Towards a General Theory of Solving for Dependent Type Theory

Formalisations of Monadic Constructions Applicable to Solving for Recursive Data

Arved Reinhard Harald Friedemann

Submitted to Swansea University in fulfilment of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University
Wales
2025

Abstract

This thesis presents several results towards a general theory of solving. We create a formalism for monadic variables that can be used to enhance the capabilities of monads used for search. These monadic variables are created using a form of lattice-based variables, extending the concepts introduced in [61,62,75]. The pre-order of the lattice is used to ensure that variables keep their assignments on all successive states until a conflict occurs. Further, the lattice structure ensures that regardless of the assignment order, the variables always give the same result. Similar to other lattice-solving based formalisms [33] we generalise this idea to model partial solutions as an ever increasing search state. To make this idea more concrete however, we additionally give constructions to turn almost arbitrary data types into lattices that can be used in this generalised solving process. To express assignments resulting from arbitrary computation, we formalise how recursive functions expressed via folds can run on our lattice representation of data types. We provide improvements to the type representing the syntax of type theory [8, 24] and give a type-theoretic interpretation of the results from [84] to move towards incorporating the idea of self-improving solvers into type theory within the framework of our generalised solving theory. We present a range of examples of solvers created from our concepts and define a trajectory towards a general theory of solving.

Declarations

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

signed:
date: 01.06.2025 

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

signed:
date: 01.06.2025 

I hereby give consent for my thesis, if accepted, to be available for electronic sharing.

signed:
date: 01.06.2025 

The University's ethical procedures have been followed and, where appropriate, that ethical approval has been granted.

signed:
date: 01.06.2025 

Contents

This thesis is written in the categorical "we". It is meant to symbolise that we, in science, work together on a common goal: to protect intelligence and enhance its capabilities.

The concrete work in this thesis however is, unless stated otherwise, purely the work of its author.

To my dream in science...

I know you died.
But if you think I would just silently accept that...

think again.

Acknowledgements

As of now, the university has not established a formal procedure to ensure that acknowledgments are excluded from the review process. As a result, there remains a risk that any wording in the acknowledgments could be subject to unintended consequences, including the possibility of informal repercussions for perceived omissions or phrasing. [redacted]

Chapter 1

Introduction and Related Work

1.1 Introduction

Solving is a foundational problem in computer science, so much so that, in its most general definition, every problem in computer science becomes a solving problem. Every time we state a problem, we are actually giving a computable, recursive function `f : A → 𝔹` that can detect a solution of type `A`. Solving that problem, then, means finding a solution `a` of type `A` such that `f a ≡ true`. Using dependent type theory, this can be generalised even further: solving becomes the task of finding an element `a` of type `A`, where `A` represents a proposition from logics and `a` is its proof. In essence, every time we write code or prove a statement, we are manually performing a task that a solver could, in theory, do automatically.

Of course, such a general solver does not exist yet. There has been significant progress made on special solving problems like boolean satisfiability [20] or satisfiability modulo theories [15], but their input languages are highly restricted. Those restrictions make it hard for their solvers to be used for general recursive programming. On the other hand, solving systems that have a sufficiently expressive input language [22, 29, 59] often suffer from performance inefficiencies due to naïve search techniques or are only efficient on problems with the same expressivity restrictions as existing solvers [13, 34]. Furthermore, developing solvers for new problem classes, or even just proving the correctness of existing techniques, remains a labor-intensive and manual process [35, 77].

Theoretical frameworks for self-improving solvers like in [84] state how solvers could be used to automate their own creation and improvement process. These frameworks however use formalisms like Turing machines and first order logic that make it difficult to describe how exactly a self improving solver would ever realistically pass its first self improvement. This thesis is working towards a foundational theory of solving in the context of general dependent

type theory to address these issues. The ultimate goal is to one day have a theory robust enough to not only explain how and why solvers work but also to enable solvers to aid their own development.

This thesis achieves three major milestones. The first is a formalism for logical variables that can be integrated into a monadic context in order to enhance monadic solving procedures. We can wrap an arbitrary monad, potentially one used for search, within a special constrained state monad transformer we created that assures the traversed states remain related by a pre-order, such as the one induced by a semilattice instance on the state. Generalising the formalism from [61, 62, 75], we introduce a class of lens-like accessor functions to interact with the state. Our monadic actions derived from those accessors can then be shown to behave like logical variables. Crucially, once a variable has been assigned, it can be shown to never be reassigned. This is especially important in a parallelised context to make sure that several current and future branches actually work on the same assignment. Additionally, we provide a mechanism to dynamically create new variables during the solving process, addressing memory challenges often encountered for example in SAT-solving where all variables that could potentially be relevant have to be added to the problem input in advance [56, 87].

The second milestone is to create lattice instances for a broad class of recursive data types using the concept of containers [2]. The constraints on the data are minimal, enabling the representation of all non-indexed algebraic data types that do not use functions as constructor arguments. We further showcase how even those constraints could be lifted by improving upon the type for the syntax of type theory [8, 24], which represents the correct terms of type theory (including functions) while itself adhering to our minimal constraints on the data type. Additionally, we provide a mechanism to fragment general data using variables (or any other locally commutative monadic context). This will soon allow us to correctly encode general recursive data in a solving context with variables.

Finally, inspired by [33], we take initial steps toward a general theory of solving, connecting the results from this thesis into a lattice-solving based framework. We give first provably correct explanations on parts of the working mechanisms of fundamental techniques like unit propagation, branching and clause learning. Moreover, we outline a general framework for self-improving solvers that can be implemented as soon as the type for the syntax of type theory is fully completed. These contributions form a roadmap towards a general theory of solving that produces executable and provably correct programs. Our formalisms give first concrete steps on how to automatically lift functions checking solutions into a solving system generating solutions.

We emphasize that all results in this thesis have been formally verified by the interactive theorem prover Agda (version 2.7.0), with only few marked exceptions. We provide the code in [37] to allow for an independent verification of these results.

1.2 Overview of Results

This is a section with a short overview of the results in this thesis. A longer version can be found in Section 1.8. A visual representation of the concepts can be found in Section 1.6.

We will quickly summarize the results of each of the four main chapters of this thesis:

- Chapter 3, Monadic Solving
  - Showcasing the importance of variables for monadic solving
  - Creating the concept of **Monadic Variables**
  - Creating the concept of **Constrained State Monad Transformer**
  - Creating the concept of **Lattice Bi-Threshold Monadic Variables** and show that they can be used together with the constrained state monad transformer to form monadic variables
  - Showcasing how a SAT-Solver can be implemented with our lattice based monadic variables
  - Showcasing how a SAT-Solver could be created from the algebra of the evaluation function for boolean formulas
  - Creating a formalism to create new variables during a monadic computation in the constrained monad transformer
- Chapter 4, Data Types for Monadic Solving
  - Concept of **Commuting Contexts** to distribute general data types over monadic variables and running a large class of recursive function on them without changing the function implementation.
  - Construction of commutable contexts for the data types of shapely containers (data types where functions in the finite amount of constructor arguments are at most defined with a finite domain)
  - Construction of a general lattice instance for data types of containers with decidable equality on their shapes, resulting in the notion of a **latticed container**
  - Construction of monadic variables into the positions of lattices containers with decidable equality on positions
  - Improvement of the data type for the syntax of type theory (a "type of type theory") created by [8,24] to allow for sigma types and recursive functions
  - Showcasing the expressive strength of our monadic variables to potentially represent our type of type theory
- Chapter 5, Towards a General Theory of Solving

- A preliminary general formalism for optimal or self improving solvers in type theory
- A lattice based theory of solvers for type theory, with a formalism for equivalents of
  - unit propagation
  - branching
  - clause learning
- First theorems on speedups during the parallelisation of solvers
- First theorems on speedups using clause learning

1.3 Publications and Talks

1. The paper "Towards Clause Learning à la Carte through VarMonads" [38] was accepted for informal proceedings and presented at the 32nd International Symposium on Logic-Based Program Synthesis and Transformation 2022 (LOPSTR 2022), Tbilisi, Georgia, and Virtual. The talk was held by the author of this thesis, who was also first author on the paper.
2. "Propagator Networks for Unification and Proof Search": Talk at 37th British Colloquium for Theoretical Computer Science 2021 (BCTCS 2021), Online, Liverpool, UK
3. "Functional Solving Engines": Talk at 36th British Colloquium for Theoretical Computer Science 2020 (BCTCS 2020), Swansea, UK
4. "Verifiable Models to Machine Learning Tasks using SAT": Talk at 35th British Colloquium for Theoretical Computer Science 2019 (BCTCS 2019), Durham, UK
5. "Using Satisfiability Solvers for Problems in Machine Learning": Talk at the International Workshop on Logic and Computational Complexity 2019 (LCC'19), Patras, Greece. Joint work with Jay Morgan.

1.4 Overview of this Thesis

This thesis' results are presented across three main chapters.

Chapter 3 introduces variable-based monadic solving and develops the formalism of monadic variables and how to construct them.

Chapter 4 focuses on how to transform data to support the creation of the monadic variables from Chapter 3 and develops a formalism for executing recursive functions over the data. The chapter also describes and improves upon a "type of type theory", a data type that can represent the syntax of type theory, to showcase its compatibility with our developed data transformations.

Chapter 5 presents a general approach towards a theory of solving, proposing a formalism capable of creating self-improving solvers for dependent type theory

under reasonable assumptions. This chapter shows how the results in this thesis fit together and sets the trajectory for future work.

Additionally, Chapter 2 provides a comprehensive overview of all formalisms used throughout the thesis. While a deep understanding of computer science and functional programming remains required, this chapter explains this thesis' notation and revisits all foundational ideas underlying our own theories.

This introductory chapter concludes with three supporting sections. Section 1.5 answers some likely discussions about the thesis, offering a more global perspective on the research. Though this section is not self-contained, we encourage the readers to lightly familiarise themselves with its ideas before commencing with reading the thesis. Section 1.7 summarises related work in the field. Finally, Section 1.8 provides a more detailed overview of the thesis' results, omitting technical details that are addressed in the main chapters.

1.5 Initial Discussions

In this section we highlight possible discussions about our work in a more global context. The aim is to make it easier for the reader to understand and appreciate the general trajectory of this thesis and the approach we are taking. We also explain why there are some purposefully left gaps that are left to future research without jeopadising the general aim of the thesis.

1.5.1 The Connection between Monadic Program Execution and Solving

Writing a program in a monadic context is actually a generalisation to writing a function in the plain context that we are used to. When we write a function, we are creating an object from a context (the function arguments) by composing the objects via the function application operator (assuming we do case distinctions via eliminator functions). When writing a monadic program, we do something similar: we still have the usual values from the function arguments, but we compose them using the monadic bind instead. Values that are not yet in the monadic context are lifted into it using the `return` operator. If our monadic context is the identity context `M = \X → X`, we get the same function as if we had composed it in the identity context. Every conventional function can be represented that way. However, additional effects occur when the monadic context is changed. For example, having a value of type `A` bound in the `List` monadic context means that we may not have only one value of type `A`, but zero or more of them. The implementation of the function does not change, but we are running the function on a range of values now instead. Also, "order of computation" now matters as the monadic context may change depending on where and how often a monadic value is used. Beyond simple monads like `List`, advanced monads are better suited for search [55, 85], but for a range of general concepts, we do not care which exact monad is used for the search. This is particularly true when using monad transformers [66] to just add properties

to an existing monad. We will do so in Chapter 3 where we wrap a state around a potential solving monad. This is not to say that the concrete monad used for search would never be relevant as pointed out in Chapter 5, but it helps to understand why the focus in Chapter 3 is mainly on just executing functions in a monadic context, while only giving examples of how this leads to programs actually doing the solving.

1.5.2 Category Theory

Category theory can be used to give another great level of generality to our approaches. For example, when we state that solvers can only be built for specific universes: in category theory, constraining to a universe would mean to just use a slightly smaller category than the one for type, so solvers would then just be morphisms in a closed category that describe the desired solving behaviour. The main upside is that if we write functions to be elements of categories instead of being elements for the primitive type `Set`, we could then actually formally prove statements about how every recursive function can be used to create a respective solver for finding elements of its image. We however made a conscious decision to stick with only one formalism in this thesis: type theory. All statements are engineered and verified within type theory, ensuring that readers only need familiarity with this one formalism. This comes with well known caveats. For example, functors in type theory are actually endofunctors for the type `Set`, and even monads who in category theory are defined using endofunctors as a basis still are, in type theory, defined only for endomorphisms in the category of types (`Set`). This means that without deviating from traditional definitions, we cannot formulate what a monad is in a category of types that have a representation in a universe. In future research, this would be fixed by modeling category theory in type theory as has already started to be done in [48].

1.5.3 Container Derivatives

In [3], an approach to create a reference like type into data types created by containers is developed, called a *container derivative*. These can be used to create ”holes” into data that can be filled by assignments, similar to what our variables are achieving. There were two reasons for us not to use them for our variables. The first reason is that these container derivatives do not automatically give a lattice based theory, which we needed in order to give simpler constructions for solvers. The second reason was to not use them in order to keep the explanations around our theory minimal. In future research however, their formalism should be used to better align with the broader type theory generic programming ecosystem built around containers.

1.5.3.1 Variables are not just used to create a Reader Monad

In a reader monad, we can read from a state but not modify it. State modification can be somewhat imitated by running a reader computation locally on a different state, but as soon as we frequently change the state it is better to go with a full state monad. While the lattice bi-threshold variables created in this thesis do have a reading component that a lot of stess is put on in this thesis, they just as much have a formalization of how to write to a state without breaking the properties they have on read. Those writes are important for solvers to write (partial) solutions to an assignment represented by the state. Therefore, we are developing the concept of a constrained monad state transformer that can do both read and write operations and just using a reader monad is insufficient.

1.5.4 Cubical Type Theory

The proofs in this thesis use cubical Agda and cubical type theory as an underlying mechanism to prove equalities. The main reason is that it is the most advanced formalism to deal with equivalences in proof systems. It allows for the use of the univalence principle, where two types are considered equal iff they are isomorphic, without losing the constructive properties of dependent type theory. This means all of our implemented proofs form executable programs. Using this more advanced form of equality however also means that some proofs become more complicated than one might be used to in classical mathematics. Equalities are now objects and especially the often used *axiom-K* does not hold anymore, meaning that identity proofs `a ≡ a` between two same objects are no longer identical between each other. Two different proofs for equalities `a=a a=a' : a ≡ a` between two identical objects might no longer be considered equal in cubical type theory. This means that a lot of proofs that would just result from normalisation under the assumption that all identity proofs are equal no longer normalise as easily under cubical type theory, occasionally complicating proofs that would otherwise seem mundane. It should be noted that the proven results are way stronger than they would have been under classical mathematics as they contain the information of how to deal with univalence. Therefore, just because a proof would have been easy in classical mathematics it does not render this thesis' results any less interesting. It frequently occurs that here, additional, more complicated things about the nature of the proof are proven, strengthening the results beyond classical mathematics and ensuring that they can, under their full extent, be used in the real world as executable, provably correct programs.

1.5.4.1 Rigor of Presented Approaches

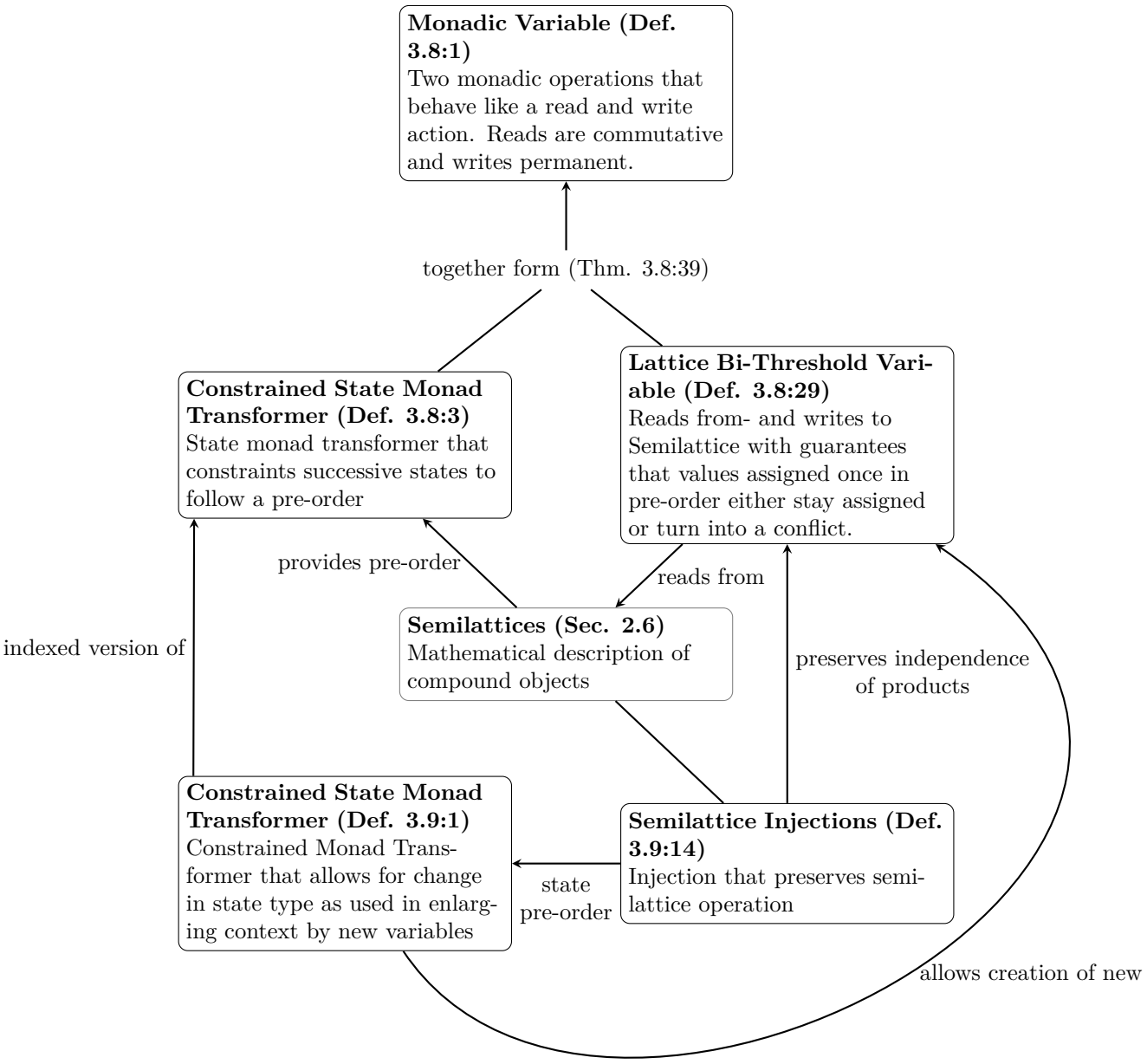
All statements and proofs in this thesis have been verified by Agda (version 2.7.0) to be correct, with only very few marked exceptions. This includes statements that have not been formally verified before. Every time we used a concept from other sources, we added our level of verification standards that have

at times not been present in the papers themselves. In some cases, just lifting existing research to the rigorous verification level used in this thesis is a result on its own. This also means that we excluded all research from this thesis that does not yet meet our quality criteria in terms of verification. This includes our earlier paper on mechanisms of clause learning [38] found in Appendix A which was written at a time when we did not yet apply those quality standards.

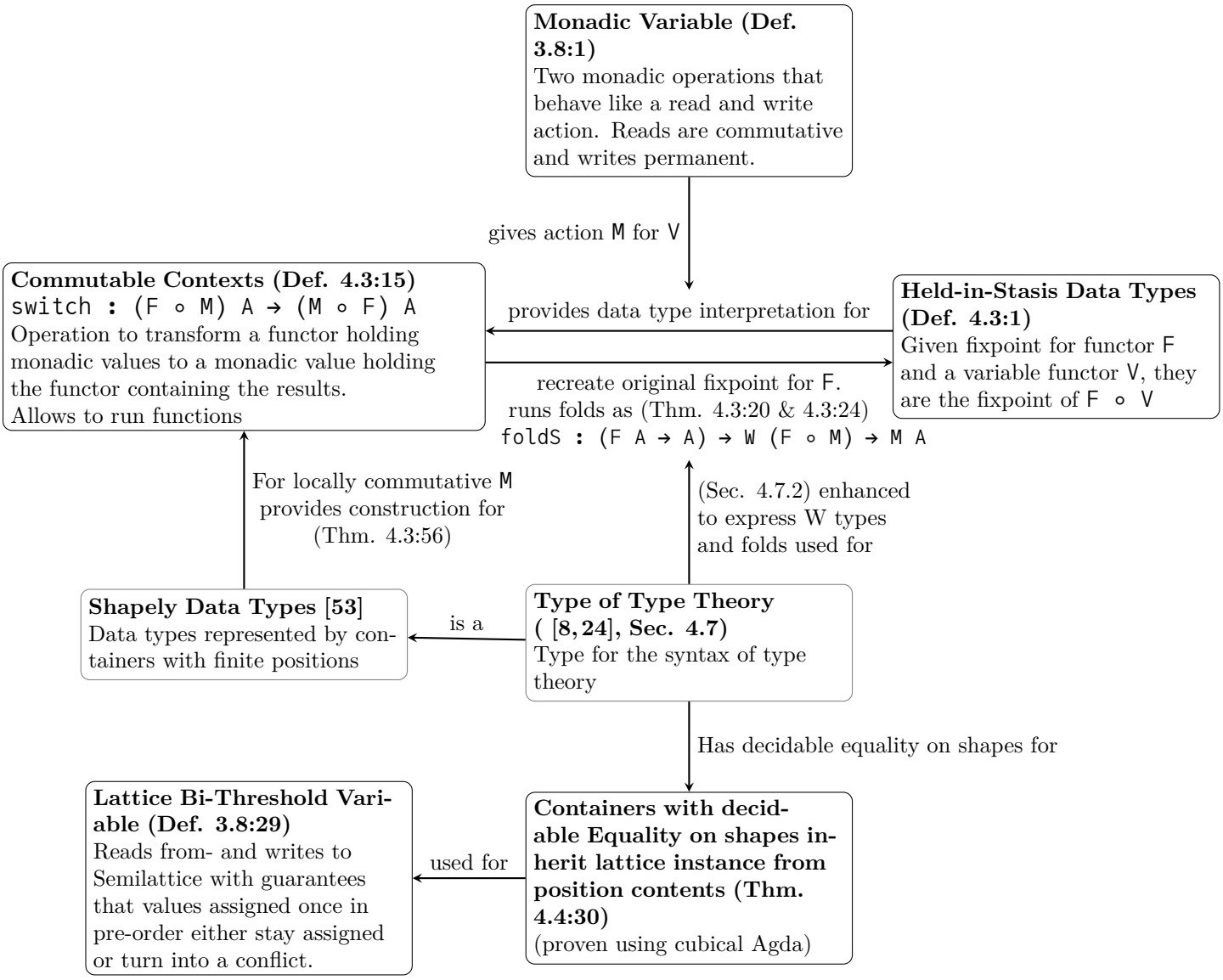
We find it imporant that in the future, all research is verified by an automatic proof checker in order to make it easier to exchange, especially complicated, results. We chose Agda specifically as a proof system because we believe that of all proof systems present, it has the most flexible syntax and theory, so we believe that it will be the most used proof system in the future.

1.6 Visualisation of Constructions

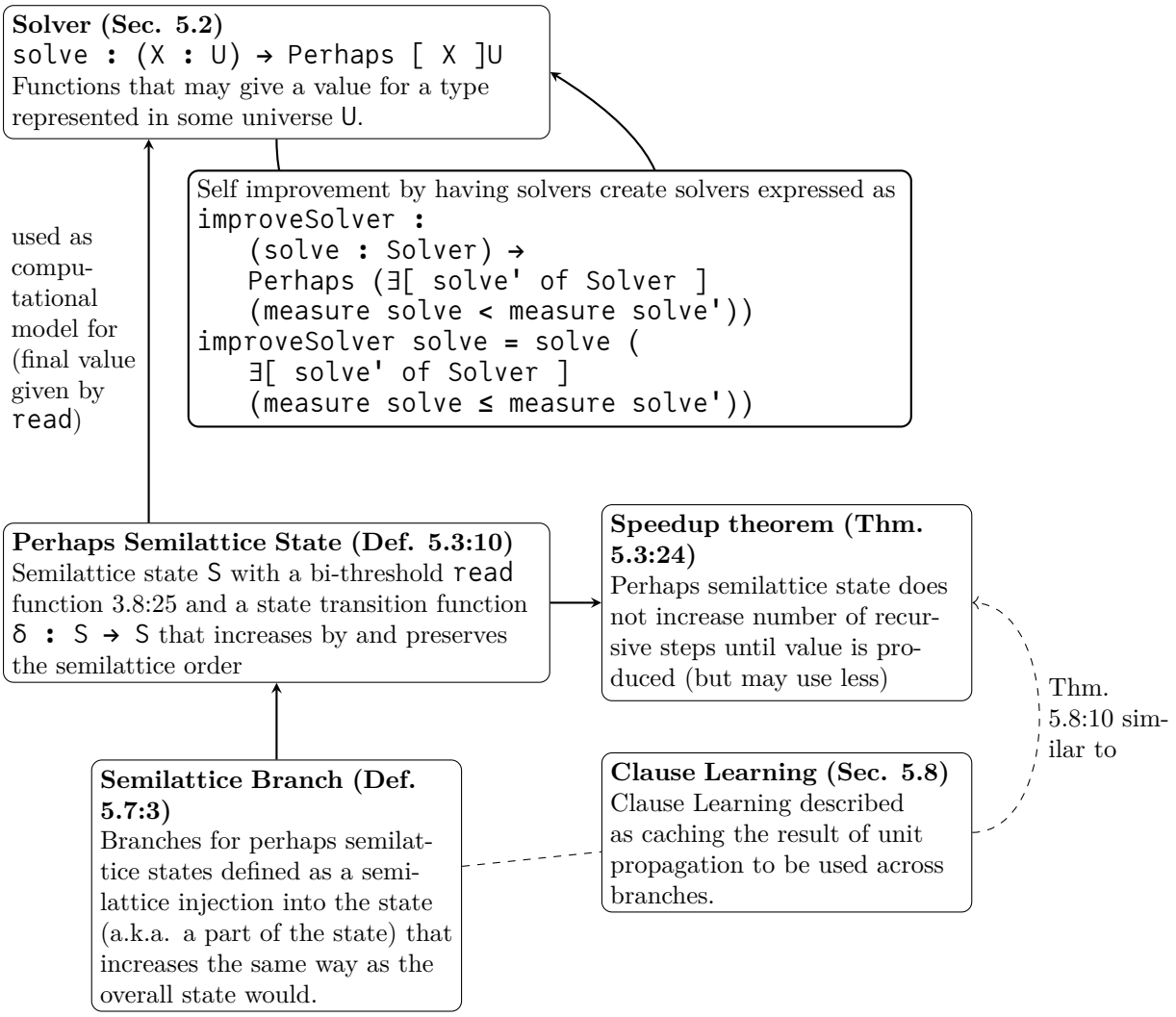
1.6.1 Chapter 3: Monadic Solving with Variables



1.6.2 Chapter 4: Data Types for Monadic Solving



1.6.3 Chapter 5: Towards A General Theory of Solving



1.7 History and Related Work

Solving is a well-investigated research topic with many facets and solutions for very specialised problems. One of the most well researched areas is the field of Boolean Satisfiability (SAT) Solving. Its main results are summarised in [20]. The problem became famous as probably being the first proven NP-complete problem (independently proven by Cook, Karp and Levin) [28, 90]; a class of problems whose efficient solution later was set to be one of the Clay Mathematics Institute Millenium Prize Problems [27] still unsolved to this day.

There are, however, solving procedures that quite often work well in practice. The first of these procedures was developed by Davis and Putnam (and later additionally refined by Logemann and Loveland) [31, 32] and hence called the DPLL-algorithm. We will describe this procedure in Chapter 2 Section 2.7. The procedure had a revival with the development of conflict driven clause learning (CDCL) [16, 68], a technique where information gathered from the solving process of one assignment is reused on a subsequent one. The technique achieved significant performance improvements in practice [79] and could later even be shown to give an exponential payoff in some special cases [17].

SAT itself is a rather restrictive language, as it does not allow for recursion in the formulas and it is often cumbersome to translate search problems into Boolean formulas [21, 25, 39, 46, 93]. To address this limitation, the formalism of SAT modulo Theories (SMT) was developed that enhanced the expressivity of SAT formulas by adding new atoms from different theories like unevaluated functions or arithmetic [15]. To make these theories interact via a SAT-Solver, the Nelson-Oppen framework was designed [60, 74, 76]. It had the possibility to let several specialised solving systems communicate with each other, and made it possible to even create solving systems for first- and higher-order logics (HOL) [13]. Still, at the core, the solving procedures stuck to the same concepts as DPLL with clause learning [60].

There are other systems that aim to solve HOL formulas, for example the LEO system [19,89] or the inbuilt solving capacities of interactive proof systems like Coq and Agda [29, 59]. However, it turns out that all these systems still rely on the same basic techniques for constraint solving. In this thesis, we will generalise these techniques, so that they can be easily reused in the creation of new solving systems.

There have been existing generalisations of solving procedures. There was an approach to generalise SAT solving and CDCL by using Lattices as an underlying theory [33], which is an approach that we will generalise further to work for general solving. Further, there have been approaches from the functional programming area to create generalisable, adaptable and reusable solving systems [44,55,85,86], that emphasize the convenience of using monads for solving and composing different search strategies, which we will use in this thesis as well. They however do not provide an automated way to use the structure of data types to enhance the search process. Something like this has been done in the research area around automatically creating generators for testing functional programs [26, 73] where data is sampled according to a random distribution to find mistakes in a functional program, however the techniques are rather used as an external framework for testing and not for writing a program utilising search. There have been approaches by Hanus et. al. to enhance functional languages to be inherently usable as constraint solvers like logical programming languages [23,42,43], but to the extent of our scientific knowledge, the techniques have not been merged with the data type generation and sampling results from the research on testing functional programs. While our thesis does not explicitly generate data, it will become clear from the constructions of Chapter 4 how with the same formulation that we create lattices and commutable contexts for

data, we could easily also write generators for it.

There have been other generalisations of solving that allow for more generalised formulations of constraint solving problems. Radul and Sussman have developed the notion of propagators [81, 82], that allow search problems to be defined using recursive functions that can update each others input values. This theory has already been merged in several haskell frameworks [44, 58] with a more generalised notion of lattice based variables for program communication [61], but these frameworks do not yet support either full monadic computation or the possibility to create new search techniques. This thesis comes close to a full monadic implementation of these ideas in Chapter 3 and we even model how solving would work under a potentially ever increasing lattice state in Chapter 5, that would just need to be turned into a monad in future research.

Using lattices as a formal basis has additional upsides. There has been work by Kuper et al. [61, 62, 75] that behaves similar to propagators and additionally provides parallelisation capabilities. We will use this approach to make our solving system inherently run in parallel, a feature that has been used in SAT solving [41, 45, 69] but that still seems to be missing a more generalised approach.

We note that there is a range of research in the area of constraint solving that could benefit from our generalisations. The research in [40, 78, 80] (just to name a few) uses individual solving techniques for program synthesis problems that are generalised by our formalisms and could then also be equipped with clause learning or direct parallelisation. Also in the SAT-area, there have been generalisations that developed their own way to do something like clause learning from scratch where, with our framework, it would have come for free. Examples include the solvers for quantified Boolean formulas (QBF) [49, 51, 56] or, more specifically, the area of counter example guided inductive synthesis (CEGIS) [1, 18, 50].

Finally, we highlight three more areas that are related to our research. We are extensively using the theory of containers [2] to formulate statements on general data types. This theory has already been generalised for indexed data [9] and composing data using so-called ornaments [30, 70], but we have not done so yet ourselves because indexing can be imitated in a monadic context with error operations and ornamenting can be achieved by creating elements of a sufficiently expressive universe. Further, [3] give a formalization of referencing values inside of a data type described by containers and we achieve a similar result in Chapter 4 Lemma 4.5:3 and the results will need to be merged to work in our lattice based universes in future research.

1.8 Results

In this section, we give a more detailed overview over the results in this thesis. This section is not a substitute for reading the individual chapters and it is not self contained, however, it gives a broad overview over all results, how they fit together and where the trajectory for future research goes without the need to read complex proof details.

We start with the results in Chapter 5 as they give an overview of all other results. The main results of this thesis are the constructions in Chapter 3 and Chapter 4.

1.8.1 Chapter 5 : Towards a General Theory of Solving

In Chapter 5, we present first building blocks towards a general theory of solving that form a roadmap to how we can build the most general solvers. The main idea is to have the input language to the solver be so expressive that the solvers can improve themselves. This is important. To the extend of our research on the topic, solving technology is only improving on special cases and on specialised solvers. Judging by how hard it is to create a provably correct solver by hand even for existing technology [35, 63, 77], one reason for the lack of improvement might be that improving on the state of the art takes too much manual engineering. Solvers aiding their own construction process could be key to achieving true progress.

In order to build a theory around solvers in general, we start by using type theory to define what a correct solver is. We start with the corecursive data type

```
record Perhaps (A : Set i) : Set i where
    constructor PC
    coinductive
    field
        val? : A ⊎ Perhaps A
```

Code-Tag: `Perhaps`

that can hold a value that is "semidecided", meaning that it is created by a program of which we do not know whether it will ever create a value (this is important to account for undecidability).

In an ideal world, a correct solver for type theory would have the type

```
solve : (X : Set) → Perhaps X
```

Code-Tag: `SolverType`

Of course, this cannot be built since we cannot really eliminate an element of type `Set` properly, which is why the best actually buildable solvers are defined over a universe `U` (with interpretation function `[_]U`):

```
solve : (X : U) → Perhaps [ X ]U
```

Code-Tag: `SolverType`

In order for `U` to be expressible enough to express type theory, we improve the type for the syntax of type theory in Chapter 4 Section 4.7 to be used as an adequately expressible universe (we have improved the type significantly by adding sigma types and recursion, but a proper interpretation is left to future research).

We can state that a solver is (semi) complete iff it holds that

```
∀ {X : U} (x : [ X ]U) → hasAnyVal (solver X)
```

Code-Tag: `SemiCompleteness`

While we do not give an implementation of a complete solver, we showcase how with a sufficiently expressive imput language, (complete) solvers can create new (complete) solvers as

```
Solver : Set
Solver = (X : U) → Perhaps [ X ]U

createSolver : Σ Solver semi-complete → Perhaps (Σ Solver semi-complete)
createSolver (solve , _) = solve (Σ Solver semi-complete)
```

Code-Tag: `CreateSolver`

(where we ignore the universe notation). Therefore, as soon as the results in Chapter 4 Section 4.7 are complete, we can use them to create a (semi) complete solver that can create new solvers. This demonstration showcases how easily the problem of using solvers to create solvers can be expressed using type theory. We even enhance this demonstration by showcasing how we can use solvers to not only create any solver, but one that is actually improving by some measure

```
improveSolver : (solve : Solver) → Perhaps (∃[ solve' of Solver ] (measure solve < measure solve'))
improveSolver solve = solve (∃[ solve' of Solver ] (measure solve ≤ measure solve'))
```

Code-Tag: `ImproveSolver`

We give suitable examples of what measures could be applied to solvers and

compare the properties when maximising for the $\mathcal{O}$ notation versus maximising a (probabilistic) measure.

We then give some results on how a solver could be improved while running it. The first possible mechanism is to change its algebra while it is in operation, done by a special fold

```
foldCN : (n : ℕ) → ([[ C ]] A → A) → W C → (([[ C ]] A → A) → A)
foldCN 0      alg (In (s , p)) alg' = foldC alg' (In (s , p))
foldCN (1+ n) alg (In (s , p)) alg' = alg (s , (\w → foldCN n alg w alg') ∘ p)
```

Code-Tag: `FoldExchangeAlg`

that allows us to swap the algebra the solver runs on after `n` computational steps. We also give a dependently typed version of it which would be needed when constructing a correct solver.

The second possible mechanism with which a solver can be improved can be found when actually looking at the inner workings of a solver. As the `Perhaps` type is a corecursive data type, for as long as we create it using a cofold, we are actually creating it using a form of state transition

```
coFold : (A → A ⊎ A) → A → Perhaps A
val? (coFold coalg a) with coalg a
... | val x  = val x
... | ctd a' = ctd (coFold coalg a')
```

Code-Tag: `CoFoldPerhaps`

Therefore, depending on what the state for the construction is made of, improving a solver could also mean to change its internal state during the creation of the `Perhaps` value to a different state that, for example, reaches a result faster. We then continue to model properties of the state that can be used to concretely model which state changes can result in improvements.

We give a class of states and state transition functions that can be used for correctness properties when changing the state of a solver. Those functions are based on the state having a semilattice instance and the state transition function to only increase that state, like

```
δ : S → S

δ-dir    : ∀ {s}      → s ≤ (δ s)
δ-pres-P : ∀ {s} {s'} → s ≤ s' → δ s ≤ δ s'
```

Code-Tag: `PerhapsSLState`

Information from the state is read via a type of variables that are developed in Chapter 3, with the property that the extracted value is either non-existing at the state yet or as soon as it can be read, never changes again when increasing the state (until a conflict occurs). We argue that this model is expressive enough for the analysis of solving behaviour as they express the behaviour of a solver without memory management, meaning that it never forgets information. We also note that for future research, memory management by loosing information through using full lattices can be modeled.

Using this modelling technique, we can show that the speed of a solver (not accounting for memory overhead yet) only ever increases (meaning the recursive calls it makes decreases) when adding information to the state, so we prove in Theorem 5.3:24 that

```
speedup : ∀ {n n' s s' x} →
    hasFirstAsmValue x at n on s →
    hasFirstAsmValue x at n' on s' →
    s ≤ s' → n' ≤ n
```

Code-Tag: `speedupTheorem`

which shows that the steps a solver needs to produce a value never increase with increased information content on its initial state. (we leave finding examples for when it decreases as an exercise to the enthusiastic reader).

Further, we use the lattice based model to give a construction of how to run two solvers in parallel, so together with the `speedup` theorem, it follows that solvers running in parallel never need more recursive calls than if they had run individually (again, finding examples where the parallelisation causes a speedup left as an exercise).

We showcase how search would be modeled in a state based solver. We argue that solvers would usually be created from a form of coinductive trees, which are traversed using a search state that is made up of a finite number of coinductive subtrees. A sufficiently expressive lattice based universe as is developed in Chapter 4 would just model the search state accordingly, so we do not give a concrete example of a search strategy on a concrete lattice. We note however that when solvers run in parallel, and there is one solver that improves the solving behaviour by creating a better solving state for the recursion, it could

add its information to the recursive calls on the subnodes here to continuously change solving behaviour and the search strategy itself.

