issue with this strategy is that workers may be very close to being done and still be terminated. It may be more efficient to let a worker finish a task instead of aborting the task early, but we cannot know how long the task will take to complete.

Adding a timer requirement Adding to the previous strategy, we can add a configurable timer requirement for terminating workers. For instance, we could only terminate workers that were assigned a computation where the result is no longer needed, are not done, and have been working for more than 50ms. There is still a chance that such workers finish at the moment we terminate them, but this should be rare.

An issue with this strategy is that a potentially long-running task will not be terminated if a new value is given too quickly. Let us say we perform two edits in the GUI in quick succession: the first computation may no longer be relevant, but as it has been running for such a short time, we do not terminate the worker. This could result in many workers being occupied with computing results that are not needed.

4.7 Data Flow in a Multithreaded Constraint System

The issues described in Section 4.6.1 prevent us from implementing the multithreaded constraint system as straightforwardly as desired. Specifically, not being able to send values of type `js_sys::Function` across threads means that we cannot implement subscribing as easily as in `hotdrink-rs`. Adding a callback to a `ConstraintSystem` that calls a `js_sys::Function` would fail, as the call is done within the provided thread pool implementation. We thus had to find a workaround that allows the event handlers defined in JavaScript to be executed on the main thread, which leads to a complex data flow. An overview of this data flow can be seen in Figure 4.6, and may be helpful to use as a reference while reading the rest of this section.

The solution to the problem makes heavy use of Web Workers; understanding how they communicate with the main thread is very useful when trying to understand it. First of all, assuming `w` is a web worker, messages can be sent to it with `w.postMessage(msg)`. To receive messages from it, a callback is defined as follows: `w.onmessage = e => { ... }`. The worker can define a callback `self.onmessage = e => { ... }` to specify what to do upon receiving a message, and `self.postMessage(msg)` to send messages to subscribers of the worker. See Figure 4.5 for a visual representation of this.

During the solving of a system, the methods of the plan will eventually be scheduled to be executed in a Web Worker-based thread pool capable of executing Rust code. After all of them have been scheduled,

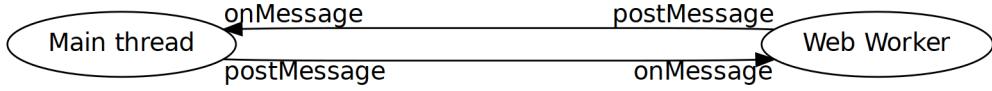


Figure 4.5: Web Worker message passing.

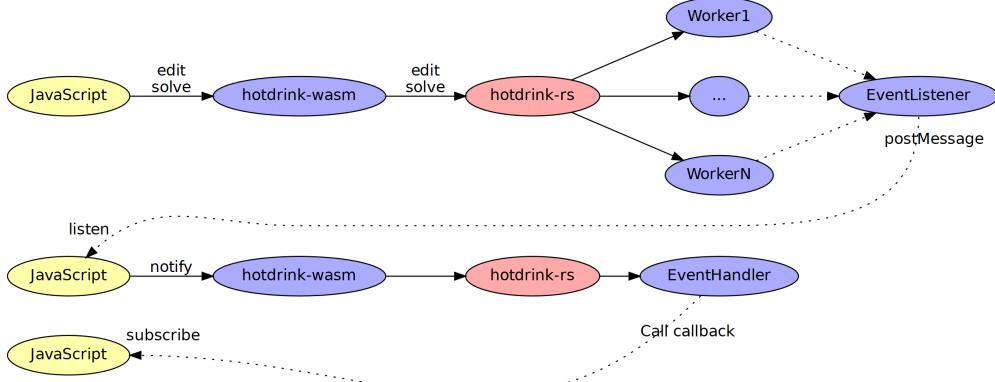


Figure 4.6: Data flow between JavaScript (yellow), hotdrink-wasm (blue) and hotdrink-rs (red). Dotted lines represent data being “sent back” via channels or callbacks; the tail label represents what the sender does to send it, and the head label represents what the receiver does to receive it.

the call to `solve` is completed, and the main thread can respond to user events while methods are being computed in the background. Upon completing the execution of a method, the responsible Web Worker will send an event to the `EventListener`.

The `EventListener` is a Web Worker-based thread that listens to events on a `Receiver`⁵, the receiving end of a Rust channel. It will then “publish” this event to its subscribers with the Web Worker message passing mechanism. The `EventListener` can be subscribed to by calling `listen` on the constraint system, at which point the event is on the main thread. This is very useful, as we can now send the event back to the constraint system with `cs.listen(e -> cs.notify(e.data))`, which will call the callbacks appropriate for the event.

We thus manage to call JavaScript callbacks on events by using Web Worker message passing to send events back to the main thread (without blocking it), while executing methods in parallel.

4.8 Generating a Multithreaded Constraint System

In the same way as `constraint_system_wrapper!` generates a WebAssembly-compatible wrapper for a single-threaded constraint system, `constraint_system_wrapper_threaded!` generates one for multithreaded constraint systems. One of the main differences between the two is that the latter requires that the programmer specifies a thread pool implementation, the number of threads, and which termination

⁵<https://doc.rust-lang.org/std/sync/mpsc/struct.Receiver.html>

strategy to use.

```
crate::constraint_system_wrapper_threaded! {
    pub struct ConstraintSystemWrapper {
        pub struct IntOrStringWrapper {
            #[derive(Clone, Debug)]
            pub enum IntOrString {
                i32,
                String
            }
        }
        thread_pool: StaticPool,
        num_threads: 4,
        termination_strategy: TerminationStrategy::UnusedResultAndNotDone
    }
};
```

Other than this difference, the usage of the macro is the same.

The generated code has to be used a bit differently than the output of `constraint_system_wrapper!`, and we recommend reading Section 4.7 first to get a better understanding of why.

In addition to the inner constraint system, and an event handler that stores JavaScript callbacks, the `constraint_system_wrapper_threaded!` must store an `EventListener` and the thread pool to use. As described in Section 4.7, the `EventListener` runs a Web Worker-based thread that listens for all events in the constraint system. With Web Workers, it is possible to attach a JavaScript callback that handles messages sent from it, which is a way of notifying the main thread of events that have happened. The user of the generated constraint system must be sure to call `cs.listen(e -> cs.notify(e.data))` to capture events that the event listener received, and to send them back to the constraint system to be handled. The `listen` function attaches a callback by setting the `EventListener` Web Worker's `onmessage` property, and the `notify` function sends the event to the constraint system. The JavaScript callbacks must always be called from the main thread; this workaround lets the main thread know when events it must react to have happened in the worker threads.

The `subscribe` function is very similar to the subscription mechanism seen in Section 4.3, other than instead of adding events to a queue to be handled immediately, they are sent to the `EventListener`, which will use `postMessage` to send the event to the main thread by calling the callback added with `listen`.

```
inner.subscribe(component, variable, |event| { event_listener_channel.send(event); });
```

The call to `subscribe` can complete before all events have been generated, and the main thread is free to continue tasks such as updating the GUI and handling user input.

```

1 let cs = new ConstraintSystem();
2 let comp = wasm.get_component(); // Get a pre-made component
3 let box = document.getElementById("a");
4 box.addEventListener("input", () => {
5     box.value = comp.edit("a", box.value) // A reference is needed here
6     cs.solve();
7 });
8 cs.add_component(comp); // `comp` is moved, and can't be used further

```

Listing 4: Use-after-move error in JavaScript.

Other than solve using the stored thread pool implementation, such as a `StaticPool` or `DynamicPool` instead of `DummyExecutor`, the rest of the macros are nearly identical.

4.9 Pitfalls

4.9.1 Use after Move

Constraint systems can be modified during runtime from JavaScript like they can be modified from Rust, but since JavaScript does not have a borrow checker, some surprising errors can occur when using the library.

Trying to use a value after it has been moved in Rust will result in a compile-time error similar to the one seen in Figure 4.7. Making the same mistake in JavaScript will instead result in a runtime error without a clear explanation. A simple example is the code seen in Listing 4, where calling `edit` on a component

```

error[E0382]: use of moved value: `component`
--> hotdrink-rs/examples/scheduling.rs:49:19
4 |     let component = component! {
... |         ----- move occurs because `component` has type `hotdrink_rs::model::Component<i32>`, which does not implement the `Copy` trait
48|     cs.add_component(component);
49|     use_component(component);
... |         ----- value moved here
... |         ^^^^^^^ value used here after move

```

Figure 4.7: Use after move in Rust.

after it has been added to the constraint system would result in a crash. This is because the component is moved on line 8, and the reference is no longer valid for the lifetime of the callback that runs line 5. When an edit is made and the callback is executed, the invalid value of `comp` is passed to Rust, producing an exception.

4.9.2 Breaking the Borrowing Rules

The borrowing rules are enforced at runtime when interacting with Rust from WebAssembly. Programs that would fail to compile in Rust will instead result in error messages such as “recursive use of an object

detected which would lead to unsafe aliasing in rust". This can happen in cases where we call a function like the one below from JavaScript.⁶.

```
#[wasm_bindgen]
fn foo(a: &mut T, b: &T) { ... }
```

If we call this function from JavaScript with the same argument twice, like `foo(x, x)`, we have a mutable and an immutable reference to the same value, which breaks the borrowing rules of Rust. Since the Rust compiler is cannot verify our JavaScript code, we get a runtime error.

In earlier implementations, methods on the constraint system wrappers required a mutable reference to modify it. If we then called another method within the callback provided to `subscribe`, we could end up in a situation where the constraint system was mutably borrowed twice.

```
cs.subscribe(..., () => { cs.edit(...); })
```

In the code above, we need a mutable reference to `cs` to add a callback, and if the callback is called before `subscribe` returns, `edit` requires another mutable reference.

This issue can be solved with *interior mutability* [21, 48]. The fields that need to be mutable are wrapped in a `Mutex`, which allows us to modify the value without having to mutably borrow the owner of the value (the constraint system wrapper). We then only require immutable references, which we can have multiple of without breaking Rust's borrowing rules. We do, however, have to be careful not to deadlock by attempting to lock the fields multiple times.

⁶Inspired by the issue at <https://github.com/rustwasm/wasm-bindgen/issues/1578>

Chapter 5

C/C++ Bindings

Since `hotdrink-rs` is implemented in pure Rust, we are not limited to compiling it to WebAssembly. `hotdrink-c` is a proof-of-concept library that demonstrates how a constraint system made with `hotdrink-rs` can be compiled to and used from C.

5.1 Creating a Dynamic Library

To compile a crate to a dynamic library that can be used from C, we add the following option to `Cargo.toml`.

```
[lib]
crate-type = [ "cdylib" ]
```

Running `cargo build` will then produce a dynamic library in the `target` folder, which can then be linked to a C program, e.g., with `gcc main.c target/libfoo.so`. An appropriate C header file can be generated with `cbindgen`¹, or written manually if desired. More information about the interoperability between Rust and C can be found in [The Embedded Rust Book](#).

5.2 Creating a C-Compatible Constraint System

The `hotdrink-c` library has the same typing problem as `hotdrink-wasm`: we cannot expose the `Component<T>` type directly. This struct is not made with support for C in mind, and it does not follow the C [ABI](#). `Component<T>` also has a generic type parameter, which causes another problem: the type is monomorphized at compile-time, meaning that the type parameter must be determined by that point. One solution to this is to wrap types like `Component<i32>`, for instance by creating a type `Component_i32` as follows:

¹We can also go the other way with `rust-bindgen`, and to WebAssembly with `wasm-bindgen` like in Chapter 4.

```

struct Component_i32 {
    inner: Component<i32>,
}

```

We can then expose an *opaque pointer* to this type (as described in the Rustonomicon [53]) to hide the inner `Component<i32>` from C, and then create an API that operates on the pointer instead. While not necessary for this example, the same could be done for `ConstraintSystem<T>`.

5.2.1 Construction and Destruction

Regarding the API, we will begin with a way to create a new component. We define a function `component_new` that returns a pointer to a `Component_i32`, which the rest of the API will continue to work with. For practical usage, one would likely create a separate constructor for each different component, e.g., one for representing a Rectangle and another for a sign-up form.

```

#[no_mangle]
pub extern "C" fn component_new() -> *mut Component_i32 {
    let my_component = hotdrink_rs::component! { ... };
    Box::into_raw(Box::new(my_component))
}

```

The `component_new` function places our struct on the heap with `Box`, and returns a pointer to use from C. To avoid memory leaks, we must also provide a way to free this memory from C. Since the memory was allocated by a `Box`, it should also be freed by one [4]:

```

#[no_mangle]
pub unsafe extern "C" fn component_free(component: *mut Component_i32) {
    Box::from_raw(component);
}

```

We can then use our newly defined type and its API from C as follows:

```

Component_i32 *component = component_new();
component_free(component);

```

5.2.2 Subscribing

To be able to do anything interesting with our functions for creating and destroying constraint systems, we must be able to subscribe to variables. We enable this by defining a function called `component_subscribe`. Besides some string conversion, the `component_subscribe` function is relatively straightforward, and does not do much more than calling methods on the inner `Component<i32>`.

```

#[no_mangle]
pub unsafe extern "C" fn component_subscribe(
    component: *mut Component_i32,
    variable: *mut c_char,
    callback: extern "C" fn(i32),
) {
    let variable = CStr::from_ptr(variable).to_str().unwrap();
    (*component).inner.subscribe(variable, move |e| {
        if let Event::Ready(value) = e {
            callback(*value)
        }
    });
}

```

Note that many errors are deliberately ignored with `unwrap()` to simplify the code. We also assume that the functions are used correctly, i.e., a valid pointer is passed in.