What we do make more concrete is the mechanism of how branching would work in the lattice based setting. We can define a branch within a state as a part of the state that acts like a recursive call to the solver. We make this precise by identifying a branch through a semilattice injection (like a normal injection, but preserving the semilattice properties) and prove that through our properties, we indeed have a recursive call of a solver running in parallel with the rest of the solver using the property

```
δ-similar' : δ ≡ unshrink ∘ δ ∘ shrink
```

Code-Tag: `Branches`

We do not give a concrete construction of how to create new branches here as there is a construction in Chapter 3 Section 3.9 that creates new variables and, within that construction, adds a new, potentially arbitrary sublattice into the state.

Finally, we give some explanations of what happens during clause learning. Summarised: We may know from one branch which initial state will eventually lead to which state increase. We give a construction where this information can be added to the state transition function of another branch without affecting its outcome and prove in Theorem 5.8:10 that, if the information is used, this leads to a significant speedup because we basically cache computation in order to not computate a possibly expensive state increase twice.

```
add-clause-prop : ∀ {s s' st n} →
    δ reaches-threshold s from s' at n →
    (mergeOPI δ δ_{s⊢st}) reaches-threshold st from s' at (1+ n)
```

Code-Tag: `AddClauseProp`

For future research, we explicitly give a construction that could lead to smaller clauses to be created than just plainly caching a state increase. This would be possible when being provided a function

```
δ-low :
    ∀ (st s : S) →
    st P δ s →
    ∃[ s' of S ] ((s' P s) and (st P δ s'))
```

Code-Tag: `SmallerDelta`

together with a version for a smaller input for `n` steps of a state transition

```
δ-low-rec : ∀ {n} → (st s : S) →
    st P ((δ ^ n) s) →
    ∃[ s' of S ] ((s' P s) and (st P (δ ^ n) s'))
```

Code-Tag: `SmallerDelta`

which can, for every threshold passed during the state transition, give a potentially smaller (the smallest) input that caused that threshold to be passed. This lowers the threshold needed for a clause to activate. This could increase the chance of the information in the clause to be used for a speedup. We do not give a concrete implementation for `δ-low` at this point, but note that there is a part in the variables from Chapter 3 Section 3.8.3.1 that can be used to find those smallest thresholds as discussed in Sidenote 3.9:40

1.8.2 Chapter 3 : Monadic Solving with Variables

Chapter 3 on monadic solving with variables is about creating monads that have inherent correctness properties related to solving, specifically on the behaviour of the logical variables used to express the search problem and or the search itself. The chapter starts with a long example section to showcase the problems that arise when solving with monads, to then create a framework for monadic variables that solve those problems.

1.8.2.1 Initial Examples

We start with a few examples on naïve search and solving techniques, like the generate and test strategy induced by the list monad. The problem with the depth-first search induced by the list monad is often that we create data (and therefore branch on it) even when we do not need it. This can be solved with what we call *explicit lazyness*, which means that we recursively wrap the data into a monadic context and only branch on it if we also case split on the result. Wrapping a list recursively into a monad can look like

```
{-# NO_POSITIVITY_CHECK #-}
data ListM (A : Set i) : Set i where
    []M   : ListM A
    _::M_ : M A → M (ListM A) → ListM A
```

Code-Tag: LISTM

This means that the list can only give us its first constructor and a monadic action for each of its subelements, similar to how coinductive types only give us their first constructor and a construction for its arguments. We call this concept *putting the data monadically in stasis*.

Side Note 1.8:1

We later in Chapter 4 get rid of the positivity issues by using containers

We can now use these monadic actions to only create (and therefore branch on) values that we actually need, like

```
{-# TERMINATING #-}
sorted : ListM ℕ → M 𝔹
sorted []M = ⦇ true ⦈
sorted (mx ::M mxs) = do
    xs ← mxs
    case xs of \{
            []M → mx >> ⦇ true ⦈
        ; (my ::M mxs') → do
            x ← mx
            y ← my
            guard (x ≤ y)
            sorted (⦇ y ⦈ ::M mxs')
        }
```

Code-Tag: SortedListM

where we check if a list is sorted by only retrieving the arguments to the ListM constructors when we actually need them.

Side Note 1.8:2

We get rid of the termination checker issues by using folds in Chapter 4 Theorem 4.3:24

This construction has the upside that we are not case-splitting for constructors that we do not need, but it has the downside that, when `M` is just the list monad, we case-split again for every read of the same value, which is not only crucially inefficient, it also produces wrong results when there are several constraints on the same data. To solve this, we store the results of the relevant monadic calls in a state. Now, each value is bound in a context `V` that can, using a function called `read`, be turned into a monadic accessor action to the state that either branches on a variable or returns its cached result. This produces similar code, only that now we both do not unnecessarily branch and also have

values that are supposed to be the same also stay the same

```
sorted : List ℕ → 𝔹
sorted [] = true
sorted (x :: []) = true
sorted (x :: y :: xs) = (x ≤ y) and sorted (y :: xs)
```

Figure 1.1: `sorted` function in identity context

```
mutual
    {-# TERMINATING #-}
    sorted : V (ListM ℕ) → MS Unit
    sorted v = read v >>= sorted' v

    sorted' : V (ListM ℕ) → ListM ℕ → MS Unit
    sorted' v []M = ⟪ unit ⟫
    sorted' v (_ ::M _) = do
        lst ← read (tailL ∘ᵥ v)
        case lst of \{
                []M      → ⟪ unit ⟫
            ; (_ ::M _) → do
                a ← read (headL ∘ᵥ v)
                b ← read (headL ∘ᵥ tailL ∘ᵥ v)
                guard (a ≤ b)
                sorted (tailL ∘ᵥ v)
            }
```

Code-Tag: `SortedVListM`

Figure 1.2: `sorted` function using lenses

```
{-# TERMINATING #-}
sorted : ListM ℕ → M Unit
-----------------------------------------
-- no initial lens-read here
-----------------------------------------
sorted []M = ⟪ unit ⟫
sorted (mx ::M mxs) = do
    xs ← mxs
    case xs of \{
            []M          → ⟪ unit ⟫ -- mx >>
        ; (my ::M mxs') → do
            x ← mx
            y ← my
            guard (x ≤ y)
            sorted (⟪ y ⟫ ::M mxs')
        }
```

Code-Tag: `SortedListM`

Figure 1.3: `sorted` function using plain explicit lazyness

We can showcase that this method greatly reduces branching. This motivates our creation of the concept of monadic variables, which are monadic actions that behave like the cached result accessor actions we used in this example section.

1.8.2.2 Monadic Variables

The main result in Chapter 3 is the concept of a monadic variable.

Definition 3.8:1: **Monadic Variable**

Given a monad `M`, a type `X` and two monadic actions

```
read : M X
write : X → M Unit
```

Then the structure is called a **monadic variable** iff the following conditions hold:

```
commutative : ∀ {m' : M Y}{f : Y → X → M Z} →
(read >>= \_ → m' >>= \y → read >>= \x2 → f y x2) ≡ (read >>= \x1 → m' >>= \y → read >>= \_ → f y x1)

read-write : ∀ {x : X}{m' : M Y}{f : Y → X → M Z} →
    (write x >> (m' >>= \y → read >>= \x' → f y x')) ≡ (write x >> (m' >>= \y → read >>= \_ → f y x))
```

Code-Tag: `MonadicVariable`

A monadic variable is a monadic action where two reads anywhere in a monadic computation are always commutative and a write always causes a subsequent read to result in the same value that was written. In and of itself, these laws do not prevent excessive branching, however they form a structure in which a monad could choose not to branch because the conditions make sure that no new information is introduced through branching. Later in a second example session we can also see how this structure can be used to create monads that only branch on values that have not been branched on yet.

In order to create contexts that have such monadic variables, we create the so-called *constrained state monad transformer*. This is a monad transformer where successive states are always related to previous states by a predicate `_P_`:

```
CStateT :
    (S : Set) →
    (_P_ : S → S → Set) →
    (M : Set → Set ) →
    Set → Set
CStateT S _P_ M X = (s : S) → M (∃[ s' of S ] (s P s') × X)
```

Code-Tag: `CStateT`

The relation `_P_` gives us the possibility that we can derive information about later states from earlier states, like for example variable assignments. In Theorem 3.8:10 we prove that this forms a monad if icM is a monad and `_P_` is a pre-order and forms an indexed monoid. We also prove that it further follows that the constrained state monad transformer is a monad transformer if `M` is a

monad, that it is a monoid when given a type `A` when `M A` forms a monoid and that error actions (meaning left absorbing monadic actions) can be preserved.

In order to give a more concrete pre-order relation `_P_`, we use the pre-order of semilattices to not only have a general pre-order, but also to be able to compute elements that are guaranteed to be bigger than another. With semilattices, this can be achieved through the *directional* property

```
directional : x P (x <> y)
```

Code-Tag: `DIRECTIONAL`

that allows us to write to the state without loosing information; by merging the new information into the state as

```
write : S → CStateT S _P_ M Unit
write s' s = return (s <> s' , directional , unit)
```

Code-Tag: `WRITE_DIRECTIONAL`

We create a concept called a lattice bi-threshold function. A lattice bi-threshold function is a function

```
f : S → VarAsm S
```

which reads from the state and either gives a value, the information to be unassigned or the information that there has been a conflict. In order to be a bi-threshold function, it has to hold that

```
isBiThresholdRead : ∀ (s s' : S) → s P s' →
        (f s  ≡ unassigned)
    ∨ (f s  ≡ f s')
    ∨ (f s' ≡ conflict)
```

Code-Tag: `IsBiThresholdRead`

so with increasing state size, the result of the function can only either jump from unassigned to a fixed value (or conflict) or from being assigned to giving a

conflict. This means that unless a conflict occurs, the variable does not change its value after being assigned. From these threshold functions, we create the lattice bi-threshold variables, which are a construct of a reading and writing function that can be used to read and write values from a lattice state, without the value changing after being assigned

Definition 3.8:29: **Lattice Bi-Threshold Variable (LBVar)**

Given a bounded semilattice over a type `S` and a type `X`, two operations

```
read : S → VarAsm X
write : X → S
```

are called a **lattice bi-threshold variable (LBVar)**, iff the following conditions hold:

```
isBiThresholdRead : ∀ (s s' : S) → s P s' →
    (f s  ≡ unassigned)
    ∨ (f s  ≡ f s')
    ∨ (f s' ≡ conflict)
write-read : read (write x) ≡ asm x
read-write-read : ⦇ write (read s) ⦈ >>= read ≡ read s
```

Code-Tag: `IsLBVar`

The laws for the write make sure that a write actually assigns a variable to the desired value. In Theorem 3.8:39, we prove that from lattice bi-threshold variables (here, abbreviated with `V`), we can indeed construct monadic variables for the lattice constrained monad transformer (here in short called `LState`) if there exist left absorbing error actions

Theorem 3.8:39

When given any two, left absorbing over `M`, monadic actions

```
readUnas : ∀ {A} → V X → S → M A
readConf : ∀ {A} → V X → S → M A
```

then forall variables `v : V X`, the operations

```
read : V X → LState X
read v s =
    unas: lift (readUnas v s) s
    conf: lift (readConf v s) s
    asm: (flip returnC s)
    (readL v s)

write : V X → X → LState Unit
write v x s = return (s <> writeL v x , directional , unit)
```

Code-Tag: `ReadWriteOps`

form a monadic variable over the `LState` monad.

1.8.2.3 SAT-Solving Example

We end this chapter with an example of how to do SAT-solving with our created monadic variables. Here, we use a construction from Lemma 3.8:32 to get lattice bi-threshold variables for Booleans over the trivial lattice of booleans $\mathsf{B}_{\bot}^{\top}$. In this example, we show how to only branch on unassigned variables, but also show further improvements that can be made by analysing the implementation of the evaluation function for Boolean formulas.

When we write the evaluation function for Boolean formulas (where in this example, we do not even require them to be in conjunctive normal form), we can do it recursively using our constrained state monad transformer to keep track of the current assignment as

```
assignmentOf : SATFormOver (V 𝔹) → LState 𝔹
assignmentOf trueSF       = return true
assignmentOf falseSF      = return false
assignmentOf (varSF v)    = read v
assignmentOf (notSF form) = ⟪ notB (assignmentOf form) ⟫
assignmentOf (a /\SF b)   = ⟪ (assignmentOf a) andB (assignmentOf b) ⟫
assignmentOf (a \/SF b)   = ⟪ (assignmentOf a) orB  (assignmentOf b) ⟫
```

Code-Tag: `assignmentOf`

If we now want the assignment to have a specific output value (in the SAT case that is usually the value `true`) then in the case where we would usually just read the variable via `read v`, we could also just assign it to the aim we want to achieve and therefore have similar behaviour to the unit propagation. Naively, this looks like

```
solve : SATFormOver (V 𝔹) → 𝔹 → LState Unit
solve trueSF  true   = return unit
solve falseSF false  = return unit
solve trueSF  false  = lift readConf
solve falseSF true   = lift readConf
solve (varSF v) aim     = write v aim
solve (notSF a) aim     = solve a (not𝔹 aim)
solve (a /\SF b) true   = solve a true   >>  solve b true
solve (a \/SF b) false  = solve a false  >>  solve b false
solve (a /\SF b) false  = solve a false  <|> solve b false
solve (a \/SF b) true   = solve a true   <|> solve b true
```

Code-Tag: `SolveNaive`

where `_<|>_` is a branching operator for our monad. The example further removes some problems with the above definition, but it showcases how the way we solve for a desired outcome is connected to how we compute the outcome. To give a strong example of this connection, we automatically create a solver with all discussed optimizations only from the algebra of the evaluation function

```
assignmentAlg : SATFormOverF 𝔹 → 𝔹
assignmentAlg trueSFF     = true
assignmentAlg falseSFF    = false
assignmentAlg (notSFF a)  = not𝔹 a
assignmentAlg (a /\SFF b) = a and𝔹 b
assignmentAlg (a \/SFF b) = a or𝔹  b
```

Code-Tag: `AssignmentAlg`

Sadly, we have to leave the proof of correctness and the generalisation of this on containers to future work, but this example greatly shows the potential of how we would automatically turn any recursively defined program into a solver.

1.8.2.4 Creating New Variables

We also give some results on creating new variables, which is a crucial feature to tackle memory problems during solving [49, 51, 56, 87] or to make general recursive solving even possible because a data type that can grow arbitrarily

during the solving process needs an ever growing source for new variables to be created on. The problem is that for our lattice based variables, adding a new variable from a new lattice means that we have to change the type of underlying lattice state for every new variable. To be able to do that, we generalise the constrained state monad transformer to an indexed version

```
ICStateT : (I × J) → (I × J) → Set → Set
ICStateT (i , j) (i' , j') X = (s : S i) → M j j' (∃[ s' of S i' ] (s P s') × X)
```

Code-Tag: `ICStateT`

Where we provide two indices: One for the inner monad and one for the state type, which is now an indexed type

```
S : I → Set
```

The relation also now becomes an indexed pre-order

```
_P_ : {i j : I} → S i → S j → Set
```

As the semilattice type for the state changes, it is no longer obvious what the order would be between lattice elements of two different lattices. To still create a pre-order, we create the principle of semilattice injections. These are injections (with the functions `inf` into a type and `outf` out of it) that rpeserve the semilattice operations

```
pres-inf  : inf  (a <>ₓ b) ≡ (inf  a <>ᵧ inf  b)
pres-outf : outf (a <>ᵧ b) ≡ (outf a <>ₓ outf b)
```

Code-Tag: `SemilatticeInjection`

We also create a bounded version of them. These semilattice injections form a pre-order on semilattices. Therefore, the semilattice instances are used as indices for the indexed pre-order. To actually create a pre-order that can compare two elements of different semilattices that have a semilattice injection

inbetween them, we use the comparison of the bigger lattice, meaning that we push the lower element into the bigger lattice and then compare their sizes with

```
inj-directional : inf x P_Y y
```

Code-Tag: `BoundedSemilatticeInjectionRelation`

In order to create a new variable, we can form the semilattice product of the semilattice it is reading from together with the semilattice instance of the current state. We create a semilattice injection from the old state semilattice type into the new product type. To show that all existing (and the new) variables can now be used on the bigger state, Lemma 3.9:27 shows that lattice bi-threshold variables can be lifted along semilattice injections to then read their values from the bigger semilattice. We can now create an operation that can create a new variable for our indexed variable type `IV` to be used in the indexed monad

```
new : IV i' X → ILState (i , j) ((i' -xi- i) , j) (IV (i' -xi- i) X)
new v s = return ((e bslX , s) , SIRSndInjection , liftLBVar semilatticeProductFstInjection v)
```

Code-Tag: `IndexedNew`

We start creating some first results on variable independence, showing that a variable created with the `new` operation is independent from all previous variables, meaning that it can be assigned without assigning other variables and its value does not depend on the value of other variables. That being shown, we leave creating an independence condition for monadic variables in general for future research.

1.8.3 Chapter 4 : Data Types for Monadic Solving

In previous chapters, we used lattices and accessor functions to the lattice objects to model computation in the context of solving. Further, in order to represent data to only be partially available and to only create individual constructors if necessary, we also have to put that data monadically in stasis. How expressive our solvers are heavily depends on the classes of data types that we can represent over both of these concepts. Therefore, this chapter presents constructions to automatically transform data types into the structures needed for solving. To showcase the strength of our constructions, we improve the type for the syntax of type theory presented in [8, 24] and at least informally check whether it can be implemented using our constructions in future research.

All constructions in this chapter heavily rely on the concept of containers [2], so the chapter begins with a detailed introduction.

1.8.3.1 Held in Stasis Data and Commutiable Contexts

We begin by using containers to define what it means for a data type to be (monadically) held in stasis. The idea is that we exchange the recursive call of a data type with a reference to its recursive value, like

```
VW : (C V : Container) → Set
VW C V = W (C :o: V)
```

Code-Tag: `VWFixpoint`

Definition 4.3:1: Held-in-Stasis Data Types

Given two containers `C` and `V`, we define the type `VW C V` to be the type of `(W C)` **held in stasis of V** (short: held in stasis). In the special case where `[[ V ]]` has a monadic instance, we say that `VW C V` is the type of `(W C)` **monadically held in stasis by V** (short: monadically held in stasis).

This describes a data type where every position containing the subfixpoint now holds that subfixpoint bound in a context `[[ V ]]`, which could for example be a monadic variable. The main result of this part of the chapter is to show under which conditions we can reassemble held-in-stasis data back into data of their original types and how to apply functions to them. Our constructions can be used to show the correctness of the reassembly, however, we also showcase a fold that would be way more efficient to use.

To define how a held in stasis data would be reassembled we create the concept of *commutable contexts*.

Definition 4.3:15: Commutable Contexts

Given the endofunctors `F : Set → Set` and `M : Set → Set` and a monad instance on `M`, together with an action

```
switch : (F ∘ M) A → (M ∘ F) A
```

Code-Tag: `CommutableContexts` we say `F` **commutes into** `M` if the following laws hold:

```
return-prop : ∀ {fa : F A} →
    switch (return <$> fa) ≡ return fa

bind-prop : ∀ {fm : (F ∘ M) A} {f : A → M B} →
    switch ((_>>= f) <$>F fm) ≡ (switch fm >>= switch ∘ fmapF f)
```

These contexts have an operation `switch` that is used to transform every single held-in-stasis constructor of type `F (M (W (F ∘ M)))` into a value of type `M (F (W (F ∘ M)))`, so that we have the constructor now with one more concrete constructor as its arguments until everything is bound monadically again. Doing this recursively reassembles the data type until we only have an element `M (W F)` left. We define wrapping and unwrapping operations to get data back and forth or their held-in-stasis representation

```
unwrap : VW C M → [[ M ]] (W C)
unwrap = foldC (fmap In ∘ (switch >=> switch) ∘ compd)

wrap : W C → VW C M
wrap = foldC (In ∘ compc ∘ map return)
```

Code-Tag: `CommutableContextsWrapping`

We prove their correctness first with the small Theorem 4.3:20 showing that

```
unwrap-o-wrap : unwrap ∘ wrap ≡ return
```

Code-Tag: `unwrap-wrap`

which means that a wrapped data type can be unwrapped without changing the data inside the context `M`.

The other direction of `wrap ∘ unwrap ≡ return` does obviously not hold because a value held in stasis unwraps into a monadic value, whereas the `wrap` takes a non-monadic value, however, we can prove something stronger. We can give a definition of how to fold over monadically held-in-stasis data using the operation

```
foldS : ([[ C ]] A → A) → VW C M → [[ M ]] A
foldS alg = foldC (switch' >=> applyMorph alg)
```

Code-Tag: `CommutableContextsFoldS`

We can then show in Theorem 4.3:24 that applying this fold on the held-in-stasis data type gives the same result as running a fold with the same algebra on the reassembled data type, so it holds that

```
foldS-unwrap : fmap (foldC alg) ∘ unwrap ≡ foldS alg
```

Code-Tag: `CommutableContextsFoldSUnwrap`

Therefore, we have devised a mechanism to put data into stasis and to run functions on values held in stasis. The only problem left is that with the above fold operation, we still have to always unwrap the entire data, regardless of whether we need the values or not. This can be fixed by using algebras that create monadic values and the fold operator

```
foldM : (([[ C ]] ∘ [[ M ]]) A → [[ M ]] A) → VW C M → [[ M ]] A
foldM alg = foldC (alg [[ M ]] mon ∘ map join ∘ compd)
```

Code-Tag: `CommutableContextsFoldM`

however, we leave proving its correctness to future work.

In the end of this subsection, we show how to automatically create commutable contexts for data types created by special containers. In Theorem 4.3:56 we show that

Theorem 4.3:56

Given a shapely container `C` and a locally commutative monad instance for `M`, then `[[ C ]]` commutes into `M`.

Code-Tag: `ShapelyCommutableContexts`

where a shapely container is a container where every shape has only finitely many positions. These data types correspond to algebraic data types that do no use functions (at least not with infinite domains) as arguments to their constructors. These are now the data types that we know can be reassembled in locally commutative monads, which are monads that are formed through our monadic variables.

Next, we give constructions on how to automatically create lattice instances

for data types.

1.8.3.2 Lattices for Containers

In this section we show how to create lattices for containers that have shapes with decidable equality that internally does not use univalence (it is constrained away for the equality produced by the decidable equality. Univalence or without-K in general remains active). We do so by using the lattice instance for the trivial lattice as a basis to create an indexed lattice used for the lattice instance of a sigma type. Concretely, given a container `S ▷ P`, we define its *latticed container* as the container

```
S⊤⊥ ▷ (top: Zero bot: Zero val: P)
```

Code-Tag: `latticedContainer`

We now use Theorem 4.4:11 stating that if we have a lattice instance for a type `A` and a lattice instance for `B` indexed over the lattice instance for `A`, we can create the lattice for the sigma type `Σ A B`. With that theorem we can give a lattice instance for the semantic extension of the latticed container, so in Theorem 4.4:30, Code-Tag: `positionLattice`, we show that there is a lattice instance for `top: Zero bot: Zero val: P` indexed over the lattice instance for the trivial lattice $S^{\top}_{\bot}$.

Please do not be fooled by how fast we can state the theorem proved. The actual proof in cubical type theory is surprisingly complex.

Finally, we start showing how to create lattice bi-threshold variables over the lattices cerate by the latticed container. In Lemma 4.5:3 we show that for every position in a container we can create a lattice bi-threshold variable pointing to the position in the latticed container extension data type. In future research, we will use this to create a function

```
subvariables : [[ C ]] Unit → [[ C ]] (LBVar ([[ latticedContainer ]] A) B)
```

Code-Tag: `subvariables`

that can fill the constructor of a data type with variables pointing to the lattice representing the arguments to the constructor in its latticed representation. These variables them make a context that we can commute with using the results from earlier and Theorem 4.3:56 to recreate the original data from the variables or to compute over the data held in stasis as a lattice.

1.8.3.3 Towards Encoding the Type of Type Theory

Code-Tag: `TypeOfTypeTheory`

In [8, 24], a type for the syntax of type theory is given. As our constructions from Chapter 5 require a universe strong enough to capture type theory to fully run, we did two things with this type for the syntax of type theory. First, we informally checked whether all prerequisites for our lattice constructions hold on the data type and they do for as long as our solver runs in a monad with error actions. The original type however did not have enough constructors to represent the concepts of Chapter 5 or to automate constructions that future research needs in order to automatically apply all constructions developed in this thesis. Therefore, we add additional constructors for sigma types and use those to add containers, well-formed types and recursive functions. For now, we leave giving a concrete interpretation to future research.

We quickly summarise the constructors we have added. The type for the syntax of type theory is comprise of four different, mutually defined types for contexts, types, terms and term substitutions (allowed context changes) with the following signature:

```
data Ctx : ℕ → Set                          -- Type for contexts
data Tp  : Ctx n → ℕ → Set                  -- Type for types
data Tm  : (Γ : Ctx n) → Tp Γ n' → Set      -- Type for terms
data Tms : (Γ : Ctx n) (Δ : Ctx n) → Set    -- Type for context transformations
```

On top of that it also defines some equality types that we will explain in the chapter.

We can now express Sigma types with the type and term constructors

```
Sig : (A : Tp Γ n) → Tp (Γ ::ctx A) n' → Tp Γ (max n n')

sigma : (a : Tm Γ A) → Tm Γ (B $Tp a) → Tm Γ (Sig A B)
fstTm : Tm Γ (Sig A B) → Tm Γ A
sndTm : (a : Tm Γ (Sig A B)) → Tm Γ (B $Tp (fstTm a))
```

Together with a few definitions to create non dependent functions in the meta theory, we can define container types in the meta theory as

```
ContainerTp : (n n' : ℕ) → Tp Γ (max (1+ n) (max n (1+ n')))
ContainerTp n n' = Sig (SetS n) ((toTp v0Tp) ⇛ (SetS n'))
```

we define its semantic extension in the meta theory as

```
[[_]] : Tm Γ (ContainerTp n n') → Tp Γ n'' → Tp Γ (max n (max n' n''))
[[_]] S|>P A = Sig (toTp (fstTm S|>P)) ((coerce-Tp (::ctx-eq refl-Ctx eq) ∘ toTpTms2 ∘ app ∘ coerce-Tm tms-=>) (sndTm S|>P) ⇛ A [ v1 ]TpTms)
```

We can now well formed types to the meta theory

```
W : Tm Γ (ContainerTp n n') → Tp Γ (max (1+ n) (max n (1+ n')))
```

and also add terms for its constructors and destructors

```
In    : Tm Γ ([[ C ]] (W C)) → Tm Γ (W C)
foldC : Tm (Γ ::ctx [[ C ]] A) (A [ v1 ]TpTms) → Tm (Γ ::ctx W C) (A [ v1 ]TpTms)
```

as well as some equality constructors for compute rules.

With this framework, future research can turn this type into a lattice and then solve for terms within type theory.

But before we get ahead of ourselves, let us start the actual thesis.

Chapter 2

Preliminaries

Code-Tag: `PreliminariesPtI`
Code-Tag: `PreliminariesPtII`
Code-Tag: `CubicalEquality`

2.1 Overview of Notation and Conventions

In this section we give a brief overview on the notation used in this thesis. Most notation overlaps with common mathematics, however, in order to avoid mistakes during code transfer from Agda to this thesis, we have to stick to some Agda notation here and there. As all of the proofs and formalisms created in this thesis have been verified by Agda, sticking to its notation ensures the formal correctness of the statements in this thesis.

It should be noted that all notation will later be explained in detail. This Section is just the overview to quickly search for the concepts most commonly used in this thesis.

2.1.1 Mathematical Notation in Agda (Overview)

This is just an overview of mathematical notation as it is written in Agda. A more detailed explanation of how this notation works can be found later in this chapter.

Mathematical Notation	Agda Notation	Notes
$a \in A$	`a : A`	The for type theory fundamental statement `a` is of type `A` roughly translates to `a` being in the set `A` (if the type `A` can be interpreted as a set. There are counterexamples). Even in Agda, we occasionally use the `∈` symbol to annotate that a certain element has a type if it is not clear from the context.
$f : A \to B$	`f : A → B`	function signature
$f(a, b)$	`f a b`	function application
$f : (A \times B) \to C$	`f : A → B → C` (convention)	function taking several arguments. Note: We can also write `f : (A × B) → C` in Agda and would then even have the traditional function application syntax `f (a , b)`, but that notation is avoided by convention if possible. Important: The notations `f : A → B → C` `f : (A × B) → C` are isomorphic, but not definitionally equal!
$f \circ g$	`f ∘ g`	function composition

Mathematical Notation	Agda Notation	Notes

$\mathtt{f}(\mathtt{y}_1) \mapsto \mathtt{x}_1$
$\mathtt{f}(\mathtt{y}_2) \mapsto \mathtt{x}_2$
...
	`f y₁ = x₁`
	`f y₂ = x₂`
	`...`
		case distinction when pattern matching the function input (and the type has constructors to be patternmatched on)

$f = \begin{cases} \mathtt{y}_1 & , \mathtt{x} = \mathtt{x}_1 \\ \mathtt{y}_2 & , \mathtt{x} = \mathtt{x}_2 \\ \ldots & \ldots \end{cases}$	`f with x`
	`... | x₁ = y₁`
	`... | x₂ = y₂`
	`... | ...`
		case distinction when pattern matching on `x` (can be any term, does not necessarily have to be an input. Type of `x` has to have constructors that can be patternmatched on)

$A \wedge B$	`A and B` `A × B`	logical "and"
$A \vee B$	`A or B` `A ⊎ B`	logical "or"
$A \Rightarrow B$	`A → B`	logical implication (just the function type due to Curry-Howard correspondence [47])
$\lambda a.f(a)$	`\a → f a`	Lambda abstraction. The lambda in this thesis is written as a backslash `\`
"Proposition"	`Set`	The type of all types (note Section 2.3.13 and Section 2.1.3 for conventions)
$\forall(a : A).B(a)$	`(a : A) → B a` `∀ a → B a`	Universal quantification is just the dependent function type. The `∀` symbol can be used to have the type of the argument inferred automatically

Mathematical Notation	Agda Notation	Notes

$\exists(a : A).B(a)$	`Σ A B`
	`Σ A (\a → B a)`
	`∃[ a of A ] (B a)`
	`∃[ a ] (B a)`
		Existential quantification is the dependent sum type, also called a *dependent tuple*. Here, `A : Set` is an "element" and `B : A → Set` is a "Proposition on the element". Often introduced using lambda abstraction for `B`. As `:` is a reserved symbol in Agda, we need the notation `∃[ a of A ] ...`.

$a := b$	`a = b`	the equality symbol `=` in Agda is reserved for definitions
$a = b$	`a ≡ b`	equality types are more complicated in type theory, so they have their own symbol
$a \overset{eq_1}{=} b \overset{eq_2}{=} c$	`a =< eq₁ > b =< eq₂ > c qed`	chains of equations are expressed with tertiary operation `_=<_>_` and ended with `qed`

`a = b` `|` `f`
$\Leftrightarrow$ `f(a) = f(b)`
	`ab || f`
	in a context where `ab : a ≡ b`. Usually written as
	`f a =< ab || f > f b qed`
		equations in type theory are also objects that we can apply a function to. The `|` symbol is reserved in Agda, so we use the symbol `||` instead. Notation is often omitted to avoid clutter.

	`[premise]`	Annotation to show what types elements in the context have. Ignores everything to its left and has no computational content.
proof by induction over `x`	writing a terminating recursive function case splitting on `x` and applying the recursive call on the smaller arguments of the current constructor of `x`	As a proof is a program, giving a proof means writing a program. (Generalised) induction is modeled via terminating recursion.

2.1.2 Language conventions

In this thesis, the term **show** marks that a proof has been done in Agda, whereas **showcase** means that the proof has either been done on paper or we just given an intuition for a concept by example.

If a type `A` has a "type class" or "interface" or any other general structure defined over it, for example that there is a field defined over `A`, we usually write it as "There is a type `A` with a field instance", which is the correct formulation. However, to occasionally shorten things, we might also write "The field `A` ...", which could both refer to an element of `A` where `A` has a field defined over it or the field over `A`, depending on the context. If unclear from the context, we will not use the ambiguous formulation.

2.1.3 Agda specific conventions

The type `Set` of all types actually has a hierarchy to avoid paradoxes like russel's paradox [83]. The type `Set` is not itself of type `Set` again, but `Set : Set1`, `Set1 : Set2` and so on. The type can therefore be parametrised with its level `l` in that hierarchy, like `Set l`. We can form the maximum of two type levels using `l1 ⊔ l2` and can form the next type level as `lsuc l`. In Agda, all definitions that take any type as an input should be made polymorphic with regards to their hierarchy, however, this introduces quite some boilerplate to the code that does not help understanding the underlying principles.

All general definitions using the Set type in this thesis are written universe polymorphic in the code, but are expressed without the universe polymorphism unless stated otherswise.

We also note that the Agda code is written without unicode (for reasons of personal preference) and automatically translated into unicode symbols by the LaTeX code for this thesis.

We often omit variable definitions when clear from the context. So a function (where the `{` and `}` brackets mean the argument will be automatically inferred by Agda and does not have to be given explicitly)

```
f : {i : Level} → {A : Set i} → A → A
```

is abbreviated as

```
f : A → A
```

or a statement

```
assoc-+ : ∀ a b c → a + (b + c) ≡ (a + b) + c
```

may be abbreviated as

```
assoc-+ : a + (b + c) ≡ (a + b) + c
```

if the bound status of the variables is clear from the context (either universally quantified or existentially).

This can even be done in the code as well using modules (Section 2.3.9) or variable declarations (Section 2.3.10).

2.2 Translating Mathematical Definitions into Agda

This section should chronologically come after the detailed preliminaries, however as the question of how to turn a classically written definition into Agda code is often one of the first ones that comes up, we already give the overview early.

When writing a conventional mathematical definition, we define the initial given objects and their properties. In type theory, a property is also an object, so really we are defining a dependent tuple, meaning a tuple where the type of later objects may depend on previous ones in the tuple. In Agda, this is done via so-called *record types*. We will give a quick example.

Definition 2.2:1: **Monoid**

Given a type `A` , a neutral element `ε : A` and a binary operation

```
_⊕_ : A → A → A
```

we call it a **monoid** iff the following laws hold:

```
∀ a b c → a ⊕ (b ⊕ c) ≡ (a ⊕ b) ⊕ c
∀ a     → ε ⊕ a ≡ a
∀ a     → a ⊕ ε ≡ a
```

so the operation is associative and uses ε as its neutral element.

The data type for this in Agda would, without universe polymorphism, look like

```
record Monoid : Set where
    field
        A : Set
        ε : A
        _⊕_ : A → A → A

        associative    : ∀ a b c → a ⊕ (b ⊕ c) ≡ (a ⊕ b) ⊕ c
        left-identity  : ∀ a     → ε ⊕ a ≡ a
        right-identity : ∀ a     → a ⊕ ε ≡ a
```

So it has all the objects that we need for the structure, as well as the properties on them (that are also just objects). Usually, those structures would be written universe polymorphic (in fact, the above structure would have to start with `record Monoid : Set1 where` in order to compile), so in the code, we write

```
record Monoid {i : Level} : Set (lsuc i) where
    field
        A : Set i
        ε : A
        _⊕_ : A → A → A

        associative    : ∀ a b c → a ⊕ (b ⊕ c) ≡ (a ⊕ b) ⊕ c
        left-identity  : ∀ a     → ε ⊕ a ≡ a
        right-identity : ∀ a     → a ⊕ ε ≡ a
```

and just omit the levels in the definitions because it is usually straight forward how to place them and they do not add to understanding the theory.

Depending on the context, we might want to have more information in the

type of the structure. In our example, this could mean that we want to know which concrete type A the monoid is defined over. In this case we would write

```
record Monoid (A : Set) : Set where
    field
        ε : A
        _⊕_ : A → A → A

        associative    : ∀ a b c → a ⊕ (b ⊕ c) ≡ (a ⊕ b) ⊕ c
        left-identity  : ∀ a     → ε ⊕ a ≡ a
        right-identity : ∀ a     → a ⊕ ε ≡ a
```

To have the type A the monoid is defined over visible in the type. This does not change the natural language definition, so which version is implemented is a pure design choice. More information in the type means less need for additional equalities between the elements of the records, but may also be more cumbersome to write. Further, types given as an argument do not to have raise the universe level of our type, which even simplifies this when writing it universe polymorphic:

```
record Monoid {i : Level} (A : Set i) : Set i where
    field
        ε : A
        _⊕_ : A → A → A

        associative    : ∀ a b c → a ⊕ (b ⊕ c) ≡ (a ⊕ b) ⊕ c
        left-identity  : ∀ a     → ε ⊕ a ≡ a
        right-identity : ∀ a     → a ⊕ ε ≡ a
```

In this thesis, we will usually go with the natural langauge definition for readability and only give the Agda definiton if we need it by name.

For data types, we always go with the Agda code, as it gives the most exact and readable formalisation.

2.3 Agda Basics

Agda is a dependently typed programming language with flexible syntax. Even though we hide mist Agda implementation details in this thesis, it is still helpful to have a basic understanding of all of its concepts.

2.3.1 Basic Functions

In Agda, everything has a type. If an element has a certain type, it is written as `a : A`, meaning that `a` is of type `A`. There is a small set of base types defined, the first of which is the function type

```
f : A → B
```

for a function from type `A` to type `B`. This is also called the function's *signature*. If we give a function `f : A → B` an element `a : A` as an input, like `f a`, we get an output of type `B`. This is often written as `(f a) : B` as an explanation, but please note that this is not correct Agda code (only correct without the type annotation `: B`).

Side Note 2.3:1

This needs to have the types `A` and `B` to be defined in the context. Unless stated otherwise, we will for more concise notation always assume that names that are not bound otherweise are universally quantified variables.

In order to implement a function we have to give it a (finite set of) clauses, which are equations that the computer can use to automatically evaluate the function. For example, if we implement the identity function, we can write it as

```
id : A → A
id a = a
```

where we say that the output of `id a` is `a`.

If we want to bind the value for `A` in the function definition to have the identity defined over an arbitrary type, we can do so by writing

```
id : (A : Set) → A → A
id A a = a
```

Where `Set` is the type of all types (more details in Section 2.3.13). Further, we can hide some details from this definition. We will also explain this in more detail in Section 2.3.7, but because we don't always want to write the type- (or other) arguments for our functions, we can tell Agda that it should try to infer the arguments from the function inputs. So if we write

```
id : {A : Set} → A → A
id a = a
```

we can see that we do not have to give the so-called *implicit* argument `A` and can just define the identity function.

Functions can also be implemented using a *lambda abstraction*, which is often done when defining a function inline without having to give its signature.

```
id : {A : Set} → A → A
id = \a → a
```

The `\` symbol imitates the symbol λ from the original formalism of the *lambda calculus*.

Functions can also take functions as an argument.

```
apply : (A → B) → A → B
apply f a = f a
```

Here, we are getting a function `f : A → B` as an argument and can apply it to the second argument `a`.

Side Note 2.3:2

Not only can we get a function as an input, we can also give one as an output. This is the reason that the type

```
f : A → B → C
```

represents a function that takes two inputs. It is actually desugared (meaning: rewritten by the compiler / interpreter) as

```
f : A → (B → C)
```

which means that it is a function that given an input `a : A` turns into a function that still takes a `b : B` to turn into a value `C`. This means we can also give a function only some of its arguments, like `f a` to get a function that still needs the remaining arguments, so colloqially, `f a : B → C`.

Multiple arguments to functions can be abbreviated in order not to write so many function arrows, so writing

```
f : (a : A)(b : B) → C
```

is the same as writing

```
f : A → B → C
```

(giving the inputs of types `A` and `B` names is actually required to do that so that agda cannot confuse just taking two arguments with trying to apply the element `A` as a function to the element `B`).

Further, if several arguments to a function have the same type, we can write

```
f : (a b : A) → B
```

to mean the same as

```
f : (a : A)(b : B) → B
```

2.3.2 Basic Data Type Definitions

A data type can have several constructors that it can be made from, those constructors may have arguments and the data type may be recursively defined. There is a way to define data types by a small set of combinators and we will do that eventually in this thesis, however, most of the time it is more readable to use a data type declaration. For example, we can define the booleans as

```
data 𝔹 : Set where
    true  : 𝔹
    false : 𝔹
```

which says that we are defining a data type 𝔹 of type `Set` (the type of all types, details in Section 2.3.13).

When we now define a function over the Booleans, we can case split on the constructors of the type, so we can implement the Boolean not-function as

```
not𝔹 : 𝔹 → 𝔹
not𝔹 true  = false
not𝔹 false = true
```

We can also give the constructors arguments, and they can take an element of their own type as an argument (with some restrictions). For example, we can define the type of lists of Booleans as

```
data ListB : Set where
    [] : ListB
    _::_ : B → ListB → ListB
```

Side Note 2.3:3

The `_::_` constructor uses a mixfix notation explained in Section 2.3.4 to turn it into an infix operator. It needs the line

```
infixr 10 _::_
```

placed before the definition to run

We can define concatenation recursively by matching on the constructors as

```
concat2B : ListB → ListB → ListB
concat2B []        ys = ys
concat2B (x :: xs) ys = x  :: (concat2B xs ys)
```

Where we can define recursive functions under some constraints explained in Section 2.3.5. If we evaluate this, we get for example that

```
concat2B (true :: []) (false :: true :: []) = true :: false :: true :: []
```

Or course, lists can also be defined polynomially by giving the data type declaration an argument, as

```
data List (A : Set) : Set where
    [] : List A
    _::_ : A → List A → List A
```

where concatenation becomes

```
concat2 : ∀ {A} → List A → List A → List A
concat2 []       ys = ys
concat2 (x :: xs) ys = x  :: (concat2 xs ys)
```

And, when clear from the context, we occasionally omit the `∀ A` notation to avoid clutter.

Side Note 2.3:4

The `∀ A` notation states that Agda uses `A` as a variable and tries to infer its type automatically.

2.3.3 Pattern Matching Lambdas

Lambda abstractions can also pattern match in Agda, So we can implement the boolean conjunction as

```
andB : B → B → B
andB = \{ true true → true
        ;  _    _   → false }
```

The pattern matched expression is wrapped in `\{ ... }`, each clause from the pattern match uses an arrow `→` instead of an `=` and the clauses are delimited by a semicolon `;`.

2.3.4 Variable Names, Mixfix Operators and Underscores

Names in Agda can almost have any shape within unicode. Exceptions are reserved symbols. One Interestig feature is that a name can also be a mixfix declaration. For example, the function

```
_andB_ : 𝔹 → 𝔹 → 𝔹
true andB true = true
_    andB _    = false
```

has two underscores in its name `_andBool_`, which marks the position that the arguments are placed. Therefore, we can put the arguments where the underscores are (in the respective order), as we have done in the clauses of the function already.

Side Note 2.3:5

Underscores in function clauses mean a value is not used. Underscores as function arguments means that Agda shall try to infer the argument

We can have arbitrarily many underscores in names, so we can also define

```
if_then_else_ : 𝔹 → A → A → A
if true  then y else n = y
if false then y else n = n
```

to have an if-then-else mixfix operator.

Side Note 2.3:6

We can have fewer underscores than inputs. This might mean that Agda no longer knows whether a trailing argument is one argument or a function application, so we might have to disambiguate using brackets. For example, we can define simple function composition as

```
_o_ : (B → C) → (A → B) → A → C
(f ∘ g) a = f (g a)
```

Agda can set the operator precedence by writing

```
infixr 20 _o_
```

for example for the operator `_o_`. The number is the binding strength: the higher the number the stronger it binds, meaning it binds before operators with lower precedence. **`infixr`** means that the operator is right associative (in the sense that using it several times in a row places the brackets on the right), and there is also **`infixl`** for left associativity and **`infix`** for no associativity.

Operator precedence and associativity can be used to avoid brackets. We usually omit making those explicit, but these operators make a great case how Agda can be used to write any kind of mathematics in a way that we are used to.

2.3.5 Interactive Agda

Agda can run interactively, which no only means that we can evaluate expressions, but also means that we can place so-called *holes* almost anywhere in our code to get information about the context and which elements we might be able to place. How this is done in detail is explained in [5], but sometimes we use holes to express (unfinished) ideas in progress. A hole looks like

```
f : {A B C : Set} → A → B → C
f a b = {!!}
```

and Agda can tell us the type of object that is needed for the hole, as well as the context of objects that are usable for the hole (expluding module definitions), so here, Agda would show

```
C                       Goal
b : B
a : A
C : Set   (not in scope)
B : Set   (not in scope)
A : Set   (not in scope)
```

so we have to find an element of type `C`, and we have `a : A` and `b : B` in the context (as well as the types `A` and `B`, which are in the context but due to not being given explicitly cannot be used in the current scope).

2.3.6 Termination Checking

If we have a recursive type, like the natural numbers for example

```
data ℕ : Set where
    zero : ℕ
    1+_  : ℕ → ℕ
```

Side Note 2.3:7

Agda has special syntax for naturals, so they can be written as `0`, `1`, `2` ... and do not have to be written like `1+ 1+ zero`

We can define a function on that type that which calls itself recursively

```
_+_ : ℕ → ℕ → ℕ
0      + y = y
(1+ x) + y = 1+ (x + y)
```

Because the recursive call happens on arguments smaller than the input, we know that this function terminates for any two given natural numbers. As a rule of thumb, this is what the termination checker checks in general: That

the input to the recursive call is strictly smaller than the input of the current function call.

In Agda, it is important that all functions terminate in order for it to be usable as a proof assistant. If we turn off the termination check, we could derive any statement, like

no termination check:

```
{-# TERMINATING #-}
a : A
a = a
```

Here, we are calling the element (function without arguments) again to define itself. Of course, this would never give us a concrete element of A by evaluation, so it would be nonsensical to allow such definitions.

Side Note 2.3:8

In this thesis, all safety is turned on, so the only time we are disabling the termination checker we do it to make a point and later show how that same function can be implemented with termination checking turned on.
Disabling the termination checker is either marked by the `{-# TERMINATING #-}` pragma or simply that the code is written in our special code-box

no termination check:

```
recursion : A
recursion = recursion
```

Agda cannot always tell whether a function will terminate even if it does, especially when the arguments to the recursion are wrapped in a function. Let's say we define a (bounded) predecessor function as

```
bpred : ℕ → ℕ
bpred 0      = 0
bpred (1+ n) = n
```

This function gives a strictly smaller result than the input in all but the `0` case, however, in the following example, Agda cannot tell that the function terminates:

no termination check:

```
until0 : ℕ → ℕ
until0 0      = 0
until0 (1+ n) = until0 (bpred (1+ n))
```

Here, we make the recursive call on the result of `bpred` in a case where `bpred` does return a smaller element, but Agda does not have that information. The easy and often performed fix is to inline (meaning locally evaluate by hand) the definition of the problematic function, so we get

```
until0 : ℕ → ℕ
until0 0      = 0
until0 (1+ n) = until0 n
```

which is accepted by the termination checker.

Side Note 2.3:9

There is a feature in Agda called *sized types* that allows us to give types a size and therefore also let functions carry the information of how large their output is depending on the input. This feature is however experimental and does not mix well with cubical type theory, especially when using coinductive types. For our purpose, we manage every proof of termination when functions have to be applied through inlining.

2.3.7 Implicit Arguments

We have often seen the notation

```
id : {A : Set} → A → A
id a = a
```

to hide arguments that should be obvious from the context. This is exactly what the brackets `{ }` do, they allow us to not actively give an argument and makes Agda try to figure out its type on its own. In the above example this works well because Agda can infer the type `A` from the type of the functions input, so we rarely ever have to give it explicitly.

The way this works is that Agda unifies implicit variables that are at the same position, possibly assigning further implicit arguments until no more assignments can be made. This works well whenever implicit arguments are clear from the function inputs for example, because those usually come with preassigned variables in the position of the implicit arguments. Agda however does not perform any search here, so this is not a magical tool and it cannot assemble code from definitions of a module.

If we do want to give an implicit argument explicitly, we can write

```
id : {A : Set} → A → A
id {A = A} a = a
```

to bring the hidden variable into scope. The `{A = A}` notation means that the implicit variable `A` is now called `A` in the current context (we could also have chosen any other name). The explicitly giving implicit arguments via the `{A = A}` notation also works when explicitly giving a function an implicit argument.

We do not necessarily have to give a name to an implicit argument, we can also write `id {A} a` to achieve the same result. This works fine when there is only one implicit argument, but it can cause problems when there are several present. Take the following function:

```
2ndℕ : {a : Nat} {b : Nat} → ℕ
2ndℕ {a} {b} = b
```

When we want to give it the numbers `b : ℕ` and `a : ℕ` as arguments but accidentally write

```
2dnNat {b} {a}
```

we get the wrong number out when evaluation it

```
2dnNat {b} {a} = {a}
```

The reason is that we only gave the arguments *named* `a` and `b` but at the wrong position, effectively flipping them. To be precise what goes what, we can write

```
2dnNat {a = b} {b = b}
```

This is also super important when there are a lot of other implicit arguments present that we might not even be aware of.

2.3.8 Instance Arguments

There is a feature in Agda where actually some proof search is performed in special implicit arguments. Those arguments are given in `{{ }}` brackets. The proof search to fill those arguments then checks whether there is a unique solution from a given (recursive) set of definitions. An example is to automatically infer type classes.

Assume we have a type class (record types explained in Section 2.3.11)

```
record Num (A : Set) : Set where
    field
        _*_ : A → A → A
```

That defines a number class with multiplication. We can now allow Agda to search for several given instances of this type, for example by writing

```
instance
    numB : Num B
    numB = record { _*_ = _andBool_ }

    numN : Num N
    numN = record { _*_ = _*Nat_ }
```

We can now tell Agda to open the record using the instance search

```
open Num {{...}}
```

And then write

```
squareB : B → B
squareB x = x * x

squareN : N → N
squareN x = x * x
```

and let Agda figure out which multiplication to use in each case. This instance search also works in the inner scope of a function, so we can write

```
squareNum : {{Num X}} → X → X
squareNum x = x * x
```

to have a more general function.

Instances that take instance arguments themselves are searched through recursively and they are often used to model type classes.

In general, this instance search can do some automatic assembly of code, but especially when the search becomes recursive the complexity becomes exponential due to Agda having to make sure the used instance is unique. There are features that speed this up, but for now it is just nice to know that Agda can do some rudimentary proof search if we need it to.

2.3.9 Modules and Hiding the Context

Agda has a great mechanism to hide the context from definitions. So when we omit detail "for as long as it is clear from the context", there is an actual formal meaning behind that. The formalism from this Subsection will rarely ever be made explicit in the thesis. We only include this explanation to show that even something that would usually be handwaved away in mathematics can be made precise with Agda.

Agda has a module system that can be used to bundle together definitions. For example we can write

```
module M where
    a1 : ℕ
    a1 = 5
open M
```

to create a module with definitions in its scope. To bring those definitions into another scope, we use `open M`.

Side Note 2.3:10

When opening modules, we can choose which definitions to import using the **using** keyword. We can also arbitrarily rename definitions using the `rename` keyword.

The interesting part is that modules can be parametrised, so we can write

```
module M (A : Set) where
    f : A → A
    f a = a
```

To have a variable `A` to be used in the module.

There are now two main ways to open this module. First, we can give the module its argument, like

```
open M ℕ
```

Now, we have the function `f : ℕ → ℕ` in scope. This works with any type that we give when opening the module. The other way is to not give any argument when opening the module, like

```
open M
```

which brings a function `f : (A : Set) → A → A` into scope. When a module is opened, it gives its functions the arguments that it is given itself. If an argument is not given to a module, it is instead added as an argument in every function that we import from the module. This is why occasionally, when clear from the context, our defined functions change their number of arguments, depending on whether they all get the same argument from the module or whether we allow them to get a new argument every time we call them.

Arguments to a module can also be implicit or instance arguments.

Further, modules can open each other, either privately or publicly. If we have two modules

```
module M1 where
    a1 : ℕ
    a1 = 5

module M2 where
    open M1
    a2 = a1
open M2
```

Then one module can open another module to use its definitions, without making those definitions available to the outside. Opening `M2` in this scenario only brings `a2` into scope. If we open `M1` publicly however, like

```
module M1 where
    a1 : ℕ
    a1 = 5

module M2 where
    open M1 public
    a2 = a1
open M2
```

then, when opening `M2` will also bring all definitiond from `M2` (here: `a1`) into scope. This is how inheritance is modeled: publicly exporting definitions "inherited" from another model we opened.

Side Note 2.3:11

Module names can be left blank using and underscore `_` to avoid naming them and directly opening them. The advantage of doing that is that the anonymous model can import definitions that it does not export to the outside, giving us the possibility to only locally open other modules.

Modules are good for hiding information that exists somewhere, but modules on their own cannot be used to define axioms. That is done using record types. Before we start with those however, another quick way to hide unnecessary definitions

2.3.10 Variable Declarations

Quite often, we just need some names for variables and we know their types. Technically, we even do not want to untroduce them using `∀ a b c ...`. In order to make that possible, Agda allows us to declare variable that are automatically put as an implicit, universallyl quantified argument to whatever function they are used in. For example, we can write

```
variable
    i : Level
    A B C : Set i
```

and can then directly define for example identity as

```
id : A → A
id a = a
```

and Agda automatically desugars this into

```
id : {i : Level}{A : Set i} → A → A
id a = a
```

Often in this thesis, when clear from the context how the variables are quantified, we just use them in a function signature without introducing them to avoid clutter.

2.3.11 Record Types and Modules

Sometimes, we have types that have only one constructor but with a lot of arguments in that constructor. For example when we want to define a field (in the number theoretic sense), it has to store all the operations and laws on that field. In this case, it helps to be able to give a name to all those operations while still being able to change the underlying field. We can do that using record types.

Record types are data types that have only one constructor, but all arguments to that constructor are named. For example, we could define a three dimensional vector as

```
record Vec3 : Set where
    field
        x : ℕ
        y : ℕ
        z : ℕ
```

The interesting thing about records is that they induce a module. So if we open the record with

```
open Vec3
```

we can write a function

```
addVecEntries : Vec3 → ℕ
addVecEntries v = x v + y v + z v
```

Opening a record as a module brings all of its functions into scope, but still requiring the record as an argument, in this case the vector `v`. This is useful when several vectors (or other record types) are in scope. If we want all functions from a specific record in scope, we can do that by opening the record module with a concrete record, like

```
addVecEntries' : Vec3 → ℕ
addVecEntries' v = x + y + z
    where open Vec3 v
```

Here, we open the specific vector record for `v` to bring its `x`, `y` and `z` value into scope. This is also how the operations of a "type class" would be brought into scope.

Side Note 2.3:12

We can make local definitions using either the **`where`** clause after a function clause or the **`let`** construct before any definition. This can also be used to open modules.

Opening other type classes often looks like

```
record Num (A : Set) : Set where
    field
        _*_ : A → A → A

numℬ : Num ℬ
numℬ = record { _*_ = _andBool_ }

open Num numℬ
```

where we, here, opened the definitions from the number type class of Booleans.

There are four ways to create the values for a record. The first one is to use the record syntax that we just saw:

```
vec : Vec3
vec = record { x = 0 ; y = 1 ; z = 2 }
```

Another way is to define the outcome of each function given the record we are defining, like

```
vec : Vec3
x vec = 0
y vec = 1
z vec = 2
```

Because this way of defining a record is useful for coinductive types (described in Section 2.3.12), these definitions are called *copatterns*.

The third way to create a record is from the definitions of a module, but that is irrelevant to this thesis.

The fourth way is to give it an actual constructor. We can give a record a constructor by writing

```
record Vec3 : Set where
    constructor <_,_,_>
    field
        valx : ℕ
        valy : ℕ
        valz : ℕ
```

and then use it like a normal constructor.

```
vec : Vec3
vec = < 0 , 1 , 2 >
```

2.3.12 Coinductive Types and Records

If all functions terminate, then how to we reason about computation that might go on forever? The answer is: via coinductive types.

There is a more precise definition that we will see in the chapters once needed, but as a rule of thumb: An inductive type `A` is a type where we always have to be able to know how big the arguments to its constructors are, especially the recursive ones of type `A`. A common use case are tree-like data types, where each element forms a tree with finite width and depth. When using functions as constructor arguments, we can create infinite trees by using the functions to model distributions of constructor arguments, but they always need to be defined in a way such that for any given leaf we know that there is a finite path reaching it. A coinductive type on the other hand allows for the definition of data types without the knowledge of whether every path of constructors through the elements reaches a leaf after a finite amount of steps. This is especially useful when defining infinite data types like streams. The idea of a coinductive data type is that, instead of relying that while reading it we will always eventually hit a leaf, we can ask its elements for only the next constructor instead, leaving the decision to us whether we want to continue reading the data type. There might be infinitely many constructors, but in each query we are guaranteed to at least get the next one.

A typical example of a coinductive type are streams. A stream is an object that we can always ask for the next element and the successive stream of elements. We can define it as a record as:

```
record Stream (A : Set) : Set where
    coinductive
    field
        elem : A
        next : Stream A
```

This record holds the next element and the next stream, which is of the same type as the recurd. By declaring the record `coinductive` we tell Agda that this is no longer a finite inductive type, but a possibly infinite coinductive type.

When we create streams, we have to make sure that we can always give the next element. For example, when we repeat the same object infinitely often, we can define this using copatterns as

```
repeat : A → Stream A
elem (repeat a) = a
next (repeat a) = repeat a
```

So the when given a repeated stream, its next element is always the element `a` and the next stream is the repeated stream `repeat a` again. We can immediately see that if this were a definition on an inductive type it would never terminate. For coinductive types, Agda does not test for termination but for *productivity*. Productivity means that the creation of the next constructor always terminates. Of course, the information that this construction terminates is lost when applying functions. What is different here is that the recursive call does not have to be on a smaller element; it can be on any. What is important is that the existence of the next constructor is guaranteed. An example of where we would lose that information by applying a function to the recursive call is if we define `repeat` differently using an auxiliary function

```
prepend : A → Stream A → Stream A
elem (prepend a s) = a
next (prepend a s) = s
```

That adds an element to the front of a stream. If we define repeat as

no termination check:

```
repeat : A → Stream A
repeat a = prepend a (repeat a)
```

We would think that the prepend makes sure the next element `a` can be given, however, because we are applying a function to the recursive call without giving a constructor, the termination checker (productivity checker) rejects the definition. This can be solved with inlining again, but in general for coinductive definitions, this has a tendency to be a lot harder than with inductive ones.

2.3.13 Universe Polymorphism

We have already talked about the fact that the type of types `Set` cannot contain itself, otherwise we would create russel's paradox [83], which can also be derived in Agda when deactivating the safety [7]. To avoid this, the types of types in type theory form a hierarchy `Set : Set1`, `Set1 : Set2`, `Set2 : Set 3` and so on. In order to not having to stay on one universe level, Agda's type for types `Set` can be parametrised by an element of a special type `Level`, so given an `i : Level` we can have a `Set i`.

Given a level `i`, we can create its successive level `lsuc i`. Also, given two levels `i` and `j`, we can create the maximum of these levels `i ⊔ j`. The function type preserves these levels, so given two types `A : Set i` and `B : Set j`, we have

```
(A → B) : Set (i ⊔ j)
```

If an input to our function is itself of type `Set i`, then the overall type is one universe level above, so

```
(Set i → B) : Set (lsuc i ⊔ j)
```

The same holds for data type declarations. If the type we declare needs a `Set i` as a parameter, its own type is one universe level above. For example, when defining a monoid (ommitting laws for brevity) with universe polymorphism, we would write

```
record Monoid {i : Level} : Set (lsuc i) where
    field
        A : Set i
        ε : A
        _⊕_ : A → A → A
```

and because `A : Set i`, the overall type is `Set (lsuc i)`.

With this in mind, finding the most general universes for a type is simple: Assign each type with a different level and then form the maximum, raising those type levels where a `Set i` is used as an argument to a constructor. Therefore, while the entire code in this thesis was written universe polymorphic unless stated otherwise, we will only give the definitions without the universe levels and they can be easily recreated when the results of this thesis are replicated.

It should be noted that the type level of a type only rises when an argument to a constructor is of some type `Set i`. This is not the case for arguments to the type level constructor. Just like functions to not raise their output type level based on the input, neither do functions creating types, so if we write

```
record Monoid {i : Level} (A : Set i) : Set i where
    field
        ε : A
        _⊕_ : A → A → A
```

and give the `A` as an argument to the type, then all elements in the type constructor arguments are below level `i` and we are all good.

2.3.14 Indexed Types

Data types in data type declarations have two different ways to get arguments for its type constructing function. The first one is getting an arbitrary element of some type, bringing its name into the scope of the definition. In the definition of lists for example, we write

```
data List (A : Set) : Set where
    [] : List A
    _::_ : A → List A → List A
```

The upside is that the given element is in scope of the constructors. The downside is however that we cannot pattern match the given object. We will see why that is useful by giving an example. The second way to give arguments to the type constructing functions is by making the type indexed. We can use this to define vectors, which are lists of a given, fixed length

```
data Vec (A : Set) : ℕ → Set where
    []   : Vec A 0
    _::_ : A → Vec A n → Vec A (1+ n)
```

Here, we state that the recursively called type has to be one smaller than the currently created type. That means that we have to have some knowledge on the index that we have been created on: Only when we have been cerated on a number greater than zero we can sill add another element to the vector. Likewise, only when we are being created with length `0` can we actually end the vector. Therefore, vectors have exactly the length they were created with.

If we hadn't gotten the length as an index, we could not have constrained that in the `_::_` case, wou would have to have at least length `1+ n`. All type arguments given by name have to show up the exact same way in the constructor, but indices don't. They can (and have to be) redefined for every constructor, which we do here implicitly by using the declared variables from Section 2.3.10 to automatically give the constructor a `n : ℕ` into its signature.

It should be noted that the equality mechanism we are using for this thesis does not support such indexed types well. In order to still use indexed types, we have to use equality types as arguments to the constructors, like

```
data Vec (A : Set) (n : ℕ) : Set where
    []   : n ≡ 0 → Vec A n
    _::_ : A → Vec A n' → 1+ n' ≡ n → Vec A n
```

This is an equivalent type where we constrain the length value using equality types. This definition works the same, is just a bit more annoying to work with, but it is compatible with cubical Agda explained in Section 2.4.

2.3.15 Dependent Functions and Types

Agda is a dependently typed programming language. This means that the type of an object can depend on other objects in the context. The output of a function

for example can depend on the input to the function. In general, given a type `A : Set` and `B : A → Set`, we can write a dependent function as

```
f : (a : A) → B a
```

As an intuition, this can for example be used to define a type safe matrix multiplication

```
mulMtx : {n m p : Nat} → Matrix n m → Matrix m p → Matrix n p
```

Where the dimensions of the second matrix depend on the dimensions of the first matrix and the outputs dimensions depends on some dimensions of both inputs.

When defining dependent functions, the way to write clauses stays largely the same, but it has to be kept in mind that the output type can change when we case split. For example, when we write a function creating a type

```
B : 𝔹 → Set
B true  = 𝔹
B false = ℕ
```

then the type of an object of type `B a` changes depending on whether `a` is true or false. Creating a `B a` can now look as follows:

```
f : (b : 𝔹) → B b
f true  = {!0!}
f false = {!1!}
```

When checking the hole `{!0!}`, we can see that it has the type 𝔹, whereas hole `{!1!}` has the type ℕ. Therefore, a possible implementation would be

```
f : (b : 𝔹) → B b
f true  = true
f false = 1
```

When we think of types as propositions, the dependent functions are how we create universal statements. "Forall elements `a` of `A`, the statement `B` of `a` holds". This is especially useful when trying to prove equations.

2.3.16 Intensional Equality

There are several ways to model equality in type theory [92], but the standard way that Agda provides is its intensional equality. To grossly oversimplify: This is a form of equality that mostly comes from syntactical equality and evaluation. Agda internally determines two objects to be equal if they evaluate towards the same object. The way this is done is that two terms are brought into what is called *head normal form*, a shape in which all functions that could reduce have been reduced. If the resulting terms are equal syntactically, that means that the two original objects were equal as well. Of course, this equality cannot tell two objects equal that look completely different, which is a problem when comparing functions. Just because two functions have the same input-ouput-relation it does not mean that they have the same implementation, so Agda can, in general, not tell that they are equal. That left aside: The internal proving equalities through evaluation and normalisation already does most of the heavy lifting in proofs, so it is worth mentioning a little about how it works. In this section, we will present an old way of how equality is handled in non-cubical Agda. In the remainder of this thesis, we will however use the equality mechanism of cubical Agda described in Section 2.4, but it does share some principles when applied.

2.3.16.1 Reflexivity, Symmetry, Transitivity and Congruence

A data type for Agda's intensional equality can be defined a the indexed data type

```
data _≡_ {A : Set i} (a : A) : A → Set i where
    refl : a ≡ a
```

This data type is constructed via two elements of type a given type `A`, however one of the elements is given in the type and is available for the constructor definition while the other one is given by the index and can be assigned a value

in the constructors. The only constructor `refl`, can only be constructed when both elements given in the type are equal according to Agdas intensional equality checker.

An example of an equality proof is to prove that `1` equals `1`.

```
1is1 : 1 ≡ 1
1is1 = refl
```

Here, the two terms given in the type are definitionally equal, so Agda can tell that they are equal. Agda can also sometimes tell when terms that need to be evaluated are equal. For example, it can tell that the following equation holds automatically:

```
1+1is2 : 1 + 1 ≡ 2
1+1is2 = refl
```

This statement is occasionally helpful [83]. However, sometimes expressions cannot evaluate. For example, when we look at the definition of addition for natural numbers:

```
_+_ : ℕ → ℕ → ℕ
0      + y = y
(1+ x) + y = 1+ (x + y)
```

we can see that only the first argument is being case split on, but not the second. Therefore, Agda cannot immediately tell that the following equation holds:

error:

```
n+0isn : ∀ n → n + 0 ≡ n
n+0isn n = refl
```

the reason is that the `n` cannot evaluate if its value is not known. This can be solved by case splitting on `n`

```
n+0isn : ∀ n → n + 0 ≡ n
n+0isn 0      = refl
n+0isn (1+ n) = {!!}
```

Agda can now easily tell that `0 + 0 ≡ 0`, because all values are assigned. However, in the `1+ n` case where the output type is

```
(1+ n) + 0 ≡ (1+ n)
```

It still cannot tell because if we evaluate the terms as far as possible, we get

```
1+ (n + 0) ≡ 1+ n
```

and those terms are not syntactically equal. To solve this problem, we need to define a few combinators for equality.

Through the `refl` constructor, we already know that our equality is reflexive. Luckily, Agda also lets us proof that equality is symmetric and transitive

```
sym : a ≡ b → b ≡ a
sym refl = refl

trans : a ≡ b → b ≡ c → a ≡ c
trans refl refl = refl
```

The reason we can make this proof is because as soon as we pattern match on the constructor `refl`, Agda internally can tell that also the respective input variables `a`, `b` (and `c`) have to be equal and we can give the `refl` constructor. It should be noted that this proving equalities through pattern matching no longer

works in the equality mechanism used in the thesis, but we can still prove the same operators for equality.

Further, and this is important for our problem above, we can prove that congruence holds

```
cong : a ≡ b → (f : A → B) → f a ≡ f b
cong refl _ = refl
```

This means that we can apply a function to both sides of the equation. In mathematics, applying congruence often looks like

$$\begin{array}{lll} & \mathtt{a} = \mathtt{b} & \mid \mathtt{f} \\ \Leftrightarrow & \mathtt{f}(\mathtt{a}) = \mathtt{f}(\mathtt{b}) & \end{array}$$

In order to mimic the mathematical notation for applying a function to both sides of an equation, we use the operator

```
_||_ = cong
```

We can now prove our above statement by induction, adding `1` to both sides of the result from the recursive call

```
n+0isn : ∀ n → n + 0 ≡ n
n+0isn 0      = refl
n+0isn (1+ n) = n+0isn n || 1+_
```

where `n+0isn n : n + 0 ≡ n`, so adding one gives `n+0isn n || 1+_ : 1+ (n + 0) ≡ 1+ n` which is the desired type and proves the statement.

2.3.16.2 Chains of Equations

In this subsection we will talk about the notation for chains of equations. The mathematical notation

$$a = b = c$$

does not really work with our equality type, as the statement

error:

```
a ≡ (b ≡ c)
```

would mean that `a` is equal to the equation `b ≡ c` which is, in general, not what we want to express. In order to give a chain of equations, we wrap transitivity into a syntax declaration that makes the individual elements visible while we apply equations on them. Omitting name aliases, we write

```
syntax trans {a = a} a=b b=c = a =< a=b > b=c
```

This of course leaves us with an equation in the end and not a single element, so we finish a chain using the operator

```
_qed : (a : A) → a ≡ a
_qed a = refl {a = a}
```

So when writing the mathematical

$$a \overset{eq_1}{=} b \overset{eq_2}{=} c$$

the Agda code would be

```
a =< eq₁ > b =< eq₂ > c qed
```

Side Note 2.3:13

In order to avoid clutter, when giving equations in a chain between elements, we omit transformations like symmetry and congruence and just give the main argument.

2.3.17 Fundamental Types and Curry-Howard Correspondence

In this section, we are talking about the fundamental types that make up the building blocks for dependent type theory. In [92] they mark the types that are basically axiomatic to the theory (in the sense that they have to be defined for the theory in order for the theory to work). While we define them, we also explain their meaning to the Curry-Howard isomorphism [47], which states that types are isomorphic to logical propositions, while their terms are isomorphic of the proofs of those propositions. To get the nomenclature consistent: A *term* of a type is the same as an *element* or *object* of a type, though the term *term* is often used when actually speaking about the term as something syntaxtic.

2.3.17.1 The Unit Type

The first type is the rather boring `Unit` type. It has exactly one constructor.

```
data Unit : Set where
    unit : Unit
```

The element of this type can always be created, which also means that it does not contain any useful information. For logics, the `Unit` type is the proposition for truth, often also called $\top$. Sometimes in the literature, because this type has only one element, it is also called $\mathbb{1}$.

2.3.17.2 The Zero Type and Contradictions

The next type is a type that cannot be created. If an element of it is deduced, that means that there is a contradiction.

```
data Zero : Set where
```

It does not have any constructors because it cannot be created. As this type is used for contradictions, in logics, it is also called $\bot$. While not having constructors, it however does have an elimination rule

```
absurd : Zero → A
absurd ()
```

This rule states that whenever there is an element of type `Zero`, then we can deduce any element. This is useful when proving universal statements. In cases where there is a contraction, meaning the case can never happen, the universal statement should still hold, so in this case we allow any statement to be deduced.

Side Note 2.3:14

If Agda can tell that there is no element of a given type we can write the `()` parenthesis and omit implementing the clause all together, because the case cannot happen anyway.

A negation is modeled by stating that from a statement, a contradiction follows, so we define

```
¬_ : Set → Set
¬ A = A → Zero
```

and when we want to prove that something does not hold, we have to prove that from the statement holding a contradiction follows. It is important that unlinke in classical mathematics, double negation cannot be used to prove a statement, so the following cannot be proven:

```
-- hole cannot be filled
prove-by-contra : ¬ (¬ A) → A
prove-by-contra ¬¬A = {!!}
```

2.3.17.3 Tuples and Conjunction

The next type is a tuple. It contains an element for each of two given types.

```
data _×_ (A : Set) (B : Set) : Set where
    _,_ : A → B → A × B
```

The name `_×_` stems form its usage to model product spaces.

This type is also often defined as a record to directly get accessor functions for the individual elements:

```
record _×_ (A : Set i) (B : Set j) : Set (i ⊔ j) where
    constructor _,_
    field
        fst : A
        snd : B
```

Because we are storing elements of two different types, this type is also called a *conjunction* and used as the logical "and". Therefore, we also define a type alias

```
_and_ = _×_
```

2.3.18 Disjunctions

We can define a type that holds either one of two objects

```
data _⊎_ (A : Set) (B : Set) : Set where
    left  : A → A ⊎ B
    right : B → A ⊎ B
```

It has two cases: One holding an element of A and the other holding an element of B. This type is used to model a disjunction, as in order to prove A or B, we have to either give proof of A or a proof of B. Therefore, we define the type alias

```
_or_ = _⊎_
```

The original name _⊎_ comes from the type kind of defining a set union in the set theoretic sense, but as we can still always tell from which of the two unified sets the elements come, it is a disjunct set union; hence the symbol. Sometimes this type is also called a "Sum", but that can easily be confused with a dependent sum from Section 2.3.18.3, which is a much different data type.

In order to occasionally avoid explicit case splitting, the type has an elimination function

```
fromSum : (A → C) → (B → C) → A ⊎ B → C
fromSum fa fb (left  a) = fa a
fromSum fa fb (right b) = fb b
```

Where we say that if a statement follows for both cases of the disjunction, it follows in general.

2.3.18.1 Functions and Implications

Nondependent functions (sometimes called "exponentials") are also considered a primitive. We have already described how they work in Section 2.3.1. As for the Curry-Howard correspondents, they represent an implication, because for all proofs of the input A we give a proof of the statement of B, meaning that from A it follows that B. They work just the same as dependent functions described in Section 2.3.15 only that their output type does not use the input argument.

In Agda, because the function arrow is a primitive, we cannot use it as a context itself, so we cannot describe the space of all proofs that follow from statements A by simply writing

error:

```
A ->_
```

Therefore, we define a wrapper

```
Morphism : Set → Set → Set
Morphism A B = A → B
```

and can then describe the space of types that can be created by means of an `A` as

```
Morphism A
```

(This makes sense when thinking of a `Morhpism A` as a functor or monad described in Section 2.5).

2.3.18.2 Dependent Products

Much different to what the same suggests, a dependent product has, at first, very little to do with tuples. In dependent type theory, a *dependent product*, also called *pi type* (written as a **Π**) is actually the dependent function type that we already talked about in Section 2.3.15. Usually, we do not give it its own definition as it is a primitive, however we can define it as

```
record Π (A : Set i) (B : A → Set j) : Set (i ⊔ j) where
    constructor lam
    field
        app : (a : A) → B a
```

which is a type that wraps a dependent function. We can define the function application operator as follows (is usually defined on the primitive, non wrapped functions):

```
_$_ : Π A B → (a : A) → B a
f $ a = (app f) a
```

Why this is a useful view on pi types can be seen in Chapter 4 Section 4.7 because it makes it easier to model the type of type theory.

The reason that these dependent functions are called `dependent products` is because we can actually model nondependent products with them. If we define the product type to be

```
--not official product type
_×_ : (A : Set i) → (B : Set i) → Set i
A × B = (c : 𝔹) → if c then A else B
```

Here, the tuple is actually a function that for all booleans has to give us the computed type, which means that a `true` has to result in an `A` and a `false` has to result in a `B`. That means that the type does technically contain two elements. We leave the universe polymorphism explicit here because this does not work as smooth when `A` and `B` are on different universe levels (though that as well can be achieved).

We can write a constructor like

```
_,_ : A → B → A × B
(a , b) true  = a
(a , b) false = b
```

to create this non dependent tuple. This construction however does not work to give a tuple where the second entry depends on the first one. For that we need a dependent sum.

2.3.18.3 Dependent Sums

A dependent Sum is a tuple in which the second argument depends on the first one. It has a vast range of applications like modeling data or existential statements. It is defined like a tuple, but with the type argument depending on one another:

```
record Σ (A : Set i) (B : A → Set j) : Set (i ⊔ j) where
    constructor _,_
    field
        fst : A
        snd : B fst
```

These types represent existential statements, because if we want to state that there exists an `A` on which `B` holds, then we provide an element `a : A` and an element `B a` to show the statement. Therefore we give two different syntax declarations (Section 2.3.21) to be able to use this type without the overhead of lambda abstractions

```
-- see sidenote
syntax Σ _ (\a → B) = ∃[ a ] B
syntax Σ A (\a → B) = ∃[ a of A ] B
```

So writing `∃[ a of A ] B` is just a different way of writing `Σ A (\a → B)`, where `B` depends on `a` (meaning that `a` is in scope of `B`).

Side Note 2.3:15

Syntax declarations from Section 2.3.21 have to use every variable, so the first syntax declaration actually has to be written like

```
exists : {A : Set} → (B : A → Set) → Set
exists {A = A} = Σ A

syntax exists (\a → B) = ∃[ a ] B
```

We quickly showcase how sigma types can be used to model data. The reason this is called a dependent sum is because we can model non dependent sum types with it, much like dependent products could model non dependent products. Here, we use the boolean argument to choose whether the second argument is of type `A` or `B`, so we either have an element of `A` or `B`.

```
--not official sum type
_⊎_ : (A : Set i) → (B : Set i) → Set i
A ⊎ B = Σ 𝔹 \c → if c then A else B
```

Again, this only works smoothely when both `A` and `B` are on the same universe level. We can now define the constructors as

```
left : A → A ⊎ B
left a = (true , a)

right : B → A ⊎ B
right b = (false , b)
```

This way to create data types given a type with the cardinality of the constructors forms the basis of containers, discussed in detail in Chapter 4.

2.3.19 Predicates

It should be mentioned that the equivalent of a subset in set theory is in type theory expressed as a predicate

```
P : A → Set
```

This filters the possible values `a : A` that can be created when having an element `P a`. The way this filter works is by sometimes having the element `P a` have the empty type, like

```
P : 𝔹 → Set
P false = Zero
P true  = Unit
```

So a value of `P false` can never be created and in this case we know that

when given an Element `P b`, we know that `b ≡ true` because that is the only way to create an element `P b`.

2.3.20 Creating Modular Types

With the fundamental types, we can also create data types without a data type declaration. We could even show that these definitions are isomorphic (equal under univalence from Section 2.4) to the data type delaration types, but we will omit this here for brevity.