5.2.3 Editing

We can then create a function `edit` for performing edits to variables. If the values in the constraint system can have multiple different types, we can create more variants of the function, such as `component_edit_f64`, or alternatively create a wrapper type like in Section 4.2.3.

```

#[no_mangle]
pub unsafe extern "C" fn component_edit(
    component: *mut Component_i32,
    variable: *mut c_char,
    value: i32,
) {
    let variable = CStr::from_ptr(variable).to_str().unwrap();
    (*component).inner.edit(variable, value).unwrap();
}

```

5.2.4 Solving

Finally, we need to be able to solve the component after making an edit. This can easily be done with a call to `solve` on the inner `Component<i32>`.

```

#[no_mangle]
pub unsafe extern "C" fn component_solve(component: *mut Component_i32) {

```

```
(*component).inner.solve().unwrap();  
}
```

5.3 Using the API

With the features described above, the API is complete enough to demonstrate its complete usage from C. Note that since C does not have anonymous functions, passing in callbacks is not as streamlined as it would be in many other languages. This would be even worse if the example implementation from this chapter required handlers for the other event types. To improve the ergonomics of writing event handlers, one could make a custom `Callback` struct with optional handlers per event, but the following is sufficient for this example:

```
#include <stdio.h>  
  
void callback_a(x: i32) { printf("a = %i\n", x); }  
void callback_b(x: i32) { printf("b = %i\n", x); }  
void callback_c(x: i32) { printf("c = %i\n", x); }  
  
int main(void) {  
    Component_i32 *component = component_new();  
    component_subscribe(component, "a", callback_a);  
    component_subscribe(component, "b", callback_b);  
    component_subscribe(component, "c", callback_c);  
    component_edit(component, "a", 10);  
    component_solve(component);  
    component_free(component);  
}
```

Running the program will now give the following output (comments added for clarity).

```
// Initial values  
a = 0  
b = 0  
c = 0  
// Set a to 10  
a = 10  
// After solve  
c = 10
```

Chapter 6

Performance Analysis

This chapter describes the performance of `hotdrink-rs` and how it was measured. The performance is also compared with the performance of earlier implementations of HotDrink.

6.1 Constraint Systems Used in Benchmarks

A selection of constraint systems with different “shapes” were constructed to test the different implementations’ on their worst- and best-case scenarios. Each of the following sections describes a specific kind of constraint system and why it was chosen.

6.1.1 Linear-oneway

The linear-oneway system seen in Figure 6.1 is for testing how well the implementation manages to avoid unnecessary calls to the simple planner. An optimal implementation will only require a single call before it detects that no other stay constraints may be added, and then terminates.



Figure 6.1: Linear-oneway.

For this system, a stay constraint added to any variable but `var0` will fail. If a stay constraint is added to `var3`, then the constraint between `var2` and `var3` cannot be enforced since its only method also writes to `var3`. This would make two methods write to the same variable, which would not result in a valid solution graph. Optimizations should avoid trying to add stay constraints to variables where it would not succeed.

6.1.2 Linear-twoway

The linear-twoway system seen in Figure 6.2 is an extension of the linear-oneway system. Once a single constraint is enforced, the direction of the data flow can be determined, avoiding further calls to the simple

planner. If any variable in the chain gets a stay constraint added to it, then the other constraints it is attached to must pick the methods that read from that variable, and the flow will lead away from it. Since all the other variables are then definitely written to, no more stay constraints may be added.

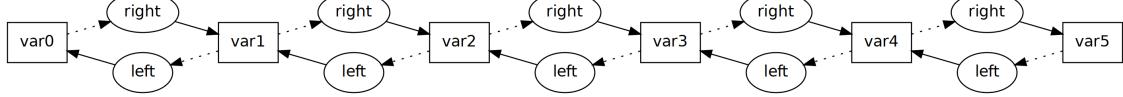


Figure 6.2: Linear-twoway.

For instance, if a stay constraint is successfully added to var_3 , then the constraint between var_2 and var_3 cannot be enforced with *right*, as that would result in two writes to the same variable. The same applies to *left* from the constraint between var_3 and var_4 . This means that the two constraints that var_3 are connected to each have a single method remaining, and those may then be selected. Selecting them will in turn cause var_2 and var_3 to be written to, which eliminates even more methods. This continues until all methods not in the solution graph are eliminated.

6.1.3 Ladder

The ladder system in Figure 6.3 is more difficult for the pruner to handle, with the dataflow direction not being uniquely determined unless specific variables are definitely written to. This gives us an idea of how the planner fares when pruning is not necessarily possible. The ladder constraint system is taken from the test models in the repository of the TypeScript implementation of HotDrink [31].

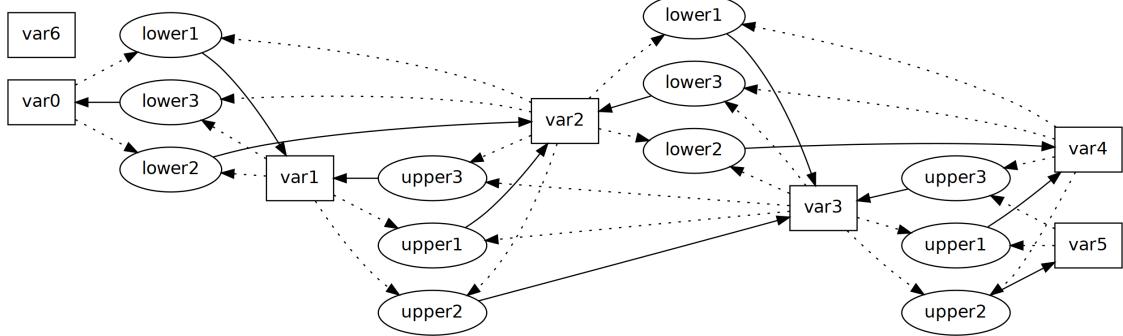


Figure 6.3: Ladder.

Adding a stay constraint to var_0 eliminates the method lower_3 that writes to it, but there are still two more methods left: lower_1 and lower_2 . This means that we cannot cascade pruning through the entire system. We cannot eliminate lower_1 or lower_2 until we add a stay constraint to either var_1 or var_2 .

6.1.4 Unprunable

The “unprunable” system in Figure 6.4 is made specifically to make pruning as difficult as possible. Once a stay constraint is added, at most one method can be eliminated, and the change will not cascade as it does in the linear systems. This gives us a worst-case example for planning and shows how important it is that the simple planner is fast, since it may be called many times.

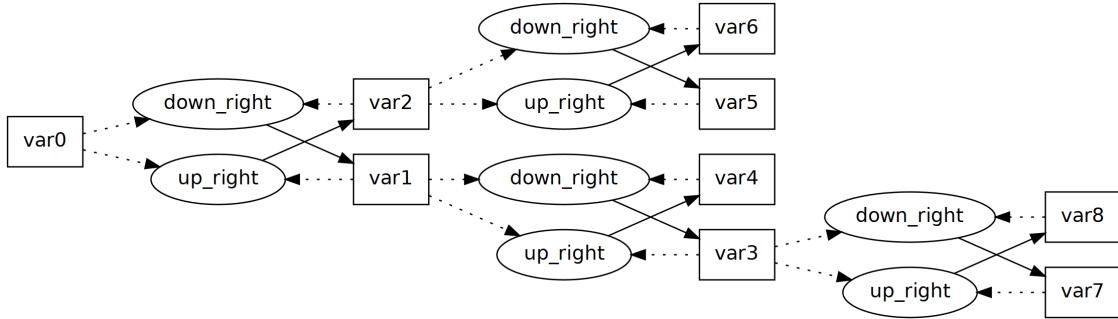


Figure 6.4: Unprunable.

Adding a stay constraint to var_0 will not eliminate $down_right$ nor up_right from the constraint between var_0 , var_1 and var_2 . It is not until we add a stay constraint to var_1 or var_2 we can remove any of the methods. Similarly, we cannot eliminate any methods from the constraint between var_1 , var_3 and var_4 until either var_3 or var_4 gets a stay constraint.

6.1.5 Random

While it is difficult to predict how the library will be used in the real world, attempting to generate a partially random constraint system can provide some insight into the expected performance. The algorithm is implemented for `hotdrink-rs` and reused in benchmarks for WebAssembly, and also implemented for the Flow and TypeScript implementations of HotDrink. To guarantee that the resulting system can be solved, the algorithm works similarly to a reverse planner. We know that there must at all times be a free variable with an accompanying free method, and start by adding that.

At the start, there are no constraints, which means that the system is trivially solvable. For each iteration, we write to a previously unused variable, which will then make it a free variable. We also write to a previously used variable to encourage the graph to be connected. This means that at this point, we can enforce the constraint by selecting the method that writes to that variable. The next iterations may make the previously free variable connected to multiple constraints, but will always add a new free variable. To solve the system, all we have to do is to select the method that writes to the latest free variable in the last constraint that was added. The actual implementation allows for setting the number of variables per constraint, and how many methods it can have. This provides a little more variation in the constraint systems. A constraint system generated by the algorithm can be seen in Figure 6.5.

Algorithm 1: RandomConstraintSystem

Input: The desired number of constraints
Output: A random constraint system of the requested size

```
1 used_variables  $\leftarrow \{0\}$ ;  
2 unused_variables  $\leftarrow \{1 \dots num\_constraints\}$ ;  
3 while actual constraints < desired constraints do  
4   used  $\leftarrow$  a random used variable;  
5   if unused_variables =  $\emptyset$  then  
6     | add a fresh unused variable  
7   end  
8   unused  $\leftarrow$  a random unused variable;  
9   others  $\leftarrow$  a set of random variables;  
10  c  $\leftarrow$  constraint between  $\{used, unused\} \cup others$ ;  
11  c.add(m(others  $\rightarrow$  used));  
12  c.add(m(others  $\rightarrow$  unused));  
13  for v  $\in$  others do  
14    | c.add(m(others  $\setminus v \rightarrow v$ ));  
15  end  
16  add c to set of constraints  
17 end
```

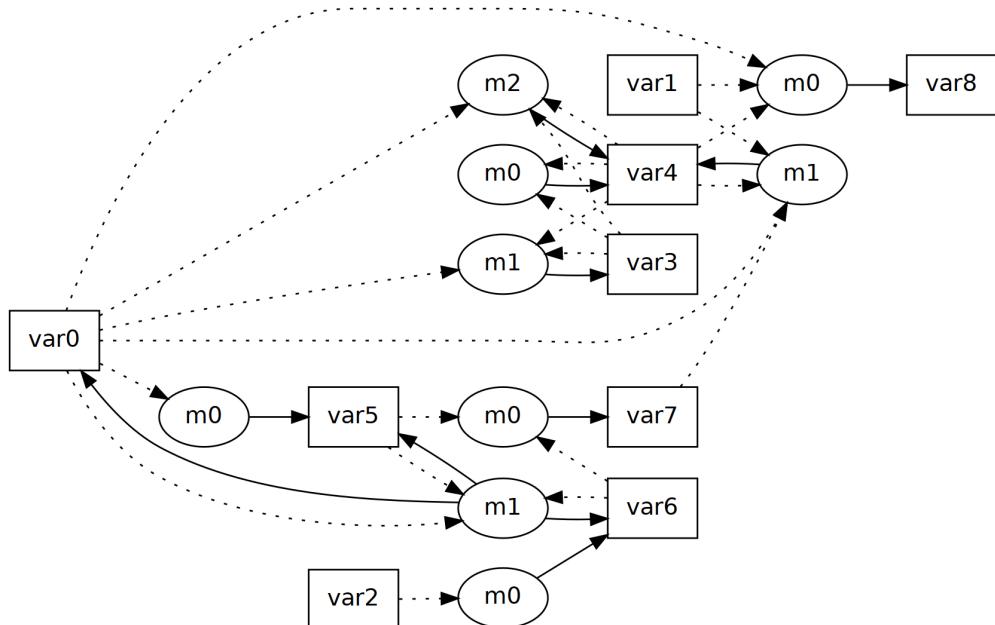


Figure 6.5: A randomly generated constraint system.

Table 6.1: Simple planner benchmarks in milliseconds (CPU: Intel i5–8265U).

Type	# of constraints	hotdrink-rs
Linear-oneway	75000	27
Linear-twoway	75000	28
Ladder	75000	22
Random	75000	92
Unprunable	75000	68
Linear-oneway	100000	39
Linear-twoway	100000	40
Ladder	100000	29
Random	100000	139
Unprunable	100000	90

6.2 Optimization Methodology

Using flamegraphs [5] to get a better understanding of the performance has been vital for the optimization of `hotdrink-rs`. Guessing which part of the code is the bottleneck is unlikely to give good results, as what we expect to be slow is often optimized away [66].

The main strategy for optimizing `hotdrink-rs` was to begin by generating a flamegraph of the planner or solver to find areas of the code to optimize. Through the development of the library, we came across several functions that took up a disproportionate amount of time, either from performing more work than required, or too many allocations, making them clear optimization targets. After attempting to optimize the functions, the performance was measured again with a new flamegraph and the statistical benchmarking with `Criterion.rs` to verify potential improvements.

A flamegraph of a call to the `solve` method can be seen in Figure 6.6. We see that the simple planner takes up about half the execution time for this particular constraint system, meaning that any improvements made to it would likely affect the hierarchical planner a fair amount as well.

6.3 Simple Planner Benchmarks

For our benchmarks, we chose a number of constraints that would take approximately 100 milliseconds to find a plan for, since this is around the time where GUI responsiveness is perceived as instantaneous [40, p. 135]. Note that we would also have to have time to execute the plan.

As seen in the results in Table 6.1, the simple planner is capable of finding a solution for constraint systems with more than 75000 constraints in the allotted time. For all systems other than the randomly generated one (which tends to have more methods per constraint), even 100,000 constraints is possible. The results were produced with the command `cargo bench simple_planner`, executed in the `hotdrink-rs/hotdrink-rs` directory.