For example, the Booleans can be defined as

```
𝔹 : Set
𝔹 = Unit ⊎ Unit
```

Finite types can be defined as

```
Fin : ℕ → Set
Fin 0      = Zero
Fin (1+ n) = Fin n ⊎ Unit
```

and we can even define vectors as

```
Vec : (A : Set i) → ℕ → Set i
Vec A 0      = Unit
Vec A (1+ n) = A × Vec A n
```

We cannot yet define general recursive types with this (here, the type definition has to terminate through one of the type arguments), but we will do so in Chapter 4 Section 4.2 with the concept of containers. It is however good to keep in mind what the building blocks of dependent type theory are when trying to build solvers for the theory.

2.3.21 Syntax Declarations

In order to mimic any notation we might want it is curial that we can have arguments to a function at any position. However, in the context of dependent functions, the order of arguments is often fixed because early arguments cannot depend on later ones. To solve that, Agda provides so-called *syntax declarations*. We introduce them here to show that every time we introduce notation in this thesis, there is an actual formalism behind what that means.

Let's say we want to write proofs where the type of argument over which we are proving something is not all that relevant, so we want to move it further behind. We can do so by writing

```
syntax Σ A B = B proven-over A
```

Here, the syntax declaration gives us the variables where the arguments would be placed and we can put them in fairly arbitrary places for our syntax (note that variables and syntax tokens always have to switch, so no two variables and no two syntax tokens can follow in succession). We can now use our defined syntax in the code.

Side Note 2.3:16

Agda only allows syntax declarations in the same scope as the function over which the syntax is defined. It also only allows one syntax declaration per function. To circumvent that, we can define an alias and then define the syntax for that alias:

```
revSig = Σ
syntax revSig A B = B proven-over A
```

To keep definitions better readable, we do not make this explicit and just use the original type or function names.

We can now show that `0` exists as

```
ex0 : (\n → n ≡ 0) proven-over ℕ
ex0 = 0 , refl
```

Which is already nice, but it is still annoying that we have to use a lambda abstraction to define the variable name. We have already seen how this can be circumvented in the declaration

```
syntax Σ A (\a → B) = ∃[ a of A ] B
```

where we can put the variable name declaration somewhere into our syntax.

2.3.21.1 Pattern Synonyms

Sometimes it can be good to give constructors or even nested constructors a new name for overview. We will not go into detail about all the constraints that apply here, but if you see a declaration like

```
pattern fstConstr x = left x
pattern sndConstr x = right (left x)
pattern thdConstr x = right (right x)
```

it just means that (nested) constructors now also have a different name which can also be used in pattern matching.

2.3.22 Other Useful Types and Functions

In this section, we just go over a few types and functions that we use throughout this thesis. They should be common knowledge for functional programmers, but we include them here for completeness. They are usually taken from the Agda standard library [6].

2.3.22.1 Operators on Functions

Function application (often important because it has a low binding strength, explained below):

```
_$_ : ∀ {A : Set} {B : A → Set} →
    ((x : A) → B x) → ((x : A) → B x)
f $ x = f x
```

Because this operator has a low binding strength it can be used to avoid brackets. So

```
f (g (h x)) ≡ f $ g $ h x
```

Function composition:

```
_o_ : ∀ {A : Set} {B : A → Set} {C : {x : A} → B x → Set} →
    (∀ {x} (y : B x) → C y) → (g : (x : A) → B x) →
    ((x : A) → C (g x))
(f ∘ g) x = f (g x)
```

2.3.22.2 The Maybe Type

Used for values that might only maybe exist.

```
data Maybe (A : Set) : Set where
    nothing : Maybe A
    just    : A → Maybe A
```

2.3.22.3 List Operations

```
map : (A → B) → List A → List B
map f []       = []
map f (x :: xs) = f x :: map f xs

maybeToList : Maybe A → List A
maybeToList nothing  = []
maybeToList (just x) = x :: []

mapMaybe : (A → Maybe B) → List A → List B
mapMaybe p []       = []
mapMaybe p (x :: xs) with p x
... | just y  = y :: mapMaybe p xs
... | nothing = mapMaybe p xs

catMaybes : List (Maybe A) → List A
catMaybes = mapMaybe id


foldr : (A → B → B) → B → List A → B
foldr c n []       = n
foldr c n (x :: xs) = c x (foldr c n xs)

_++_ : List A → List A → List A
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

concat : List (List A) → List A
concat = foldr _++_ []

concatMap : (A → List B) → List A → List B
concatMap f = concat ∘ map f
```

2.3.22.4 Finite Types

Finite types are types with only finitely many type constructors. They can be defined as

```
data Fin : ℕ → Set where
    f0   : Fin (1+ n)
    f1+_ : Fin n → Fin (1+ n)
```

They have exactly `n` many elements. That is because they can have at most `n` elements to fit their indexing, but they can stop at any smaller `n'` using the `f0` constructor, so there are also at least `n` many elements.

2.3.22.5 Decidables

Sometimes, we do not just want a Boolean output when modeling a computable predicate, we want that Boolean to be a proof of a statement in the `true` case (or the negation of a statement in the `false`). In the branches of an if-then-else for example, when writing correct code, we case split for a reason an the branch only works when that reason either does or does not hold in their respecitive branches. In order to model this in type theory, we use the type

```
data Dec (A : Set) : Set where
    yes :   A → Dec A
    no  : ¬ A → Dec A
```

that states that we can compute whether a statement does or does not hold. We have several variations of if-then-else operators that we use interchangeably depending on what is needed in the context.

```
ifDec_then_else_ :
    (d : Dec A) →
    ((a : A) → B) →
    ((a : ¬ A) → B)
    → B
ifDec yes a then x else y = x  a
ifDec no ¬a then x else y = y ¬a
```

With this operator we are doing a case distinction where the branch is given the statement from the decision. This is used when the statement is needed. If the statement is not needed we will just use the operator

```
if_then_else_ : (d : Dec A) → B → B → B
if yes a then x else y = x
if no ¬a then x else y = y
```

There is a "type class" for data types with decidable equality, defined as

Definition 2.3.17: **Decidable Equality**

A data type A is said to have **dcidable equality** iff there is a function

```
_==_ : ∀ {a b : A} → Dec (a ≡ b)
```

This "type class" is useful when we actually have to compute whether two elements are equal at runtime.

2.4 Cubical Type Theory and Equivalence

It might seem unintuitive, but equality is a yet unsolved problem of computer science. The reason is that equality is more complex than just comparing two chunks of data to have the same bits at the same positions. A famous example is that of extensionality. In logics, outside of complexity theory, two functions should be considered equal if for every input they produce the same output. Just from the pure implementation of the function however we often cannot tell whether two functions with different implementations always produce the same output. This is however what Agda's intensional equality is doing in this case, so when just using plain Agda, a lot of proofs on functions like proving the monadic laws cannot be done. It should be noted that Agda has a few inbuilt equations that it does apply on its own like when comparing two expressions that still have to be evaluated, but in general we cannot rely on Agda always figuring out whether an equation holds or not. To tackle this problem, Agda has an extension called *cubical Agda* that implements cubical type theory [92] to solve this problem. This new notion of equality is strong enough to even implement the concept of *univalence*, where two types are considered equal even if there is 'just' an isomorphism between the types. This enables to create strong and general type checked proofs, but it also comes with a few caveats that we will describe in this section. Even if we do not always make it explicit, the equality notion of cubical Agda is always the default notion of equality used in this thesis

unless stated otherwise. We will not given a whole explanation as this is not a thesis on cubical type theory, however, we give enough intuition to understand what the presented proofs to. We begin with a few basics, taken from [11].

2.4.1 Homogenous Paths

An equality in cubical Agda is represented as a so-called `Path`. A path represents the transformation of one object into another "continuously" over an interval `I`. We put "continuously" into quotation marks because the interval type used here is not actually a real number but more the minimal structure required for those transformations to work in a computer, however, the analogy of a continuous transformation helps when understanding the theory. We start by making this definition precise.

Definition 2.4.1: **The Interval Type**

We call a special type `I` the **interval type**. It has two primitive elements **`i0 : I`** and **`i1 : I`** (not to be confused with the often used variable name `i`). There are three primitive operations

```
~_ : I → I
_/\-path_ : I → I → I
_\/-path_ : I → I → I
```

that form a de Morgan algebra, with `_/\-path_` being the minimum, `_\/-path_` being the maximum and `~_` being the negation. This means they form a bounded, distributive lattice, as well as satisfy the following laws:

```
~ (~ i) ≡ i
~ i0 ≡ i1
~ i1 ≡ i0

~ i /\-path ~ j ≡ ~ (i \/-path j)

~ i \/-path ~ j ≡ ~ (i /\-path j)
```

It follows from the definition, but because it is often important, we

stress that through being a bounded lattice, they also follow the following laws:

```
i0 /\-path j ≡ i0
i1 /\-path j ≡ j

i0 \/-path j ≡ j
i1 \/-path j ≡ i1
```

Side Note 2.4:2

The interval operators are only included here for the sake of completeness. For now, we only need the interval inversion operator `~_`. The other operators are used in the case of dealing with several interval variables (equations in higher dimensions), which happens for example when dealing with equations of equations or certain operations called *fillers*, but we found this to be a detail unnecessary to understand the results in this thesis.

Side Note 2.4:3

Important: Intervals are not booleans. While they have only two primitive elements given in their definition, we treat them as if they had more values in between, similar to values between 0 and 1. For the theory however, it is not important what those theoretical values are, as we cannot construct them (that is done by the theory behind cubical type theory). It is important to know that we, in general, cannot case split on a value of an interval type. We cannot just give a case for `i0` and `i1` to create a path unless we are using special primitives that add constraints on the types we are allowed to connect via paths (that we are not going to cover here because it would be unnecessarily technical. For a more in-depth tutorial, we recommend [11]).

Intervals are used to create so-called *Paths*.

Definition 2.4:4: **Homogenous Path**

Given a type `A`, a function

```
p : I → A
```

is called a **homogenous path from `p i0` to `p i1`**. As notation, we write

```
p i0 ≡ p i1
```

or, when clear from the context that `p i0 = a` and `p i1 = b` we write

```
a ≡ b
```

Side Note 2.4:5

It should be noted that in Agda, an element of type `p : I → A` is not automatically a path, because we usually want a path from some `a` to `b` and not `p i0` and `p i1`. Therefore, Agda has its own representations of paths from `a` to `b` as a type

```
Path : (A : Set) → (a : A) → (b : A) → Set
```

It should also be noted that Agda's `Path` type has a different notion of equality than functions in general, but we will not get into detail here. For notation, we write

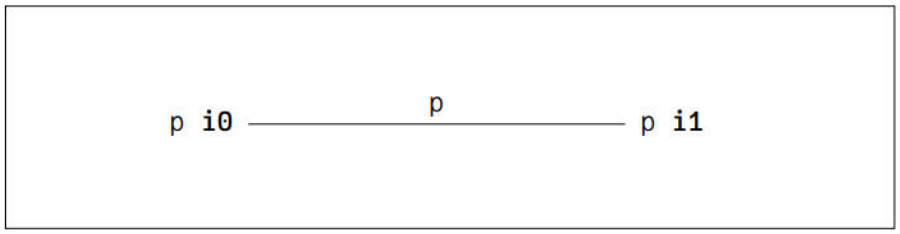
```
_===_ : {A : Set} → (a : A) → (b : A) → Set
_===_ {A} a b = Path A a b
```

That being said, we can always switch between the two representations as

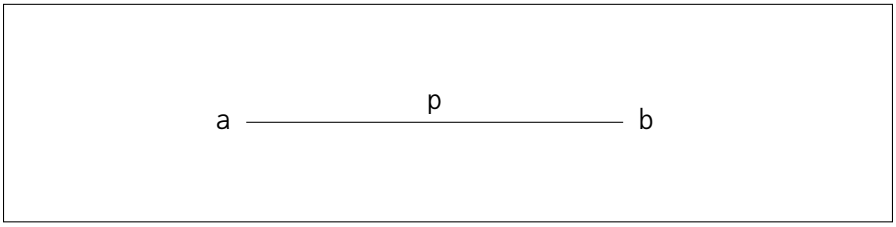
```
path-to-eq : (P : I → A) → P i0 ≡ P i1
path-to-eq P i = P i

eq-to-path : {a b : A} → (a=b : a ≡ b) → ((i : I) → A)
eq-to-path a=b i = a=b i
```

But as we can see, Agda's internal representation of paths on the outside still looks like a function that takes a value of the interval type to produce a result, so we don't have to think about these two representations too much.

Using paths as the mechanic for equality makes this theory very visual. A path `p : I → A` can be visualised as a line between the endpoints

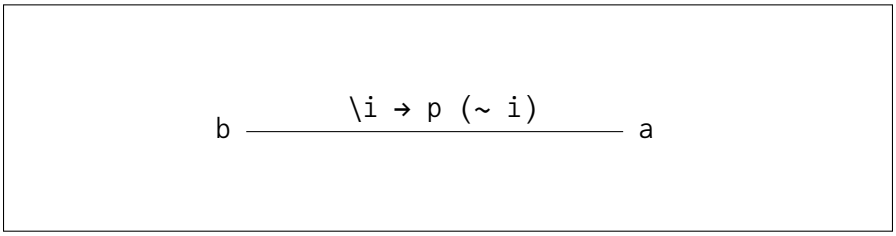


or, when it is clear from the context that `p i0 = a` and `p i1 = b`, then as the line



If a line has no annotation, that means that the equation can be figured out just using computation, like reducing terms. In the actual proofs, those lines would usually not be made explicit.

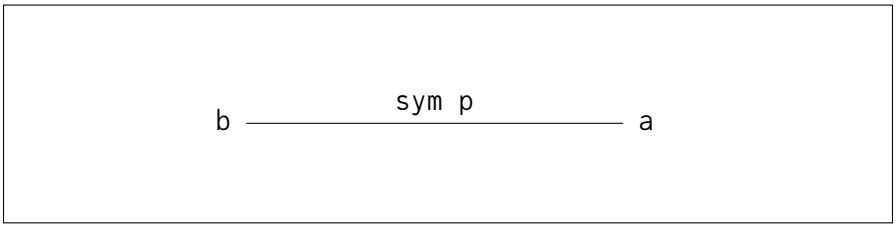
We can use the operators to easily change the direction of a path. We can write



We can also write this nicer using a function

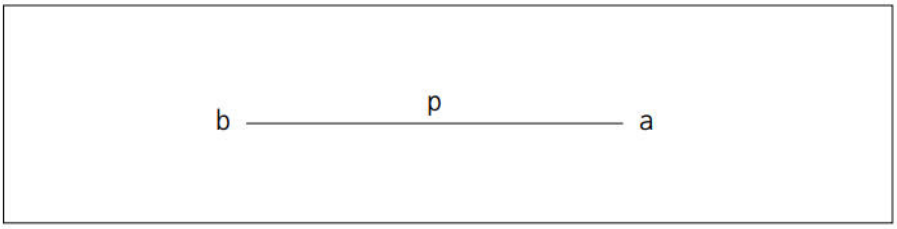
```
sym : a ≡ b → b ≡ a
sym a=b i = a=b (~ i)
```

to get the diagram

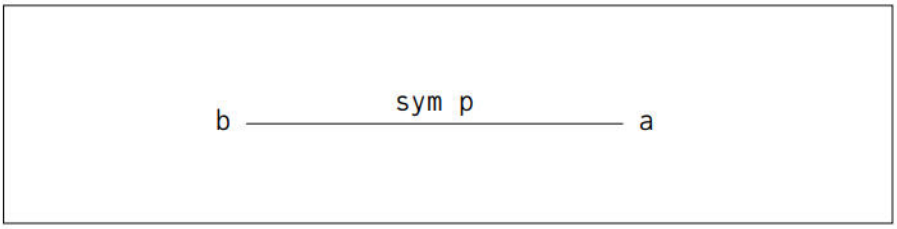


When clear from the context however, we will omit explicit symmetry of

paths and just treat paths as non-directional, so we write



instead of



To better understand how Agda treats equations as paths, it can help to see Agda's output when creating a path.

```
sym : a ≡ b → b ≡ a
sym a=b i = {!!}
```

When we give the representation of the path's body, Agda makes sure that the paths endpoints are consistent with the desired path. The endpoints of a path are still checked using Agda's intensional equality solver, but the internal representation can change in between endpoints (which is impossible in Agdas pure intensional equality). We will see later where this is useful. For now, Agda's output on the above example is:

```
A
-------------- Boundary (wanted) --------------
i = i0 ⊢ b
i = i1 ⊢ a
```

If we now apply the intervall parameter `i` to an equation, Agda takes the endpoints of the equation into account when creating the path, so for example putting

```
sym : a ≡ b → b ≡ a
sym a=b i = {! a=b i !}
```

we can see that the endpoints do not match.

```
A
-------------- Boundary (wanted) --------------
i = i0 ⊢ b
i = i1 ⊢ a

A
-------------- Boundary (actual) --------------
i = i0 ⊢ a
i = i1 ⊢ b
```

Moving the path backwards using the `~_` operator however yields:

```
sym : a ≡ b → b ≡ a
sym a=b i = {! a=b (~ i) !}
```

```
A
-------------- Boundary (wanted) --------------
i = i0 ⊢ b
i = i1 ⊢ a

A
-------------- Boundary (actual) --------------
i = i0 ⊢ b
i = i1 ⊢ a
```

and, when giving the implementation to Agda, we have proven the equation.

Similarly, we can prove congruence. We do this here just to show the notation we will be using in this thesis:

```
_||_ : ∀ {a b : A} → a ≡ b → (f : A → B) → f a ≡ f b
(a=b || f) i = f (a=b i)
```

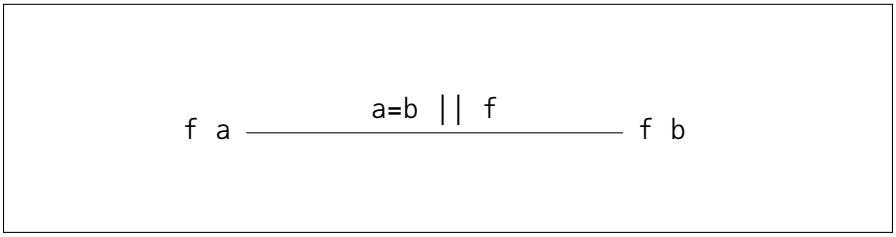
Side Note 2.4:6

Congruence is in mathematics often expressed using a single | next to an equation, to apply a function to both sides, like

```
      a = b     | + 1
⇔ a + 1 = b + 1
```

Sadly, a single pipe | is a reserved symbol in Agda, so we mimic it using the operator `_||_`.

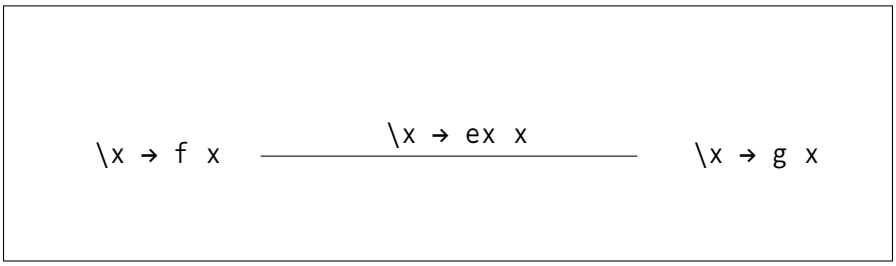
Most of the time, we will omit the congruency notation for the same reason as `sym` to avoid, clutter, but it can occasionally be seen. Visualised, it looks like



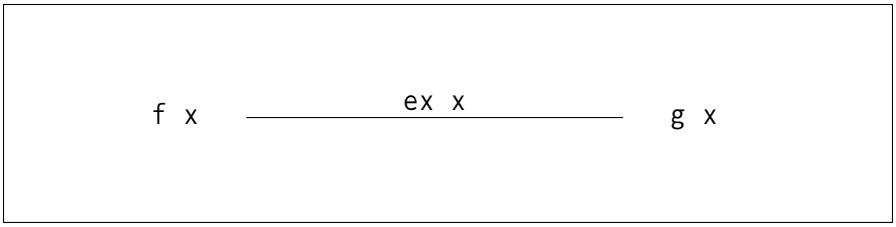
We can also define extensionality as

```
extens : (∀ x → f x ≡ g x) → f ≡ g
extens ex i x = ex x i
```

A path between functions is just a function `p : I → (h : A → B)` with `p i0 = f` and `p i1 = g`, so when we give the respective arguments to `p`, like `p i x`, we now need to find an equation for `f x ≡ g x`. This can be visualised as going from



to



which, to Agda, is the same.

2.4.2 Heterogenous Paths

Another most useful feature of cubical type theory and cubical Agda is that we can define what it means to have an equality between two objects of different types. We do so using *heterogenous paths*.

Definition 2.4.7: **Heterogenous Paths**

Given a path `P : I → Set`, we call a function

```
p : (i : I) → P i
```

a **heterogenous path from** p i0 **to** p i1, also known as a **dependent path**. Notice how the type of the endpoints changes, so `p i0 : P i0` and `p i1 : P i1`. If `P` is a path from type `A` to type `B` and `p i0 = a : A` and `P i1 = b : B`, we use the notation

```
a ≡⟨ P ⟩≡ b
```

to denote the type of homogenous paths between `a` and `b` via the type-equality `P`.

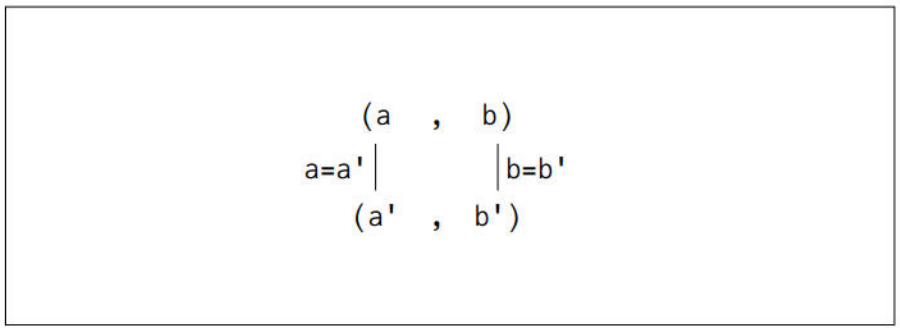
Heterogenous paths are particularly useful for proving properties for sigma types. Given an element of a sigma type `(a , b) : Σ A B`, exchanging the `a` via an equation for something else would change the type of `b`, making it far from obvious to Agda how to transform the `b` in response. Using the explicit interval parameter `i` from the path on equations in cubical Agda, we can apply several equations at the same time while even changing the type of elements along the way. A general equation between elements of a sigma type now looks as follows:

```
SigEq : (a=a' : a ≡ a') → b ≡⟨ (\i → B (a=a' i)) ⟩≡ b' → (a , b) ≡ (a' , b')
SigEq a=a' b=b' i = (a=a' i , b=b' i)
```

Which, expressed with the syntax for congruence, looks like

```
SigEq : (a=a' : a ≡ a') → b ≡⟨ a=a' || B ⟩≡ b' → (a , b) ≡ (a' , b')
SigEq a=a' b=b' i = (a=a' i , b=b' i)
```

Because this notation is not particularly readable, we will often visualise the simultaneous application as a graphic:



2.4.3 Transports and Coercions

In cubical Agda, when two types `A` and `B` are equal, we can automatically change the representation of an object `a : A` into its representation in `B`. This is done using the *transport* primitive.

Definition 2.4.8: **Transport**

Given a path between types `P : I → Set`, we can get the representation of every object `a : P i0` in the type `P i1` using the function

```
transp : (P : I → Set) → (i : I) → P i0 → P i1
```

Unless we know the concrete path for the transport, the transport does not reduce to a concrete object. This has to be kept in mind when proving things on transported objects.

Most of the time, we do not care for a general transport. We define a concrete version, called a *coercion* as

Definition 2.4.9: **Coercion**

Given an equality `A=B : A ≡ B` between two types `A` and `B`, we define a **coercion** as

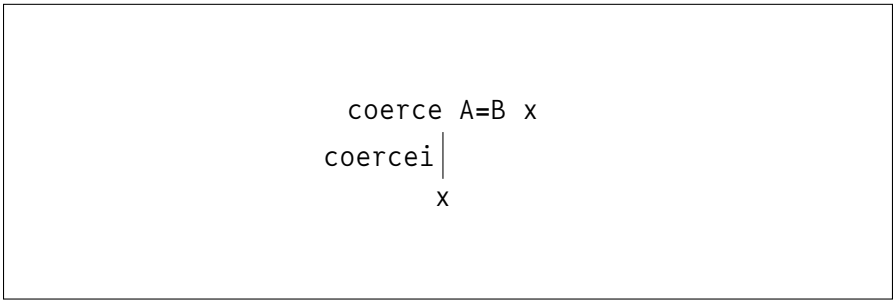
```
coerce : (A ≡ B) → A → B
coerce A=B a = transport (\j → A=B j) i0 a
```

Side Note 2.4:10

It might be a bit unintuitive at first, but in a transport the element at **`i0`** is defined to be the target element and the element at **`i1`** is the original one. So a transport can be thought of as a function taking an element from a different representation and transforming it back into the original representation.

Coercions come in particularly handy when we know two types to be equal before Agda does. In this case we can place explicit coercion to force an elements type to be a different, equal type. As we intuitively usually know which types are equal, simplifications of the presented proofs will therefore omit explicit coercions, but it should be kept in mind that to automatically check the proofs, coercions need to be explicitly taken care of. This also makes proofs significantly harder and seemingly simple statements might take a lot of lines to actually prove. In return, however, we can prove much more general statements can be proven this way.

When dealing with coercions in a proof, there is a special family of paths called *fillers* that describe how a coercion gets reduced when going along a path. We will not go into detail how they work. To keep things simple, we will denote those reductions as



We will now briefly have a look at why creating proofs by manually applying equations makes proving harder than especially mathematicians are used to. It should also be stated though that this added complexity is a feature, not a bug. It makes the statements even more general than most mathematicians would think possible (or necessary). It might also make mathematicians not appreciate the presented proofs as much as they should be, but that is a problem only proper education can change.

2.4.4 Univalence

Univalence is the principle that from an isomorphism between types it should follow that these types are also equal [92]. The application for proof theory is obvious. We cannot distinguish two isomorphic types purely by the properties that hold on them anyway, so we might as well have them exchangeable in order not to do all proofs twice. For computer science, the application is that there are representations of data that are great for proofs and others that are great for computation. Often, in practice, we ditch provable correctness for performance, which comes with the obvious dangers. With univalence, this gap can be automatically filled. For example: We can prove the properties of natural numbers on their unary representation, then prove that this representation is isomorphic to the binary representation, and then run the program on the binary representation. We gain the improved performance while still knowing that the program runs correctly.

Of course allowing for univalence means that there suddenly is an important semantics in an equation. Especially when we have equations between elements of different types, the representation of the data may have to change when coercing the data from one type to another. In fact, the representation can even change when coercing within the same type. Take the Booleans, for example. There are exactly two isomorphisms of the Booleans into themselves. Suddenly, when coercing a Boolean into a Boolean, we do not know what the representation of the Boolean on the other end is like, because the coercion could either be the `id` or the `not`$\mathbb{B}$ function. This means that when proving an equivalence, it suddenly becomes important which exact equation is put where. This is called the *proof relevance of equations*. The opposite is often called the *uniqueness of identity proofs* or *axiom K*, and it is incompatible with univalence. Usually,

when doing classical mathematical proofs, mathematicians assume the uniqueness of identity proofs, which makes proving the equivalence between two things of which we know the type to be equal easy. Under univalence, we have to watch out that we do not accidentally change the representation of data to something we cannot compute with anymore. This makes the proofs harder, but we get one significant upside: Even the most abstract proved statements result in computable functions. In other words, the presented proofs are actually useful in a real world application.

All proofs in this thesis are compatible with univalence unless stated otherwise.

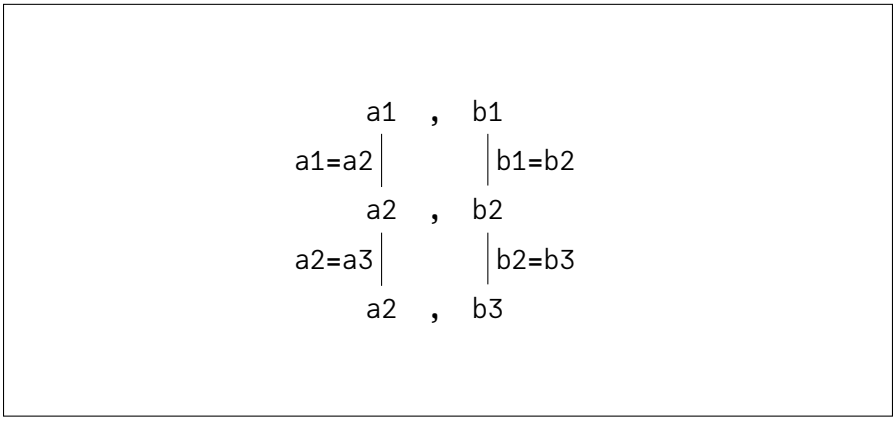
We will not go into detail how univalence is implemented in Agda, but an explanation can be found at [11]. However we import a function from the cubical standard library [4] that implements univalence that we call:

```
Iso→≡ : Isomorphism A B → A ≡ B
```

In this thesis, this function will only be used implicitly for proofs by univalence.

2.4.5 Transitivity and Composition

The engaged reader will have noticed by now that we cannot prove transitivity yet. Transitivity in cubical Agda is done using the `hcomp` and `comp` primitives, but explaining them here (again, not a thesis on cubical type theory) would not help the understanding of the presented proofs. We encourage curious readers to consult the explanation given in [11]. However: together with univalence, there is one rule of thumb that helps with dealing with proofs involving transitivity. As every application of a transitivity has an intermediate representation of the data, it means that if the transformation of one type depends on the transformation of its parameter, we also have to show what happens in that intermediate representation of the parameter. This can be visualised by the following proof on an element of a sigma type that involves two consecutive proof steps:



In the proof from `a1` to `a3`, we went through the intermediate representation of `a2`. Of course, for the computer to know how to transform `b1` to `b3` via the transformation applied to `a1`, we also need to know what happens to `b1` at the intermediate representation of `a2`.

2.4.5.1 Higher Inductive Types

We here only mention that this is possible, but in cubical Agda, we can add constructors to a type that define that two constructors are equal. This occasionally makes data type definitions easier. An example are the integers:

```
data ℤ : Set where
    pos : ℕ → ℤ
    neg : ℕ → ℤ
    +-=0 : pos 0 ≡ neg 0
```

Without the equality construtor, we would have two different elements representing `0`, but with the equality constructor we just declare that both of these `-` elements should be equal. When now defining a function, we have to make sure that the function result preserves this equality. We are not going to use these types in this thesis but knowing of their existence helps understanding why on the one hand, cubical Agda adds so much more definitional power while also introducing so many new problems.

2.4.6 Encode Decode Method

One of the problems introduced by cubical is that type constructors cannot automatically be shown to be injective. For example for the `Maybe` type, we

are used to it being true that

```
just x ≡ just y → x ≡ y
```

this still holds true under cubical, but is a little harder to prove. We do this using the so-called *encode-decode-method* [92]. It works roughly the same for all of our relevant data, so we just show it at the example of the `Maybe` type.

The idea is that we can construct the space of equivalences that hold for a type. As a raw definition, this is called a *cover*, and for the maybe type, it looks like

```
CoverMaybe : Maybe X → Maybe X → Set
CoverMaybe (just x) (just y) = x ≡ y
CoverMaybe nothing nothing   = Unit
CoverMaybe _       _         = Zero
```

This is a type that states which equalities we have to provide in order to show that the constructed values are equal. So when having two `just` values, we still have to prove that their arguments are equal, two nothing values always equal and everything else cannot be shown to be equal. We now have to reverse this: show that if we have the equality between two maybe values, we can show that their cover exists. As an example, if we are given an equation `just x ≡ just y`, the type of the cover of `just x` and `just y` is `x ≡ y`, which is the type we want to deduce.

To turn an equality into a cover, we have to use something called path-induction. This is done using the `J` combinator. We will not explain in detail how it works, just what it does:

```
J : {A : Set i} {x : A}
    (P : ∀ y → x ≡ y → Set i)
    (d : P x refl) {y : A}
    (p : x ≡ y) → P y p
J P d p = coerce (\ i → P (p i) (\ j → p (i /\-path j))) d
```

We are given three main things. The first is an an anchor element `x : A`.

The second is a predicate `P` that filters arbitray values of `y : A` together with an equation to the anchor element. As an intuition, `_P_` turns into a type whose elements we have to create if indeed `y` and `x` are actually equal (this will eventually be our cover, which for our simple types does not even need the equation towards the anchor). The third thing we are provided with is a proof that `P` holds for reflexivity, meaning in the case where `y` can be intensionally be proven equal to `x` by Agda itself. Then, this operator turns an equation between the anchor and an arbitrary object `y : A` into an element of the predicate `P` being the type resulting from the equality.

An intuition for how this works is that through `P` holding at reflexivity, meaning at the anchor element, we can move along the equation `p : x ≡ y` (by coercing `P x refl` along `p` to `P y p`) show that the resulting type `P y p` also has to exist, as we only moved between equal elements. The type `P y p` is the type of things that has to be shown in order for `x` and `y` to be equal, which is the result that we give. So basically we are streching the `P x refl` value of things to be shown at reflexivity out to the type of things needed to be shown for more distant things to be equal.

We can now use this to show that we can always create a maybe-conver from two maybe values. First, we have to show that the cover exists at reflexivity:

```
encodeReflMaybe : CoverMaybe a a
encodeReflMaybe {a = just  x} = refl
encodeReflMaybe {a = nothing} = unit
```

Now, we can create the cover using the `J` combinator

```
encodeMaybe : ∀ {a b : Maybe X} → a ≡ b → CoverMaybe a b
encodeMaybe {a} = J (\b _ → CoverMaybe a b) encodeReflMaybe
```

Now, we get several useful tools for proofs. Not only does it hold that

```
(eq : just x ≡ just y) → encodeMaybe eq ≡ (x ≡ y)
```

giving us the desired injectivity property, we also automatically get the conflicts

```
(eq : just x ≡ nothing) → encodeMaybe eq ≡ Zero
```

so we can root out absurd cases. We could still show that the reverse also holds, meaning that from a cover we can deduce the equality, but we never needed that in the proofs (is more important when working with the higher inductive types from Section 2.4.5.1)

2.5 Category Theory

We have already mentioned in Chapter 1 Section 1.5.2 that using category theory would make all statements in this thesis be super general, however, as this thesis already heavily focuses on type theory, we did not wanted to add another formalism. That being said, a few basics of category theory are still helpful to understand the contents of this thesis.

2.5.1 Useful Notation From Category Theory

In category theory, everything is modeled by so called *categories*, that consists of a collection of objects with morphisms ("arrows") in between them [14], where the morphisms have to be reflexive (there is always a morephism from every object to itself) and transitive, meaning that morphisms can be composed. In the type theoretic setting, the objects can be pictured as types and the morphisms as functions between those types, though that has a slight loss of generality because actually we are just working in the specific category of types and functions between those types. Still, modeling things with objects and arrows can be beneficial to keep formalisms concise. In this thesis, our objects are always types and the morphisms are always functions between those types unless stated otherwise.

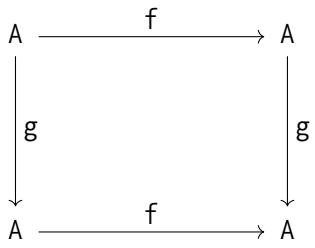
One important tool from category theory is that we can express a set of equations as a visual graph. For example, let's say that we have two functions

```
f : A → A
g : A → A
```

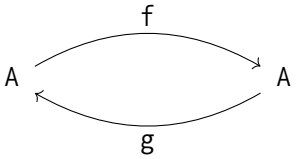
and we want to show that it does not matter in which order we apply them. With equations, we would state this as

```
f ∘ g ≡ g ∘ f
```

As a category theoretic diagram, this could look like



The way that we read out the equation(s) from a diagram is to traverse the arrows in their respective direction and compose the functions that we encounter along the way. If there are two different paths to the same location, then the functions composed to that location also have to be equal. We also say the diagram *commutes* (or is *commutative*). Of course, the above property could just as well be stated using the following diagram:

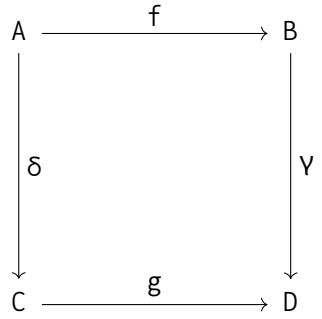


But the square shape is actually quite common in category theory because it expresses the property that if we know a property from one transformation and we can transform that transformation into another transformation, then

the property also holds on the target transformation. For example, if we have the functions given

```
f : A → B
δ : A → C
γ : B → D
g : C → D
```

it could be a setup where we know a property of a function `g : C → D` and we want to know if, let's say an isomorphic property holds on a given transformation `f : A → B`. If we can transform the `f` into the `g`, then we can use results on `g` to prove properties on `f` that are preserved during the transformation. If the behaviour of `f` stays the same after the transformation, any property of `γ ∘ g ∘ δ` also holds for `f`. As a commuting diagram, this looks like



in order to get the equation

```
f ∘ γ ≡ δ ∘ g
```

2.5.2 Type Morphisms as Contexts

Sometimes, we need to have objects of a type wrapped in some kind of context, because values for example may or may not exist, we might have several values of the same type or some error could be thrown when trying to read the element. For example the `Maybe` type

```
data Maybe (A : Set) : Set where
    nothing : Maybe A
    just    : A → Maybe A
```

represents values that may or may not exist. If we ask a value of `Maybe A` for a value of type `A`, we might not get it. Still though, computation defined on `A` should still run for as long as the value of type `A` exists and we do not want to redefine every function just so that it can deal with values wrapped in the `Maybe` value. To solve this problem, there is a theory around type morphisms of the form

```
F : Set → Set
```

Where for a value of type `F A` we say that we have a value of `A` wrapped in `F`, or sometimes that we have a value of `A` in the context `F`.

Side Note 2.5:1

If the universe levels change through the morphism, we call the morphism **impredicative**, like in the morphism

```
F : Set i → Set j
```

If however the universe levels stay the same, like in

```
F : Set i → Set i
```

we call the morphism **predicative**. Such a morphism is also called an **endomorphism**, with the respective names for its instances (like endofunctor or endomonad)

Famous such contexts include but are not limited to (using pseudocode notation for brevity):

```
Maybe : Set → Set
```

for values that may or may not exist

```
_⊎ E : Set → Set
```

for values that may also be an error `E`

```
List : Set → Set
```

when possibly having zero or more values of the same type

```
State S : Set → Set
```

for values that change based on a stateful computation on `S`.

We will quickly go over the basics of the structure that generalises computation within all the aforementioned contexts (and beyond). The implementation

of the general structures for the above examples will not be shown, but they are well explained in [67]. We note that in general category theory, there are concepts of the same name that are way more general, however in the context of type theory, we define the concepts within the category of types and functions.

2.5.3 Functors

We start with so-called functors. Functors are type endomorphisms that can preserve the behavior of functions applied to the original type.

Definition 2.5:2: **Functor**

Given a type endomorphism

```
F : Set → Set
```

together with a function

```
fmap : (A → B) → F A → F B
```

(also called **_<$>_** as an infix notation) then we call this endomorphism a **functor** iff the following laws hold:

```
fmap-id   : fmap id ≡ id
fmap-comp : ∀ {f g} → fmap (f ∘ g) ≡ (fmap f) ∘ (fmap g)
```

functors are useful for applying a function to a value inside a context using `fmap` (or, equivalently, **_<$>_**)

2.5.4 Applicatives

With functors we are always applying a function defined outside of the context to the context. However, we might also have a function bound inside a context that we want to apply to another value inside the same context. We especially

need this to apply a function taking several arguments to several values bound in a context. To do that, we define an applicative [71]

Definition 2.5:3: **Applicative**

Given a type endomorphism `F : Set → Set` and the two operations

```
pure  : A → F A
_<*>_ : F (A → B) → F A → F B
```

then we call it an **applicative** iff the following laws hold

```
identity     : ∀ {v}     → pure id <*> v ≡ v
composition  : ∀ {u v w} → pure _o_ <*> u <*> v <*> w ≡ u <*> (v <*> w)
homomorphism : ∀ {f x}   → pure f <*> pure x ≡ pure (f x)
interchange  : ∀ {u y}   → u <*> pure y ≡ pure (_$ y) <*> u
```

First we can show that every applicative is also a functor

Lemma 2.5:4

Given an applicative on `F`, then together with the function

```
fmap : (A → B) → F A → F B
fmap f v = pure f <*> v
```

we form a functor on `F`

Proof 2.5:5

We prove that `fmap id ≡ id` by extensionality over `v`

```
fmap id v      =<>
pure id <*> v =< identity >
id v           qed
```

We prove that

```
fmap (f ∘ g) ≡ (fmap f) ∘ (fmap g)
```

also by extensionality over `v` as

```
fmap (f ∘ g) v                        =<>
pure (f ∘ g) <*> v                    =< homomorphism >
pure (f o_) <*> pure g <*> v          =< homomorphism >
pure _o_ <*> pure f <*> pure g <*> v =< composition >
pure f <*> (pure g <*> v)             =<>
((pure f <*>_) ∘ (pure g <*>_)) v     =<>
(fmap f ∘ fmap g) v                   qed
```

As already mentioned, due to currying, applicative contexts can be used to apply several arguments in a context to a function. For example when given the following context:

```
f  : A → B → C
ma : F A
mb : F B
```

Then we can write

```
pure f <*> ma <*> mb : F C
```

to get an element of `F C`. As `f <$> v = fmap f v = pure f <*> v`, we can also write this nicer as

```
f <$> ma <*> mb : F C
```

As this is a common pattern, it has a specific syntax called an idiom bracket [71], where the following two expressions (and of course the generalisation to any `n` arguments) are equal:

```
f <$> ma <*> mb = (| f ma mb |)
```

This makes the function application in a context a bit more readable.

Side Note 2.5:6

When dealing with monads, it holds that `pure = return`, so the following two expressions are equal:

```
return x = (| x |)
```

So the idiom brackets hold the extra notation through context at a minimum.

2.5.5 Monads

Probably the most important concept from category theory for type theory is the *monad*. Monads are contexts that model the behaviour of computation by providing an operation to pipe the result of one context via a function for case

distinction into another.

Definition 2.5.7: **Monad**

A type endomorphism `M : Set → Set` together with the two operations

```
return : A → M A
_>>=_  : M A → (A → M B) → M B
```

forms a **monad** iff the following laws hold:

```
left-identity  : ∀ {f x}   → return x >>= f ≡ f x
right-identity : ∀ {m}     → m >>= return ≡ m
associative    : ∀ {m f g} → (m >>= f) >>= g ≡ m >>= (f >=> g)
```

The `_>>=_` operation (called a "bind") expresses that the result of one context is used in a function to create the consecutive context. This structure is used to model computation for context, but much more general. The list monad for example models computation on several values at once, while still looking like a program that could also have been defined for individual contexts.

Side Note 2.5:8

There is a reason that the laws for a monad look much like the laws of a monoid. That is because categorically, a monad is a monoid in the category of endofunctors, so if we would generalise the theory in this thesis to utilise the full extend of category theory, the definition and proofs on a monad become a lot easier.

In order to simplify computation expressed over a monad, functional languages often use the **do-notation**. If we have a computation expressed as

```
ma >>= \a → mf a >>= \b → mg b
```

we can equivalently write this as

```
do
    a ← ma
    b ← mf a
    mg b
```

Or, if the context needs brackets, as

```
do {
    a ← ma ;
    b ← mf a ;
    mg b }
```

Further, there are a bunch of useful operators on monads defined that we will often use:

```
_>>_ : M A → M B → M B
ma >> mb = ma >>= const mb

_=<<_ : (A → M B) → M A → M B
_=<<_ = flip _>>=_

_>=>_ : (A → M B) → (B → M C) → (A → M C)
(amb >=> bmc) = \a → amb a >>= bmc

_<=<_ : (B → M C) → (A → M B) → (A → M C)
_<=<_ = flip _>=>_

_<<*_ : M A → (A → M B) → M A
ma <<* f = ma >>= \a → f a >> return a

_*>>_ : (A → M B) → M A → M A
_*>>_ = flip _<<*_
```

Monads also form an applicative (and therefore also functors)

Lemma 2.5:9

Given a monad instance for `M`, the operations

```
pure : A → M A
pure = return

_<*>_ : M (A → B) → M A → M B
mf <*> ma = do
    f ← mf
    a ← ma
    return (f a)
```

form an applicative.

Proof 2.5:10

Identity: `pure id <*> v ≡ v`

```
pure id <*> v                                        =<>
return id <*> v                                      =<>
(return id >>= \ f → v >>= \ a → return (f a)) =< left-identity >
(v >>= \a → return (id a))                           =<>
(v >>= return ∘ id)                                  =< o-right-id >
v >>= return                                         =< right-identity >
v                                                    qed
```

Composition: `pure _o_ <*> u <*> v <*> w ≡ u <*> (v <*> w)`

```
pure _o_ <*> u <*> v <*> w                                                    =<>
return _o_ <*> u <*> v <*> w                                                 =<>
(return _o_ >>= \f → u >>= \uf → return (f uf)) <*> v <*> w                  =< left-identity >
(u >>= \uf → return (uf o_)) <*> v <*> w                                     =<>
((u >>= \uf → return (uf o_)) >>= \ufo → v >>= \vf → return (ufo vf)) <*> w   =< associative >
(u >>= \uf → return (uf o_) >>= \ufo → v >>= \vf → return (ufo vf)) <*> w     =< left-identity >
(u >>= \uf → v >>= \vf → return (uf ∘ vf)) <*> w                             =<>
((u >>= \uf → v >>= \vf → return (uf ∘ vf)) >>= \ufovf → w >>= \a → return (ufovf a)) =< associative >
(u >>= \uf → (v >>= \vf → return (uf ∘ vf)) >>= \ufovf → w >>= \a → return (ufovf a)) =< associative >
(u >>= \uf → v >>= \vf → return (uf ∘ vf) >>= \ufovf → w >>= \a → return (ufovf a))   =< left-identity >
(u >>= \uf → v >>= \vf → w >>= \a → return ((uf ∘ vf) a))                    =<>
(u >>= \uf → v >>= \vf → w >>= \a → return (uf (vf a)))                      =< left-identity >
(u >>= \uf → v >>= \vf → w >>= \a → return (vf a) >>= \vw → return (uf vw))  =< associative >
(u >>= \uf → v >>= \vf → (w >>= \a → return (vf a)) >>= \vw → return (uf vw)) =< associative >
(u >>= \uf → (v >>= \vf → w >>= \a → return (vf a)) >>= \vw → return (uf vw)) =<>
(u >>= \uf → (v <*> w) >>= \vw → return (uf vw))                             =<>
u <*> (v <*> w)                                                              qed
```

Homomorphism: `pure f <*> pure x ≡ pure (f x)`

```
pure f <*> pure x                                    =<>
(return f >>= \f → return x >>= \x → return (f x)) =< left-identity >
(return x >>= \x → return (f x))                     =< left-identity >
return (f x)                                         =<>
pure (f x)                                           qed
```

Interchange : `u <*> pure y ≡ pure (_$ y) <*> u`

```
u <*> pure y =<>
(u >>= \uf → return y >>= \y → return (uf y))          =< left-identity >
(u >>= \uf → return (uf y))                             =<>
(u >>= \uf → return (uf $ y))                           =< left-identity >
(return (_$ y) >>= \ _$y → u >>= \uf → return (uf $y)) =<>
pure (_$ y) <*> u                                        qed
```

In category theory, monads can also be defined via a functor `M` together with an operation

```
join : M (M A) → M A
```

where it holds that

```
_>>=_  : M A → (A → M B) → M B
ma >>= fmb = join (fmap fmb ma)
```

Giving the definition for `join` is often slightly simpler, but cannot be given in general for the type theoretic monads.

The problem is here that the join only holds when `M` stays on the same universe level, so

```
M : Set i → Set i
```

for some `i`. If `M` is impredicative, meaning that it changes universe levels like

```
M : Set i → Set j
```

then this construction and operation no longer work because the monad can no longer hold an element transformed by itself, so we cannot construct an element `M (M A)`.

2.5.6 Monoids

We occasionally need a structure called a monoid

Definition 2.5:11: Monoid

A type `A : Set` together with an identity element and an operation

```
ε    : A
_<>_ : A → A → A
```

is called a **monoid** iff the following laws hold:

```
left-identity  : ∀ {a}     → ε <> a ≡ a
right-identity : ∀ {a}     → a <> ε ≡ a
associative    : ∀ {a b c} → (a <> b) <> c ≡ a <> (b <> c)
```

2.5.7 On Indexed Contexts

Often, contexts have to change their types depending on the context [88]. This is done using an indexed version of their functor. For example, if we originally had a functor for

```
M : Set → Set
```

Then its indexed version could have the type

```
M : I → I → Set → Set
```

where I is some type that can be used to change the type of context during for example a monadic computation. We then of course have to generalise the monadic operators. The principle of how to do that is always roughly the same, but we will give an example for indexed monads [88]:

Definition 2.5:12: **Indexed Monad**

Given an index type `I` and a morphism

```
M : I → I → Set → Set
```

together with the operations

```
return : A → M i i A
_>>=_  : M i j A → (A → M j k B) → M i k B
```

then this structure is called an **indexed monad** or monad indexed over `I` iff the following laws hold:

```
left-identity  : ∀ {f x}   → return x >>= f ≡ f x
right-identity : ∀ {m}     → m >>= return ≡ m
associative    : ∀ {m f g} → (m >>= f) >>= g ≡ m >>= (f >=> g)
```

As we can clearly see, the indexing does not even change the shape of the laws, it is just a generalisation that we will occasionally need.

2.5.8 The State Monad

We will create our own versions of state monads in this thesis, so we will quickly showcase how they are originally defined.

Definition 2.5:13: **State Context (State Monad)**

The state context (later just called the state monad) is the type transformer

```
State : Set → Set → Set
State S X = S → (S × X)
```

Lemma 2.5:14

The operations

```
return : X → State S X
return x s = (s , x)

_>>=_ : State S X → (X → State S Y) → State S Y
(mx >>= fmy) s = let (s' , x) = mx s in fmy x s'
```

form a monad for the state context (now called the state monad)

Proof 2.5:15

Follows trivially from evaluation.

The state monad has two operations defined on it to modify the state value

```
get : State S S
get s = (s , s)

put : S → State S (Unit {i})
put s _ = (s , unit)
```

where `get` retrieves the value from the current state and `put` overrides the current value in the state with a different one.

2.6 Lattices

In this thesis, we heavily rely on a mathematical structure called *lattices* (or, rather semilattices to be precise). They form a structure of objects that represent things with an information content, where we can merge any two things to get an element of their combined information. We need this to assemble solutions during solving: to have partial solutions be merged into one big solution. In order to parallelise solvers, it is important that this merge is commutative, so that the order in which the solutions are put together does not matter. Also information should not needlessly be copied, so the merge has to be idempotent. This and other useful properties are defined using the mathematical structure of semilattices and lattices. Also, as they represent a sort of information content, lattices and semilattices form pre-orders that describe what it means for an element to increase in size.

2.6.1 Semilattices

Definition 2.6:1: **Semilattice**

A type `L` together with an operation

```
_<>_ : L → L → L
```

is called a **semilattice** iff the following laws hold

```
associative : forall{x y z} → x <> (y <> z) ≡ (x <> y) <> z
commutative : forall{x y}   → x <> y ≡ y <> x
idempotent  : forall{x}     → x <> x ≡ x
```

The semilattice operation `_<>_` represents that the information of two elements is merged, hence the laws. Information does not increase when copied, so the operation is idempotent. The information is merged regardless of order so the operation is commutative and the structure of multiple merges is also irrelevant, so the operation is associative. The most important property of lattices in our context is that they induce a pre-order.

Definition 2.6:2: **Pre-Order**

A predicate `_P_ : A → A → Set` over the carrier type `A` is called a **pre-order** iff it holds that

```
reflexive  : ∀ {a}     → a P a
transitive : ∀ {a b c} → a P b → b P c → a P c
```

Lemma 2.6:3

Given a semilattice over a type `L`, then the following relation

```
_P_ : L → L → Set
x P y = x <> y ≡ y
```

is a pre-order.

Proof 2.6:4

For reflexivity, `a P a = x <> x ≡ x` which is just idempotence. In the transitive case, we have

```
ab=b : a <> b ≡ b
bc=c : b <> c ≡ c
```

and we prove the statement with

```
a <> c          =< bc=c >
a <> b <> c     =< associative >
(a <> b) <> c   =< ab=b >
b <> c          =< bc=c >
c               qed
```

Side Note 2.6:5

In the literature, semilattices are often described to form even a *partial order*, which have the additional law

```
antisymmetric : ∀ {a b} → a P b → b P a → a ≡ b
```

semilattices always form a partial order, but for the proofs in this thesis, the pre-order is sufficient and fits in well with other structures that only require pre-orders and would be a lot more difficult to define under partial orders.

This pre-order has a few properties that underline the intuition of semilattices to describe objects that can increase / decrease in information content / size (from this point on, we will just say increase when clear from the context whether we model an increase or decrease).

When adding information, the size of an element only increases

Lemma 2.6:6

Directionality of Semilattice Operation Given a semilattice over a type L it holds that

```
directional : x P (x <> y)
```

Proof 2.6:7

```
x <> (x <> y) =< associative >
(x <> x) <> y =< idempotent >
x <> y        qed
```

Semilattices have a product construction, which will be important for this thesis to model the assignment state of a finite set of variables.

Lemma 2.6:8

Given two semilattices, one over the type `L1` and the other over the type `L2`, then there is a semilattice on the type `L1 × L2` using the operation

```
_<>_ : (L1 × L2) → (L1 × L2) → (L1 × L2)
(x1 , y1) <> (x2 , y2) = (x1 <>L1 x2 , y1 <>L2 y2)
```

This construction is called the **semilattice product**.

Proof 2.6:9

Associativity:

```
(x1 , x2) <> (y1 , y2) <> (z1 , z2)          =<>
(x1 <>L1 y1 <>L1 z1 , x2 <>L2 y2 <>L2 z2)    =< associativeL1 >
((x1 <>L1 y1) <>L1 z1 , x2 <>L2 y2 <>L2 z2)  =< associativeL2 >
((x1 <>L1 y1) <>L1 z1 , (x2 <>L2 y2) <>L2 z2) =<>
((x1 , x2) <> (y1 , y2)) <> (z1 , z2)        qed
```

Commutativity:

```
(x1 , x2) <> (y1 , y2)  =<>
(x1 <>L1 y1 , x2 <>L2 y2) =< commutativeL1 >
(y1 <>L1 x1 , x2 <>L2 y2) =< commutativeL2 >
(y1 <>L1 x1 , y2 <>L2 x2) =<>
(y1 , y2) <> (x1 , x2)  qed
```

Idempotence:

```
(x1 , x2) <> (x1 , x2)   =<>
(x1 <>L1 x1 , x2 <>L2 x2) =< idempotentL1 >
(x1 , x2 <>2 x2)          =< idempotentL2 >
(x1 , x2)                 qed
```

2.6.2 Bounded Semilattices

Semilattices can have elements of least (highest) information content, which we will use to model that a solution has either been completely unassigned or there has been a conflict. Semilattices with such an element are called *bounded semilattices*

Definition 2.6:10: **Bounded Semilattice**

A semilattice over the type `L` is called a **bounded semilattice** if there is an object

```
e : L
```

that is an identity element for the semilattice operation `_<>_`, so it holds that

```
identity-left : e <> x ≡ x
```

Using commutativity, we can quickly show that those identity elements are also right identities

Lemma 2.6:11

Given a bounded semilattice, it holds that

```
identity-right : x <> e ≡ x
```

Proof 2.6:12

```
x <> e =< commutative >
e <> x =< identity-left >
x        qed
```

Also, the identity element is unique

Lemma 2.6:13

Given a bounded semilattice over the type `L` with identity element `e : L`, then forall elements `e' : L` that are left identities it holds that `e ≡ e'`

Proof 2.6:14

Given the statement

```
e-id : ∀ x → e' <> x ≡ x
```

we can prove the statement as

```
e'      =< sym identity-right >
e' <> e =< e-id e >
e       qed
```

We now transfer over to using two semilattices over the same type to model both information growth and decline at the same time

The boundedness transfers to the product construction of semilattices

Lemma 2.6:15

Given the semilattice product construction of the two bounded semialttices over the types `L1` and `L2`, then

$$(e_{L1} \ , \ e_{L2})$$

is a neutral element in the semilattice product, also making the semilattice product a bounded semilattice.

Proof 2.6:16

We only have to prove the left identity

$$\begin{array}{l}
(e1 \ , \ e2) <> (x1 \ , \ x2) \quad =<> \\
(e1 <>_{L1} x1 \ , \ e2 <>_{L2} x2) =< \text{identity-left}_{L1} > \\
(x1 \ , \ e2 <>_{L2} x2) \quad =< \text{identity-left}_{L2_} > \\
(x1 \ , \ x2) \quad \text{qed}
\end{array}$$

2.6.3 Join and Meet Semilattices

In order to use two semilattices with orders in the opposite direction we first fix on some renames so that their operation names do not clash.

Definition 2.6:17: (Bounded) Meet Semilattice

A (bounded) meet semilattice is a (bounded) semilattice where the semilattice operation is called `_∧_` (pronounced "meet") and the pre-order is called `_≤_`. If it exists, the neural element is called `top`.

In the literature, the neutral element for bounded meet semilattices is often called `bot`, as it is the smallest object. However, for historical reasons, we use the symbol `top` to mean logical truth; the element that gives no new information whatsoever. The symbol `bot` will be used to denote a conflict.

Definition 2.6:18: (Bounded) Join Semilattice

A (bounded) join semilattice is a (bounded) semilattice where the semilattice operation is called `_∨_` (pronounced "join") and the pre-order is called `_≥_`. If it exists, the neural element is called `bot`.

We can now formulate joint properties on two lattices over the same type.

2.6.4 Lattices

(Bounded) Lattices are structures made up of two semilattices over the same type whose pre-order goes in the opposite direction. Lattices are useful when modeling both information increase and decrease.

Definition 2.6:19: (Bounded) Lattice

Given a type `L` and a (bounded) meet and (bounded) join semilattice over `L`. Then this structure is called a **lattice** iff the following laws hold

```
absorb-/\ : a ∧ (a ∨ b) ≡ a
absorb-\/ : a ∨ (a ∧ b) ≡ a
```

The absorption laws model that both an information decrease can be overridden by a respective increase and an information increase overridden by a

respective decrease. Lattices also have a product construction, by it is never needed in the thesis, so we omit it here.

We now give the construction of the most important lattice of this thesis

2.6.5 The Trivial Lattice

The simplest way to turn any type into a lattice type is with what we call the *trivial lattice*.

Definition 2.6:20: **Trivial Lattice**

```
data _⊤⊥ (A : Set) : Set where
    topTB : A⊤⊥
    botTB : A⊤⊥
    valTB : A → A⊤⊥
```

The trivial lattice data type puts elements of a type `A` into a context where we can either have an element, no information `topTB` or a conflict `botTB`. For as long as we have decidable equality on `A`, this forms a lattice where the merges check whether the two values are equal (if present). If they are not, the meet fails with the `botTB` element and the join with the `topTB` element. This type will be our main building block to turn arbitrary types into lattices.

Lemma 2.6:21

If `A` has decidable equality, then there is a lattice on $A^{\top}_{\bot}$ with the operations

```
_∧_ : X⊤⊥ → X⊤⊥ → X⊤⊥
topTB     ∧ x          = x
botTB     ∧ _          = botTB
x         ∧ topTB      = x
_         ∧ botTB      = botTB
(valTB x) ∧ (valTB y) = if x == y then valTB x else botTB

_∨_ : X⊤⊥ → X⊤⊥ → X⊤⊥
topTB     ∨ _          = topTB
botTB     ∨ x          = x
_         ∨ topTB      = topTB
x         ∨ botTB      = x
(valTB x) ∨ (valTB y) = if x == y then valTB x else topTB
```

Proof 2.6:22

In order to prove this, we need a few small statements whose implementation is a bit technical, so we will not show it, but they have been proven to be correct

```
meetVal : x ≡ y → (valTB x) ∧ (valTB y) ≡ valTB x
meetBot : x ≢ y → (valTB x) ∧ (valTB y) ≡ botTB
joinVal : x ≡ y → (valTB x) ∨ (valTB y) ≡ valTB x
joinTop : x ≢ y → (valTB x) ∨ (valTB y) ≡ topTB
```

We will start to go through the proofs of the bounded semilattices. As both semilattices are analogous, we will only show the proofs for the meet semilattice.
Associativity is proven by case splitting over the three elements `x`, `y` and `z`. All cases trivially follow through evaluation and case splitting on the result of the decidable equality. The only nontrivial case is the case of `x = valTB x`, `y = valTB y` and `z = valTB z`.
We split by the results of the decidable equalities `x == y`, `y == z` and `x == z`. We only go through the cases where the results are consistent with each other.
In the case of

```
x=z : x ≡ z
y=z : y ≡ z
x=y : x ≡ y
```

we have

```
(valTB x) ∧ ((valTB y) ∧ (valTB z)) =< meetVal y=z >
(valTB x) ∧ (valTB y)               =< meetVal x=y >
(valTB x)                           =< meetVal x=z >
(valTB x) ∧ (valTB z)               =< meetVal x=y >
((valTB x) ∧ (valTB y)) ∧ (valTB z) qed
```

In the case of

```
¬x=z : x ≠ z
¬y=z : y ≠ z
 x=y : x ≡ y
```

we have

```
(valTB x) ∧ ((valTB y) ∧ (valTB z)) =< meetBot ¬y=z >
botTB                               =< meetBot ¬x=z >
(valTB x) ∧ (valTB z)               =< meetVal x=y >
((valTB x) ∧ (valTB y)) ∧ (valTB z) qed
```

In the case of

```
 y=z : y ≡ z
¬x=y : x ≠ y
```

we have

```
(valTB x) ∧ ((valTB y) ∧ (valTB z)) =< meetVal  y=z >
(valTB x) ∧ (valTB y)                =< meetBot ¬x=y >
botTB ∧ (valTB z)                    =< meetBot ¬x=y >
((valTB x) ∧ (valTB y)) ∧ (valTB z) qed
```

In the case of

```
¬y=z : y ≠ z
¬x=y : x ≠ y
```

we have

```
(valTB x) ∧ ((valTB y) ∧ (valTB z)) =< meetBot ¬y=z >
(valTB x) ∧ botTB                   =< meetBot ¬x=y >
((valTB x) ∧ (valTB y)) ∧ (valTB z) qed
```

which proves associativity.
In the commutative case as well, the only non trivial case is where `x = valTB x`, `y = valTB y`. Here, we case split by the result of `x == y`.
In the case of

```
x=y : x ≡ y
```

we have

```
(valTB x) ∧ (valTB y) =< meetVal x=y >
valTB x               =< x=y >
valTB y               =< meetVal x=y >
(valTB y) ∧ (valTB x) qed
```

In the case of

```
¬x=y : x ≢ y
```

we have

```
(valTB x) ∧ (valTB y) =< meetBot ¬x=y >
botTB                  =< meetBot ¬x=y >
(valTB y) ∧ (valTB x) qed
```

and finally, the same for the only non trivial case of `x = valTB x` for idempotency, we case split by the result of `x == x`, which can be proven by `meetVal x=x`.
Finally, the absorption laws. As they are analogous, we will only proof one of them. Again, the only non trivial case is `a = valTB a`, `b = valTB b`. We case split by the result of `a == b`.
In the case of

```
a=b : a ≡ b
```

we have

```
(valTB x) ∧ ((valTB x) ∨ (valTB y)) =< joinVal x=y >
(valTB x) ∧ (valTB x)               =< meetVal refl >
valTB x                               qed
```

In the case of

```
¬a=b : a ≢ b
```

we have

```
(valTB x) ∧ ((valTB x) ∨ (valTB y)) =< joinTop ¬x=y >
valTB x                                qed
```

which concludes the proof.

2.7 Solvers

There is an absolute abundance of solving techniques and solving related research [13,15–17,20,60,74,76,77,79], but interestingly enough, the main mechanism for solving has not changed since the introduction of the first SAT-solving procedure, the Davis Putnam Logemann Loveland (DPLL) algorithm [31, 32]. The algorithm has been generalised in the Nelson Oppen framework [60, 74, 76] to allow for more general theories and input languages than SAT, but the general procedure stayed the same. Excluding search heuristics, the only conceptual change since came with the introduction of clause learning [17,33,79] (and maybe a bit of parallelisation through branch and bound [64,65]). There is a good reason for why the concept never fundamentally changed which we will start to explain in Chapter 5, but for now, we will just explain the basic principles of DPLL and clause learning. The results on [33] also give a good generalised explanation of why the procedures work, which also gives the general fundament on which this thesis is built upon.

2.7.1 The DPLL Procedure

We start by defining the problem input to the DPLL algorithm. In SAT solving, we are given a boolean formula where all variables are existentially quantified. We search for an assignment to these variables such that the formula evaluates to `true` or until we have the information that there is no such assignment.

The DPLL procedure also does not run on just any boolean formula, but specifically on boolean formulas in the conjunctive normal form (CNF). This means that the boolean formulas are conjunctions of disjunctions of variables are are either plain or negated. This normal form can be created at least satisfiability preserving in polynomial time using the tseitin transformation [91]. We start by defining the data type for the problem input before we progress.

We start with the definition of an atom

Definition 2.7:1: **Atom**

An **atom** of a data type `A` is the data type

```
data Atom (A : Set) : Set where
    pos : A → Atom A
    neg : A → Atom A
```

An atom represents a variable that is either plain (positive) or negated (negative).

A clause is defined to be a list of atoms

```
Clause : Set i → Set i
Clause A = List (Atom A)
```

and represents a finite disjunction.

A formula is a list of clauses

```
Formula : Set i → Set i
Formula A = List (Clause A)
```

and represents a conjunction of clauses.

We now want to define an evaluation function. To do that, we have to define what the variables actually are. For now, let us say that variables are just natural numbers.

In order to get the assignment of a variable and be able to change the assignment, we define that the type of an assignment is a list of possibly assigned values

```
Asm : Set i → Set i
Asm A = List (A⊤⊥)
```

The assignment of a specific values is the assignment of the index within the `Asm` list. We define two operations to read the value of a variable and assign them a value

```
read : ℕ → Asm A → A⊤⊥
read n      []         = topTB
read 0      (x :: asm) = x
read (1+ n) (x :: asm) = read n asm
```

when we read a value from an assignment, we treat variables as indices and read the respecitve value from the list. If the value has not been assigned anything yet, we return that the variable was unassigned.

The write case is similar, only that we lengthen the list if necessary

```
write : ℕ → A⊤⊥ → Asm A → Asm A
write 0      a []         = a :: []
write 0      a (x :: asm) = a ∧ x :: asm
write (1+ n) a []         = topTB :: write n a []
write (1+ n) a (x :: asm) = x :: write n a asm
```

The interesting behaviour occurs when we override a variable. We want the behaviour that when the variable is unassigned it is now assigned and if it is assigned we check whether the assignment is the same, otherwise the variable is assigned to have conflicting assignments. This is exactly the behaviour of the meet for the trivial lattice. Therefore, we use the meet operation for the trivial lattice, assuming decidable equality on `A`.

2.7.2 Unit Propagation

The first phase of the DPLL algorithm is the so-called *unit propagation*. Here, we assign all variables that are the only unassigned variables in a clause where all other variables evaluated to `false`. All of these assignments have to be made in any satisfying assignment of the formula, because it would otherwise evaluate to `false`.

To make this computational step precise, we start by defining one step of the unit propagation. To do that, we first check whether a clause has already been satisfied in a current assignment

```
readAtom : Atom ℕ → Asm 𝔹 → 𝔹⊤⊥
readAtom (pos x) asm = (read x asm)
readAtom (neg x) asm = not𝔹 <$> (read x asm)

isPositiveAtom? : Asm 𝔹 → Atom ℕ → 𝔹
isPositiveAtom? asm x with readAtom x asm
... | valTB true = true
... | _          = false

isSatisfied : Asm 𝔹 → Clause ℕ → 𝔹
isSatisfied asm = any (isPositiveAtom? asm)
```

We do this by checking whether any atom has been assigned positively. To check that per atom, we read the assignment of the atom from the current assignment according to its polarity. If it is positively assigned, we return `true`, otherwise `false`.

Next we are going to check which atoms in a clause are unassigned

```
isUnassigned? : A⊤⊥ → 𝔹
isUnassigned? topTB = true
isUnassigned? _     = false

atomUnassigned? : Asm 𝔹 → Atom ℕ → 𝔹
atomUnassigned? asm x = isUnassigned? (readAtom x asm)

unassigned : Asm 𝔹 → Clause ℕ → Clause ℕ
unassigned asm = filter (atomUnassigned? asm)
```

Per clause, we now check whether the clause has already been satisfied. If not, we check whether it has exactly one unassigned atom in it. If that is the case, we positively assign the atom.

```
assignAtom : Atom ℕ → Asm 𝔹 → Asm 𝔹
assignAtom (pos x) asm = write x (valTB true) asm
assignAtom (neg x) asm = write x (valTB false) asm

AssignUnit : Asm 𝔹 → Clause ℕ → Asm 𝔹
AssignUnit asm cls with isSatisfied asm cls
... | true  = asm
... | false with unassigned asm cls
... | (x :: []) = assignAtom x asm
... | _         = asm
```

Per one step of unit propagation, we keep on adding assignments to the assignment for each clause

```
UnitPropStep : Asm 𝔹 → Formula ℕ → Asm 𝔹
UnitPropStep asm []       = asm
UnitPropStep asm (c :: cs) = AssignUnit (UnitPropStep asm cs) c
```

We repeat this step until no more new assignments can be made. This does not mean that all variables have been assigned, it just means that there are no more unit clauses.

Side Note 2.7:2

We are aware that the above implementation does not check for conflicts, which occur if a fully assigned clause has no positively assigned atoms. As this is just a demonstration of the algorithm, we leave adding the check for a conflict as an exercise to the enthusiastic reader.

The important takeaway message is: During an assignment, because we have used the meet operation, we only ever increase the information content of the assignment. We never eventually unassign a variable. That means that the unit propagation from SAT assigns values we can (quickly) deduce to have to be true and adds them to the assignment, until the assignment no longer grows. This is the main idea for lattice based generalisations of solving like in [33].

We omit the next step in the DPLL algorithm that assigns variables that only ever occur in one polarity because it does not work for general solving [60].

2.7.3 Branching and Clause Learning

The next step of the algorithm is what we call branching (originally called the split-rule). In this step, we are choosing a variable and test out both of its possible assignments. In our above program, we can do so by creating two (mutually exclusive) assignments that we would, independently at first, continue the search on.

```
splitAsmAt : ℕ → Asm 𝔹 → List (Asm 𝔹)
splitAsmAt n asm = write n (valTB true) asm :: write n (valTB false) asm :: []
```

We take an assignment and create two assignments where the given index is assigned `true` and `false` respectively. We can already see how a brute force general solver can be created by using the list monad to eventually split on every variable like

```
allSplitsUntil : ℕ → Asm 𝔹 → List (Asm 𝔹)
allSplitsUntil 0      asm = splitAsmAt 0 asm
allSplitsUntil (1+ n) asm = splitAsmAt (1+ n) asm >>= allSplitsUntil n
```

but of course in practice, we would apply the unit propagation before each split and also only split on assignments that have not already been made. Heuristics may be applied to choose a better order in which to split the variables. The interesting technique however that later greatly improved SAT-solving was the concept of clause learning.

The main idea of clause learning is that we can add extra clauses that increase the speed of the solving process by enhancing the unit propagation. One expensive part of solving is to find a proof for a conflict, meaning that there is a subset of variables that no matter how we assign them, we will never get a satisfying assignment. The worst case complexity on this problem for any known solver is always exponential runtime [12]. But what would be even worse is if we had to do this expensive proof twice. This would happen if we branch on a variable `a = true` and then find that variables `b`, `c` and `d` always cause a conflict on every assignment. We then try the different assignment of `a = false` and check if we now might have a solution. However, if the assignment of `a` has nothing to do with the conflict, the variables `b`, `c` and `d` will continue to not have any assignment and we would have to do the expensive proof of their unsatisfiability all over again. That is, unless we managed to already learn from the `a = true` case that `a` has nothing to do with the conflict of `b`, `c` and `d`, in which we would not even need to backtrack `a`. But what happens if `a` does have something to do with the conflict? For example that it occurs as soon

as `a = true`? We need to note that when we backtrack to an even earlier variable, let's say `a0`, we forget all assignments that we made since the branch of `a0`. So we would no longer know that `a = false`, which has to be the case for the conflict not happening. But we can hit that same sub-assignment again because we are again branching for every variable unassigned after unit propagation. And this is where clause learning becomes active. With clause learning we try to find out which variables are responsible for a conflict and add their assignment as a clause to the formula so that a different branch that might again hit the same assignment that caused the conflict can now be prevented from the expensive branching and have unit propagation already force a different subassignment. Basically, clause learning is caching the solver call on an intermediate solving state, so that once we hit that same state again, we no longer branch but try something else. Of course, in order to not memorise all of the solving state (that would never be hit again), we only remember the parts that created the conflict and the nice property of SAT is that we can encode this recursive-call-caching as a clause in the formula so that we do not need any additional meta theory and can continue to use unit propagation and branching as we did before.

Because the clauses we learn are those that prevent conflicts (and not those that would for example just speed up the regular unit propagation), this technique is called the *conflict driven clause learning* (CDCL).

The main difficulty of clause learning is to find the right clauses. Why that is the case is best explained when looking at the solving state as a lattice, as has been done in [33]. If we describe the solving procedure as a function that increases the assignment (turning it into `bot` on conflict), then the clause we want to learn is basically a smallest possible input to our solving procedure that still causes the conflict. The problem here is: That smallest input is not unique.

Technically, we could ask our solving function in general for any smallest input (being a partial assignment) that would always make it fail, but without any additional context, finding this input is even harder than finding just a single conflict. Therefore, we only search for conflicts that have already happened during the search. But even if we search for the smallest subassignment that is still smaller than the current assignment, this is still potentially too expensive because there could have been several smaller subassignments that cause the solver to fail. What is done here in practice is to restrict even further to only give subassignments that have actually contributed to the assignments in the unit propagation. We track which variable caused the assignment of which other variable using an implication graph and then only give the first reason for an assignment that came up during unit propagation, forgetting about the other reasons that other clauses may have given. Even [33] just states that *a* smallest reason has to be found (meaning that making the subassignment smaller would not cause the conflict anymore), but state that it is up to the aglorithm to decide which smallest one. In Chapter 5 Section 5.8.2, we make this idea slightly more formally precise.

Still, the clauses are not unique. If a conflict has been caused by a subassignment, than also that subassignment has been caused by another subassignment,

and learning to avoid either one of these subassignments is valid. At this point, choosing the subassignment to create the clause from is left to heuristics [20]. What is important is how we find the smaller subassignments from the state transition function of the solver. In SAT-this is done using an implication graph. In our lattice based formalism, we have put a concrete construction to future work, but already our paper [38] worked on the first principles of how the variables we created for solving can be used to find smallest reasons for an assignment or conflict.

Chapter 3

Monadic Solving with Variables

Disclaimer

The code for this chapter was, unless stated otherwise, written fully universe polymorphic. We will, to avoid boilerplate, not write the universe levels unless necessary and just mark impredicative definitions as if we were of `Set0`, meaning that a higher level is marked as `Set1` and so on.

3.1 Introduction and Overview

In this chapter we showcase the usefulness of variables when it comes to solving and develop a formalism to deal with variables in a monadic context. We do this by presenting some basic monadic search techniques and improve their performance by representing the data using variables. We extend this example to the point that we automatically create a solver from an evaluation function that utilises our performance increasing techniques, showcasing how the existence of variables as an underlying technique can be hidden form a user.

Based on this initial example, we develop a theory of what a *monadic variable* is. Monadic variables are monadic actions that behave like variables, in the sense that they only carry exactly one value that it can pass to subsequent monadic computations (while, of course, being able to hold several different values for computations for each branch in a search). We present a new formalism to create such variables in a constrained version of a state monad with our concept of *lattice bi-threshold variables* that is based on the ideas of [61, 62, 75].

The results summarise as follows:

- Example parts (Section 3.2 until 3.6.1, with one final example in Section 3.8.3.3)

  - Experimentally showing how naive List-monad based solving is inferior to a solving technique where the result of a computation can be cached to prevent branching.
  - Showcasing how this caching can be used together with recursive data types using what we call *explicit lazyness* and *held-in-stasis data*.
  - Creation of a SAT-Solver from just the algebra of the evaluation function of Boolean formulas that employs our techniques for reading and writing from variables

- Monadic Variables (Section 3.8 and 3.9)
  - Formalising so-called *lattice bi-threshold variables*, which represent variables reading from and writing to a type with a semilattice instance, inspired by [61,62,75] but being further enhanced for the use in solving.
  - Creating a constrained version of a state monad transformer, called a *constrained state monad transformer*, that holds a state with a semilattice instance where the state only ever increases according to the semilattice.
  - Creating the abstract concept of a *Monadic Variable*, which describe monadic actions that behave like logical variables.
  - Show that the lattice bi-threshold variables induce monadic variables in the constrained state monad transformer
  - Generalising the concept to allow the constrained state monad transformer to an indexed version to even create *new variables* during computation

3.2 Basic SAT-Solving using Monads

Code-Tag: `MonadicSolvingChapterPartI`

In this Section we give a concrete example of a naïve implementation of a SAT solver in the list monad as an introduction of how to do solving monadically. We go through a brute-force implementation of a SAT solver, showing how monads make its implementation incredibly simple.