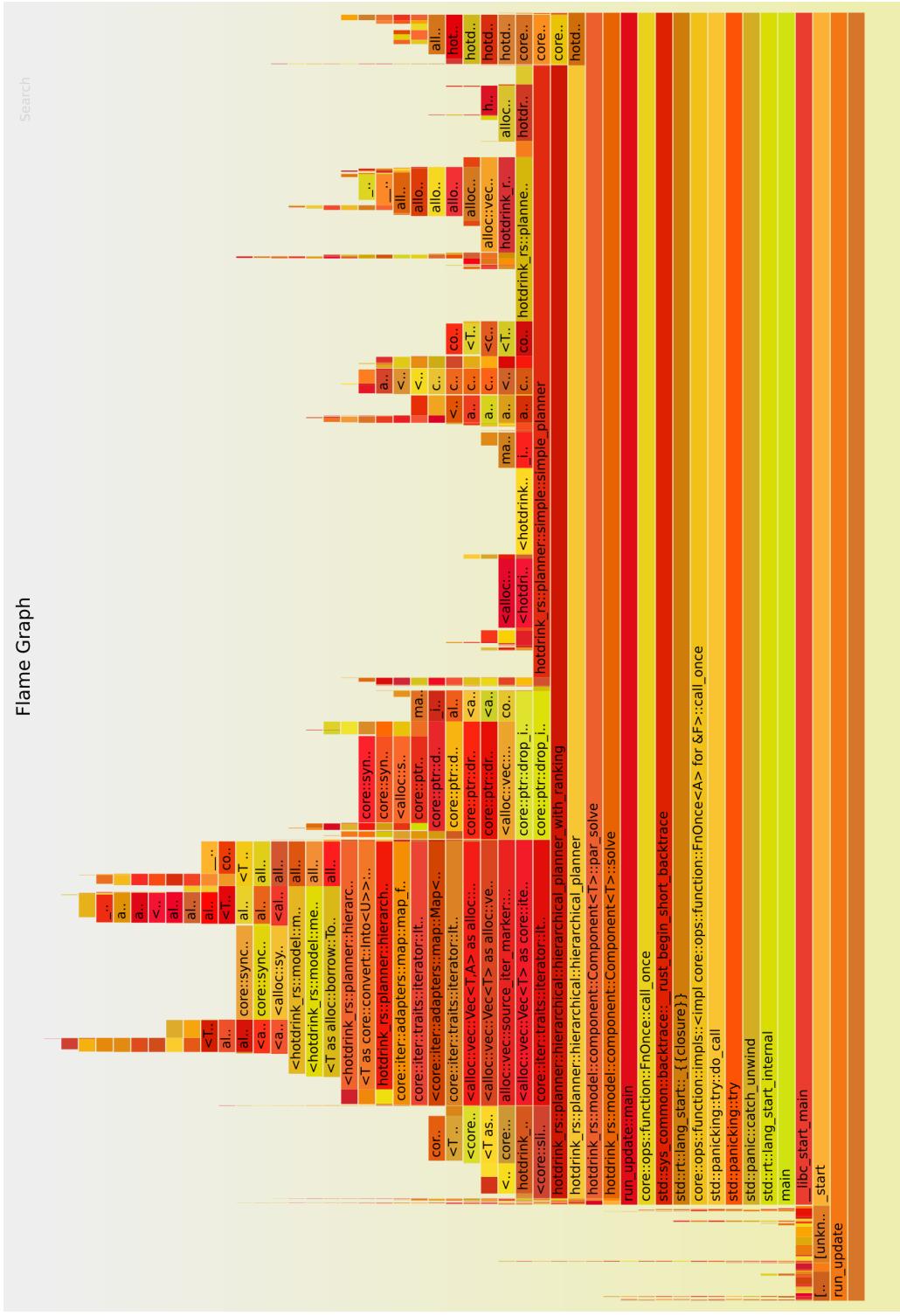


Figure 6.6: Flamegraph of a call to the solve method. This can be generated in the project directory `hotdrink-rs/hotdrink-rs` with the command `cargo flamegraph --example run_solve`.

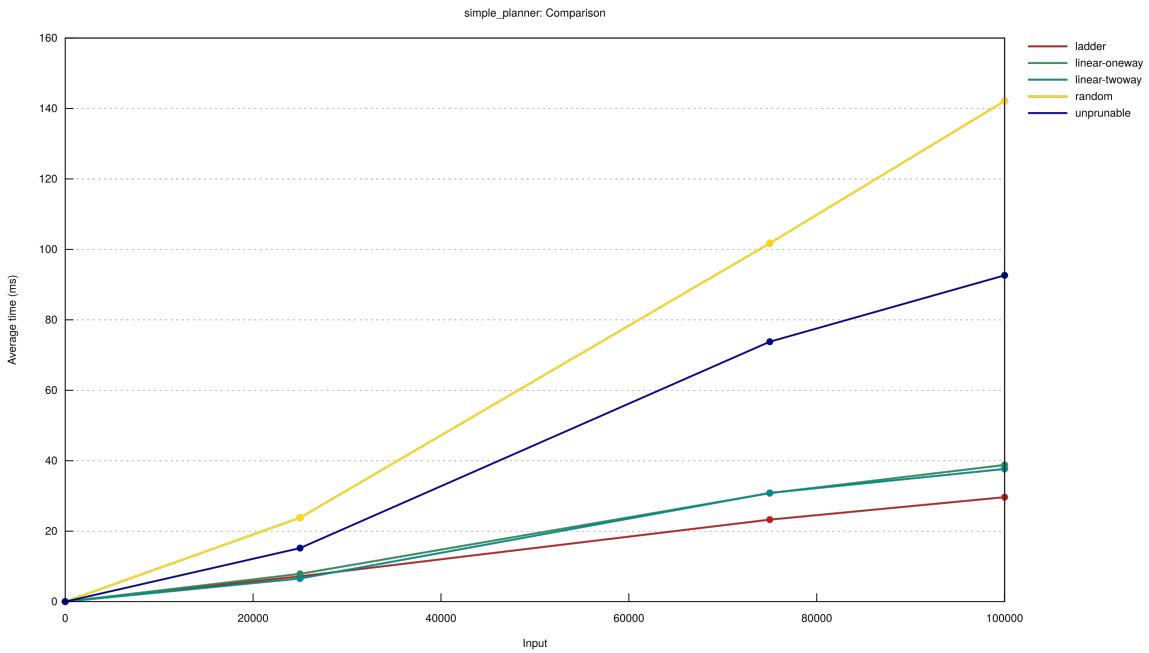


Figure 6.7: Simple planner performance on different constraint systems.

Table 6.2: Hierarchical planner benchmarks in milliseconds (CPU: Intel i5–8265U).

Type	# of constraints	hotdrink-rs
Linear-oneway	20000	54
Linear-twoway	20000	57
Ladder	1000	62
Random	1000	68
Unprunable	500	144

Looking at how the simple planner performs on different constraint systems in Figure 6.7, we see that while the random and unprunable systems are much slower to plan for than the others, the order of growth appears to be approximately the same. The growth appears to be linear, which is what we expect from the simple planner.

6.4 Hierarchical Planner Benchmarks

The benchmarks for the hierarchical planner are quite different from the ones for the simple planner, and can be seen in Table 6.2. The first result is that the hierarchical planner can handle much fewer variables in the allotted time, and the second is that the structure of the constraint system affects the result a lot. The results were produced with the command `cargo bench hierarchical_planner`, executed in the `hotdrink-rs`/`hotdrink-rs` directory.

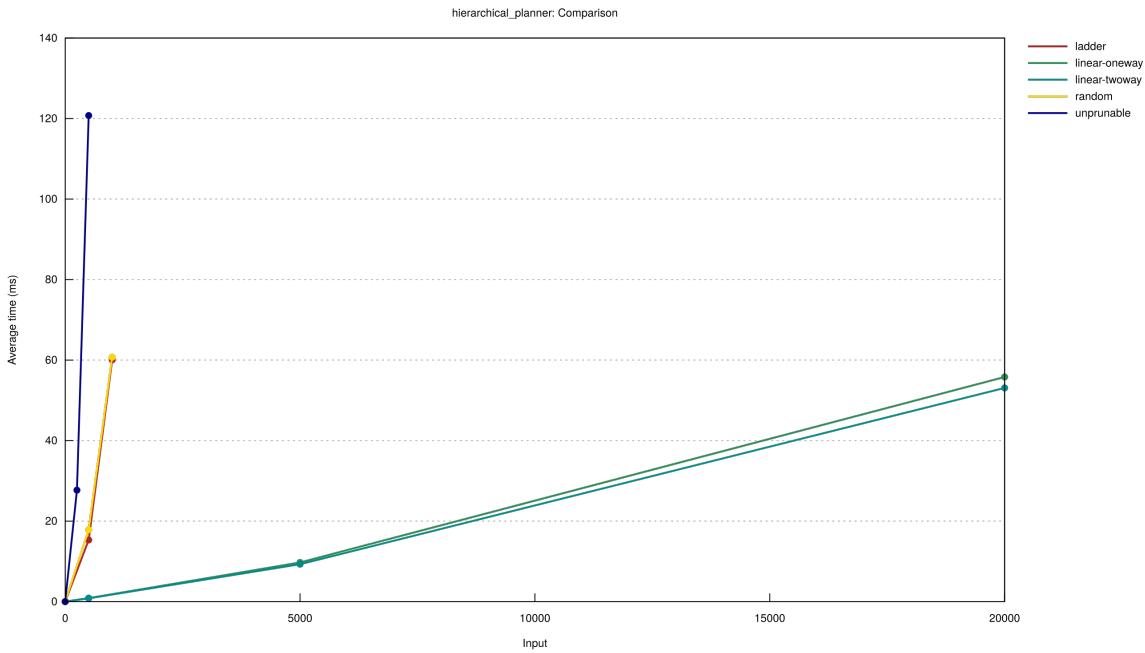


Figure 6.8: Hierarchical planner performance on different constraint systems.

How much the constraint system structure affects the result becomes very clear when comparing planning for different numbers of constraints, as seen in Figure 6.8. While the time to plan for the linear systems maintains a near-linear growth, planning for the random and unprunable quickly takes many times longer when the input size increases.

This difference is not surprising, since a plan for the linear systems can be found in a single call to the simple planner, while the less pruning-friendly systems may require a number of calls equal to the number of variables.

6.5 Solver Benchmarks

If we include both the planning phase (with the hierarchical planner) and actually executing the plan, the number of variables we can have in a system continues to decrease, even when the methods in question are trivial. The results can be seen in Table 6.3, and can be replicated by running `cargo bench solve`.

Since we are still using the hierarchical planner, there is still a rather large difference in performance between the different constraint systems, as seen in Figure 6.9.

Table 6.3: Solver benchmarks in milliseconds (CPU: Intel i5–8265U).

Type	# of constraints	hotdrink-rs
Linear-oneway	5000	119
Linear-twoway	5000	124
Ladder	1000	64
Random	1000	71
Unprunable	500	115

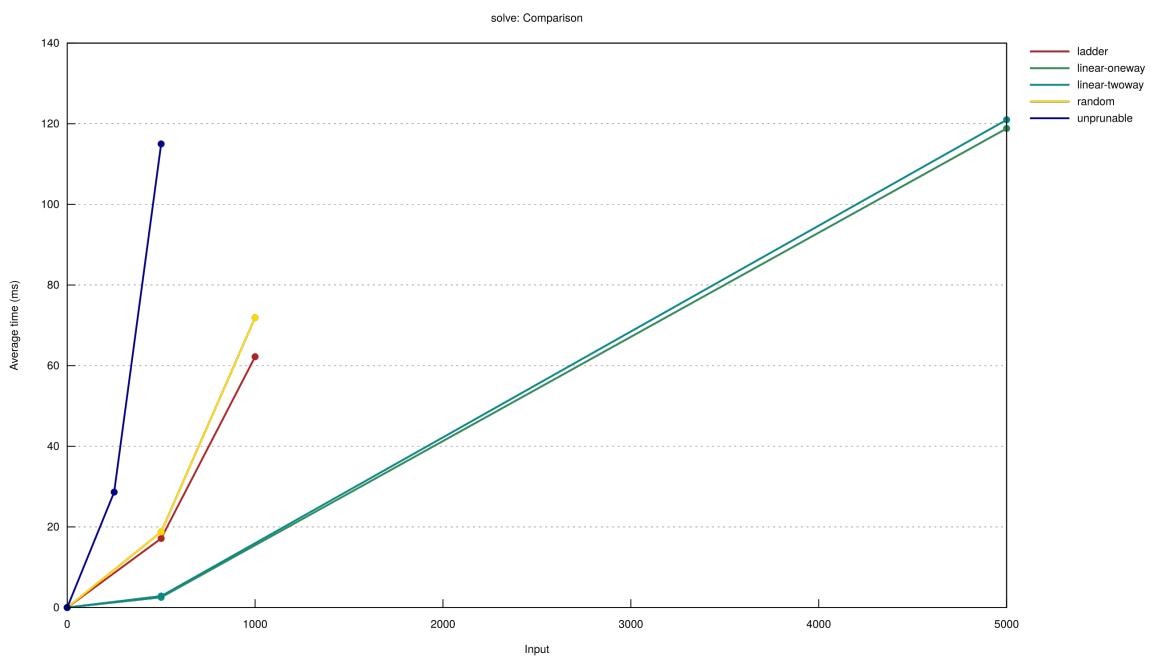


Figure 6.9: Solver comparison.

6.6 Comparison to Other Implementations

This section compares the performance of the different implementations of HotDrink. The comparison will benchmark `hotdrink-rs`¹ compiled to both native code and WebAssembly, the Flow-implementation of HotDrink², and the TypeScript implementation of HotDrink³. `hotdrink-rs` compiled to WebAssembly and the older implementations were executed in the browser, specifically in Chromium version 90.0.4430.212.

The comparison was done by performing a complete solve of the different constraint systems to get accurate results of the libraries' performance in practice, as opposed just measuring planning speed. The benchmarks can be run as follows:

Rust Execute the command `cargo bench thesis_update` in `hotdrink-rs/hotdrink-rs`.

WebAssembly Execute the following command in the directory `hotdrink-rs/hotdrink-wasm`, and visit `localhost:8000` to start the benchmarks. The console must be opened before the benchmarks have started, or after they are complete, as they are being executed on the main thread.

```
wasm-pack test --chrome --release
```

Flow Obtain a compiled version of the library, place it in `hotdrink-rs/benches/hotdrink-flow/` with the file name `hotdrink.js`, then open `index.html`.

TypeScript Obtain a compiled version of the library, place it in the directory `hotdrink-rs/benches/hotdrink-typescript/` with the file name `hotdrink.min.js`, then open `index.html`.

The results can be seen in Table 6.5. The performance with few constraints is unstable and may give different numbers for different executions. In addition, there are many factors that may affect the results:

1. Different languages.
2. Different execution environments.
3. While all the planning algorithms are based on QuickPlan, there are still many differences. `hotdrink-rs` relies heavily on pruning to lower the number of simple planner calls, the Flow implementation does some pruning and is partially incremental, and the TypeScript implementation is incremental without any pruning.

All this makes it difficult to see exactly what causes the differences in performance. See Table 6.4 for an overview of the biggest differences in the planner implementations.

Since the Rust- and WebAssembly-implementations both use pruning, they can solve the linear systems very quickly, as they only need to call the simple planner once to determine which methods must be selected.

¹<https://github.com/HotDrink/hotdrink-rs/tree/325185f47ffc5b39199d62bb83a297b309ec0cc8>

²<https://git.app.uib.no/Jaakko.Jarvi/hd4/-/tree/ab7a100a/>

³<https://github.com/HotDrink/hotdrink/tree/c96f3b03c932206ff5fe17c7e1c1517c64f54fc7>

Table 6.4: Planner feature comparison between HotDrink-implementations.

Implementation Language	Pruning	Incremental
Rust	Yes	No
WebAssembly	Yes	No
Flow	Partial	Partial
TypeScript	No	Yes

As we move towards less prunable systems, such as the ladder, random and unprunable, the planning time increases a lot, as expected. In these systems, the planner has to call the simple planner n times, where n is the number of variables in the component. In general, the WebAssembly implementation is approximately twice as slow as the Rust implementation but behaves in the same way since they use the same underlying algorithm.

The Flow- and TypeScript-implementations are much slower for most of the systems, but there are some notable exceptions. The Flow-implementation manages to solve the linear-twoway systems fairly quickly, but it is much slower on the linear-oneaway systems, even though pruning should work for either. The TypeScript implementation does not do any pruning at all, which is a likely cause of its much lower performance on the linear systems. Both of them, however, perform a lot better than Rust and WebAssembly on the unprunable systems. This is likely due to their incremental planners, as *The Upstream Constraint Technique* [73, p. 45] allows them to inspect much fewer constraints: in an unprunable system with N constraints, the incremental implementation lets them get away with analyzing approximately $\log_2 N$ constraints since this constraint system is shaped like a binary tree. This leads to completely different time complexities, and the difference in performance grows greater the more constraints there are in the system, as seen in Figure 6.10.

Table 6.5: Constraint system solve benchmarks in milliseconds (CPU: Intel i5–8265U).

Type	# of constraints	Rust	Wasm	Flow	TypeScript
Linear-oneway	100	0.3	1	52	104
Linear-twoway	100	0.4	1	20	107
Ladder	100	1	3	35	180
Random	100	1	4	36	732
Unprunable	100	5	16	23	37
Linear-oneway	500	3	10	425	2819
Linear-twoway	500	3	11	90	2657
Ladder	500	17	62	286	5299
Random	500	19	54	352	19211
Unprunable	500	116	331	118	179
Linear-oneway	1000	7	26	1669	11802
Linear-twoway	1000	7	26	203	10172
Ladder	1000	63	251	1640	19342
Random	1000	73	212	1269	83456
Unprunable	1000	472	1274	269	352

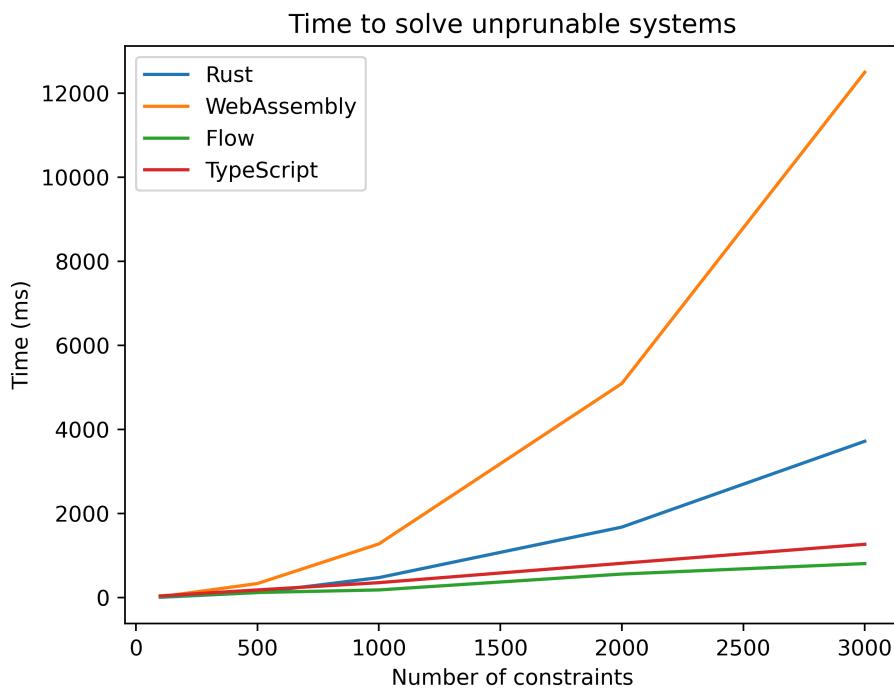


Figure 6.10: Time to solve unprunable systems. The poor performance of the Rust and WebAssembly implementations of HotDrink on unprunable constraint systems becomes more apparent in larger constraint systems.

Chapter 7

Memory-Efficient Data Structures

This chapter describes the work of minimizing the amount of memory used by methods and constraints. Specifically, it includes three different sets of data structures that can be used to model a constraint graph. As well as the obvious benefit of it using less memory, this may also improve performance by allowing use of bitwise operations, and by making the data structures used in planning more cache-friendly. `hotdrink-rs` uses a variant of the first technique presented. The alternative techniques defined in this section are intended to guide future improvements to the library.

The described data structures will only include the parts that are relevant to define a constraint graph; data such as names and the method bodies is not included. We are only interested in the relative memory usage of the different techniques, and make a number of simplifications. For instance, we assume that the number of variables and methods are the same for each constraint. Also, while `hotdrink-rs` technically allows for all variables involved in a constraint to be both read from and written to by its methods, this should be fairly rare; we thus assume that each method only uses each variable once, either as an input or output, but not both. Stay constraints can easily be represented with a single index representing the variable that the stay constraint is for, and are not taken into account. While we must know how many variables there are in the component containing the constraints, all three strategies are independent of the component's representation; we can thus compare the data structures by looking at the memory usage of only the constraints. See Table 7.1 for an overview of symbols that are used.

To have concrete numbers to work with, we assume that a 64-bit architecture is used. An overview of the sizes of different data structures can be seen in Table 7.2.

Table 7.1: Symbols and their meanings.

Symbol	Meaning
V	Variables of a component
R	Variables of a constraint
M	Methods of a constraint

Table 7.2: Sizes of types on a 64-bit architecture. The variable n is the number of elements in the Vec/BitVec. Both Vec and BitVec have a capacity (64 bits), length (64 bits), and a pointer to the heap where their elements are stored (64 bits), for a total of 192 additional bits.

Type	Size in bits
usize	64
Vec<T>	$192 + n \cdot \text{sizeof}(T)$
BitVec<Lsb0, u8>	$192 + \lfloor \frac{n+7}{8} \rfloor \cdot 8$

7.1 Naive Implementation

Variable objects are stored in the component in an array. A method object must map between variables in a component and the sequence of input arguments to the method, and similarly for the outputs, the results of the method. We begin by defining `NaiveMethod`, which has two vectors of indices (referring to variables of the Component) that represent the method's inputs and outputs.

```
struct NaiveMethod {
    inputs: Vec<usize>,
    outputs: Vec<usize>
}
```

With this simple representation, each Vec takes up a constant 192 bits, in addition to the indices contained within them that take up 64 bits each.

Definition 7.1.1 (Size of naive method). $sz_{nm}(R) = 384 + |R| \cdot 64$

A `NaiveConstraint` is simply a container for Methods that all use the same variables.

```
struct NaiveConstraint {
    methods: Vec<NaiveMethod>
}
```

It requires 192 bits for the Vec itself, in addition to the memory taken up by its $|M|$ `NaiveMethods`.

Definition 7.1.2 (Size of naive constraint). $sz_{nc}(R, M) = 192 + |M| \cdot sz_{nm}(R)$

The current implementation of `hotdrink-rs` uses a variation of the naive strategy, but additionally stores a `Vec<usize>` in each constraint for easy access to the variables that they use. This adds another 192 bits for the Vec, as well as 64 bits for each variable in the constraint.

Definition 7.1.3 (Size of constraint in `hotdrink-rs`). $sz_{hdrs}(R, M) = 384 + 64 \cdot |R| + |M| \cdot sz_{nm}(R)$

For context, we also show a possible definition of `Component`, which includes its size and constraints. The size itself (the number of variables) only requires 64 bits, and assuming each constraint uses the same number of variables gives us another $|C| \cdot sz_{nc}(R, M)$ bits. Finally, the Vec itself takes up 192 bits.

```

struct Component {
    size: usize,
    constraints: Vec<Constraint>
}

```

7.2 Representing Method Inputs and Outputs with Individual Bits

The first optimization is that we can operate on the bit-level instead of having an entire `usize` to tell which variables are being used. That is, instead of having a vector of indices, we simply set the individual bits at the corresponding indices. For instance, if we have 4 variables in a component and want to mark 0, 2, and 3 as inputs, we can represent the information with 4 bits. We set the bits at index 0, 2, and 3 to 1 to get 1011.

We can use the `bitvec`-crate [3] to simplify the process of storing individual bits. It allows us to create `BitVec`, a `Vec`-like type that can store individual bits. For example, `BitVec<Lsb0, u8>` means that the indexing starts at the least significant bit (`Lsb0`), and that the underlying type used is `Vec<u8>`. In addition to the 192 bits it requires when empty, an underlying type of `u8` means that it will require more memory in increments of 8: 0 additional bits to represent 0 variables, 8 additional bits to store 1–8 variables, 16 additional bits to store 9–16 variables, and so on.

Instead of having two separate `BitVecs`, we can use the same one for both inputs and outputs to avoid doubling the constant 192 bits. We then use the first half of the bits for inputs and the second for outputs. When the number of variables is low, e.g. 4, we can then fit all the indices in a single `u8`, instead of using two where only 4 of the bits are used.

```

struct BitMethod {
    ins_and_outs: BitVec<Lsb0, u8>
}

```

In addition to the bits used by a `BitVec`'s contents, we need 192 bits for the `BitVec` itself.

Definition 7.2.1 (Size of bit-method). $sz_{bm}(R) = 192 + \lfloor \frac{|R|+7}{8} \rfloor \cdot 8$

By only changing the representation of method inputs and outputs, we may leave many bits unused. For instance, let us say that there are 1000 variables in a component; we then require 2000 bits to mark method inputs and outputs. With a constraint that only uses 5 variables, each method would leave 1990 bits unused, which leads to worse memory usage than the naive solution. We need to store which variables the constraint uses, for instance in a `Vec<usize>`, and make the methods' indices refer to the variables' position in the constraint. See 7.1 for a visualization of the translation between constraint indices and component indices.

Definition 7.2.2 (Component index and constraint index). A variable's index in its component is called its *component index*. For each constraint the variable is involved in, its *constraint index* is its index in the constraint.

For instance, a variable with index 10 in the component may have a constraint index 2 if it is the third variable used in a constraint. A variable only has a constraint index in constraints it is used in, and can have a different one for each constraint.

```
struct ExtendedConstraint {
    variables: Vec<usize>,
    methods: Vec<Method>
}
```

Since we now add `variables` field with the indices of the variables involved in the constraint, our `ExtendedConstraint` will require an additional $|R| \cdot 64$ bits of memory, though it can now use `BitMethods` instead of `NaiveMethod` to offset this.

Definition 7.2.3 (Size of extended constraint). $sz_{ec}(R, M) = 384 + |R| \cdot 64 + |M| \cdot sz_{bm}(R)$

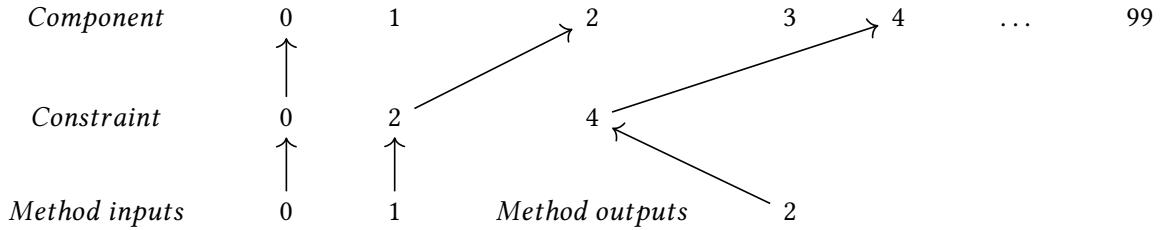


Figure 7.1: Variable lookup when using bit-indices. To find a variable's component index, we first find out which bit is set in the method inputs or outputs. If the third bit is set in the method's inputs, the variable has a constraint index of 2; we look at the third variable used by the constraint and find the value 4. Thus the method reads from the variable with component index 4.

7.3 Representing Constraint Variables with Individual Bits

We can apply the same trick as we did to method inputs and outputs to the constraint itself. Instead of having a list of indices indicating which variables it is using, we can use `BitVec` and mark the bits of the component that are involved in the constraint.

```
struct BitConstraint {
    variables: BitVec<Lsb0, u8>,
    methods: Vec<Method>
}
```

A constraint will then use $192 + \lfloor \frac{|V|+7}{8} \cdot 8 \rfloor$ bits to represent its variables, instead of the previous $|R| \cdot 64$ when using `Vec<usize>`.

Definition 7.3.1 (Size of bit-constraint). $sz_{bc}(V, R, M) = 384 + \lfloor \frac{|V|+7}{8} \cdot 8 \rfloor + |M| \cdot sz_{bm}(R)$

7.4 Comparison

Above, we explored space-saving optimizations. Below, we analyze potential savings in typical scenarios, and for different constraint system sizes. Since we ignore stay constraints, we will assume that each constraint involves at least 2 variables. We will then use at least 4 bits (2 inputs and 2 outputs) to represent the variables used by a BitMethod. Each variable will thus take up at most 2 bits each, which simplifies the formula from $sz_{bm}(R) = 192 + \lfloor \frac{|R|+7}{8} \rfloor \cdot 8$ to $sz_{bm}(R) = 192 + 2 \cdot |R|$. We apply the same reasoning to the formula for the size of BitConstraint, sz_{bc} . The formulas for calculating the size of the constraints are then as follows:

$$\begin{aligned} 384 + 64 \cdot |R| + |M| \cdot (384 + |R| \cdot 64) && (\text{Current representation}) \\ 192 + |M| \cdot (384 + |R| \cdot 64) && (\text{Naive representation}) \\ 384 + |R| \cdot 64 + |M| \cdot (192 + 2 \cdot |R|) && (\text{BitVec in methods}) \\ 384 + 2 \cdot |V| + |M| \cdot (192 + 2 \cdot |R|) && (\text{BitVec in constraints}) \end{aligned}$$

To make the math easier to follow, we rename $|V|$ to v , $|C|$ to c , $|R|$ to r , and $|M|$ to m . We also reorder parts of the formulae to get the following:

$$\begin{aligned} 384 + 64r + m(384 + 64r) \\ 192 + m(384 + 64r) \\ 384 + 64r + m(192 + 2r) \\ 384 + 2v + m(192 + 2r) \end{aligned}$$

BitVec in methods vs. naive representation

We then compare the representations, starting with the naive representation and BitVec in methods. Constraints must have at least one method to be able to be enforced, and can at most have a number of methods equal to the number of variables; any more would mean that one method outputs to a subset of another method's outputs. The method with the fewest outputs would then always be chosen over the other [73, p. 37]. Using BitVec to represent method inputs and outputs performs worse when there are few methods: we start by analyzing the case where each constraint has r variables and 1 method.

$$\begin{aligned} 384 + 64r + m(192 + 2r) &< 192 + m(384 + 64r) \\ 384 + 64r + 192 + 2r &< 192 + 384 + 64r && (m = 1) \\ 64r + 2r &< 64r && (\text{Subtract constants}) \\ 66r &< 64r \end{aligned}$$

In the result above, we see that even in the worst case, using BitVec to represent method inputs and outputs only takes up $2r$ bits more than the naive representation.