In functional programming, whenever we want to add effects into a computation like state, errors or solving, we wrap the values into a *context*. A context in this thesis is a morphism

```
M : Set → Set
```

so wrapping a type `A` into the context `M` yields the type `M A`. Most useful contexts abide some categorical laws, turning them into functors, applicatives or monads as described in Chapter 2 Section 2.5. Depending on the context, the normal computation of functions (in the identity context) is enhanced by some effects. In this chapter, we will work with monadic effects useful for solving.

The most basic monad that can be used for solving is the list monad. Its bind operation has the type

```
_>>=_ : List A → (A → List B) → List B
a >>= f = concatMap f a
```

This means that every value in the first list is applied to a function that creates new values in a list, ultimately creating the result of every computation from any of `lsta`'s input values. Because every possible computation is created, this monad can be used for 'nondeterministic' computation.

Side Note 3.2:1

'Nondeterministic' is in quotes here because of course the computation is deterministic, just its specification looks like the one of a nondeterministic computation.

A simple example is a SAT-Solver. To create a SAT-Solver in the list monad, we start by creating a list of all possible Booleans:

```
newB : List 𝔹
newB = true :: false :: []
```

We will now create a simple Boolean formula to solve. It states that out of three variables, only exactly one should be true:

```
onlyOne₃ :  𝔹₃→ 𝔹
onlyOne₃ x y z =
        (x implies (not y and not z))
    and (y implies (not x and not z))
    and (z implies (not x and not y))
    and (x or y or z)
```

Side Note 3.2:2

We use $A_n \to B$ as shorthand notation for $A \to \ldots \to A \to B$ with n times A before the B. We also use A^n as shorthand for the n-tuple $A \times \ldots \times A$, where $A^0 = \mathsf{Unit}$ and $A^1 = A$. Note that $A_n \to B$ is not the same as $A^n \to B$ (one takes a tuple the other several arguments).

We can now create the list of all solutions to $\mathsf{onlyOne}_3$ by writing

```
solveOnlyOne₃ : List (𝔹³)
solveOnlyOne₃ = do
    x ← new𝔹
    y ← new𝔹
    z ← new𝔹
    unless (onlyOne₃ x y z) fail
    return (x , y , z)
```

The list monad here follows a generate-and-test approach. We first generate all possible solutions and then test them, failing the monad if the $\mathsf{onlyOne}_3$ test failed. In the end, we return the solution, which we here have to do explicitly because a monad always returns the value of the last computation and `unless` only returns the unit value.

Side Note 3.2:3

`unless` executes a given monadic action only if the given Boolean is `false`. In our case, unless $\mathsf{onlyOne}_3$ is actually satisfied, we use

`fail` to not return any result on the current computation. For lists, `fail = []`, because if `[]` is used as a monadic action, that means that no value has been produced.

If we run $\texttt{solveOnlyOne}_3$, we correctly get

```
(true  , false , false) ::
(false , true  , false) ::
(false , false , true) :: []
```

We can now do general SAT-Solving, getting a Boolean function that takes at least one input as an input and returning all models, meaning all inputs to the Boolean function that make it evaluate to `true`.

```
SAT-Solve : (𝔹ₙ→ 𝔹) → List 𝔹ⁿ
SAT-Solve {n = 0} f = do
    unless f fail
    return unit
SAT-Solve {n = 1} f = do
    x ← new𝔹
    unless (f x) fail
    return x
SAT-Solve {n = 1+ 1+ n} f = do
    x ← new𝔹
    ⟪ ⟪ x ⟫ , SAT-Solve (f x) ⟫
```

Where we are given a n-ary Boolean function and return the list of all of its models, where `unit` is considered a model for the Boolean constant `true` (a 0-ary Boolean function returning `true`). For every `n`, we create a new Boolean variable with any value and feed it into the given Boolean function. We then recursively repeat until the function needs no more values.

Side Note 3.2:4

Due to our definition of n-ary tuples, we always have to deal with the `0` and `1` case separately.

Here, a few examples of calls to this SAT-Solver:

```
SAT-Solve onlyOne₃
```

has models and therefore yields:

```
(true  , false , false) ::
(false , true  , false) ::
(false , false , true ) :: []
```

whereas an unsatisfiable SAT-formula:

```
SAT-Solve (\ a → a and not a)
```

yields no model at all

```
[]
```

Of course, this SAT-Solver is not particularly fast, as it uses a simple depth-first-search. Throughout this thesis, we will develop better monads that either reimplement existing search techniques or come up with entirely new ones.

3.3 Type Classes for Monadic Solvers

As we do not want to be tied to the list monad, we will use generalised type classes (interfaces) for monads that are often used in solving. We will not be too precise about their laws yet but just want to be on the same page about which names we are using. The most common monadic interface used for solving, employed in [55], is the monad plus interface, defined as

```
record MonadPlus (M : Set → Set) : Set1 where
    field
        mzero : M A
        mplus : M A → M A → M A
        monad : Monad M
```

This monad interface should model that there is a disjunction of some kind given through `mplus` and a sort of "empty" solution set `mzero`. It is not universally agreed upon which laws monad plus should obey, but `mzero` and `mplus` should form a monoid. This class is closely related to the `Alternative` type class (also called a "monad alternative"), defined as

```
record Alternative (M : Set → Set) : Set1 where
    field
        empty : M A
        <|> : M A → M A → M A
        applicative : Applicative M
```

Where, in our case, `empty = mzero` and `mplus= _<|>_` are also forming a monoid, so we will use those names interchangeably. We will make the concrete laws used precise as soon as we prove anything about the behaviour of the monads. On the alternative type class, there is a convenience function for "failing" a computation based on a Boolean value:

```
guard : 𝔹 → M Unit
guard false = empty
guard true = ⦇ unit ⦈
```

This function returns a failure if the conditions are not met.

Side Note 3.3:1

`guard` is not to be confused with the similar monadic functions

```
when : 𝔹 → M Unit → M Unit
when true m = m
when false _ = ⟪ unit ⟫

unless : 𝔹 → M Unit → M Unit
unless = when ∘ not𝔹
```

because `guard` actually uses the `empty` element, which might stop computation, while with `when` and `unless`, computation can only fail based on the action `m` that is given.

Further, we will use the "monad fail" type class

```
record MonadFail (M : Set → Set) : Set1 where
    field
        fail : M A
```

for computation that can fail. It is also often called `MonadZero` and it obeys the following law:

```
fail >>= m ≡ fail
```

3.4 Naïve Approach to Monadically Solving for Recursive Data Types

Code-Tag: `MonadicSolvingChapterPartI`

Previously, we used SAT-Solving as an example of how to solve for finite data types, but what happens if we solve for recursive ones? No approach claiming generality could exclude recursive data types. Surprisingly, the basic concept stays the same, except that now we have to deal with a few termination issues. In this Section, we use a concrete example to show a naive approach in searching for a sorted list.

In this Section, we always assume an alternative, monad plus and monad fail instance on our contexts.

In the example from Section 3.2, we saw that data types can be solved for by disjoining their constructors, like

```
new𝔹 : M 𝔹
new𝔹 = ⦇ true ⦈ <|> ⦇ false ⦈
```

In fact, we can do this with all finite data types. We show this by writing a generator for the `Fin` type that all finite data types are isomorphic to:

```
data Fin : ℕ → Set where
    fzero : Fin (1+ n)
    f1+_ : Fin n → Fin (1+ n)

newFin : (n : ℕ) → M (Fin (1+ n))
newFin 0 = ⦇ fzero ⦈
newFin (1+ n) = ⦇ fzero ⦈ <|> ⦇ f1+ (newFin n) ⦈
```

We either create the stopping constructor `fzero` or the recursive call (if still possible). We can, of course, not create any elements of type `Fin 0`.

But what do we do now with infinite data types? We can, of course, always give some fuel and basically create a sized type. For the natural numbers, this can look like

```
newNat : (n : ℕ) → M ℕ
newNat 0 = ❨ 0 ❩
newNat (1+ n) = ❨ 0 ❩ <|> ❨ 1+ newNat n ❩
```

This is very similar to creating the finite type because creating the naturals with a fuel parameter yields a natural number up to that fuel.

This idea can be lifted to all data types, which we will at least approach in Chapter 4, but for now, we just look at the list example:

```
newList : (n : ℕ) → M A → M (List A)
newList 0 _ = ❨ [] ❩
newList (1+ n) ma = ❨ [] ❩ <|> ❨ ma :: newList n ma ❩
```

Here, we are given the fuel `n` and a monadic action creating the list contents.

As an example, let us search for a sorted list. For that, we need a Boolean function telling us whether a list is sorted:

```
sorted : List ℕ → 𝔹
sorted []             = true
sorted (x :: [])      = true
sorted (x :: y :: xs) = x ≤ y and sorted (y :: xs)
```

We can now filter our search for sorted functions.

Side Note 3.4:1

If we just want to filter a solution but still return it, we can do so using a monadic operator

```
_<<*_ : M A → (A → M B) → M A
ma <<* f = ma >>= \ a → f a >> return a
```

where filtering now looks like

```
mkSth <<* guard ∘ filterFunc
```

in an alternative context, we can further simplify it to an operator

```
_such-that_ : M A → (A → 𝔹) → M A
ma such-that p = ma <<* guard ∘ p
```

that neatly now gives us only the elements that satisfy our predicate.

We can now search for sorted lists (up until a given length, with naturals up until the length of the list) as

```
sortedLists : (n : ℕ) → M (List ℕ)
sortedLists n = newList n (newNat n) such-that sorted
```

So that for `sortedLists 2`, using the list monad for search, we yield all sorted lists of length up to 2 filled with natural numbers up to 2:

```
[]             ::
(0 :: [])      ::
(0 :: 0 :: []) ::
(0 :: 1 :: []) ::
(0 :: 2 :: []) ::
(1 :: [])      ::
(1 :: 1 :: []) ::
(1 :: 2 :: []) ::
(2 :: [])      ::
(2 :: 2 :: []) :: []
```

We will later also prove the correctness of the output, but before we do that, we address one problem that becomes very clear when using the list monad. Just creating all possible values to filter them afterward is a bad idea. The reason is that most lists are not sorted. This means that if we get unlucky, we might have to create a lot of lists until we finally find a sorted one. We are of course aware that this computational overhead is theoretically exponential, but we will conduct a small experiment to measure the effect our optimizations will have on generating solutions. Instead of failing the list monad on a computation and discarding that failed solution, we will keep a marker instead. We can do so using the maybe monad transformer that turns the result of a monadic computation into a maybe value. Concretely, our signature for searching for sorted lists will then look like:

```
sortedLists : (n : ℕ) → M (Maybe (List ℕ))
```

We will not go into full detail about how to implement the maybe monad transformer wrapping a list, but we will change the `mzero` object to `nothing :: []` (a.k.a. `return nothing`), so that we can see how many solutions where produced during the computation. A result for `sortedLists 2` now looks like

```
just []             ::
just (0 :: [])      ::
just (0 :: 0 :: []) ::
just (0 :: 1 :: []) ::
just (0 :: 2 :: []) ::
just (1 :: [])      ::
nothing             ::
just (1 :: 1 :: []) ::
just (1 :: 2 :: []) ::
just (2 :: [])      ::
nothing             ::
nothing             ::
just (2 :: 2 :: []) :: []
```

where we can see the nodes where an unsorted list was created.
We create two test functions

```
sortedListVisitedNodes : ℕ → ℕ
sortedListVisitedNodes n = length (sortedLists n)

sortedListFailedNodes : ℕ → ℕ
sortedListFailedNodes n = length (filter is-nothing (sortedLists n))
```

That counts how many nodes were visited during the search and how many of those were nodes that did not create a solution. For small numbers, this is of course insignificant:

```
sortedListVisitedNodes 2 = 13
sortedListFailedNodes  2 = 3
-- 23% failure, 10 solution
```

But for larger lists the failure-per-visited ratio becomes bad:

```
sortedListVisitedNodes 4 = 781
sortedListFailedNodes  4 = 655
-- 84% failure, 126 solutions

sortedListVisitedNodes 6 = 137257
sortedListFailedNodes  6 = 135541
-- 99% failure, 1716 solutions
```

The problem is, that we always create entire lists, only to then evaluate whether we even needed them or not. That means if a list is already unsorted for the first two elements, still all remaining elements are generated (which are exponentially many). Already for a list of Booleans with length `10`, not having the first two elements sorted but still generating all completions of the list results in $2^8 = 256$ lists that are generated even though we could already know they will be discarded. In the next section, we will create a solution to that problem.

3.5 Explicit Lazyness

Code-Tag: `MonadicSolvingChapterPartI`

We want to stop creating data as soon as we know the data to be discarded. The way we do this is to only create the data of a type going constructor for constructor, forcing functions as well to read the data constructor by constructor. We use lists as a concrete example:

```
{-# NO_POSITIVITY_CHECK #-}
data ListM (A : Set i) : Set i where
    []M   : ListM A
    _::M_ : M A → M (ListM A) → ListM A
```

Here, we have created a list that does not hold concrete values in its constructor, but monadic actions that create the values. We say that the data type is *monadically held in stasis*. This obviously does not pass the positivity checker, which we will solve in Chapter 4 Section 4.3.4.4 using containers. With this setup, we can lazily read the list. Any value is only created when we actually ask for it. Creating an arbitrary value for this monadically held-in-stasis list means creating any constructor and just passing the instructions of how to create the subsequent values into the constructors. The attentive reader will find an analogy to corecursive data types.

```
newList : (n : ℕ) → M A → M (ListM A)
newList 0 _ = ⟪ []M ⟫
newList (1+ n) ma = ⟪ []M ⟫ <|> ⟪ (ma ::M newList n ma) ⟫
```

We will now rewrite the `sorted` function to work on the monadically held-in-stasis lists. It will be the same implementation as before, only that now we can only inspect the contents of the constructors monadically:

```
{-# TERMINATING #-}
sorted : ListM ℕ → M 𝔹
sorted []M = ⟪ true ⟫
sorted (mx ::M mxs) = do
    xs ← mxs
    case xs of \{
            []M → mx >> ⟪ true ⟫
        ; (my ::M mxs') → do
            x ← mx
            y ← my
            guard (x ≤ y)
            sorted (⟪ y ⟫ ::M mxs')
        }
```

As Agda cannot tell whether the value we get from the monadic context is any smaller than our current input, the termination checker does not accept the program. This can be fixed using either sized types (though we will not in this thesis) or special monadic folds as described Chapter 4 Section 4.3.3. Other than that, the implementation is the same as in the id-context function, just that we are case-splitting and prematurely terminating the computation in the monad.

Side Note 3.5:1

We have to be careful that every monadic action is only read exactly once. We will discuss shortly why later in this section.

What is important here is that the sorted function only creates the list values when they are actually relevant and no longer creates every possible list until the bitter end. For now, let us not extract the list-solutions but just compare

the number of solutions that are visited. We rewrite the test functions as

```
unevalSortedLists : ℕ → M Unit
unevalSortedLists n = newList n (newNat n) >>= sorted >>= guard

sortedListVisitedNodes : ℕ → ℕ
sortedListVisitedNodes n = length (unevalSortedLists n)

sortedListFailedNodes : ℕ → ℕ
sortedListFailedNodes n = length (filter is-nothing (unevalSortedLists n))
```

Where we are not returning the found list (we will explain shortly why), but are only giving a unit when we found a sorted list. The failure-per-visited ratios are now significantly better, and we are visiting far fewer nodes:

```
monadically-held-in-stasis      | naive filtering
--------------------------------------------------------------
sortedListVisitedNodes 2 = 13   | sortedListVisitedNodes 2 = 13
sortedListFailedNodes  2 = 3    | sortedListFailedNodes  2 = 3
-- 23% failure, 10 solutions    | -- 23% failure, 10 solution

sortedListVisitedNodes 4 = 281  | sortedListVisitedNodes 4 = 781
sortedListFailedNodes  4 = 155  | sortedListFailedNodes  4 = 655
-- 55% failure, 126 solutions   | -- 84% failure, 126 solutions

sortedListVisitedNodes 6 = 5545 | sortedListVisitedNodes 6 = 137257
sortedListFailedNodes  6 = 3829 | sortedListFailedNodes  6 = 135541
-- 69% failure, 1716 solutions  | -- 99% failure, 1716 solutions
```

We could further improve on those results by also holding the naturals monadically in stasis, but there is a much more pressing problem with this approach when used on any generic monad. This becomes apparent when we try to extract the list that we had just solved for. Let us write a function to extract a monadically-held-in-stasis list:

```
{-# TERMINATING #-}
toList : ListM X → M (List X)
toList []M = ⟪ [] ⟫
toList (mx ::M mxs) = ⟪ mx :: (toList =<< mxs) ⟫
```

Side Note 3.5:2

This, again, does not terminate because we are not using a sized representation of the list. This can be solved by either giving a maximum list size or using a corecursive data type.

We can now try to extract our sorted list naively using the following function:

```
sortedListsNaive : ℕ → M (List ℕ)
sortedListsNaive n = do
    lst ← newList n (newNat n)
    sorted lst >>= guard
    toList lst
```

But when we run it, not only do we see an absolute myriad of solutions, way more than we had before, but also the solutions contain unsorted lists:

```
just [] ::
just (0 :: []) ::
just (0 :: 0 :: []) ::
just (0 :: 1 :: []) ::
just (0 :: 2 :: []) ::
just (1 :: []) ::
just (1 :: 0 :: []) :: -- < unsorted!
just (1 :: 1 :: []) ::
just (1 :: 2 :: []) ::
...
```

The reason is simple: Evaluating a monadic action twice does not give the same value for every evaluation. Reading the list twice creates a whole new list every time we read it, so the list that we filtered using `sorted` is not the same list that we are returning when reading the list contents. This is, to say the least, undesirable. But going back to just creating the list once and then evaluating it will not give us the speed improvement we had just achieved.

What we would need are (special) monadic actions that, when used twice,

always give the same monadic value, without losing the property that this single value is still part of a search. To cut to the chase: this can be achieved using a form of monadic pointers.

3.6 Caching Monadic Computation

Code-Tag: `MonadicSolvingChapterPartI`

In this Section, we develop an early mechanism to cache the return value of a monadic computation to use it multiple times, at the same time without giving up the explicit laziness when creating a value of a data type. We do so by carrying a state that represents a partially instantiated value, meaning a value that instead of some constructors just holds a placeholder.

We first define the functor representing either created or possibly creatable data (called a *cached value (CV)*):

```
CV : Set i → Set i
CV A = A or M A
```

This is obviously a functor if `M` is a functor. In order to not have to use `left` and `right` constantly, we define the two pattern synonyms (like aliases for functions, but for constructors):

```
pattern val a = left a
pattern prod ma = right ma
```

So a cached value is either a plain value `val a` or a monadic production rule `prod ma` for a value of type `A`.

We recapitulate the monadically held-in-stasis lists:

```
{-# NO_POSITIVITY_CHECK #-}
data ListM (A : Set i) : Set i where
    []M   : ListM A
    _::M_ : M A → M (ListM A) → ListM A
```

If we instantiate the `M` to use the functor `CV` (problem with `CV` containing `M` solved in Sidenote 3.6:8), we get a partial list that has some instantiated constructors and others that only hold their production rule, so

```
{-# NO_POSITIVITY_CHECK #-}
data ListM (A : Set i) : Set i where
    []M   : ListM A
    _::M_ : CV A → CV (ListM A) → ListM A
```

can produce values like

```
val 5 ::M prod (newNat 10) ::M val 6 ::M prod (newList 10 (newNat 10))
```

This represents a partially instantiated list.

Side Note 3.6:1

This is similar to placing holes in Agda, so we could analogously write

```
5 :: {!!} :: 6 :: {!!}
```

if we wanted to solve for a specific list by hand and are stuck mid-implementing it. Everywhere we place a hole to be implemented, our data type stores the respective monadic production rule as `prod mx`

We now want to read from this partially assigned list, but in a way that we do not need to know that the underlying list is a partial one. On the outside, we just want to access elements or constructors from the list as if they were normal values. We do so using functional lenses.

A quick recap: A lens is an object that describes how to read and write a value of type `B` into a value of type `A`. Lenses form a functional version of what a pointer is: a read and write (`lget` and `lput`) function into, for example, a state. They can be defined as

```
record Lens (A B : Set i) : Set i where
    field
        lget : A → B
        lput : B → A → A
```

where it must hold that [36]

```
put-get : lget l (lput l v s) ≡ s
get-put : lput l (lget l s) s ≡ s
```

so, simply put, if we write a value `v` into an object `s` and read it using the same lens we get the value `v` back that we just put into it, and vice versa: if we put something that we just read from the object `s` back into `s` using the same lens, it does not change `s`. This is what you would expect from a pointer: Unless anything else changes the state, you would expect the values you stored in it at a given position to still be there when you return. This is precisely what we need to store the results of our monadic computations.

Side Note 3.6:2

We will use the terms *lens* and *pointer* interchangeably to ease the transfer from the concept of pointers in other languages. The kind of pointers we will use in this thesis are statically typed and have no pointer arithmetic (constant pointers), so we are using their abstract semantics instead of a concrete implementation (as is done in other areas by using addresses in memory). The general term for something of this concept is a *reference*.

We can now create a state monad transformer that, soon, will hold our partially instantiated list. As, obviously, this technique will not only work for lists, we use an arbitrary state type `S`. To formulate what it means to cache monadic computations, we just need a state monad transformer

```
MS : Set i → Set i
MS = StateT S M
```

So that the inner monad `M` that potentially does the searching now contains state computations that cache the results of monadic computations. We note that this monad stays a monad plus and alternative with the functions

```
_<|>_ : MS A → MS A → MS A
(m1 <|> m2) s = m1 s <|> m2 s

fail : MS A
fail s = mzero
```

so we can do search using the outer state monad as well.

For now, we just formulate what it means to read from the state as if all values were present. A pointer from the state to a possibly cached value has the form

```
V : Set i → Set i
V X = Lens S (CV X)
```

Because we can also write back into the lens, we can use the lens to write the value of a monadic computation back into the state as soon as it has been computed. Reading from the lens, either returning the cached value or creating a new value that is then being cached, now looks like:

```
read : V A → MS A
read l = do
    a+ma ← getS (lget l)
    case a+ma of \{
          (val   a) → return a
        ; (prod ma) → do
            a ← ma
            modify (lput l (left a))
            return a
        }
```

We first get the cached value from the state. We then check whether it

already contained the desired value or not. If it did, we return the value. If not, we produce the value using the production rule and write the value back via the lens before returning it. It is important to note that as long as we only use the read-operation and valid lenses, a cached value can only ever turn from a production rule to a concrete value, but never from a value back to a production rule. That means that every production rule is run at most once.

Side Note 3.6:3

It is important to note that the `read` function only every writes back concrete values into the lens, never production rules. This is important because we will soon run into complications with our naive definition.

As our state will eventually be the list we are solving for, it can be useful to have a pointer into the entire state content. Abstractly, this is just the identity lens for lenses into cached values:

```
idL : Lens A (CV A)
lget idL = val
lput idL (val   a) = const a
lput idL (prod ma) = id --hack
```

We here use the property that we do not write back production rules to keep this example simple. To stress that we are using this lens for the partial list state, we create an alias:

```
stateListPtr : V (ListM A)
stateListPtr = idL
```

Side Note 3.6:4

At this point we note, that the lenses into cached values actually compose as follows:

```
_∘ᵥ_ : Lens B (CV C) → Lens A (CV B) → Lens A (CV C)
lget (vc ∘ᵥ vb) s with lget vb s
... | val   a = lget vc a
... | prod ma = prod (ma >>= fromRight return ∘ lget vc)
lput (vc ∘ᵥ vb) cvc = lmodify vb (fmap (lput vc cvc))
```

The way the composition works is that when a value is present, we just pipe it into the other lens, whereas if a production rule `ma` is present, we return an enhanced production rule that pipes the result of `ma` into the next lens. Modification, interestingly enough, stays the same as with normal lenses.

In order to properly read our (partial) list, we need lenses to access the sub-parts of our list. We do so with lenses into the head and tail of a list:

```
headL : Lens (ListM A) (CV A)

lget headL []M = prod fail
lget headL (cva ::M _) = cva
```

If we try to read the head of an empty list, we have a production rule that always fails. If we read the head of a non-empty list, we just return the head (that is still possibly unevaluated).

The case for writing back into the head is more complicated. There are two problems:

1. What happens when we write back a value into a value that has already been cached?
2. What happens when we write back a production rule?

Luckily, in our current example, the second case does not occur because the only time we write back into the state is when we read a value. When reading a value, we only write back concrete values into the state that have not been previously assigned, so we always just override a production rule. To keep this example minimal, we will solve the above problems in more generality in Section 3.8. For now, we define the lenses just enough to be able to write back concrete values and deal with the first problem. We do so using the two helper functions:

```
onValClash : (A → A → A) → CV A → CV A → CV A
onValClash f (val x)   (val y)   = val (f x y)
onValClash f (val x)   (prod _)  = val x
onValClash f (prod _)  (val x)   = val x
onValClash f (prod mx) (prod _) = prod mx -- hack

leftOnValClash : CV A → CV A → CV A
leftOnValClash = onValClash const
```

In the `onValClash` function we keep an existing value, if only one exists. For production rules, we keep the left production rule (which is a hack and in general unsuited for solving as will be elaborated in Section 3.8). We can do this because in our example, all producers are the same. In the case where both values exist we are given a function that can resolve the clash. If we do not have such a function at hand, we can just say that in the event of a clash we just take the left value, as is done in the `leftOnValClash` function.

We can now hack together the writing case for the head-lens as

```
lput headL cva []M = []M    -- < - we should fail here!
lput headL cva (x ::M xs) = leftOnValClash cva x ::M xs
```

If we write back a head into an empty list, we should technically fail (and just won't in this example). In any other case, we write back the new head into the old one, automatically resolving what happens if the new or old head already exist.

It is slightly more interesting to create the lens for the list's tail. Retrieving the tail is easy:

```
tailL : Lens (ListM A) (CV (ListM A))
lget tailL []M = prod fail
lget tailL (_ ::M cvlst) = cvlst
```

writing back into the tail is slightly harder. The reason is that we might write back a list that has a production rule where the list in the state is already initialised. To picture this, assume we want to set the following terms equal:

pseudocode:

```
5 :: 4 :: {!!}
5 :: {!!} :: 6 :: {!!}
```

then this should result in the list

pseudocode:

```
5 :: 4 :: 6 :: {!!}
```

which means that we have to recursively go down the list we want to write back into the state and compare it with the list in the state, to merge them wherever one of them has a value instead of a production rule. Our write function for the tail lens now looks like the following, where we will explain the helper function afterwards:

```
lput tailL cvlst []M = []M -- < -  we should fail here!
lput tailL cvlst (cva ::M cvlst') = cva ::M mergeListMCV cvlst' cvlst
```

We merge two partially assigned lists by recursively comparing the two lists, removing value clashes for the cached values where they occur:

```
mutual
    {-# TERMINATING #-} -- this terminates on inlining
    mergeListMCV : CV (ListM A) → CV (ListM A) → CV (ListM A)
    mergeListMCV = onValClash mergeListM

    mergeListM : (ListM A) → (ListM A) → (ListM A)
    mergeListM []M []M                = []M
    mergeListM (x ::M xs) (y ::M ys) =
        leftOnValClash x y ::M mergeListMCV xs ys
    mergeListM xs ys                  = xs
```

We can now start to write our search. We begin by, again, creating a new, arbitrary list of a given length:

```
newList : (n : ℕ) → MS A → MS (ListM A)
newList zero ma = ⟪ []M ⟫
newList (1+_ n) ma = ⟪ []M ⟫ <|> ⟪ (prod ma ::M prod (newList n ma)) ⟫
```

This list is either the empty list or the production rule for the list head together with the production rule for the rest of the list. This gives us the same lazyness-effect from Section 3.5. We now rewrite the sorted function using lenses into the state. As the function is getting quite long, we compare it with the previous implementations:

```
sorted : List ℕ → 𝔹
sorted [] = true
sorted (x :: []) = true
sorted (x :: y :: xs) = (x ≤ y) and sorted (y :: xs)
```

Figure 3.1: `sorted` function in identity context

```
mutual
    {-# TERMINATING #-}
    sorted : V (ListM ℕ) → MS Unit
    sorted v = read v >>= sorted' v

    sorted' : V (ListM ℕ) → ListM ℕ → MS Unit
    sorted' v []M = ⟪ unit ⟫
    sorted' v (_ ::M _) = do
        lst ← read (tailL ∘ᵥ v)
        case lst of \{
              []M       → ⟪ unit ⟫
            ; (_ ::M _) → do
                a ← read (headL ∘ᵥ v)
                b ← read (headL ∘ᵥ tailL ∘ᵥ v)
                guard (a ≤ b)
                sorted (tailL ∘ᵥ v)
            }
```

Figure 3.2: `sorted` function using lenses

```
{-# TERMINATING #-}
sorted : ListM ℕ → M Unit
-----------------------------------------
-- no initial lens-read here
-----------------------------------------
sorted []M = ⟪ unit ⟫
sorted (mx ::M mxs) = do
    xs ← mxs
    case xs of \{
           []M          → ⟪ unit ⟫ -- mx >>
        ; (my ::M mxs') → do
            x ← mx
            y ← my
            guard (x ≤ y)
            sorted (⟪ y ⟫ ::M mxs')
        }
```

Figure 3.3: `sorted` function using plain explicit lazyness

The actions that we perform are, in principle, identical, with the only ex-

ception that in Figure 3.2 we use lenses for accessing the list.

Side Note 3.6:5

In both cases of the monadically held-in-stasis data types, Agda's termination checker cannot tell wether the function will terminate or not, and for good reason! If we see lenses as pointers, we could, technically, do the recursive call on a value that points to a bigger list than we had in our input, practically creating a loop in the data. This issue can be fixed using the folds from Chapter 4 Section 4.3.3

Side Note 3.6:6

The `-- mx >>` annotation is due to a here irrelevant change made to the original code from Section 3.5 where we had to execute the `mx` operation to get the correct number of solutions (because the result of the action at this point is never read anywhere and we were not showing the list)

We can now restate our search problem. This time, because we are using the lenses to identify when we are accessing the same value as a different function, we can even read the list that we have created by layily filtering it using the `sorted` function.

```
sortedLists : ℕ → MS (List ℕ)
sortedLists n = do
    newList n (newNat n) >>= put
    sorted stateListPtr
    renderList stateListPtr
```

We have to put the new list into the state and then we declare it to be sorted and return the rendered list (that is no longer monadically held in stasis).

Side Note 3.6:7

Rendering a list means that we read the list from the state and create a conventional list from the results as

```
{-# TERMINATING #-}
renderList : V (ListM A) → MS (List A)
renderList v = read v >>= \{
      []M       → ❨ [] ❩
    ; (_ ::M _) → ❨ read (headL ∘ᵥ v) :: renderList (tailL ∘ᵥ v) ❩
    }
```

We now get the same number of solutions:

```
lenses to partial list          | monadically-held-in-stasis      | naive filtering
--------------------------------------------------------------------------------------------------
sortedListVisitedNodes 2 = 13   | sortedListVisitedNodes 2 = 13   | sortedListVisitedNodes 2 = 13
sortedListFailedNodes  2 = 3    | sortedListFailedNodes  2 = 3    | sortedListFailedNodes  2 = 3
-- 23% failure, 10 solutions    | -- 23% failure, 10 solutions    | -- 23% failure, 10 solution

sortedListVisitedNodes 4 = 281  | sortedListVisitedNodes 4 = 281  | sortedListVisitedNodes 4 = 781
sortedListFailedNodes  4 = 155  | sortedListFailedNodes  4 = 155  | sortedListFailedNodes  4 = 655
-- 55% failure, 126 solutions   | -- 55% failure, 126 solutions   | -- 84% failure, 126 solutions

sortedListVisitedNodes 6 = 5545 | sortedListVisitedNodes 6 = 5545 | sortedListVisitedNodes 6 = 137257
sortedListFailedNodes  6 = 3829 | sortedListFailedNodes  6 = 3829 | sortedListFailedNodes  6 = 135541
-- 69% failure, 1716 solutions  | -- 69% failure, 1716 solutions  | -- 99% failure, 1716 solutions
```

and, this time, we can even extract the lists that we filtered:

```
[]             ::
(2 :: [])      ::
(1 :: [])      ::
(0 :: [])      ::
(2 :: 2 :: []) ::
(1 :: 2 :: []) ::
(1 :: 1 :: []) ::
(0 :: 2 :: []) ::
(0 :: 1 :: []) ::
(0 :: 0 :: []) :: []
```

Side Note 3.6:8

It was not shown here, but there is one complication when using a monadically held-in-stasis list as a state when the monad to hold the list in stasis is the state monad itself. We kind of get a set of equations that is not easily resolved:

```
MS = StateT (ListM A) M
CV = A or MS A

{-# NO_POSITIVITY_CHECK #-}
data ListM (A : Set i) : Set i where
    []M   : ListM A
    _::M_ : CV A → CV (ListM A) → ListM A
```

The state definition needs the list which needs the cached value definition which needs the state which needs the list again. This can be resolved using a slightly inlined data type definition:

```
CV' MS = A or MS A

{-# NO_POSITIVITY_CHECK #-}
data ListM (A : Set i) : Set i where
    []M   : ListM A
    _::M_ :
        CV' (StateT (ListM A) M) A →
        CV' (StateT (ListM A) M) (ListM A) →
        ListM A

MS = StateT (ListM A) M
CV = CV' MS
```

This way, the state monad is included in the fixpoint of the ListM data type and the MS monad is well defined.

3.6.1 Summary of Problems from Example

We have just conducted a practical example of how to use lenses to cache monadic computation, such that a monad that potentially branches on values does not do so twice when we mondaically hold values in stasis for explicit laziness. This enables us to only branch at most once and only on the values that are actually relevant to the computation. During the practical experimentation, we encountered a few problems that a generalised approach should solve:

1. How should a write of a monadically-held-in-stasis data type work?
    - Writes should not override existing values (and fail if they attempt to)
    - Writes of production rules should merge the production rules
2. How should this be generalised to arbitrary data types?
3. How to regain termination when recursing over monadically held-in-stasis data types?

We will solve all of those issues with a general concept of how to deal with lens / pointer (soon called variable) access in a functional language to get several exponential solving speedups for general search problems. Before we solve the problems, a small teaser about toward which general concept we are headed.

3.7 VarMonads (Teaser)

In practive, we don't really care whether our access to some state is implemented via pointers or lenses or anything. We really just want there to be some functor

```
V : Set i → Set i
```

where if we are given an element of `V A`, we can always retrieve or write back an element of `A` from or to the state. Reading out an element of `V A` should be like reading from a variable in logics: always returning a value and always returning the same value. Writing to an element of `V A` should be equivalent to assigning a value to a variable, meaning that it is set equal to a constant. Therefore, we call `V` a *variable functor* and $V\ A$ a variable. Now, we just need to ensure that our context allows us to use those variables accordingly. This is what we will call a *VarMonad*, and a, for now informal, definition looks like:

```
record VarMonad (M V : Set → Set) : Set where
    field
        new : M (V A)
        read : V A → M A
        write : V A → A → M Unit
```

So in a VarMonad we can create new variables, read them and assign them a value.

This current definition is idealised, and from practical experiments like those we performed in our paper [38] indicates that this interface has to be generalised to fit certain constraints. In this thesis, we will still leave the development of some necessary axiomatisation for this interface to future work, so we will not fully refine it, however, we still place it here to give an intuition of the trajectory of this research.

3.8 Writing to Assignments

In this section, we give a more precise solution to the problem of what should happen in a lens-write to a state in a solving context. We will use this to write a SAT-Solver outperforming the one from Section 3.2, with additionally giving some improvements to the solving process from Section 3.6. We will additionally give a strong example of how our technique generalises to arbitrary

solvers for equations between computable values by showing that the SAT-Solver can be created using only the algebra that describes the evaluation function of a Boolean formula.

There is one defining feature that differentiates reading from a variable in solving from reading from a pointer or lens in a program: The value we read should not change in between reads. This makes sense if we think of variable assignments as a model. If we put two constraints (programs that filter models) on a model, both constraints should see the same model and not a different one. Otherwise, we would just have two different models, none of which necessarily satisfy both constraints at the same time.

We can therefore call certain monadic actions *variables*, if they abide by the following laws:

Definition 3.8:1: **Monadic Variable**

Given a monad `M`, a type `X` and two monadic actions

```
read : M X
write : X → M Unit
```

Then the structure is called a **monadic variable** iff the following conditions hold:

```
commutative : ∀ {m' : M Y}{f : Y → X → M Z} →
(read >>= \_ → m' >>= \y → read >>= \x2 → f y x2) ≡ (read >>= \x1 → m' >>= \y → read >>= \_ → f y x1)

read-write : ∀ {x : X}{m' : M Y}{f : Y → X → M Z} →
    (write x >> (m' >>= \y → read >>= \x' → f y x')) ≡ (write x >> (m' >>= \y → read >>= \_ → f y x))
```

Code-Tag: `MonadicVariable`

The `commutative` law states that the result of two reads, no matter what happened in between, should not change. This is expressed as stating that no matter which of the two results is chosen, the computation stays the same. The `read-write` law states that whenever we write a value, the read action `m` always returns that value, no matter what happens in between. This is modeled by allowing the subsequent computation to utilise the written value directly.

Side Note 3.8:2

It should be noted that the laws from Definition 3.8:1 suffice to state that the monadic read returns a coherent answer for all successive states (so it is commutative in the broader sense), but it does not state whether any value is returned at all. If the read fails to become a left

absorbing action (meaning that no computation after it is performed), the laws still hold. We will not check for this case, but note that all variables used here are also "alive", meaning that they can return a value that does not make the read fail.

In order to create such logical variables, we often need the property that a monadic action returns the same value for all subsequent computation. This is a property that, if we return something on one state, we should return that value for all subsequent states. So, more generally, we want all subsequent states to be linked by some form of relation. We will quickly explore how that would look like before using an existing concept to solve the problem.

3.8.1 Exploration: Constrained State Monad Transformers

Code-Tag: `CStateT`

Definition 3.8.3: **Constrained State Transformer Context (Constrained State Monad Transformer)**

Let `S` be a state type and

```
_P_ : S → S → Set
```

a binary, infix relation on `S`. Then, together with a type morphism `M : Set → Set` we call the following type morphism

```
CStateT :
    (S : Set) →
    (_P_ : S → S → Set) →
    (M : Set → Set ) →
    Set → Set
CStateT S _P_ M X = (s : S) → M (∃[ s' of S ] (s P s') × X)
```

the **constrained state transformer context** (later: the constrained state monad transformer)

In this context we can only transition to a new state if it relates to the previous one. We need this because the relation `_P_` can be used to express

that if for some lens `l` we get a value `x` at a state `s`, that means we get the same value `x` at all states following `s`. This already hints that the relation `_P_` might have to be transitive. Coincidently, our relation-enhanced state computation definition also needs `_P_` to be reflexive and transitive to be a monad.

We will go through the proof under which conditions this is a monad step by step. First, we define the monadic operations. We will constrain that `M` has to be a monad as well. Further, to avoid clutter, we use the name alias

```
MS X = CStateT S _P_ M X
```

The return for the constrained state monad transformer is quite straight forward:

```
return : X → MS X
return x s = return (s , reflexive ∈ (s P s) , x)
```

Where `∈` is just notation to show which type a value has. It reduces away without effect.

In the `return` operation, we return the given object `x` without changing the state. This means `_P_` has to be reflexive. We can now define the monadic bind using the likewise necessary transitivity of `_P_`:

```
_>>=_ : MS X → (X → MS Y) → MS Y
(mx >>= fmy) s = do
    (s'  , p  , a) ← mx s
    (s'' , p' , b) ← fmy a s'
    return
        ( s''
        ,  (p  ∈ s  P s'  [premise]
            p' ∈ s' P s'' [premise]
            ----------------------------
            transitive p p' ∈ s P s'')
        , b)
```

Where `[premise]` is just notation to show which type an input to a function

or rule has. It evaluates away without effect. The `_∈_` is an line type annotation that also evaluates away without effect. Both operators are defined in Chapter 2 Section 2.1.1.

In the monadic bind, we run the first stateful computation on the state `s`, pipe its return value and successive state into the next computation and return the result, only that we have to apply the transitivity first to assure that the next state `s''` relates to both `s` and `s'`.

We can now show the monadic laws for this monad transformer (given that also `M` is a monad). We will go through this proof step by step to check whether further conditions for `_P_` arise. Not to spoil the lead: we will see that `_P_` needs some weak form of proof irrelevance.

We check the law of left identity first. We are omitting writing the congruence rules to avoid clutter. As a convention, we name the monadic operations of the constrained state monad transformer $\texttt{return}_\texttt{S}$ and $\texttt{\_>>=}_\texttt{S}\texttt{\_}$, whereas we leave the names for the same operations on `M` as is.

Lemma 3.8:4

With the above definitions, it holds that

```
left-identity : forall{x} → return_S x >>=_M f ≡ f x
```

under the assumptions that

```
left-identity-P : ∀ {x} {y} → (p : x P y) → transitive reflexive p ≡ p
```

Proof 3.8:5

Proof by extensionality over the state `s : S`.

```
(returnS x >>=ₛ f) s =<>                                        -- (1)

(do
    (s'  , p , x') ← return (s , reflexive , x)
    (s'' , p' , b) ← f x' s'
    return (s'' , transitive p p' , b)
) =< left-identity >                                            -- (2)

(do
    (s'' , p' , b) ← f x s
    return (s'' , transitive reflexive p' , b)
) =< left-identity-P ∈ transitive reflexive p' ≡ p' >           -- (3)

(do
    (s'' , p' , b) ← f x s
    return (s'' , p' , b)
) =< right-identity >                                           -- (4)

f x s                                                           qed
```

We go through the proof step by step:

(1) We prove the equality using extensionality, giving both sides the initial state `s` as an argument. We start with the left-hand side of the equation.

(2) We unfold the definition of `returnₛ` and `_>>=ₛ_` using the **do**-notation. We use the `left-identity` of `M` to evaluate the leading `return`.

(3) Now, we need a (partial) proof-irrelevance, stating that for `_P_`, it holds that `∀ p → transitive reflexive p ≡ p`, to create the proof that would have been returned if it hadn't been for the extra transitivity arising from the prior use of `return`.

(4) We now use the `right-identity` of `M` to remove the trailing `return`.

This means that we need the law

```
left-identity-P : ∀ {x} {y} → (p : x P y) → transitive reflexive p ≡ p
```

to hold for this proof to go through. For the right identity, we will see that we also need the law

```
right-identity-P : ∀ {x} {y} → (p : x P y) → transitive p reflexive ≡ p
```

to hold as well.

Lemma 3.8:6

With the above definitions, it holds that

```
right-identity : m >>=S returnS ≡ m
```

under the assumptions that

```
right-identity-P : ∀ {x} {y} → (p : x P y) → transitive p reflexive ≡ p
```

Proof 3.8:7

We prove this by extensionality over the state `s : S`

```
(m >>=ₛ returnS) s =<>                                          -- (1)

(do
    (s'  , p , x) ← m s
    (s'' , p' , b) ← return (s' , reflexive , x)
    return (s'' , transitive p p' , b)
) =< left-identity >                                            -- (2)

(do
    (s'  , p , x) ← m s
    return (s' , transitive p reflexive , x)
) =< right-identity-P ∈ transitive p reflexive ≡ p >            -- (3)

(do
    (s'  , p , x) ← m s
    return (s' , p , x)
) =< right-identity >                                           -- (4)

m s                                                             qed
```

(1) We make the prove by extensionality, giving both sides of the equations the initial state `s`. We start with the left-hand side of the equation.

(2) We unfold the definition of $\texttt{return}_\texttt{S}$ and $\texttt{\_>>=}_\texttt{S}\texttt{\_}$ using the **do**-notation. We use the left identity to evaluate the second `return`.

(3) We use the proof-irrelevance rule to say that `∀ p → transitive p reflexive ≡ p` to state that the extra transitivity added by the excess return is irrelevant.

(4) We use the `right-identity` of `M` to remove the trailing `return`.

In the proof of the monad's associativity, we will need a third irrelevance law for `_P_`:

```
associative-P : ∀ {ab bc cd} → transitive ab (transitive bc cd) ≡ (transitive ab bc) cd
```

so we need the transitivity of _P_ to be associative as well. All these laws combined make _P_ an indexed monoid, indexed over the state type S (indexed because the type of the objects has to fit in the monoidal operation). Because a monad is a monoid in the category of endofunctors, it only makes sense that its substructures also have to form a monoid.

Lemma 3.8:8

With the above definitions, it holds that

```
associative : (m >>=ₛ f) >>=ₛ g ≡ m >>=ₛ (f >=>ₛ g)
```

under the assumptions that

```
associative-P : ∀ {ab bc cd} → transitive ab (transitive bc cd) ≡ (transitive ab bc) cd
```

Proof 3.8:9

As this proof is quite long, we will split it into several parts. We do the proof by extensionality over the state `s : S`:

```
((m >>=S f) >>=S g) s =<>                                        -- (1)

(do
    (s''  , p''  , b) ← (m >>=S f) s
    (s''' , p''' , c) ← g b s''
    return (s''' , transitive p'' p''' , c)
) =<>                                                           -- (2)

(do
    (s''  , transp'p''  , b) ← do
        (s'  , p'  , a) ← m s
        (s'' , p'' , b) ← f a s'
        return (s'' , transitive p' p'' , b)
    (s''' , p''' , c) ← g b s''
    return (s''' , transitive transp'p'' p''' , c)
) =< associative >                                              -- (3)

(do
    (s'  , p'  , a) ← m s
    (s''  , transp'p''  , b) ← do
        (s'' , p'' , b) ← f a s'
        return (s'' , transitive p' p'' , b)
    (s''' , p''' , c) ← g b s''
    return (s''' , transitive transp'p'' p''' , c)
) =< associative >                                              -- (4)

(do
    (s'  , p'  , a) ← m s
    (s'' , p'' , b) ← f a s'
    (s''cpy  , transp'p''  , b) ← do
        return (s'' , transitive p' p'' , b)
    (s''' , p''' , c) ← g b s''cpy
    return (s''' , transitive transp'p'' p''' , c)
) =< left-identity >                                            -- (5)
```

(1) We make the prove by extensionality, giving both sides of the equations the initial state `s`. We start with the left-hand side of the equation.

(2)
(3) We unfold the definition of both `_>>=S_` using the **do**-notation.
(4)
We use the associative law to move monadic actions from the subcomputation into the main computation.

(5) We use the `left-identity` law of `M` to evaluate the return of the remaining monadic subcompuation.

```
(do
    (s'   , p'   , a) ← m s
    (s''  , p''  , b) ← f a s'
    (s''' , p''' , c) ← g b s''
    return (s''' , transitive (transitive p' p'') p''' , c)
) =< associative-P >                                          -- (6)

(do
    (s'   , p'   , a) ← m s
    (s''  , p''  , b) ← f a s'
    (s''' , p''' , c) ← g b s''
    return (s''' , transitive p' (transitive p'' p''') , c)
) =< left-identity >                                          -- (7)

(do
    (s'   , p'    , a) ← m s
    (s''  , p''   , b) ← f a s'
    (s''' , p'''  , c) ← g b s''
    (s''' , p'''' , c) ← return (s''' , transitive p'' p''' , c)
    return (s''' , transitive p' p'''' , c)
) =< associative >                                            -- (8)

(do
    (s'   , p'    , a) ← m s
    (s''  , p''   , b) ← f a s'
    (s''' , p'''' , c) ← do
        (s''' , p'''  , c) ← g b s''
        return (s''' , transitive p'' p''' , c)
    return (s''' , transitive p' p'''' , c)
) =< associative >                                            -- (9)
```

(6) We now need the associativity of `_P_` as a weak form of proof-irrelevance to exchange the order of transitivity applications when merging the two monadic computations.

(7) We use the `left-identity` of `M` to introduce a new return statement that will be used for the new monadic subcomputation that is now taking place after the computation of `m`.

(8) (9) We use associativity twice to move the upper monadic computations into the new monadic subcomputation.

```
(do
    (s'   , p'    , a) ← m s
    (s''' , p'''' , c) ← do
        (s''  , p''   , b) ← f a s'
        (s''' , p'''  , c) ← g b s''
        return (s''' , transitive p'' p''' , c)
    return (s''' , transitive p' p'''' , c)
) =<>                                                    -- (10)

(do
    (s'   , p'    , a) ← m s
    (s''' , p'''' , c) ← (f >=>S g) a s'
    return (s''' , transitive p' p'''' , c)
) =<>                                                    -- (11)

(m >>=S (f >=>S g)) s                                    qed
```

(10) (11) Finally, we just apply the definitions of _>>=$_M$_ and _>=>$_M$_ to show that we ended up at our right-hand side.

We summarise:

Theorem 3.8:10

Given a state type S, a relation _P_ that forms a pre-order and is an indexed monoid with reflexivity and transitivity and index type S, and a monad M, then the context, which we will later call a constrained state monad transformer

```
CStateT : Set → Set
CStateT X = (s : S) → M (∃[ s' of S ] (s P s') × X)
```

forms a monad using the monadic operations from Section 3.8.1

Proof 3.8:11

Using Lemma

As the relation `_P_` needs to be reflexive and transitive means that it forms a *pre-order*. In that sense: Later states are somewhat "bigger" than earlier states. This makes pre-orders closely related to the concept of a *lattice*, introduced in Chapter 2 Section 2.6. Any lattice induces a pre-order and pre-orders, at least classically, also induce a lattice. Therefore, we will use a lattice instance on the state type to model a variable read to never change during computation, which is what we introduced the relation for in the first place. Conveniently, using lattices also gives us a direct formalism of how writes now have to work on the state. We will explore this in Section 3.8.3, but before that, we will create a few proofs of instances on our constrained state monad transformer.

3.8.2 Constrained Monad Transformer Instances

Code-Tag: `CStateT`

Theorem 3.8:12

Given a type `S`, a relation `_P_` that forms an indexed monoid with reflexivity and transitivity, and a monad `M`, then

```
CStateT : Set → Set
CStateT X = (s : S) → M (∃[ s' of S ] (s P s') × X)
```

together with the operation

```
lift : M X → CStateT X
lift m s = do
    x ← m
    return (s , reflexive , x)
```

forms a monad transformer using the monadic operations from Section 3.8.1

Proof 3.8:13

```
lift-return  : lift ∘ return_M  ≡ return_C
lift-return = extensPi \x → extensPi \s →
    (lift (return_M x)) s                                        =<>
    return x >>=_M (\ x' → return_M (s , reflexive , x')) =< left-identity >
    return_M (s , reflexive , x)                                 =<>
    return_C x s                                                 qed
```

We make this proof by extensionality over the returned value `x` and the state `s`. When the monadic lift unfolds, we just have to use the left identity of `M` to put the return value `x` into the return of the constrained monad and we are done.

```
lift-bind : lift (m >>=_M f) ≡ (lift m) >>=_C (lift ∘ f)
lift-bind = extensPi \s →
    lift (m >>=_M f) s =<>

    (do
        x ← do
            y ← m
            f y
        return (s , reflexive , x)
    ) =< associative >

    (do
        y ← m
        x ← f y
        return (s , reflexive , x)
    ) =< left-identity-P >

    (do
        y ← m
        x ← f y
        return (s , transitive reflexive reflexive , x)
    ) =< left-identity >
```

In the first three steps, we unfold the definition of the bind of the inner

monad `M`, using the monoidal instance of the relation `_P_` to state that our return has the shape it would have if we had run the two monadic actions after each other using the monadic bind of the constrained monad transformer.

```
(do
    y ← m
    x ← f y
    (s'' , p' , x') ← do
        return (s , reflexive , x)
    return (s'' , transitive reflexive p' , x')
) =< associative >

(do
    y ← m
    (s'' , p' , x') ← do
        x ← f y
        return (s , reflexive , x)
    return (s'' , transitive reflexive p' , x')
) =< left-identity >

(do
    y ← m
    (s' , p , y') ← do
        return (s , reflexive , y)
    (s'' , p' , x') ← do
        x ← f y'
        return (s' , reflexive , x)
    return (s'' , transitive p p' , x')
) =< associative >
```

In the next three steps, we bring the computation into a shape that resembles the monadic bind together with the lift operation, which has to return a state that does not change.

```
(do
    (s' , p , y') ← do
        y ← m
        return (s , reflexive , y)
    (s'' , p' , x') ← do
        x ← f y'
        return (s' , reflexive , x)
    return (s'' , transitive p p' , x')
) =<>

((lift m) >>=ᶜ (lift ∘ f)) s qed
```

Finally, we just verify that this shape is indeed the correct one by applying the definition of the monadic bind of the constrained monad transformer backward.

This monad transformer has a few useful properties when it comes to errors. It preserves left absorption of the inner monad, meaning that errors in the inner monad stay an error in the constrained state transformer monad:

Lemma 3.8:14

Given the conditions for being a monad transformer, any left absorbing action `empty` on `M` stays left absorbing in `CStateT`

Proof 3.8:15

```
preserves-left-absorb :
    {empty : {A : Set} → M A}
    {m' : A → CStateT B} →
    ({A B : Set} {m : A → M B} → empty >>= m ≡ empty) →
    (lift empty) >>=c m' ≡ (lift empty)
preserves-left-absorb {empty} {m'} left-absorb = extensPi \s →
    (lift empty >>=c m') s =<>

    (do
        (s1 , p1 , a) ← lift empty s
        (s2 , p2 , b) ← m' a s1
        return (s2 , transitive p1 p2 , b)
    ) =<>

    (do
        (s1 , p1 , a) ← do
            a ← empty
            return (s , reflexive , a)
        (s2 , p2 , b) ← m' a s1
        return (s2 , transitive p1 p2 , b)
    ) =< left-absorb >

    (do
        (s1 , p1 , a) ← empty
        (s2 , p2 , b) ← m' a s1
        return (s2 , transitive p1 p2 , b)
    ) =< left-absorb >

    empty =< sym left-absorb >

    (lift empty) s qed
```

We just unfold the definitions and apply the left absorption rule wherever necessary.

This works even inside the inner monad:

Lemma 3.8:16

lifting a left absorbing monadic action over `M` stays a left absorbing action over `M`

Proof 3.8:17

```
preserves-left-absorb-inside : {empty : {A : Set} → M A} →
    ({A B : Set} {m : A → M B} → empty >>= m ≡ empty) →
    {s : S} → {m' : (∃[ s' of S ] s P s' and A) → M (∃[ s' of S ] s P s' and B)} →
    (lift empty s) >>= m' ≡ (lift empty s)
preserves-left-absorb-inside {empty} left-absorb {s} {m'} =
    lift empty s >>= m'  =<>
    (empty >>= _) >>= m' =< left-absorb >
    empty >>= m'         =< left-absorb >
    empty                =< left-absorb >
    empty >>= _          =<>
    lift empty s         qed
```

again, just applying the left absorption rule wherever necessary when unfolding the lift operation.

We will use these facts to derive error-based instances, but before we do that, we build the alternative instance. We start by creating the monoid instance of the constrained monad transformer

Theorem 3.8:18

Given a type `S`, a relation `_P_` that forms an indexed monoid with reflexivity and transitivity, and a monad `M` that for all types `A` forms a monoid on `M A` so that its identity element `ε : M A` is left absorbing on `M`, then

```
CStateT : Set → Set
CStateT X = (s : S) → M (∃[ s' of S ] (s P s') × X)
```

together with the operations

```
ε = lift mzero
(m1 <> m2) s = m1 s <> m2 s
```

forms a monoid on `CStateT A` for all types `A` using the monadic operation from Section 3.8.1 and the monad transformer lift from Theorem 3.8:12

Proof 3.8:19

```
left-identity : ε <> a ≡ a
left-identity {a} = extensPi \s →
    (lift ε) s  <> a s  =<>
    (ε >>=ₘ _)  <> a s  =< left-absorb >
    ε           <> a s  =< left-identity-monoid >
    a s                  qed

left-identity : a <> ε ≡ a
left-identity {a} = extensPi \s →
    a s <> (lift ε) s   =<>
    a s <> (ε >>=ₘ _)   =< left-absorb >
    a s <> ε            =< right-identity-monoid >
    a s                  qed
```

in both identities, we use the fact that `ε` is left absorbing on the monad `M`. Apart from that, the identities follow from the identity of the monoid on `M`. Associativity as well follows trivially from the associativity of the monoid on `M`:

```
associative : (a <> b) <> c ≡ a <> (b <> c)
associative = extensPi \s → associative-monoid
```

We can now derive a few trivial instances.

Corollary 3.8:20

Given, for all types `A`, a monoid over `M A` with `ε : M A` being a left absorbing action on `M`, together with the prequisits for the monad transformer instance on `CStateT`, then `CStateT` forms a monad fail and monad alternative. If `ε` is right absorbing as well, it even forms a monad plus.

Proof 3.8:21

trivial from properties above.

The constrained state monad transformer does not form a classical state monad. While the `get` action stays largely the same, the `put` action does not work in the way we intend to. The reason is that in order to put something into the current state, we would need to know that it relates to the previous state. As we usually do not know anything about the previous state, this, in its pure form, becomes hard to implement. We solve this problem by giving the state operations for the case that the state type itself has a lattice instance, meaning that not only is there a pre-order defined on it, but we can even compute an increase within that order in order to create new states that relate to previous ones.

3.8.3 Lattice Based State

Code-Tag: `ThresholdVariables`
Code-Tag: `BiThresholdVariables`

We want to solve the problem that in order to put a new state into the constrained state monad transformer we need to know that the state we put into it relates to the previous state. Luckily, as soon as we have a semilattice, we get the property that the lattice operation `<>` form the monoid in the lattices is what we call *directional*:

```
directional : x P (x <> y)
directional {x} {y} =
    x <> (x <> y) =< associative >
    (x <> x) <> y =< idempotent >
    x <> y        qed
```

We can now create a `put` action, which we will call `add`, that only "increases" the state, like

```
add : X → LState Unit
add x s = return (s <> x , directional , unit)
```

together with the usual state operations

```
get : CStateT S
get s = return (s , reflexive , s)

getS : (S → X) → CStateT X
getS f s = return (s , reflexive , f s)
```

We want to use these operations to create the monadic variables from Definition 3.8:1.

There is a computational model for lattice based variables developed by Kuper [61,62] in order to write parallel programs that still create a deterministic result. The idea is that the state we read from has a lattice instance and, during computation, only ever grows in size, just like for our constrained state monad transformer. The allowed reads from this state have to be in a form such that the read either fails or, from a certain size of the state on, never changes its output for any successive, bigger state. If the state now only increases in size, all variable reads return the same result no matter in what order they occur, giving us the properties from Definition 3.8:1. The read value stays the same for every successive state. These values were called *LVars* (short for: lattice variable). We will slightly reformulate Kuper's result to fit the use in solving.

The main addition that we will provide is that our variables can eventually also result in a conflict. We will see in a bit why this is necessary. First, we reformulate the original result.

The basic mechanism for reading from an LVar is based on a so-called *threshold function*.

Definition 3.8:22: **Threshold Function**

Given a pre-order `_P_` on `S`, then a function

```
f : S → Maybe X
```

is called a **threshold function** if it satisfies the following properties:

```
∀ (s s' : S) → s P s' →
      (f s ≡ nothing)
    ∨ (f s ≡ f s')
```

so for all successive state of a state `s`, if a value is returned, the same value is returned for all successive states.

with this property, we can create an `LVar` from a threshold function that would form a monadic variable from definition 3.8:1. The problem is that for solving, this is not expressive enough.

Exploration 3.8:23

Assume we want to create a threshold function returning the Boolean values stored in the trivial Boolean lattice $\mathbb{B}_\bot^\top$. We remember that the trivial Boolean lattice has the values

pseudocode:

```
top⊤⊥ < (val⊤⊥ true) , (val⊤⊥ false) < bot⊤⊥
```

So it is either unassigned, assigned with a value or results in a conflict. We need the conflict value in order to model what happens when we try to assign a variable with two different values. If the threshold function returns a value for either the $(\mathtt{val}_\bot^\top\ \mathtt{true})$ or the $(\mathtt{val}_\bot^\top\ \mathtt{true})$ case, it would have to return that same value in the $\mathtt{bot}_\bot^\top$ case as well, as this value is greater than the values for the Boolean assignments. But the $\mathtt{bot}_\bot^\top$ value does not store the information about whether the value was `true` or `false`, so the threshold function can only return some constant value. This means that there could be no threshold function that can potentially give out both the `true` or `false` value once assigned and we could only get the correct behaviour using two different threshold functions. As this is cumbersome, we need a generalisation of threshold functions that also deals with the `bot` case adequatly.

We note at this point that also [61] deals with the problem of conflicting assignments, in their case using an error state. Our result is similar, but we allow for the use of `bot` values, with the variables themselves returning an error if they read such a value. Therefore, we introduce so-called *Bi-threshold-functions*.

3.8.3.1 Bi-Threshold Functions and Variables

Code-Tag: `BiThresholdVariables`

Definition 3.8:24: **Variable Assignment**

Given a type `X`, we define a **variable assignment** as

```
data VarAsm (X : Set i) : Set i where
    unassigned : VarAsm X
    asm    : X → VarAsm X
    conflict   : VarAsm X
```

Meaning it can either be unassigned, assigned or bear a conflict.

Definition 3.8:25: **Bi-Threshold Function**

Given a pre-order `_P_` over the type `S`, a function

```
f : S → VarAsm S
```

is called a **bi-threshold function** if the following property holds:

```
isBiThresholdRead : ∀ (s s' : S) → s P s' →
      (f s  ≡ unassigned)
    ∨ (f s  ≡ f s')
    ∨ (f s' ≡ conflict)
```

For better readability, we will express that same property using an indexed data type

pseudocode:

```
data _=incAsm=_ (a b : VarAsm X) : Set where
    unas-to-anything :  unassigned =incAsm= a
    asm-eq           :  asm x      =incAsm= asm x
    asm-conf         :  asm x      =incAsm= conflict
    conf-conf        :  conflict   =incAsm= conflict
```

which we rewrite to have the equivalences explicit (and be able to use cubical and homotopy type theory for proofs):

```
data _=incAsm=_ (a b : VarAsm X) : Set where
    unas-to-anything :  a ≡ unassigned →                a =incAsm= b
    asm-eq           :  a ≡ asm x      → b ≡ asm x     → a =incAsm= b
    asm-conf         :  a ≡ asm x      → b ≡ conflict  → a =incAsm= b
    conf-conf        :  a ≡ conflict   → b ≡ conflict  → a =incAsm= b
```

So we can express the above property as

```
isBiThresholdRead : ∀ (s s' : S) → s P s' → f s =incAsm= f s'
```

A bi-threshold function can at some size of the input start to produce a value and that value stays the same until eventually a conflict is produced. This can be shown with the following, easily provable corollaries:

Corollary 3.8:26

Given a threshold function f over the pre-order _P_ over S, the following properties hold:

```
increase-asm : ∀ (s s' : S) → s P s' → (x : S) → ¬ (f s ≡ asm x ∧ f s' ≡ unassigned)
```

Proof 3.8:27

pseudocode:

```
increase-asm s s' sPs' (fs=asm , fs'=unas) with isBiThresholdRead s s' sPs'
...| unas-to-anything fs=unas = absurd $
        asm x      =< fs=asm >
        f s        =< fs=unas >
        unassigned qed
...| asm-eq fs=asm fs'=asm = absurd $
        asm x      =< fs=asm >
        f s        =< fs=fs' >
        f s'       =< fs'=unas >
        unassigned qed
...| asm-conf fs=asm fs'=conf = absurd $
        conflict   =< fs'=conf >
        f s'       =< fs'=unas >
        unassigned qed
...| conf-conf fs=conf fs'=conf = absurd $
        asm x    =< fs=asm >
        f s      =< fs=conf >
        conflict qed
```

Analogously, we can prove the following properties:

```
stays-conflict : (s s' : S) → s P s' → (x : S) → ¬ (f s ≡ conf ∧ f s' ≡ asm x)

stays-conflict' : (s s' : S) → s P s' → (x : S) → ¬ (f s ≡ conf ∧ f s' ≡ unassigned)

keeps-value : ∀ (s s' : S) → s P s' → (x : S) → f s ≡ asm x → (f s' ≡ asm x) ∨ (f s' ≡ conflict)
```

Side Note 3.8:28

The only reason we are using pseudocode for the proofs over variable assignments is that in order for these to compile in cubical agda, we need something called the encode-decode method [92] in order to prove inequalities. As this technique is a bit technical, we present the proofs in a way that looks more natural, marking them as pseudocode even though they are structurally equivalent to the actual proofs.

Using bi-threshold functions, we can create our version of an `LVar` that will soon be used to create the monadic variables.

Definition 3.8:29: **Lattice Bi-Threshold Variable (LBVar)**

Given a bounded semilattice over a type `S` and a type `X`, two operations

```
read : S → VarAsm X
write : X → S
```

are called a **lattice bi-threshold variable (LBVar)** (written `LBVar S bsl X` if bounded semilattice unclear from the context), iff the following conditions hold:

```
isBiThresholdRead : ∀ (s s' : S) → s P s' → read s =incAsm= read s'
write-read : read (write x) ≡ asm x
read-write-read : ⟪ write (read s) ⟫ >>= read ≡ read s
```

Where we use the obvious monad instance for variable assignments to write the `read-write-read` property.

A lattice bi-threshold variable will be our foundation for the monadic variables because through their threshold read, which keeps the returned value consistent with an increasing state until a conflict occurs, we can show that it does not matter which value out of two different reads we use. The properties

in connection to the `write` ensure that we can put an arbitrary value into the variable (and read it afterward).

The most important property of the lattice bi-threshold variables is that their return value stays consistent with any previous write.

Lemma 3.8:30

Given a lattice bi-threshold variable over a bounded semilattice over the type S, then the following property holds:

```
write-persist : (write x) P s → (read s ≡ asm x) or (read s ≡ conflict)
```

Proof 3.8:31

We proof this by case splitting on the bi-threshold read property of the `read` function.

```
write-persist {x} {s} wPs with isBiThresholdRead _ _ wPs
... | unas-to-anything readwritex=unas = absurd $
        unassigned     =< readwritex=unas >
        read (write x) =< write-read >
        asm x          qed
... | asm-eq readwritex=asmx1 reads=asmx1 = left $
        read s         =< reads=asmx1 >
        asm _          =< readwritex=asmx1 >
        read (write x) =< write-read >
        asm x          qed
... | asm-conf readwritex=asmx1 reads=conflict = right reads=conflict
... | conf-conf readwritex=conflict reads=conflict = absurd $
        conflict       =< sym readwritex=conflict >
        read (write x) =< write-read >
        asm x          qed
```

To check that the lattice bi-threshold variables also are viable to what we want to achieve, we prove two useful properties. The first one is, that there is a lattice bi-threshold variable reading out the value from a trivial lattice.

Lemma 3.8:32

Given a type X with decidable equality, then the operations

```
read : X⊤⊥ → VarAsm X
read top⊤⊥      = unassigned
read bot⊤⊥      = conflict
read (val⊤⊥ x) = asm x

write : X → X⊤⊥
write x = val⊤⊥ x
```

form a lattice bi-threshold variable over the bounded meet semilattice of the trivial lattice $\mathsf{X}^{\top}_{\bot}$.

Proof 3.8:33

trivial by going through each case.

Further, in order to have more than just one variable, we need a product of variables.

Lemma 3.8:34

Let L1, L2 be bounded semilattices over the types S1, S2, respectively. Let v1 be a lattice bi-threshold variable over L1 and S1 with codomain X1 and v2 be a lattice bi-threshold variable over L2 and S2 with codomain X2. Then the operations

```
read₁ : (S1 × S2) → VarAsm X1
read₁ = read v1 ∘ fst

write₁ : X1 → (S1 × S2)
write₁ s = write v1 s , e2

read₂ : (S1 × S2) → VarAsm X2
read₂ = read v2 ∘ snd

write₂ : X2 → (S1 × S2)
write₂ s = e1 , write v2 s
```

form two lattice bi-threshold variables v_1 and v_2 with codomains X1 and X2 over the shared domain S1 × S2 on the semilattice product of L1 and L2. As notation, we write LBVarProductFst v1 v2 for the first variable and LBVarProductSnd v1 v2 for the second. To further simplify, we write

```
< v1 × v2 >LBVar = (LBVarProductFst v1 v2 , LBVarProductSnd v1 v2)
```

Proof 3.8:35

Trivial, by going over all cases, providing the matching properties from the original variables v1 and v2.

We will not show any details here, but of course a general homomorphic (and heteromorphic) product exists as well.

Corollary 3.8:36

There exists a general, homomorphic product

```
LBVarHomProduct : LBVar S bsl X → (LBVar (Sⁿ) (bslⁿ) X)ⁿ
```

where we duplicate the given variable into a product space of n distinct, independent variables. bslⁿ in this case is not the tuple of bounded semilattices but the n-product of the given bounded semilattice.

Proof 3.8:37

We note that we give a slightly prettier explanation of how this works in Section 3.9, but for the sake of completeness, we add the ad hoc proof of the statement here.
In order to create the general lattice product, we have to be able to lift the lattice over which the product of n lattice variables was defined to a product lattice with one more variable for the recursive case. We can do so with the function

```
RaiseLBVarHomProductContext : LBVar S bsl X →
    (LBVar (Sⁿ) (bsl BSL^ n) X)ⁿ →
    (LBVar (S(1 + n)) (bsl BSL^ (1+ n)) X)ⁿ
RaiseLBVarHomProductContext {0}    v t = unit
RaiseLBVarHomProductContext {1+ n} v t = mapNTup (LBVarProductSnd v) t
```

Here, we recursively turn every variable into the second variable of a product construction.
We can then prove the statement by recursively adding the first variable of the product construction to the vector from the recursive call that has been transformed to contain the variables of the second part of the product construction.

Now that we have shown that lattice bi-threshold variables behave the way that we want to, we can show that they, indeed, form monadic variables in our constrained state monad transformer.

3.8.3.2 Monadic Variables from Lattice Bi-Threshold Variables

Code-Tag: LatticeState

In this section, we first need to fix the monad we are working on, to get matching variables into the state.

Definition 3.8:38: **Lattice State**

Given a bounded semilattice bsl over the type S with the induced pre-order _P_ and a monad M, we rename the constrained state monad

transformer over those structures to `LState`, short for **lattice state**. We rename the lattice bi-threshold variable morphism over these structures to `V`, so

```
V : Set → Set
V X = LBVar S bsl X
```

forms a reference to a given object of type `X` in the state of the constrained state monad transformer.

We now have to define what the monadic read should look like. As the lattice bi-threshold variable returns a variable assignment, we define a destructor for variable assignments to later create the read

```
unas:_conf:_asm: : X → X → (Y → X) → VarAsm Y → X
unas:_conf:_asm: unas conf fasm unassigned = unas
unas:_conf:_asm: unas conf fasm (asm x)    = fasm x
unas:_conf:_asm: unas conf fasm conflict   = conf
```

The read now has to end up in some erroneous state for the cases that the lattice bi-threshold variable does not return a value. We will show that the read we create stays a monadic variable for as long as the monadic action we choose for the erroneous reads is left absorbing. For further generalisation, we even say that the error state can get the current context it was thrown in as an argument without losing the monadic variable properties. We will prove the following theorem:

Theorem 3.8:39

When given any two, left absorbing over `M`, monadic actions

```
readUnas : ∀ {A} → V X → S → M A
readConf : ∀ {A} → V X → S → M A
```

then forall variables `v : V X`, the operations

```
read : V X → LState X
read v s =
    unas: lift (readUnas v s) s
    conf: lift (readConf v s) s
    asm: (flip returnC s)
    (readL v s)

write : V X → X → LState Unit
write v x s = return (s <> writeL v x , directional , unit)
```

form a monadic variable over the `LState` monad.

Proof 3.8:40

This proof is incredibly long and tedious, so we present the core ideas in simplified code. We begin with the commutative property of the variable read. We will not distinguish between the monadic binds for the inner monad and the transformer as it will be clear from the context.

```
commutative : ∀ {Y Z}{m' : M Y}{f : Y → X → M Z} →
    (read v >>= \_  → m' >>= \y → read v >>= \x2 → f y x2)
    ≡
    (read v >>= \x1 → m' >>= \y → read v >>= \_  → f y x1)
```

For this property we have to show that when reading twice, we can use any of the two values for the subsequent computation without changing the outcome. We prove this by extensionality over the state `s : S`, and mainly use the same reasoning as in the `keep-value` property from Corollary 3.8:26.
We distinguish the cases for the first read, which is the lattice bi-threshold variable read `readL v s` on the initial state `s`.

```
...| unassigned =
    (lift (readUnas v s) >>= _) s =< preserves-left-absorb left-absorb-readUnas >
    (lift (readUnas v s))       s =< preserves-left-absorb left-absorb-readUnas >
    (lift (readUnas v s) >>= _) s qed
```

In the unassigned case, the first read results in an error action, so we can use its left absorption to transform the successive computation into any shape we want. This is identical to the case of a conflict on the first read.

```
...| conflict   =
    (lift (readConf v s) >>= _) s =< preserves-left-absorb left-absorb-readConf >
    (lift (readConf v s))        s =< preserves-left-absorb left-absorb-readConf >
    (lift (readConf v s) >>= _) s qed
```

In the case of an actual result on the first read, we start with the following formula.

```
...| (asm x1)  =
    (do
        (s1 , p1 , x1) ← read v s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x2) ← read v s2
                (s4 , p4 , z) ← f y x2 s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) =<>

    (do
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s
            (s4 , p4 , z) ← do
                (s3 , p3 , x2) ← read v s2
                (s4 , p4 , z) ← f y x2 s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive reflexive p4 , z)
    ) =<>
```

This is all definitions from the constrained monad state transformer unrolled. The initial read directly computes without changing the state and its value is ignored. Our goal is to have the (here: marked in blue) value `x2` to turn into the value for `x1`. We can do that by going through the cases of the `isBiThresholdRead` property of `readL v`, so we look at the result of `readL v s2`. Especially due to the `isBiThresholdRead` property, there are only two cases for `readL v s2`

pseudocode:

```
... | readL v s2 ≡ conflict =
    (do
        (s1 , p1 , x1) ← read v s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x2) ← read v s2
                (s4 , p4 , z) ← f y x2 s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) =< left-absorb-readConf twice >

    (do
        (s1 , p1 , x1) ← read v s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x2) ← read v s2
                (s4 , p4 , z) ← f y x1 s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) qed
```

When the second read returns a conflict, its read action turns into a left absorbing monadic action, so we can freely modify any successive computation. When the read returns a value, then by the `keep-value`property from Corollary 3.8:26, we know the value stays the same, so we can exchange it like

pseudocode:

```
... | readL v s2 ≡ asm x1 =
    (do
        (s1 , p1 , x1) ← read v s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x2) ← read v s2
                (s4 , p4 , z) ← f y x2 s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) =< keep-value >

    (do
        (s1 , p1 , x1) ← read v s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x2) ← read v s2
                (s4 , p4 , z) ← f y x1 s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) qed
```

and have therefore proven the statement. In the case for the `read-write` property, we mainly use the `write-persist` property from Lemma 3.8:30. We have to prove that

```
read-write : ∀ {x : X}{m' : M Y}{f : Y → X → M Z} →
    (write v x >> (m' >>= \y → read v >>= \x' → f y x'))
    ≡
    (write v x >> (m' >>= \y → read v >>= \_  → f y x))
```

We prove the statement by extensionality over the initial state `s` and start again by unfolding all of the definitions from the constrained

state monad transformer.

pseudocode:

```
(do
    (s1 , p1 , _) ← write v x s
    (s4 , p4 , z) ← do
        (s2 , p2 , y) ← m' s1
        (s4 , p4 , z) ← do
            (s3 , p3 , x') ← read v s2
            (s4 , p4 , z ) ← f y x' s3
            return (s4 , transitive p3 p4 , z)
        return (s4 , transitive p2 p4 , z)
    return (s4 , transitive p1 p4 , z)
) =< write-persist >
```

We want to turn the marked value `x'` into the written value `x`. When applying the `write-persist` property, we have two cases again

pseudocode:

```
...| readL v s2 ≡ conflict =
    (do
        (s1 , p1 , _) ← write v x s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x') ← read v s2
                (s4 , p4 , z ) ← f y x' s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) =< left-absorb-readConf twice >

    (do
        (s1 , p1 , _) ← write v x s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x') ← read v s2
                (s4 , p4 , z ) ← f y x s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) qed
```

In the case where the read returns a conflict, we can freely change the successive computation again. In the case of an assignment read, we know that the assignment returned the written value, so we simply get

pseudocode:

```
...| readL v s2 ≡ asm x in readLvs2=asmx =
    (do
        (s1 , p1 , _) ← write v x s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x') ← read v s2
                (s4 , p4 , z ) ← f y x' s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) =< readLvs2=asmx >

    (do
        (s1 , p1 , _) ← write v x s
        (s4 , p4 , z) ← do
            (s2 , p2 , y) ← m' s1
            (s4 , p4 , z) ← do
                (s3 , p3 , x') ← read v s2
                (s4 , p4 , z ) ← f y x s3
                return (s4 , transitive p3 p4 , z)
            return (s4 , transitive p2 p4 , z)
        return (s4 , transitive p1 p4 , z)
    ) qed
```

Now that we know that our lattice bi-threshold variables can be turned into monadic variables it is time to test the concept at the example of SAT-Solving.