To both simplify the calculations and to see how large the potential savings can become, we will assume that the number of methods is about the same as the number of variables, i.e., $m = r$.

$$\begin{aligned}
 384 + 64m + m(192 + 2m) &< 192 + m(384 + 64m) \\
 384 + 64m + 192m + 2m^2 &< 192 + 384m + 64m^2 && \text{(Expand expressions)} \\
 384 + 256m + 2m^2 &< 192 + 384m + 64m^2 \\
 192 + 256m + 2m^2 &< 384m + 64m^2 && \text{(Subtract 192)} \\
 192 - 128m - 2m^2 &< 64m^2 && \text{(Subtract 384m)} \\
 192 - 128m - 62m &< 0 && \text{(Subtract 64m)} \\
 m &< 1.007\dots && \text{(Find positive root)}
 \end{aligned}$$

This shows that when a constraint has more than one pair of methods and variables, using BitVec to represent method inputs and outputs is better than the naive representation. Most of the constraint systems constructed in Chapter 6 have between 1 and 2 variables per method within each constraint. The only exception is the random constraint system, which may have a single method no matter how many variables there are, but tends to have around 4 variables per method.

BitVec in methods vs. current implementation

The current implementation of Constraint already has the variables field to avoid recomputing it from method inputs and outputs, which means that the benefit of minimizing memory consumption in methods is even greater. Since we then have 192 more bits within Constraint to store the Vec, and $|V| \cdot 64$ to store the variables, we end up with the following inequality instead (bit indices in methods on the left, actual implementation on the right). We again replace r with m since they are likely to be similar.

$$\begin{aligned}
 384 + 64m + m(192 + 2m) &< 384 + m(384 + 64m) \\
 384 + 64m + 192m + 2m^2 &< 384 + 384m + 64m^2 && \text{(Expand expressions)} \\
 384 + 256m + 2m^2 &< 384 + 384m + 64m^2 \\
 256m + 2m^2 &< 384m + 64m^2 && \text{(Subtract 384)} \\
 -128m - 2m^2 &< 64m^2 && \text{(Subtract 384m)} \\
 -128m - 62m^2 &< 0 && \text{(Subtract 64m)} \\
 m &< 0 && \text{(Find positive root)}
 \end{aligned}$$

Since m is always positive, we will always save memory compared to the current implementation.

Table 7.3: Relative bits used to store a component per number of methods. The number of variables in the component is 50, and the number of variables per constraint is the same as the number of methods.

	1 method	5 methods	20 methods	50 methods
hotdrink-rs	896	4224	34944	182784
Naive	640	3712	33472	179392
BitVec in methods	642	1714	6304	18184
BitVec in constraints	678	1494	5124	15084

BitVec in constraints vs. BitVec in methods

We continue by comparing the usage of BitVec in methods with using it in constraints as well. This time, the result is independent of m , which makes replacing r with m an unnecessary simplification.

$$\begin{aligned}
 384 + 2v + m(192 + 2m) &< 384 + 64r + m(192 + 2m) \\
 2v + m(192 + 2m) &< 64r + m(192 + 2m) && \text{(Subtract 384)} \\
 2v &< 64r && \text{(Subtract } m(384 + 2m)) \\
 v &< 32r && \text{(Divide by 2)}
 \end{aligned}$$

This result shows that we save memory as long as the total number of variables in the component is less than 32 times the number of variables per constraint. For instance, if each constraint uses 5 variables, we will save memory up until we have 160 variables in the component. That is, while using BitVec to represent variables used by constraints saves memory in smaller constraint systems, we want to optimize memory usage in large constraint systems (thousands of variables).

Table 7.3 and Table 7.4 show how much memory each representation consumes depending on the number of variables and methods per constraint, and a plot of nearly the same data (limiting the number of variables and methods) can be seen in Figure 7.2. In addition, to see what the comparison would look like if r and m can be different, see Figure 7.3. Based on our own experience, it is rare to create constraint systems with more than a few (2–5) variables and methods per constraint, while programmatically generated constraint systems easily reach thousands (or hundreds of thousands) of variables in total. To optimize this kind of system, limiting usage of BitVec to methods appears to be the optimal strategy. Based on the data in Table 7.3, it can reduce the required memory consumption of components by more than 53% if there are 5 variables and 5 methods per constraint, and over 80% if there are 20 variables and 20 methods per constraint.

7.5 Drawbacks

While the benefits to memory usage are clear, the space-saving strategies are more complex to implement. With the naive solution, one could simply inspect the inputs and outputs of methods to find out which

Table 7.4: Relative bits used to store a component per number of methods. The number of variables in the component is 10000, and the number of variables per constraint is the same as the number of methods.

	1 method	5 methods	20 methods	50 methods
hotdrink-rs	896	4224	34944	182784
Naive	640	3712	33472	179392
BitVec in methods	642	1714	6304	18184
BitVec in constraints	20578	21394	25024	34984

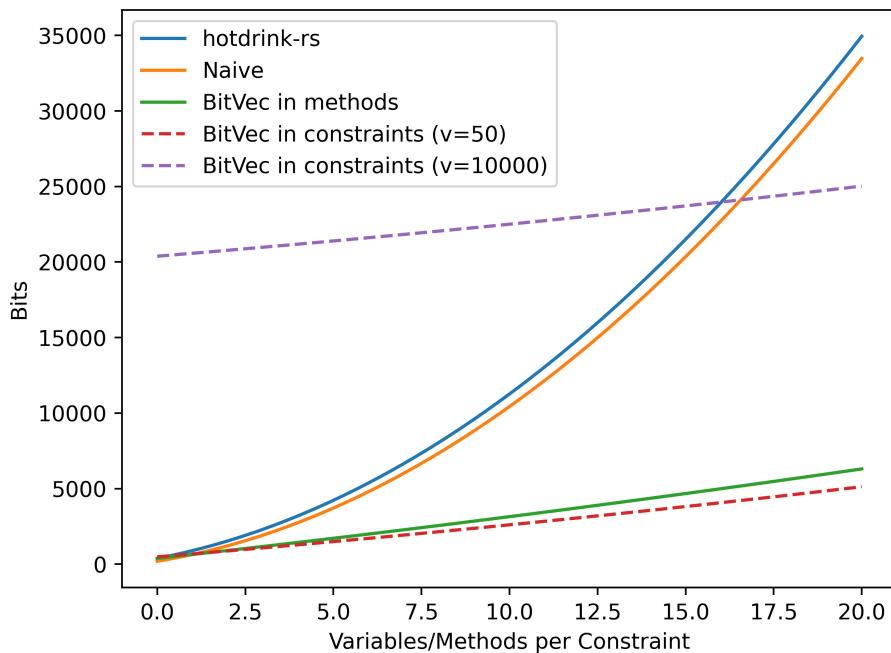


Figure 7.2: A plot of the memory usage per strategy. The memory usage of the currently implemented strategy and the naive strategy grows quite quickly when the number of variables and methods per constraint increases. The constant factor included when using BitVec in constraints causes memory usage to be high early on. This leaves using BitVec in methods as the optimal strategy.

BitVec in Methods vs. Naive Representation

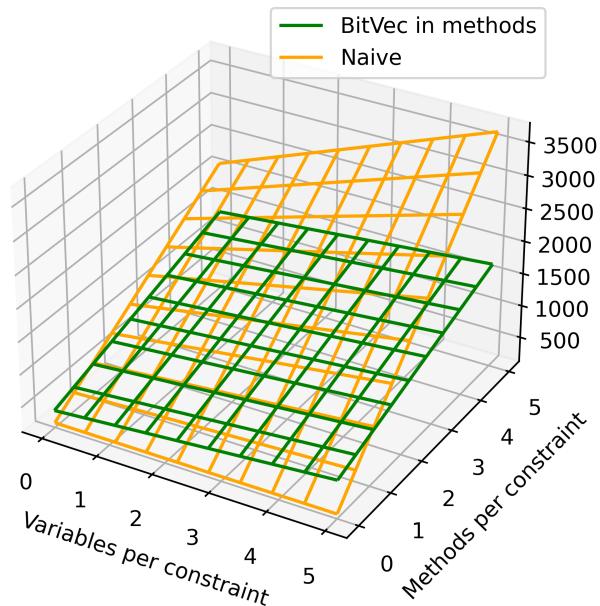


Figure 7.3: A comparison of naive strategy and using BitVec in methods. This is a more accurate representation that takes into account that r and m are separate variables. Its three-dimensional nature can make the plot difficult to understand; run `computing/memory_usage_plot_3d.py` to interact with it.

variables of a component they use, without having to translate bit indices through a constraint. To implement the change, the Method API (or how it is used) must change. Some possible solutions include

1. having the `inputs` and `outputs` methods require a constraint that can interpret the set bits,
2. only being able to access method inputs and outputs through a constraint,
3. or manually performing the translation when checking the set bits of the method.

In addition to changing the API, the extra translation may also affect performance; to find the component indices of variables used by a method we must now check each individual bit of its inputs and outputs. Using set bits to mark used variables is thus a trade-off between memory usage and performing additional work, and must be evaluated further before deciding if it is worth it. Exactly how to implement this — if at all — is yet to be decided.

Chapter 8

Discussion

This section describes some of the positive and negative experiences of implementing `hotdrink-rs`, `hotdrink-wasm` and `hotdrink-c` with Rust and WebAssembly. It also discusses how the final implementation turned out, and the results found in the performance analysis in Chapter 6.

8.1 Rust

Overall, Rust and its associated tooling offers a productive development environment. Some parts of the implementation may have been more difficult because of the choice of language, but it has resulted in a significantly faster implementation for many constraint systems. The following sections describe more specific advantages and disadvantages of the language in the context of our work.

8.1.1 Strict Type System

Many other implementations of constraint systems use languages that do not require types to be known at compile-time, such as the implementations of HotDrink in Flow and TypeScript, QuickPlan implemented in Common Lisp[73, p. 62], and ConstraintJS in JavaScript [41]. However, implementations of Blue and DeltaBlue exist in Smalltalk, C, C++, Object Pascal, and Common Lisp, so this does not always apply. TypeScript and Common Lisp can use type annotations for static type checking, but make it easy to use dynamic typing where it is necessary.

Rust is a statically, strongly typed language, which introduces a few pain points when implementing heterogeneous data structures. For instance, supporting methods with an arbitrary amount of arguments as seen in Section 8.1.2, or methods with inputs and outputs of multiple types as seen in Section 8.1.3.

The type system and tooling (such as the compiler, clippy, and the language servers) also provide a number of benefits. First of all, Rust's type system guarantees that type errors are caught at compile-time. This proved especially useful during large-scale refactoring of the library. For instance, when changing the type signature of a function, the library will no longer compile until all uses have been modified to support

the change. Ensuring that all uses are updated after such a change can quickly become a monumental task in a dynamic language where errors resulting from the change will not be detected until runtime (if at all), unless the change is supported by refactoring tools [55, p. 1].

Another benefit of types is that they serve as a form of documentation that makes it easier to use the library correctly. Combined with a language server that provides information about which types and data structures are available, types become a huge help for both users and developers of the library.

Since Rust does not have a garbage collector and allows low-level control of memory, there are many decisions to make that affect large parts of the codebase. For a simple value, we need to know what part of the program should own it, and then ensure that we only use references to it where the compiler is convinced that the reference is still valid. If there is no clear owner, and we want multiple parts of the program to maintain references to it, then we may want to choose a `Rc`, a reference counted pointer. If we want to share the value among multiple threads, we must use an atomically reference counted pointer (an `Arc`) instead, to make sure that the count is incremented atomically. If multiple threads should be able to modify the value, then we must allow interior mutability, e.g., with `Arc<Mutex<T>>` or `Arc<RwLock<T>>`. All of these decisions often propagate through the program, and while it ensures that we share data safely, it can quickly complicate the various APIs.

8.1.2 Methods With an Arbitrary Number of Arguments

Unless we generate a separate struct for each method we use, we must make the single `Method` struct general enough to represent all possible methods. If we have to represent $m_1 : a \rightarrow (b, c)$ and $m_2 : (b, c) \rightarrow a$ with the same data structure, then we could for instance use a `Vec`.

```
struct Method<T> {
    ins: Vec<usize>,
    outs: Vec<usize>,
    f: fn(Vec<T>) -> Vec<T>,
}
```

By using `Vec` we make the inner function more general, at the cost of losing compile-time errors when the method is called with the wrong number of arguments.

8.1.3 Methods With Arbitrary Return Types

To represent a method that takes both an integer and a string as input, we can use sum types as the value type in the constraint system. This would make `Method<IntOrString>` the type we are looking for, without having to change the internal representation of a method. Doing this in a dynamically typed language is again much simpler since nothing stops us from passing in and returning any type.

8.1.4 Variable Access

In languages like JavaScript and Python, each component can have a field for each variable they contain. This means that the variables could be accessed like `comp.a` and `comp.b` instead of using strings for indexing like in `comp.variable("a")` in `hotdrink-rs`. While the availability of a variable is checked at runtime in either implementation, the Rust implementation returns an `Option` to signify that the variable may not exist.

In order to get compile-time errors when accessing variables, we can instead generate a separate struct for each component, though this would require that the component cannot have new variables added during runtime. Doing this would also require many changes to the existing implementation to support the different component data structures, but may be worth investigating further.

8.1.5 Multithreading in Rust

Rust is built with good support for parallelism in mind, and it uses its ownership system to catch concurrency errors at compile-time [11]. For instance, erroneous usage of data that is not safe to pass between threads (is not `Send`), or does not synchronize access (is not `Sync`), will result in a program that fails to compile. This forces usage of atomically reference counted values (`Arc`) for shared data that needs to live until all threads are done with it, or wrapping of values in a `Mutex` if multiple threads need to modify the value. Rust has been very helpful in implementing the parallel solver correctly.

8.2 WebAssembly

While creating a simple Rust project that compiles to WebAssembly is made relatively easy with `wasm-bindgen`, introducing parallelism has complicated matters a great deal. WebAssembly is still being worked on, and information about it and related projects can sometimes quickly become outdated; this was especially noticeable with information on how to implement Web Worker-based threads. Knowing exactly what still applies can be difficult, like in Alex Crichton's article on multithreading with Rust and Wasm [8]: Spectre and Meltdown's effect on `SharedArrayBuffer` has changed the specification a lot in the last few years [59].

8.2.1 Multithreading with Web Workers

Using Web Workers for multithreading is far from as straightforward as using Rust's standard library. First of all, `wasm-pack` must be run with the `--target no-modules` option, which makes the compiled WebAssembly more difficult to use, as it is no longer a JavaScript module (though a shim generated with `wasm-bindgen` exports a symbol of the same name that can be used to make the process easier). Additional document headers must also be set to enable usage of `SharedArrayBuffer` [59]. Finally, a thread-like wrapper must be built around `web_sys::Worker`, and a script for them to run must be included. While

there are a few libraries that implement this wrapper already, as mentioned in Section 4.6.2, they do not allow cancellation of threads; we therefore made our own implementation.

Another issue is that the types `wasm_bindgen::JsValue` and `js_sys::Function` cannot be sent between threads; this has complicated the design of `hotdrink-wasm`, as discussed in Section 4.7. Exposing Rust types to JavaScript only works for ones that are compatible, or are contained within a wrapper-struct that hides them, such as the constraint system wrapper. This complicates the process of creating WebAssembly-compatible bindings in some cases, but the same applies when creating bindings for other languages, such as C, and may be unavoidable.

8.2.2 Cancellation

Different thread implementations have the capability of canceling threads, such as the `pthread` specification with `pthread_cancel` [43], `Worker.terminate()` for Web Workers in Javascript [88], or `Thread.stop` in Java [27]. Adding the same capabilities to Rust has been suggested [69], but has been rejected for many of the same reasons for why Java's `Thread.stop` was deprecated [27]: canceling threads causes many issues that are difficult to deal with.

Breaking invariants If a thread can be canceled at any point, then it may also happen during an operation that maintains some data invariant. It could for instance happen while modifying some shared data, or while writing to a file, in which case data may become corrupted [44].

Failing to free resources If the thread is immediately stopped instead of letting the thread finish on its own, it may not free all the resources it is supposed to. This does not only apply to memory, but also mechanisms for sharing memory safely, such as locks. A locked mutex may then never be unlocked, which may cause a deadlock to occur.

However, these issues do not usually apply to Web Workers. Once a worker is terminated, the script it is running will be aborted, and any resources will eventually be garbage collected [9, 67, 1]. Web Workers tend to use message passing with channels to communicate with the main thread, and thus do not require locks since there is no shared data. Being canceled in the middle of a computation means that no values are ever sent to another thread, which means that the corruption does not spread. They cannot modify the Domain Object Model (DOM) either, leaving them isolated and safe to terminate.

However, this changes when we introduce `SharedArrayBuffer`, which is used to share memory used by WebAssembly. This enables memory sharing in the Web Worker-based thread implementation in `hotdrink-wasm`, and re-introduces the issues with cancellation. Since Rust has no concept of a thread exit, the thread's stack will also remain, which causes a memory leak [8].

All of these issues may indicate that using thread cancellation to forcibly stop user code may cause more problems than it solves.

8.3 Implementation and Results

This section discusses the final implementations of `hotdrink-rs` and `hotdrink-wasm`, and how the results answer our research questions.

8.3.1 Features and API

Other than parallel solving, undo and redo, there are few new features that do not exist in earlier implementations of HotDrink. There are not many deliberate changes to the API of the library either: the changes that have been made are often there to make it work in Rust and WebAssembly, and not to improve ergonomics. One example of this is that modifications to the system should always be made through a `ConstraintSystem` or `Component`; modifying a `Variable` directly will neither notify the constraint system of the change nor update the priority of the variable. Another example is subscription to variables through the constraint system wrapper as described in Section 9.8. A lot more work can be done to make the API more user-friendly.

8.3.2 Performance

In the introduction of this thesis, we asked how much the library’s performance could be improved with Rust and WebAssembly. With the benchmarks from Section 6.6, we see that `hotdrink-rs` can be many times faster than earlier implementations in systems where the pruning optimization technique is applicable. On the other hand, it performs poorly on systems where it cannot take advantage of pruning. The incremental planners, however, appear to do the exact opposite: they perform well on the unprunable system and worse on the others. This suggests that the two optimization techniques should not be thought of as replacements of one another, but should rather be used together to ensure good performance for all systems. Making the planner implementation in `hotdrink-rs` incremental is thus likely required to guarantee that it is much faster than the earlier implementations for all systems. Note that in hand-crafted constraint systems with less than a hundred variables, the time it takes to create a plan is already negligible, even in the worst case. Thus, for most practical purposes, the relatively bad performance on unprunable systems is not a problem.

There are many differences between the different implementations, such as the language, execution environment, optimization techniques, and the exact implementation of the planning algorithm. It is thus difficult to know exactly how much each of these contributed to the differences in performance. The execution environment factor (e.g., a web browser) can be eliminated by compiling `hotdrink-rs` to WebAssembly; the difference this makes can be seen in Figure 6.10. Further comparisons of individual parts of the algorithms can be made to remedy this limitation of our work.

8.3.3 Responsiveness

We also asked if we could guarantee GUI responsiveness during solving, and have attained partial success. With constraint satisfaction methods being executed outside of the main thread, we guarantee that the user

code in methods does not block the GUI. The user is free to make further modifications, which can even cause irrelevant computations to be canceled. However, the planning step is still executed on the main thread; a constraint system of sufficient size may take long enough to plan for to create a noticeable delay for the user. Other limitations include the issues with cancellation as discussed in Section 8.2.2, and extra overhead from having multiple threads running.

Overall, the solution appears to succeed in its goal of guaranteeing responsiveness in the presence of long or non-terminating computations in methods, given that the planning step completes within the desired time frame. To improve upon the guarantee, the planning step itself could also be done in another thread in future versions of the library.

8.3.4 Memory-Efficient Data Structures

While not implemented in `hotdrink-rs`, the new data structures introduced in Chapter 7 can provide a substantial decrease in memory consumption; specifically, they can halve the memory consumption in systems with five variables and methods per constraint, and save even more memory as this number increases. During our work on optimizing the planning algorithm of `hotdrink-rs`, we found that using less memory could often give a greater speedup than trying to cache results of computations. Using the new representations of constraint systems could thus not only save memory but also provide performance benefits.

Chapter 9

Future Work

Our exploration of constraint system GUIs has opened up many new avenues for further study. This chapter includes further enhancements that we believe may be useful to implement for `hotdrink-rs` and `hotdrink-wasm`. For instance, making the planner incremental would likely improve planning time for unprunable constraint systems, and smarter scheduling of methods would increase how many computations actually run in parallel during solving.

9.1 Planner Optimization

We first describe improvements to planning.

9.1.1 Making the Planner Fully Incremental

While `hotdrink-rs` performs well in constraint systems where it is possible to use pruning, it falls short of the incremental implementations in constraint systems such as the unprunable one. By making the planner fully incremental, as described in Zanden’s paper about QuickPlan [73], `hotdrink-rs` will likely be much faster than the Flow and TypeScript implementations for all constraint systems.

A drawback of incremental planning is that we would have to store information about the latest solution, which would increase the memory consumption, but the impact is small — at most doubling the memory size requirement for the non-incremental planner. The effect this will have on the performance of the planner must still be researched.

9.1.2 Minimizing Allocation

The hierarchical planner currently requires the constraint system to be mutable in order to add stay constraints and prune the system. Since we do not want the hierarchical planner to modify the original constraint system, we create a clone instead. This, as well as all intermediate plans that are created by the simple planner, require a lot of unnecessary allocation that slows down the planning.

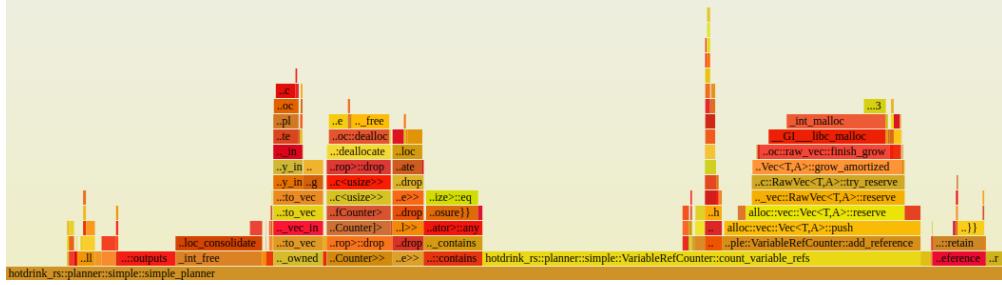


Figure 9.1: Flamegraph of the simple planner on the unprunable system.

We could solve the former issue by creating a mutable wrapper around an immutable constraint system to keep track of stay constraints and pruned parts, or simply allow the system to be mutated in a way that can be undone after planning. To improve the simple planner, a version that does not store the plan but simply checks if a plan exists would likely speed up the process. Actually constructing the plan is only necessary to do once.

9.1.3 Reusing Variable Reference Counts

The current implementation of the simple planner stores information about how many constraints still reference variables. This is done to be able to check in constant-time which variables are free. The issue is that this data is re-computed for each call to the simple planner; computing it in the hierarchical planner and gradually modifying it would likely speed up solving, especially in unprunable systems that call the simple planner many times. As seen in Figure 9.1, the `count_variable_refs` function can take up nearly half of the time spent in the simple planner.

9.2 Improved Scheduling

While the current planner respects data dependencies to maintain correctness, it does nothing to maximize parallelism. In a chain-shaped plan such as the one in Figure 9.2, one task is created per method, even though the inputs of m_2 depend on the output of m_1 . This means that the second thread is idle until the first completes. Depending on the structure of the dependency graph formed by the plan, this thread could have been used for performing other useful work instead.

A *task* in this context is just a unit of work, which will eventually be executed by a thread once one is ready. Even with a limited number of threads, an arbitrary amount of tasks can be scheduled by placing them in a queue until they can be executed. In the current implementation, a task corresponds to the execution of a single method, while smart scheduling may require that a task involves execution of multiple methods in sequence. Let the plan be the one seen in Figure 9.3. With the current scheduler, m_3 could be scheduled right after m_2 , in which case the second computation would not be able to start anyway. Meanwhile, the green or blue path could have been computed instead. In this example, the first task that

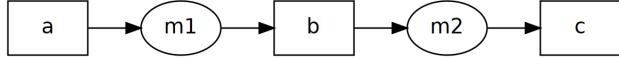


Figure 9.2: Scheduling a chain is done inefficiently, with one task per method even when one is dependent on another. The second task will then have to wait for the first to complete, wasting a thread.

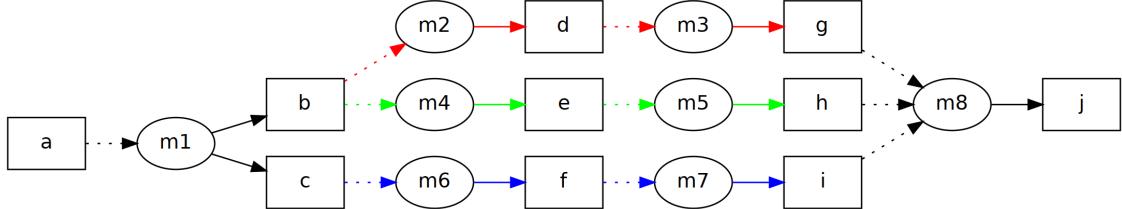


Figure 9.3: In a plan such as this one, a smarter delegation of tasks can be used to (1) execute the plan faster and (2) use fewer threads to do so.

must be completed is m_1 , followed by three separate tasks:

1. m_2 then m_3
2. m_4 then m_5
3. m_6 then m_7

Then finally the last task, the execution of m_8 .

By combining the execution of multiple methods into one task, we only require three threads to always compute all paths in the DAG in parallel, while the current solution may schedule the entire blue and green paths before starting the red path. In this case, if we are using 4 threads that compute m_2 , m_3 , m_4 and m_5 , then only m_2 and m_4 execute in parallel. Then, when one of them completes, the computation of m_6 will start.

This issue becomes worse the longer the chains are: with three chains of length n each, the entire first chain may be computed sequentially, followed by the second, then the third. This negates half the point of using multithreaded method execution¹, and would be very nice to avoid.

9.2.1 Breadth-First Scheduling

As long as the plan is still topologically ordered, we can modify its order as we please. By scheduling methods in a breadth-first manner, we prioritize methods with few dependencies.

Continuing with the example from Figure 9.3, we can change the plan from $[m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8]$ to the equally valid plan $[m_1, m_2, m_4, m_6, m_3, m_5, m_6, m_8]$. The latter is created by performing a breadth-first search through the solution graph from *root-methods*, i.e., methods with no dependencies. We now schedule methods with the fewest dependencies first, instead of wasting a thread on a method that must wait for its

¹The other one being that the main thread is not blocked.

inputs to be computed. In the previous plan, m_2 and m_3 are scheduled successively, even though m_3 cannot make progress until m_2 is done. In the new plan, we schedule m_2 and m_4 successively instead, which lets us execute them in parallel.

To implement this, start by maintaining a count of the references to each method. Schedule all methods with zero references, then for each one, decrement the reference count of all methods that depend on them.

```
let scheduled: Queue = [];
for m in plan {
    if m.refs == 0 {
        schedule(m);
        scheduled.enqueue(m);
    }
}
while !scheduled.is_empty() {
    let m = scheduled.dequeue();
    for d in m.dependents {
        if d.refs == 1 {
            schedule(d);
            scheduled.enqueue(d);
        }
        d.refs -= 1;
    }
}
```

The first step works since there are no inputs for the methods to wait for. Once there are none with no dependencies left, we must move on to scheduling ones that have one dependency. If we look at the dependencies of the methods just scheduled and find one with one reference, then we know that the current method is the only one it depends on. We can thus schedule it and decrement its reference count to zero. The methods that did not have a count of one must depend on other methods we must schedule first. Continuing the search from the newly scheduled methods lets us traverse the DAG further, repeating the process.

9.2.2 Multi-Method Tasks

Instead of having tasks consist of executing a single method, we can combine independent chains into larger tasks. This approach would require a lot more changes to the current implementation. In the previous example, we would instead have the five tasks below:

1. m_1
2. m_2, m_3

3. m_4, m_5

4. m_6, m_7

5. m_8

Each of these combined tasks would be scheduled as usual.

Any valid plan would start with task 1 to maintain topological ordering, but any ordering of the three that follow are valid. With this solution, we would not schedule tasks that depend on other non-completed tasks unless required, and each thread would compute the entire chain in one go instead of splitting it between threads.

9.2.3 Deferred Scheduling

The above scheduling strategies do not take the execution time of tasks into account. This means that they may end up scheduling tasks that can only be executed after a long time. If we instead let tasks schedule their dependencies upon completion, tasks will never be scheduled until they can actually be executed.

For instance, if we have two chains $[a, b, c]$ and $[d, e, f]$, then the breadth-first scheduling strategy with four threads would assign threads to a and d followed by b and e . If d is very slow, then the thread assigned to e will be idle for a long time. What we could do instead is to schedule all methods with no dependencies first (roots), then add a callback for them to call upon completion. This callback would decrement the dependency counter of the method's dependents, and if they have no dependencies left then they are scheduled. For the previous two-chain example, this would mean that a and d are scheduled, and upon a 's completion, b is scheduled. If b completes before d then c will be scheduled. Once d actually completes, e is scheduled, but no work was wasted on it before it could actually be computed.

Note that some extra care must be taken when implementing this in a multithreaded context. Say two threads complete task 1 and task 2 respectively, both of which are dependencies of task 3. If both threads decrement task 3's counter at the same time, then it will be 0 for both of them, which makes both threads schedule the task. This can be corrected by using the atomic operation `fetch_sub` that decrements the value, then returns the previous one.

It is also important that we do not schedule any tasks before having set all callbacks, as this could result in some tasks never being scheduled. An example of this is if we have three tasks $a \rightarrow b \rightarrow c$: a is scheduled first since it is a root, and upon completion, it schedules b . If b completes before its callback is set, then c is never scheduled. A solution to this is to set all callbacks before scheduling any tasks.

The scheduler is not implemented in `hotdrink-rs`, but an experimental implementation of the algorithm exists. The implementation can be found in the repository under `hotdrink-rs/examples`, and can be run using `cargo run --example deferred_scheduling`. If more details about the execution is desired, logging can be enabled with by setting the environment variable `RUST_LOG` to `info` or `trace`.

Algorithm 2: DeferredScheduling

Input: A plan

```
1 roots = ∅;
2 for m ∈ plan do
3     /* Find root tasks */  

4     if m.deps_left = 0 then /*/  

5         roots ← ∪ {m};  

6     end  

7     /* Set callback to schedule dependents */  

8     begin m.on_completion  

9         for d ∈ m.dependents do  

10            if d.deps_left.fetch_sub(1) = 1 then  

11                schedule(d);  

12            end  

13        end  

14    end  

15    /* Schedule root tasks */  

16 for r ∈ roots do  

17     schedule(r);  

18 end
```

9.3 Using Procedural Macros for the Component DSL

While it may be possible to add more compile-time checks to the declarative `component!` macro, we believe a better approach would be to turn it into a *procedural macro* [42, 71]. Declarative macros are limited in what they can express, e.g., they cannot easily compare identifiers, something which procedural macros have no problem with.

The first issue with the existing macro is that many type annotations must be repeated unnecessarily. Though the programmer knows that the concrete type of `i` is `i32`, the compiler loses this information when the value is stored as an enum variant; while the type can be detected during runtime, the code has already been generated by the macro, and no compile-time errors are provided to the programmer.

```
// Compile-time information about i's type is lost
let i: i32 = 5;
let ios: IntOrString = IntOrString::Int(i);
// Must now use ios as IntOrString, not i32

// We can check at runtime
match ios {
    IntOrString::Int(i) => ...,
```

```
    IntOrString::String(s) => ...,
}
```

A similar issue would apply when using the Any trait, except that we do not even know the possible variants.

```
let i: i32 = 5;
let any: Box<dyn Any> = Box::new(i);

// We can check if it has a specific type at runtime
match any.downcast::<i32>() {
    Ok(i) => ...,
    Err(e) => ...,
}
```

You can see an example of the redundant type annotations below. The programmer must specify types in method parameter lists, even though they have already done so in the variable declarations.

```
component Component {
    let i: i32, s: String;
    constraint C {
        m(i: &i32) -> [s] = ret![...]
        m(s: &String) -> [i] = ret![...]
    }
}
```

With a procedural macro, the Abstract Syntax Tree (AST) of the macro contents can be modified programmatically, which removes the need for type annotations in method parameter lists. This removes a whole class of possible user errors from the macro since users can no longer use wrong type annotations.

It may also be possible to assert that the method body returns a value of the correct type at compile-time, guaranteeing that the concrete type of a variable does not change. For instance, the current implementation allows erroneous code such as this:

```
component Component {
    let i: i32, j: i32, s: String;
    constraint C {
        m(i: &i32) -> [j] = ret![String::from("abc")];
    }
}
```

This code is accepted because the String has an Into implementation that allows it to be converted to the enum used to store values in this component. However, the concrete type of j will now change, which leads to a runtime error if an attempt to use it as &i32.

9.4 Undo and Redo in Mutable Constraint Systems

The current implementation of undo and redo is experimental and mainly serves to demonstrate another benefit of letting the constraint system control all variable values.

An issue with the current implementation is that it does not take modifications to the constraint system into account; adding a new constraint and then undoing the latest change will revert variable values, but not remove the newly added constraint. This could lead to the latest added constraint being broken. To fix this, undo and redo must not only modify variable values, but also ensure that the state of the constraint system itself matches the selected generation.

9.5 Dynamic Constraint System Construction

The `component!` macro makes it much easier to create components, but there are no macros for creating individual constraints and methods. Consequently, dynamic construction of constraint systems (adding new constraints during runtime) requires knowledge about how to create objects of these types. Boilerplate, such as automatic conversion of method inputs and wrapping of method outputs, must also be implemented manually.

An issue with having macros for individual constraints and methods is that they do not have access to the variables or types from the outer component, which could limit its capabilities.

9.6 Pre- and Postconditions

An interesting subject for further study is pre- and postconditions for constraints, or even for individual methods. The main motivation for support for pre- and postconditions would be to enable automated testing of methods, as well as to offer a way of specifying the constraint to be maintained explicitly in code. In the current implementation, constraints are only defined by how their methods are implemented; there is no guarantee the methods enforce the intended relation between variables. See the erroneous implementation of the `Sum` constraint below.

```
// a + b = c
constraint Sum {
    m(a: &i32, b: &i32) -> [c] = ret![a - b];
    m(a: &i32, c: &i32) -> [b] = ret![a - c];
    m(b: &i32, c: &i32) -> [a] = ret![b + c];
}
```

The code actually enforces the constraint $a = b + c$, not $a + b = c$, but unless this is tested, the mistake may go unnoticed. There is thus a discrepancy between the intended constraint written as a comment, and the one actually enforced by the method implementations.

To fix this, we could implement *postconditions* by adding assertions to constraints, or even individual methods.

```
// Example syntax
@postcond(a + b == c)
@testcase(a=1, b=2, c=3)
constraint Sum { ... }
```

By verifying that the postcondition holds after each method execution, the programmer could be notified if the constraint is not enforced after solving. We would thus have a form of testable documentation that is verified during runtime. This could also be extended with test case annotations that generate tests to be run by `cargo test`.

Postconditions could also be used for solver optimization: if a postcondition already holds, then there is no reason to re-enforce the constraint (unless the method has side effects). Configuration of when to check postconditions could be added; checking for errors could for instance only be used if the library was compiled in debug mode.

Preconditions could also be useful: when values are expected to satisfy certain requirements before the constraint is enforced, a precondition could be added to verify it. If methods would receive invalid values, an error could be reported to the user.

```
@precond(is_valid_email(s), "The supplied email is not valid")
constraint ConstraintInvolvingEmail { ... }
```

This is already possible by conditionally using the `fail!` macro in a method, but requires the logic to be duplicated through all the method bodies. Another possibility would be to create a separate validator constraint:

```
constraint EmailIsValid {
    // Alternative 1, may fail with `fail!`
    m(s: String) -> [s] = { ... }
    // Alternative 2, sets a status flag
    m(s: String) -> [s_status] = { ... }
}
```

The first alternative would automatically propagate the error to all methods that use the result. The second provides more flexibility, but the programmer must manually check the status flag.

9.7 Enabling and Disabling Components

Users may perform actions that make parts of a GUI irrelevant, for instance checking a checkbox. If this part is modeled as a component, it should be made easy to disable all widgets for variables within it to show

the user that they have no effect. For the programmer to do this manually, they would have to keep track of all relevant variables, and connect the widgets to the controls for enabling and disabling the component. The component itself already knows which variables it contains, so this results in the programmer having to do redundant work. A better alternative is to have two methods `enable` and `disable` implemented for components, which will then automatically send an event to all of its variables. The programmer would only have to decide when to call the methods, and then handle one additional event type for variables.

9.8 Improvements to Subscribing from JavaScript

Currently, `subscribe` function in the constraint system wrapper takes one argument per callback, such as the pending-, ready-, and error-callbacks. It may not be clear to the programmer which argument correlates to which callback, and ignoring an event type sometimes requires explicitly ignoring it by setting its handler to `undefined`. The following example demonstrates how the three callbacks can be specified.

```
comp.subscribe("a",
  v => { ... }, // handle ready-events
  undefined // explicitly ignore pending-events
  // implicitly ignore error-events
);
```

The programmer can provide fewer arguments to implicitly ignore setting trailing callbacks. Callbacks less likely to be provided should therefore be last, which is why the ready-callback is the first parameter.

It would likely be much more ergonomic to allow the programmer to pass in an object with only the event handlers they want:

```
comp.subscribe("a", {
  on_pending = () => { ... },
  on_ready   = () => { ... },
  on_error   = () => { ... },
});
```

Any unspecified callbacks are interpreted as not wanting to handle that specific event. If future implementations add more events, the current solution requires even more arguments, which would not be backward-compatible if the order changes. By using an object instead, the programmer would not have to make any changes unless they wanted to handle the new events.

The reason why we do not do this already is that `wasm-bindgen` does not support conversion of objects to an appropriate struct such as a `HashMap`.

```
#[wasm_bindgen]
pub fn subscribe(..., callbacks: HashMap<String, Function>) { ... }
```

However, by serializing the data with `serde` or `serde-wasm-bindgen` it should be possible to get this to work as desired [58, 56, 57].

Chapter 10

Conclusion

In this project, we have explored the design space of constraint-based GUI programming for web applications. In particular, we have focused on static typing and multithreading. Concretely, we implemented a new version of HotDrink with Rust and WebAssembly, examined its performance and responsiveness-guarantees, and compared it to earlier implementations.

We compared the planning algorithm of `hotdrink-rs` with two previous implementations and found that our implementation is several orders of magnitude faster on systems where the pruning optimization technique is applicable. On the other hand, the incremental planning algorithms of the previous implementations can be much faster in constraint systems where this is not the case. This indicates that pruning alone is not sufficient to provide good performance for all constraint systems, and that making the planner incremental may be an optimization that is worthwhile to implement.

`hotdrink-rs` also supports running constraint satisfaction methods in parallel, which guarantees responsiveness in the face of slow (or even non-terminating) computations. It is compatible with both Rust's standard library threads when compiling to native instructions, as well as Web Worker-based threads when compiling to WebAssembly. The Web Worker-based threads can also be canceled in order to avoid computing results that are no longer required.

Finally, we also present new data structures for storing constraints and methods that can roughly halve the memory consumption for the constraint systems used in our benchmarks, and save even more in constraint systems with more variables and methods in each constraint.

Glossary

API	Application Programming Interface. 15, 16, 18, 20, 22, 25, 26, 27, 44, 50, 99, 102
AST	Abstract Syntax Tree. 110
crate	A Rust package. 61, 71, 90
DAG	Directed Acyclic Graph. vi, 106, 107
DFA	Deterministic Finite Automaton. 13
DOM	Domain Object Model. 101
DSL	Domain-Specific Language. 4, 5
eDSL	Embedded Domain-Specific Language. 16
GUI	Graphical User Interface. i, 1, 2, 3, 4, 7, 8, 12, 13, 14, 38, 56, 57, 59, 60, 64, 65, 66, 68, 79, 102, 103, 104, 112, 115
item	Functions, types, etc. in Rust. 15
WebAssembly	A binary instruction format that can be executed in a web browser. 5, 44, 45, 48, 49, 50, 51, 54, 55, 60, 77, 101

Bibliography

- [1] *Abort a running script.* URL: <https://html.spec.whatwg.org/multipage/webappapis.html#abort-a-running-script> (visited on 2021-07-14).
- [2] *Adam and Eve.* URL: http://stlab.adobe.com/group__asl__overview.html (visited on 2021-06-13).
- [3] *bit-vec.* URL: <https://crates.io/crates/bit-vec> (visited on 2020-10-27).
- [4] *Box::leak.* URL: <https://doc.rust-lang.org/std/boxed/struct.Box.html#method.leak> (visited on 2021-06-15).
- [5] *cargo-flamegraph.* URL: <https://github.com/flamegraph-rs/flamegraph> (visited on 2021-07-09).
- [6] *concat_idents.* URL: https://doc.rust-lang.org/std/macro.concat_idents.html (visited on 2021-07-23).
- [7] *ConstraintJS.* URL: <https://soney.github.io/constraintjs/> (visited on 2021-06-14).
- [8] Alex Crichton. *Multithreading Rust and Wasm.* Oct. 24, 2018. URL: <https://rustwasm.github.io/2018/10/24/multithreading-rust-and-wasm.html> (visited on 2021-01-14).
- [9] *Dedicated workers and the Worker interface.* URL: <https://html.spec.whatwg.org/multipage/workers.html#dom-worker-terminate-dev> (visited on 2021-07-14).
- [10] *Ensure that JsValue isn't considered Send.* URL: <https://github.com/rustwasm/wasm-bindgen/pull/955> (visited on 2021-06-29).
- [11] *Fearless Concurrency.* URL: <https://doc.rust-lang.org/book/ch16-00-concurrency.html> (visited on 2021-07-14).
- [12] *Flow.* URL: <https://flow.org/> (visited on 2021-06-02).
- [13] Charles Gabriel Foust. “Guaranteeing Responsiveness and Consistency in Dynamic, Asynchronous Graphical User Interfaces”. en. Accepted: 2016-07-08T15:06:17Z. Doctoral dissertation. Texas A & M University, Jan. 2016. URL: <https://oaktrust.library.tamu.edu/handle/1969.1/156821> (visited on 2020-10-21).
- [14] Charles Gabriel Foust. *HotDrink in TypeScript.* URL: <https://github.com/HotDrink/hotdrink> (visited on 2021-06-02).

- [15] 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. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 121–130. ISBN: 9781450336871. DOI: 10.1145/2814204.2814207. URL: <https://doi.org/10.1145/2814204.2814207>.
- [16] John Freeman et al. “Helping programmers help users”. In: *Proceedings of the 10th ACM international conference on Generative programming and component engineering*. 2011, pp. 177–184.
- [17] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. “An Incremental Constraint Solver”. In: *Commun. ACM* 33.1 (Jan. 1990), pp. 54–63. ISSN: 0001-0782. DOI: 10.1145/76372.77531. URL: <https://doi.org/10.1145/76372.77531>.
- [18] *GC Post-v1 Extensions*. URL: <https://github.com/WebAssembly/gc/blob/dfc44cc394d71f6a7d94f2d87fb6b3d98b440356/proposals/gc/Post-MVP.md> (visited on 2021-07-06).
- [19] Andreas Haas et al. “Bringing the Web up to Speed with WebAssembly”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: <https://doi.org/10.1145/3140587.3062363>.
- [20] Magne Haveraaen and Jaakko Järvi. “Semantics of multiway dataflow constraint systems”. en. In: *Journal of Logical and Algebraic Methods in Programming* 121 (June 2021), p. 100634. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2020.100634. URL: <https://www.sciencedirect.com/science/article/pii/S235222082030119X> (visited on 2021-02-18).
- [21] *Interior Mutability*. URL: <https://doc.rust-lang.org/reference/interior-mutability.html> (visited on 2020-09-15).
- [22] Jaakko Järvi. *HotDrink in Flow*. URL: <https://git.app.uib.no/Jaakko.Jarvi/hd4> (visited on 2021-06-02).
- [23] Jaakko Järvi, Gabriel Foust, and Magne Haveraaen. “Specializing Planners for Hierarchical Multi-Way Dataflow Constraint Systems”. In: *SIGPLAN Not.* 50.3 (Sept. 2014), pp. 1–10. ISSN: 0362-1340. DOI: 10.1145/2775053.2658762. URL: <https://doi.org/10.1145/2775053.2658762>.
- [24] Jaakko Järvi et al. “Algorithms for User Interfaces”. In: *SIGPLAN Not.* 45.2 (Oct. 2009), pp. 147–156. ISSN: 0362-1340. DOI: 10.1145/1837852.1621630. URL: <https://doi.org/10.1145/1837852.1621630>.
- [25] Jaakko Järvi et al. “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. Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 25–32. ISBN: 9781450315760. DOI: 10.1145/2364394.2364399. URL: <https://doi.org/10.1145/2364394.2364399>.

- [26] Jaakko Järvi et al. “Property models: from incidental algorithms to reusable components”. In: *Proceedings of the 7th international conference on Generative programming and component engineering*. 2008, pp. 89–98.
- [27] *Java Thread Primitive Deprecation*. URL: <https://docs.oracle.com/javase/9/docs/api/java/lang/doc-files/threadPrimitiveDeprecation.html> (visited on 2021-07-14).
- [28] *JavaScript*. URL: <https://developer.mozilla.org/en-US/docs/Web/javascript> (visited on 2021-06-02).
- [29] *JsValue*. URL: https://rustwasm.github.io/wasm-bindgen/api/wasm_bindgen/struct.JsValue.html (visited on 2021-01-14).
- [30] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: [10.1109/SP.2019.00002](https://doi.org/10.1109/SP.2019.00002).
- [31] *ladder-100.js*. URL: <https://github.com/HotDrink/hotdrink/blob/c96f3b03c932206ff5fe17c7e1c1517c64f54fc7/test/models/ladder-100.js> (visited on 2021-07-18).
- [32] *Loading and running WebAssembly code*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly>Loading_and_running (visited on 2021-05-19).
- [33] Guillaume Cedric Marty. *How fast are web workers?* July 2, 2015. URL: <https://hacks.mozilla.org/2015/07/how-fast-are-web-workers/> (visited on 2021-01-14).
- [34] *Mitigations landing for new class timing attack*. URL: <https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/> (visited on 2021-06-29).
- [35] *Monomorphization*. URL: <https://rustc-dev-guide.rust-lang.org/backend/monomorph.html> (visited on 2021-07-06).
- [36] B.A. Myers et al. “The Amulet environment: new models for effective user interface software development”. In: *IEEE Transactions on Software Engineering* 23.6 (1997), pp. 347–365. DOI: [10.1109/32.601073](https://doi.org/10.1109/32.601073).
- [37] Brad A Myers. *Why are human-computer interfaces difficult to design and implement*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Dept of Computer Science, 1993.
- [38] Brad A. Myers. “Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs”. In: *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*. UIST ’91. Hilton Head, South Carolina, USA: Association for Computing Machinery, 1991, pp. 211–220. ISBN: 0897914511. DOI: [10.1145/120782.120805](https://doi.org/10.1145/120782.120805). URL: <https://doi.org/10.1145/120782.120805>.
- [39] Brad A. Myers et al. “Garnet Comprehensive Support for Graphical, Highly Interactive User Interfaces”. In: *Readings in Human-Computer Interaction*. Ed. by RONALD M. BAECKER et al. Interactive Technologies. Morgan Kaufmann, 1995, pp. 357–371. ISBN: 978-0-08-051574-8. DOI: [10.1016/B978-0-08-051574-8.50037-6](https://doi.org/10.1016/B978-0-08-051574-8.50037-6). URL: <https://www.sciencedirect.com/science/article/pii/B9780080515748500376>.

- [40] Jakob Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. ISBN: 9780080520292.
- [41] Stephen Oney, Brad Myers, and Joel Brandt. “ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States”. In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST ’12. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2012, pp. 229–238. ISBN: 9781450315807. DOI: 10.1145/2380116.2380146. URL: <https://doi.org/10.1145/2380116.2380146>.
- [42] *Procedural Macros*. URL: <https://doc.rust-lang.org/reference/procedural-macros.html> (visited on 2021-07-12).
- [43] *pthread_cancel*. URL: https://linux.die.net/man/3/pthread_cancel (visited on 2021-07-14).
- [44] *pthread_setcanceltype*. URL: https://linux.die.net/man/3/pthread_setcanceltype (visited on 2021-07-14).
- [45] *rayon*. URL: <https://crates.io/crates/rayon> (visited on 2020-11-10).
- [46] *Raytrace Parallel Docs*. URL: <https://rustwasm.github.io/docs/wasm-bindgen/examples/raytrace.html> (visited on 2021-01-14).
- [47] *Reactive Manifesto*. URL: <https://www.reactivemanifesto.org/> (visited on 2021-07-05).
- [48] *RefCell<T> and the Interior Mutability Pattern*. URL: <https://doc.rust-lang.org/book/ch15-05-interior-mutability.html> (visited on 2020-09-15).
- [49] Micha Reiser and Luc Bläser. “Accelerate JavaScript Applications by Cross-Compiling to WebAssembly”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 10–17. ISBN: 9781450355193. DOI: 10.1145/3141871.3141873. URL: <https://doi.org/10.1145/3141871.3141873>.
- [50] *Rust*. URL: <https://www.rust-lang.org/> (visited on 2021-06-02).
- [51] *Rust and WebAssembly*. URL: <https://rustwasm.github.io/docs/book/> (visited on 2021-07-25).
- [52] *rust-webpack-template*. URL: <https://rustwasm.github.io/docs/book/reference/project-templates.html> (visited on 2021-08-01).
- [53] *Rustonomicon*. URL: <https://doc.rust-lang.org/nomicon/ffi.html> (visited on 2021-07-26).
- [54] Michael Sannella. “Skyblue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction”. In: *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*. UIST ’94. Marina del Rey, California, USA: Association for Computing Machinery, 1994, pp. 137–146. ISBN: 0897916573. DOI: 10.1145/192426.192485. URL: <https://doi.org/10.1145/192426.192485>.

- [55] Max Schäfer. “Refactoring Tools for Dynamic Languages”. In: *Proceedings of the Fifth Workshop on Refactoring Tools*. WRT ’12. Rapperswil, Switzerland: Association for Computing Machinery, 2012, pp. 59–62. ISBN: 9781450315005. doi: 10.1145/2328876.2328885. URL: <https://doi.org/10.1145/2328876.2328885>.
- [56] *serde*. URL: <https://serde.rs/> (visited on 2021-07-16).
- [57] *serde-wasm-bindgen*. URL: <https://github.com/cloudflare/serde-wasm-bindgen> (visited on 2021-07-16).
- [58] *Serializing and Deserializing Arbitrary Data Into and From JsValue with Serde*. URL: <https://rustwasm.github.io/docs/wasm-bindgen/print.html#serializing-and-deserializing-arbitrary-data-into-and-from-jsvalue-with-serde> (visited on 2021-07-16).
- [59] *SharedArrayBuffer*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer (visited on 2021-06-28).
- [60] *SpectreAttack*. URL: <https://spectreattack.com/> (visited on 2021-06-28).
- [61] Ingvar Stepanyan. *Using WebAssembly threads from C, C++ and Rust*. July 12, 2021. URL: <https://web.dev/webassembly-threads/> (visited on 2021-07-18).
- [62] *Supported Rust Types and their JavaScript Representations*. URL: <https://rustwasm.github.io/docs/wasm-bindgen/reference/types.html> (visited on 2021-07-06).
- [63] Rudi Blaha Svartveit. *hotdrink-rs crate*. URL: <https://crates.io/crates/hotdrink-rs> (visited on 2021-07-28).
- [64] Rudi Blaha Svartveit. *hotdrink-rs docs*. URL: <https://docs.rs/crate/hotdrink-rs> (visited on 2021-07-28).
- [65] Rudi Blaha Svartveit. *hotdrink-rs repository*. URL: <https://github.com/HotDrink/hotdrink-rs/tree/thesis> (visited on 2021-07-28).
- [66] *Systems Performance Work Guided By Flamegraphs*. URL: <https://github.com/flamegraph-rs/flamegraph#systems-performance-work-guided-by-flamegraphs> (visited on 2021-07-09).
- [67] *Terminate a worker*. URL: <https://html.spec.whatwg.org/multipage/workers.html#terminate-a-worker> (visited on 2021-07-14).
- [68] *The structured clone algorithm*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm (visited on 2021-06-24).
- [69] *Thread::cancel() support*. URL: <https://internals.rust-lang.org/t/thread-cancel-support/3056> (visited on 2021-07-14).
- [70] *Threads Proposal for WebAssembly*. URL: <https://github.com/WebAssembly/threads> (visited on 2021-01-14).

- [71] David Tolnay. *Procedural Macros Workshop*. URL: <https://github.com/dtolnay/proc-macro-workshop> (visited on 2021-07-12).
- [72] Sam Van den Vonder et al. “Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 27–33. ISBN: 9781450355155. doi: 10.1145/3141858.3141863. URL: <https://doi.org/10.1145/3141858.3141863>.
- [73] Brad Vander Zanden. “An Incremental Algorithm for Satisfying Hierarchies of Multiway Dataflow Constraints”. In: vol. 18. 1. New York, NY, USA: Association for Computing Machinery, Jan. 1996, pp. 30–72. doi: 10.1145/225540.225543. URL: <https://doi.org/10.1145/225540.225543>.
- [74] *Waker*. URL: <https://doc.rust-lang.org/std/task/struct.Waker.html> (visited on 2021-08-01).
- [75] *wasm-bindgen*. URL: <https://rustwasm.github.io/docs/wasm-bindgen/> (visited on 2021-07-25).
- [76] *wasm-bindgen-rayon*. URL: <https://crates.io/crates/wasm-bindgen-rayon> (visited on 2021-07-18).
- [77] *wasm-mt*. URL: <https://crates.io/crates/wasm-mt> (visited on 2021-01-14).
- [78] *wasm-pack*. URL: <https://rustwasm.github.io/docs/wasm-pack/> (visited on 2021-07-25).
- [79] *wasm_thread*. URL: https://crates.io/crates/wasm_thread (visited on 2021-01-14).
- [80] *web_worker*. URL: https://crates.io/crates/web_worker (visited on 2021-01-14).
- [81] *WebAssembly*. URL: <https://webassembly.org/> (visited on 2021-06-02).
- [82] *WebAssembly*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly?source=techstories.org#in_a_nutshell (visited on 2021-06-04).
- [83] *WebAssembly Instance*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Instance (visited on 2021-06-29).
- [84] *WebAssembly Memory*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Memory (visited on 2021-06-29).
- [85] *WebAssembly Module*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Module (visited on 2021-06-29).
- [86] N. Wirth. “A plea for lean software”. In: *Computer* 28.2 (1995), pp. 64–68. doi: 10.1109/2.348001.
- [87] *worker.js*. URL: <https://github.com/rustwasm/wasm-bindgen/blob/837e354aefe672dc3111f6658a9b0705846e859a/examples/raytrace-parallel/worker.js> (visited on 2021-07-18).
- [88] *Worker.terminate()*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Worker/terminate> (visited on 2021-07-14).

- [89] Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. “Handling Bidirectional Control Flow”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). doi: 10.1145/3428207. url: <https://doi.org/10.1145/3428207>.