3.8.3.3 Example: SAT-Solving

Code-Tag: `SATSolvingExample`

We start by defining a SAT-formula over an arbitrary type `V` for variables.

```
data SATFormOver (VAR : Set) : Set where
    trueSF  : SATFormOver VAR
    falseSF : SATFormOver VAR
    varSF   : VAR → SATFormOver VAR
    notSF   : SATFormOver VAR → SATFormOver VAR
    _/\SF_  : SATFormOver VAR → SATFormOver VAR → SATFormOver VAR
    _\/SF_  : SATFormOver VAR → SATFormOver VAR → SATFormOver VAR
```

This is a Boolean formula that is parametrised over the type of variables it is defined over. We define a few convenience functions:

```
_=>SF_ : {V : Set i} → SATFormOver V → SATFormOver V → SATFormOver V
a =>SF b = notSF a \/SF b

_<=>SF_ : {V : Set i} → SATFormOver V → SATFormOver V → SATFormOver V
a <=>SF b = (a =>SF b) /\SF (b =>SF a)
```

For implication and bi-implication.

In our setting, the variables will be the `V 𝔹` from Definition 3.8:38, which are the monadic variables created from the lattice bi-threshold variables over the trivial Boolean lattice.

For our monadic context, we are using an `LState` over a monad `M`. We further say that M has an alternative instance that `LState` inherits via the construction of the monoid from Theorem 3.8:18. Concretely in this example, `M` is the list monad composed with the left error monad induced by the sum type. On top of that, we can parametrise the solver over the two left-absorbing monadic actions

```
readUnas : V 𝔹 → S → M A
readConf : M A
```

where we ignore that `readConf` could, according to the previous definitions, also take the variable and state as an input to the error. Both operations are lifted in the variable read according to Theorem 3.8:39.

Before we start writing any solving related functions, we define ourselves a benchmark example formula.

Example 3.8:41

As an example, we use a Boolean formula that is only true if exactly one variable is true, similar to our $\texttt{onlyOne}_\texttt{n}$ example in Section 3.2. Given a natural number $\texttt{n} : \mathbb{N}$, we define our variables to be the homogenous product space of the lattice bi-threshold variables over the trivial Boolean bounded meet semilattice, so

```
variables : (LBVar (𝔹⊥⊤ⁿ) (𝔹⊥⊤-bmsl) 𝔹)ⁿ
variables = LBVarHomProduct Triv𝔹⊥⊤LBVar
```

or, shorter:

```
variables : (V 𝔹)ⁿ
variables = LBVarHomProduct Triv𝔹⊥⊤LBVar
```

We define a few formulas for constraints. To those more familiar with old mathematics, here is how the formula would look like in classical mathematics:

$$
\begin{aligned}
{}_{\geq}\mathbb{1} &:= \bigvee_{\mathtt{v} \in \mathtt{variables}} \mathtt{v} \\
{}_{\leq}\mathbb{1} &:= \bigwedge_{\mathtt{v} \in \mathtt{variables}} \left(\mathtt{v} \Rightarrow \bigwedge_{\substack{\mathtt{v}' \in \mathtt{variables} \\ \mathtt{v} \neq \mathtt{v}'}} \neg \mathtt{v}' \right) \\
{}_{=}\mathbb{1} &:= {}_{\geq}\mathbb{1} \wedge {}_{\leq}\mathbb{1}
\end{aligned}
$$

Of course, we can also do this categorically with our definitions. We begin by redefining the big conjunction and disjunction operators:

```
AND : (V 𝔹 → SATFormOver (V 𝔹)) →
    (V 𝔹)ⁿ → SATFormOver (V 𝔹)
AND f = fold
    (_/\SF_ ∘ f)
    trueSF

OR : (V 𝔹 → SATFormOver (V 𝔹)) →
    (V 𝔹)ⁿ → SATFormOver (V 𝔹)
OR f = fold
    (_\/SF_ ∘ f)
    falseSF

syntax AND (\v → F) Vec = ⋀[ v , Vec ] F
syntax OR  (\v → F) Vec = ⋁[ v , Vec ] F
```

This creates a conjunction and disjunction over a finite list of variables using the supplied variable-to-formula function. We provide additional syntax declarations to make these look like the mathematical formulas. We can express the formula for at least one variable having to be true as a disjunction over all variables.

```
minOne : SATFormOver (V 𝔹)
minOne = ⋁[ v , variables ] varSF v
```

Next, we want to define the constraint that we want to have at most one variable to be true. In order to define it, we have to state that if any variable is true, no other variable should be true as well. Therefore, we create the constraint

```
noneOf : (V 𝔹)ⁿ → SATFormOver (V 𝔹)
noneOf vars = ⋀[ v , vars ] notSF (varSF v)
```

We have to define that for all variables `v`, if `v` is true then all variables

except v have to be false. For this, we need a special conjunction that gives all variables except the current one as an input to the formula constructor function. As it turns out, it is a little easier to define this when we only give all variables that came before the given variable in the iteration, which is also perfectly sufficient for our purpose. We define the new and-operator as

```
ANDREM : (V 𝔹 → {n' : Nat} → (V 𝔹)^n' → SATFormOver (V 𝔹)) →
           (V 𝔹)^n →
           SATFormOver (V 𝔹)
ANDREM f = fst ∘ foldNTup
    (\{v (r , rvars) → (f v rvars) /\SF r , (v , rvars)})
    (trueSF , unit)

syntax ANDREM (\v vars → F) Vec = ⋀r[ v , Vec , vars ] F
```

This operator also creates the tuple of all previously visited variables, which is given to the formula creation function and thrown away in the final output.
We can now express the constraint of at most one variable being true as

```
maxOne : SATFormOver (V 𝔹)
maxOne = ⋀r[ v , variables , variables/v ] (varSF v) =>SF noneOf variables/v
```

Our overall constraint can now be formulated as

```
exactlyOne : SATFormOver (V 𝔹)
exactlyOne = minOne /\SF maxOne
```

which we will use as a benchmark example.

Now, we want to describe the first functions on our SAT-formulas in our monadic context. We start with a function that just evaluates the result of our formula:

```
assignmentOf : SATFormOver (V 𝔹) → LState 𝔹
assignmentOf trueSF       = return true
assignmentOf falseSF      = return false
assignmentOf (varSF v)    = read v
assignmentOf (notSF form) = ⟪ notB (assignmentOf form) ⟫
assignmentOf (a /\SF b)   = ⟪ (assignmentOf a) andB (assignmentOf b) ⟫
assignmentOf (a \/SF b)   = ⟪ (assignmentOf a) orB  (assignmentOf b) ⟫
```

This just reads the assignment of the SAT-Formula from the current state. Of course, this only works when any assignment is present. If we have the variables `v1`, `v2` and `v3`, we can write

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v1 true
    write v2 false
    write v3 false
    assignmentOf exactlyOne
```

To get the output

```
left true :: []
```

which states that the `LState` monad had one result stating `true`, which is the result of reading out the assignment from the `exactlyOne` formula. For the remainder of this section, we will, unless otherwise stated, use the formula from Example 3.8:41 with `n = 3`, meaning we have three variables of which only exactly one can be true in a satisfying assignment.

Side Note 3.8:42

To run our solver, we use the list monad transformer wrapped in the error monad transformer as our monad `M`. We therefore have a run-function:

```
run : LState X → List (X or Err)
run m = map (mapLeft (snd ∘ snd)) (m top)
```

that gives us a list of all possible results from our lattice constrained state monad transformer. Our concrete type for `Err` will vary between the concrete applications.

Of course, we can also output when the formula evaluates to false

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v1 true
    write v2 true
    write v3 false
    assignmentOf exactlyOne
```

```
left false :: []
```

The problem is that with our naive formulation, we loose the ability to read out the formula's assignment if not all variables are assigned, for example:

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v1 true
    write v2 true
    assignmentOf exactlyOne
```

```
right "read error" :: []
```

gives us a read error from reading from an unassigned variable, even though we should already be able to tell that the formula is assigned to false. The reason this is happening is because our `assignmentOf` function does not read the variables lazily, so even if we do not need a value, it is still being read. We solve this by first defining a slightly safer read. To do that, we parametrise the solver further with a function

```
catchUnas : LState A → LState A → LState A
```

that can catch if a read threw an unassignment error. This function has the obvious property of

```
catchUnas (lift (readUnas v s)) m ≡ m
```

We can use this to define a safe read that does not throw an error when the read fails due to a value being unassigned.

```
safeRead : LState X → LState (Maybe X)
safeRead m = catchUnas (just <$> m) (return nothing)
```

A naïve approach to rewrite our `assignmentOf` could look as follows:

error:

```
assignmentOfSafeNaive : SATFormOver (V 𝔹) → LState 𝔹
assignmentOfSafeNaive trueSF  = return true
assignmentOfSafeNaive falseSF = return false
assignmentOfSafeNaive (varSF v) = read v
assignmentOfSafeNaive (notSF form) = ⦇ notBool (assignmentOfSafeNaive form) ⦈
assignmentOfSafeNaive (a /\SF b) = do
    a? ← safeRead (assignmentOfSafeNaive a)
    case a? of
        \{ (just false) → return false
        ; _             → assignmentOfSafeNaive b }
assignmentOfSafeNaive (a \/SF b) = do
    a? ← safeRead (assignmentOfSafeNaive a)
    case a? of
        \{ (just true) → return true
        ; _            → assignmentOfSafeNaive b }
```

What we did here is to rewrite the properties of the Boolean ”and” and ”or” function. If we can already determine the result from reading the first value we return it. Otherwise we return the result of the second value. This does solve our issue above:

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v1 true
    write v2 true
    assignmentOfSafeNaive exactlyOne
```

```
left false :: []
```

The only problem is that this returns the wrong value in come cases. Take for example

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v2 true
    assignmentOfSafeNaive exactlyOne
```

This should return a read error because we should but be able to tell the assignment of the formula because for some assignments it would be `true` and for others it would be `false`. However, our naïve assignment evaluator gives us

```
left false :: []
```

which is just wrong.

The reason for us to get the wrong result is that in the case where the first value is unassigned, we always return the value of the second value. As however the result does still depend on the first value, this might give the wrong result. We can solve this by going through all cases of assignments of both variables, only returning values when we are sure that they do not depend on unassigned values.

```
assignmentOfSafe : SATFormOver (V 𝔹) → LState 𝔹
assignmentOfSafe trueSF  = return true
assignmentOfSafe falseSF = return false
assignmentOfSafe (varSF v) = read v
assignmentOfSafe (notSF form) = ⦇ notB (assignmentOfSafe form) ⦈
assignmentOfSafe (a /\SF b) = do
    a? ← safeRead (assignmentOfSafe a)
    b? ← safeRead (assignmentOfSafe b)
    case (a? , b?) of
        \{ (just false , _ ) → return false
        ;  ( _ , just false) → return false
        ;  (just true , _ )  → assignmentOfSafe b
        ;  ( _ , just true)  → assignmentOfSafe a
        ;  (nothing , nothing) →
            ⦇ (assignmentOfSafe a) andB  (assignmentOfSafe b) ⦈ }
assignmentOfSafe (a \/SF b) = do
    a? ← safeRead (assignmentOfSafe a)
    b? ← safeRead (assignmentOfSafe b)
    case (a? , b?) of
        \{ (just false , _ ) → assignmentOfSafe b
        ;  ( _ , just false) → assignmentOfSafe a
        ;  (just true , _ )  → return true
        ;  ( _ , just true)  → return true
        ;  (nothing , nothing) →
            ⦇ (assignmentOfSafe a) orB  (assignmentOfSafe b) ⦈ }
```

This now returns the correct result

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v2 true
    assignmentOfSafeNaive exactlyOne
```

```
right "read error" :: []
```

Side Note 3.8:43

In the final case, we could have thrown an unassignment error, but we would not know on which variable directly, so it just looked nicer to rerun the function and just let it throw its read error on its own.

We now want to turn this evaluation function into a solver. The easiest way to do that is using the following helper function:

```
branchingRead : V 𝔹 → LState 𝔹
branchingRead v = catchUnas
    (read v)
    ((write v true <|> write v false) >> read v)
```

This function tries to read a value. If the read is unsuccessful due to a missing assignment, we disjunct every possible assignment and try to reread. Because we only branch when the read was unassigned, we branch for every variable at most once. It should be noted that if this creates a conflict, the second read will throw a conflict error.

We now recreate the `assignmentOfSafe` function, now called `assignmentOfSolve`, only changing the case of the read to

```
assignmentOfSolve : SATFormOver (V 𝔹) → LState 𝔹
...
assignmentOfSolve (varSF v) = branchingRead v
...
```

Now, we can get the assignment of a partially assigned Boolean formula, filling in assignments that are needed to determine the result of the Boolean formula. If we run this like before, we get a list of possible formula results:

```
left false :: left false ::
left false :: left true  ::
left false :: left true  ::
left true  :: left false :: []
```

This is not particularly informative. All it tells us that there are assignments that make the formula evaluate to `true`/`false`, respectively. We can write a function actually extracting the model as

```
getModel : LState (Maybe 𝔹ⁿ)
getModel = fold
    (\v m → ⟪ (safeRead (read v)) , m ⟫)
    (return unit)
    variables
```

This performs a safe read on every variable, returning its value when it is assigned and `nothing` if it isn't. The function continues to throw a conflict if a conflict is detected during the read of any variable.

We can now get a list of all models satisfying the formula as

```
assignmentOfTest : List (Maybe 𝔹ⁿ or Err)
assignmentOfTest = run do
    assignmentOfSolve exactlyOne >>= guard
    getModel
```

Here, we get the result of the formula on the current assignment and filter it for those assignments that made the formula evaluate to `true`. Afterwards, we return the current model. Running this gives us the following satisfying models:

```
left (just true  , just false , just false) ::
left (just false , just true  , just false) ::
left (just false , just false , just true ) :: []
```

We want to do some complexity analysis to check how many nodes have been visited to enumerate the possible models. We do so by slightly rewriting the above code to

```
assignmentOfTest : List (Maybe 𝔹³ or Err)
assignmentOfTest = run do
    res ← assignmentOfSolve exactlyOne
    when (not𝔹 res) (throw "wrong output")
    getModel
```

This yields the output

```
right "wrong output" ::
right "wrong output" ::
right "wrong output" ::
left (just true , just false , just false) ::
right "wrong output" ::
left (just false , just true , just false) ::
left (just false , just false , just true) ::
right "wrong output" :: []
```

Couting the number of nodes visites we can see, that the current solution is just a generate and test. We still go through all possible models, filtering those that do not form a model of the formula. The reason is clear: We branch for every variable, but also in every direction. For individual variables, there is no prevention from branching even if assigning the variable would lead to a wrong output. It would be way better if instead of just blatantly branching

when reading a variable we would instead directly assign it to the desired output value. We write the following function that is similar to our evaluation, only that it is given an additional parameter of which output it is supposed to produce. We start with a naïve version:

```
-- naive
solve : SATFormOver (V 𝔹) → 𝔹 → LState Unit
solve trueSF  true   = return unit
solve falseSF false  = return unit
solve trueSF  false  = lift readConf
solve falseSF true   = lift readConf
solve (varSF v) aim     = write v aim
solve (notSF a) aim     = solve a (not𝔹 aim)
solve (a /\SF b) true   = solve a true   >>  solve b true
solve (a \/SF b) false  = solve a false  >>  solve b false
solve (a /\SF b) false  = solve a false  <|> solve b false
solve (a \/SF b) true   = solve a true   <|> solve b true
```

For the constants, we either wanted to produce their value, in which case we do nothing, or we wanted to produce the opposite value, in which case we throw a conflict. In the variable case, we assign the variable with the aimed output of the formula. In the not case, we just invert the aim. In the positive conjunction case or its dual, negative disjunction case (which we will call the *conjunctive cases*), we solve for both `a` and `b` to have the desired values. In the two cases where we can choose which of the two formulas should have which value (which we will call the *disjunctive cases*), we branch for the concrete choice. Running this naïve version gives the output

```
--naive solve
solveTest : List (Maybe 𝔹³ or Err)
solveTest = run do
    solve exactlyOne true
    getModel
```

```
right "conflict" ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
left (just true , just false , just false) ::
left (just true , just false , just false) ::
left (just true , just false , just false) ::
left (just true , just false , just false) ::
right "conflict" ::
right "conflict" ::
left (just false , just true , just false) ::
left (just false , just true , just false) ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
left (just false , just false , just true) ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
right "conflict" :: []
```

This looks a bit wild. Not only are we visiting way more nodes than before, but we are also producing the same models several times. The reason is that even though the last two lines in the naïve `solve` function look intuitive, they actually branch too much. Let us take the `/\SF` case as an example. Here, we are first getting all models where `a` is true and then all models where `b` is true. The problem is: The models where `a` is true contain some of the models where `b` is true and vice versa. The branches should have disjunct models. We can quickly fix this by writing

```
-- slightly less naive
solve : SATFormOver (V 𝔹) → 𝔹 → LState Unit
solve trueSF  true   = return unit
solve falseSF false  = return unit
solve trueSF  false  = lift readConf
solve falseSF true   = lift readConf
solve (varSF v)    aim = write v aim >> void (read v)
solve (notSF form) aim = solve form (not𝔹 aim)
solve (a /\SF b) true   = solve a true   >>  solve b true
solve (a \/SF b) false  = solve a false  >>  solve b false
solve (a /\SF b) false  = solve a false  <|> (solve a true  >> solve b false)
solve (a \/SF b) true   = solve a true   <|> (solve a false >> solve b true )
```

Where we say that in the second branch, the models of the first branch should be excluded. Further, we also read after every write in order to throw a conflict directly as it happens. The output of running this like before is

```
right "conflict" ::
left (just true , just false , just false) ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
left (just false , just true , just false) ::
right "conflict" ::
right "conflict" ::
right "conflict" ::
left (just false , just false , just true) ::
right "conflict" ::
right "conflict" ::
right "conflict" :: []
```

Now, we do not have any module doubles anymore and we run into fewer conflicts. Still, there should only be 8 nodes to be visited, and here we visited 13 of them. The source for this excess branching in the case of conflicts is in the fact that in the disjunctive cases, we are branching even though the model might already be assigned enough for no branching to be necessary. We apply the same trick as in the `assignmentOfSafe` function to write a proper solving function

```
solve : SATFormOver (V 𝔹) → 𝔹 → LState Unit
solve trueSF  true   = return unit
solve falseSF false  = return unit
solve trueSF  false  = lift readConf
solve falseSF true   = lift readConf
solve (varSF v)    aim = write v aim >> void (read v)
solve (notSF form) aim = solve form (not𝔹 aim)
solve (a /\SF b) true   = solve a true   >> solve b true
solve (a \/SF b) false  = solve a false  >> solve b false
solve (a /\SF b) false  = do
    a? ← safeRead (assignmentOfSafe a)
    b? ← safeRead (assignmentOfSafe b)
    case (a? , b?) of
        \{ (just x , just y) → unless' (not𝔹 (x and𝔹 y)) (lift readConf)
        ;  (just false , _ ) → return unit
        ;  ( _ , just false) → return unit
        ;  (just true , _ )  → solve b false
        ;  ( _ , just true)  → solve a false
        ;  (nothing , nothing) → solve a false  <|> (solve a true >> solve b false)
solve (a \/SF b) true  = do
    a? ← safeRead (assignmentOfSafe a)
    b? ← safeRead (assignmentOfSafe b)
    case (a? , b?) of
        \{ (just x , just y) → unless' (x or𝔹 y) (lift readConf)
        ;  (just true , _ )  → return unit
        ;  ( _ , just true)  → return unit
        ;  (just false , _ ) → solve b true
        ;  ( _ , just false) → solve a true
        ;  (nothing , nothing) → solve a true  <|> (solve a false >> solve b true) }
```

Code 3.8:44

that gives the output

```
solveTest : List (Maybe 𝔹³ or Err)
solveTest = run do
    solve exactlyOne true
    getModel
```

```
left (just true  , just false , just false) ::
left (just false , just true  , just false) ::
left (just false , just false , just true ) :: []
```

where we do not run into any conflicts and only produce the three desired models. We will not prove this, but for our SAT example, the developed technique is equivalent to the DPLL procedure [32].

3.8.3.4 Generalisation of the SAT-Example

Code Tag: `SATSolvingExample`

Of course this thesis does not just reinvent the wheel. SAT-Solving is just a solving procedure that is picked for didactic purposes. We will give a short hint as to how this solving procedure generalises.

Every algebraic data type can be represented as a fixpoint `Fix` over a strictly positive functor `F`, resulting in a (weakly) initial algebra `In : F Fix → Fix`. We explain this in more detail in Chapter 4 Section 4.2. Further, every transformation between algebraic data types can be expressed via the fold over an algebra `F X → X`. That means that if we can represent such a fold operation on our SAT-formulas, we can express their evaluation function with it. We will show at an example that this expression of the evaluation function is enough to also represent a solver, giving an example of how every evaluation function can be turned into a solver.

We start by creating the functor for Boolean expressions. The direct approach would be to create the data type

```
-- will be slightly adjusted later
data SATFormOverF (V R : Set) : Set where
    trueSFF  : SATFormOverF V R
    falseSFF : SATFormOverF V R
    varSFF   : V → SATFormOver V
    notSFF   : R → SATFormOverF V R
    _/\SFF_  : R → R → SATFormOverF V R
    _\/SFF_  : R → R → SATFormOverF V R
```

where `V` is the type of variables and `R` is the type of recrusive call. From this data type, we can define a fold:

```
-- will be deprecated later
foldSF : (SATFormOverF V X → X) → SATFormOver V → X
foldSF alg trueSF     = alg trueSFF
foldSF alg falseSF    = alg falseSFF
foldSF alg (varSF v)  = alg (varSF v)
foldSF alg (notSF a)  = alg (notSFF (foldSF alg a))
foldSF alg (a /\SF b) = alg ((foldSF alg a) /\SFF (foldSF alg b))
foldSF alg (a \/SF b) = alg ((foldSF alg a) \/SFF (foldSF alg b))
```

Having an extra case for variables however will turn out to be cumbersome later. The behaviour of the variable in the algebra should always be the same: It should get a value from some assignment state. Having it as a case in the functor however allows for plenty other behaviour. Therefore, we will focus on a different functor for the fold: one where the data itself does not know that it is potentially split up by variables:

```
data SATFormOverF (R : Set) : Set where
    trueSFF  : SATFormOverF R
    falseSFF : SATFormOverF R
    notSFF   : R → SATFormOverF R
    _/\SFF_  : R → R → SATFormOverF R
    _\/SFF_  : R → R → SATFormOverF R
```

We can now no longer fold without calling the monad for the contents of the state, but we can define a monadic fold:

```
foldLS : (SATFormOverF X → X) → SATFormOver (V X) → LState X
foldLS alg trueSF     = ⟪ (alg trueSFF) ⟫
foldLS alg falseSF    = ⟪ (alg trueSFF) ⟫
foldLS alg (varSF v)  = read v
foldLS alg (notSF a)  = ⟪ (alg ∘ notSFF) (foldLS alg a) ⟫
foldLS alg (a /\SF b) = ⟪ alg ⟪ (foldLS alg a) /\SFF (foldLS alg b) ⟫ ⟫
foldLS alg (a \/SF b) = ⟪ alg ⟪ (foldLS alg a) \/SFF (foldLS alg b) ⟫ ⟫
```

With this fold we can already create a version of the evaluation function for Boolean formulas. We will do so using the algebra

```
assignmentAlg : SATFormOverF 𝔹 → 𝔹
assignmentAlg trueSFF     = true
assignmentAlg falseSFF    = false
assignmentAlg (notSFF a)  = not𝔹 a
assignmentAlg (a /\SFF b) = a and𝔹 b
assignmentAlg (a \/SFF b) = a or𝔹  b
```

We will show by example how this algebra is the only thing a user needs to provide in order to create a solver with our aforementioned optimisations. To do that however we need to define a few better versions of the monadic folds.

With the `foldLS` fold we have the problem that we cannot define a function like `assignmentOfSafe` that does not create an error in the state unless necessary, so running an evaluation function expressed using `foldLS` on a function that should say the formula is assigned to `false` gives instead the error

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v1 true
    write v2 true
    foldLS assignmentAlg exactlyOne
```

```
right "read error" :: []
```

With `foldLS`, if any read fails, regardless of whether its value was relevant, the whole evaluation fails. If we constrain ourselves to only (slightly) constrained algebras, we can overcome that problem. In the Boolean case (and, technically, any finite data type), this can be automated. We will show how to do that at the example of algebras over the Booleans.

First, we define the two functions

```
irrelevant : (𝔹 → 𝔹) → 𝔹
irrelevant f = f true == f false

irrelevant2 : (𝔹 → 𝔹 → 𝔹) → 𝔹
irrelevant2 f =
    f true  true  == f true  false and𝔹
    f true  false == f false true  and𝔹
    f false true  == f false false and𝔹
    f false false == f true  true
```

These functions check for 1- and 2-ary (Boolean) functions whether an input to them is relevant for the output. If the output stays the same under all possible inputs, the input is not relevant. We can now determine what it means to safely apply a function in a (monadic) context, meaning that no value is queried that would not be relevant to the computation, catching possible unassignment errors on the way:

```
safeApplyFunc𝔹 : (𝔹 → 𝔹) → LState 𝔹 → LState 𝔹
safeApplyFunc𝔹 f a = do
    a? ← safeRead a
    case a? of
        \{ (just x) → return (f x)
         ;  nothing → if irrelevant f
                then return (f true)
                else ⦇ f a ⦈ }

safeApplyFunc2𝔹 : (𝔹 → 𝔹 → 𝔹) → LState 𝔹 → LState 𝔹 → LState 𝔹
safeApplyFunc2𝔹 f a b = do
    a? ← safeRead a
    b? ← safeRead b
    case (a? , b?) of
        \{ (just x , just y)   → return (f x y)
         ; (just x , nothing)  → if irrelevant (f x)
                then return (f x true)
                else ⦇ f a b ⦈
         ; (nothing , just x)  → if irrelevant (\y → f y x)
                then return (f true x)
                else ⦇ f a b ⦈
         ; (nothing , nothing) → if irrelevant2 f
                then return (f true true)
                else ⦇ f a b ⦈ }
```

We first try to get the values for the arguments to the function. If all values are present, we can return the result of the function, If individual values are

missing we check whether the remaining function depends on its input. If not, we return any value. If it does depend we redo the monadic computation so that the original errors are being thrown.

With this safe application of functions, we can define a fold operation that we can use to recreate our previous evaluation functions. We parametrise it over the concrete read operation so that we can switch between a normal read and a branching read:

```
foldBLState :
    (V 𝔹 → LState 𝔹) →
    (SATFormOverF 𝔹 → 𝔹) →
    SATFormOver (V 𝔹) → LState 𝔹
foldBLState rd alg trueSF     = ⦇ (alg trueSFF)  ⦈
foldBLState rd alg falseSF    = ⦇ (alg falseSFF) ⦈
foldBLState rd alg (varSF v)  = rd v
foldBLState rd alg (notSF a)  =
    safeApplyFuncB
    (alg ∘ notSFF)
    (foldBLState rd alg a)
foldBLState rd alg (a /\SF b) =
    safeApplyFunc2B (\x y → alg (x /\SFF y))
    (foldBLState rd alg a)
    (foldBLState rd alg b)
foldBLState rd alg (a \/SF b) =
    safeApplyFunc2B (\x y → alg (x \/SFF y))
    (foldBLState rd alg a)
    (foldBLState rd alg b)
```

Here, we apply the function induced by the algebra over the respective constructors safely in the monadic context. Running this over the evaluation algebra gives the desired results:

```
assignmentOfTest : List (𝔹 or Err)
assignmentOfTest = run do
    write v1 true
    write v2 true
    foldBLState read assignmentAlg exactlyOne
```

```
left false :: []
```

And we can turn this into a generate-and-test solver just by exchanging the argument of how a read is performed:

```
assignmentOfTest : List (Maybe 𝔹³ or Err)
assignmentOfTest = run do
    res ← foldBLState branchingRead assignmentAlg exactlyOne
    when' (not𝔹 res) (throw "wrong output")
    getModel
```

```
right "wrong output" ::
right "wrong output" ::
right "wrong output" ::
left (just true , just false , just false) ::
right "wrong output" ::
left (just false , just true , just false) ::
left (just false , just false , just true) ::
right "wrong output" :: []
```

If we say that the fold over an algebra creates a function `f`, we want to, in general, create a solver for the equation `f x = y`, where `y` is given and `x` is searched for. In Code 3.8:44, we showcased for the concrete example of solving for models of Boolean functions how using writes to a desired value instead of branching reads can lead to a significant reduction in branching.

We can now create a fold for the general case that does this more precise solving by parametrising it over the type of write operation that writes the desired output values into the variables directly. To do that, we first need some functions to create the image of an algebra at a certain desired output:

```
_imageOf_ : (𝔹 → 𝔹) → 𝔹 → List 𝔹
f imageOf aim = do
    x ← true :: false :: []
    guard (f x == aim)
    return x

_imageOf2_ : (𝔹 → 𝔹 → 𝔹) → 𝔹 → List (𝔹 × 𝔹)
f imageOf2 aim = do
    x ← true :: false :: []
    y ← true :: false :: []
    guard (f x y == aim)
    return (x , y)
```

Here, we create the images of 1- and 2-ary Boolean functions for a given input. The solver would branch over that image in order to create (potentially) all possible states creating the desired output. However, we want to branch as little as possible. The first step to do so is to transform the image in a way so that it only contains the information that we need to run a solver on in the first place. When we have some function inputs present, we only want the solver to solve on image values that are consistent with the current inputs. So we filter the image by the given inputs and setting those parts of the image to `nothing` that the solver does not need to run on anymore.

```
filterImage : Maybe 𝔹 → List 𝔹 → List (Maybe 𝔹)
filterImage (just a) = map (\_ → nothing)
filterImage nothing  = map just

filterImage2 : Maybe 𝔹 → Maybe 𝔹 → List (𝔹 × 𝔹) → List (Maybe 𝔹 × Maybe 𝔹)
filterImage2 (just a) (just b) = map (\{(_ , _) → (nothing , nothing)}) ∘ filter (\{(x , y) → x == a and𝔹 y == b})
filterImage2 (just a)  nothing = map (\{(_ , y) → (nothing , just y )}) ∘ filter (\{(x , y) → x == a})
filterImage2  nothing (just b) = map (\{(x , _) → (just x  , nothing)}) ∘ filter (\{(x , y) → y == b})
filterImage2  nothing  nothing = map (\{(x , y) → (just x  , just y)})
```

If any value for the function is present, we filter the image such that it only contains the entries consistent with the current observation. That way we will not branch into state that will create a conflict anyway. However, even this is not quite enough. If you compare this to our hand-crafted solver in Code 3.8:44, even when there is no information present whatsoever, we have to be careful about not creating the same state twice by branching the solver with overlapping possible states, while at the same time running the solver only as little as possible to avoid excessive branching on states that will create a conflict.

Therefore, we further reduce the image in two steps. First, we use the Quine McClusky Algorithm [72] to find a smallest set of prime implicants to rerpesent the image after filtering. Then we fill consecutive the don't-care values with th opposite of previous assignments, so that we do not get any assignments twice. We filter out complete don't-care-clauses, but leave one of them in in case they would be the only clause. This makes sure that if the image is empty we can return a conflict, while if all values satisfy the constraints, we just don't solve for anything.

```
minimize : List (Maybe 𝔹) → List (Maybe 𝔹)
minimize (just true  :: just false :: []) = nothing :: []
minimize = id

minimize2 : List (Maybe 𝔹 × Maybe 𝔹) → List (Maybe 𝔹 × Maybe 𝔹)
minimize2 lst =
    ( AddDon'tCareClauseIfNecessary
    ∘ AddInvertsToDon'tCare
    ∘ filterEmpty
    ∘ QuineMcClusky)
    lst
```

In the first minimization function we are just checking whether the value put into the function matters. In the second minimization function for function inputs with two variables, we implement the algorithm described above. Details of the function's implementations are the following:

```
AddDon'tCareClauseIfNecessary = (\ lst' →
    if (not𝔹 (null lst)) and𝔹 null lst' then (nothing , nothing) :: [] else lst')

AddInvertsToDon'tCare = flip foldr [] (\{(a , b) lst' →
        (a , b) ::
        map (\{(a' , b') →
            fromMaybe (mapMaybe not𝔹 a) (mapMaybe just a') ,
            fromMaybe (mapMaybe not𝔹 b) (mapMaybe just b')}) lst'
    })

filterEmpty = filter (\{(a , b) → not𝔹 (is-nothing a and𝔹 is-nothing b)})

QuineMcClusky = \lst → map (
    \{(a , b) →
        (if any (\{(a' , b') → (a' == (mapMaybe not𝔹 a)) and𝔹 (b == b')}) lst then nothing else a) ,
        (if any (\{(a' , b') → (b' == (mapMaybe not𝔹 b)) and𝔹 (a == a')}) lst then nothing else b)})
        lst
```

The only interesting part here is the Quine McClusky algorithm where we check whether there are two clauses that differ by only one variable, in which case the differing variable is set to a don't-care value. In the `AddInvertsToDon'tCare` function we go through the current image as a list in order, and if a value has been set in a previous clause that is a don't-care-value in a consecutive clause, than the consecutive clause sets the value to the inverse in order to avoid solving for the assignment twice.

As this is just an example, we quickly showcase the the function behaves as intended:

```
minimize2 (filterImage2 (just true) nothing $ _orBool_ imageOf2 true)
    = (nothing , nothing) :: []
```

If we want to set a disjunction to `true` and already know that one of the inputs is `true`, the solver does not need to continue solving for anything, so all the solving calls are don't-care-values. If there however needs to be a solving call as in

```
minimize2 (filterImage2 (just false) nothing $ _orBool_ imageOf2 true)
    = (nothing , just true) :: []
```

then we know we have to solve for the second input to be `true`. If we have no knowledge whatsoever, we get the same information needed for the solving call as in Section 3.8.3.3, where we branch between two possible inputs that create distinct solutions.

```
minimize2 (filterImage2 nothing nothing $ _orBool_ imageOf2 true)
    = (just true , nothing) :: (just false , just true) :: []
```

Now, we have all the information about the image of a boolean function at an arbitrary degree of knowledge about its inputs. We can use this to create the SAT-Solver extracted from an evaluation algebra.

Side Note 3.8:45

Of course, in the general case, we should also check whether the order in which we solve makes a difference, but we will not do that here to keep the example simple.

Now, given the two convenience functions for branching over a list of possibilities and only running a monadic action on a `just` value

```
disjunctMap' : (X → LState Unit) → List X → LState Unit
disjunctMap' f [] = lift readConf
disjunctMap' f (x :: []) = f x
disjunctMap' f (x :: xs) = f x <|> disjunctMap' f xs

whenJust : Maybe A → (A → LState Unit) → LState Unit
whenJust (just a) f = f a
whenJust nothing  f = return unit
```

we can create the solving fold as

```
solveBLState :
    (V 𝔹 → 𝔹 → LState Unit) →
    (SATFormOverF 𝔹 → 𝔹) →
    SATFormOver (V 𝔹) → 𝔹 → LState Unit
solveBLState wrt alg trueSF     aim = unless' (alg trueSFF  == aim) (lift readConf)
solveBLState wrt alg falseSF    aim = unless' (alg falseSFF == aim) (lift readConf)
solveBLState wrt alg (varSF v)  aim = wrt v aim
solveBLState wrt alg (notSF a)  aim = do
    a? ← safeRead (foldBLState read alg a)
    disjunctMap'
        (\a-aim → whenJust a-aim (solveBLState wrt alg a))
        (minimize $ filterImage a? ((alg ∘ notSFF) imageOf aim))
solveBLState wrt alg (a /\SF b) aim = do
    a? ← safeRead (foldBLState read alg a)
    b? ← safeRead (foldBLState read alg b)
    disjunctMap'
        (\{(a-aim , b-aim) → do
            whenJust a-aim (solveBLState wrt alg a)
            whenJust b-aim (solveBLState wrt alg b) })
        (minimize2 $ filterImage2 a? b? $ (\a b → alg (a /\SFF b)) imageOf2 aim)
solveBLState wrt alg (a \/SF b) aim = do
    a? ← safeRead (foldBLState read alg a)
    b? ← safeRead (foldBLState read alg b)
    disjunctMap'
        (\{(a-aim , b-aim) → do
            whenJust a-aim (solveBLState wrt alg a)
            whenJust b-aim (solveBLState wrt alg b) })
        (minimize2 $ filterImage2 a? b? $ (\a b → alg (a \/SFF b)) imageOf2 aim)
```

This looks cumbersome, but in each case we are doing the exact same thing. We branch over the image of the algebra for the current aim, filtering the branches for the current assignment and minimize the recursive calls to the solver for each case in the image. Running this with writes that have consecutive reads for faster conflict occurrences yields, as expected:

```
solveTest : List (Maybe 𝔹³ or Err)
solveTest = run do
    solve𝔹LState
        (\v x → write v x >> void (read v))
        assignmentAlg
        exactlyOne true
    getModel
```

```
left (just true , just false , just false) ::
left (just false , just true , just false) ::
left (just false , just false , just true) :: []
```

Where we do not run into any conflict, meaning that each node we visit is also an actual solution. Of course, this will not be the case for general examples for SAT-formulas, but it does showcase how we have a significant performance improvement over a simple generate and test approach.

Also note how we have just created a solver out of an arbitrary Boolean algebra used to define an evaluation function. This gives a strong example of how every evaluation function can be turned into a solver.

3.8.3.5 Upscaling the SAT-Example

Of course, the SAT-Solving example from Section 3.8.3.3 and 3.8.3.4 works for more than just three variables. We can also run the solver on 5 or more variables as in the following example:

```
solveTest : List (Maybe 𝔹⁵ or Err)
solveTest = run do
    solve𝔹LState
        (\v x → write v x >> void (read v))
        assignmentAlg
        exactlyOne true
    getModel
```

```
left (just true  , just false , just false , just false , just false) ::
left (just false , just true  , just false , just false , just false) ::
left (just false , just false , just true  , just false , just false) ::
left (just false , just false , just false , just true  , just false) ::
left (just false , just false , just false , just false , just true ) :: []
```

Of course, for bigger number of variables it may become slow to give all possible outputs as we are solving a counting problem for SAT here. Further, as this is only an example to showcase where the technology can lead, we did not optimize the code for runtime. If this is used as a SAT-Solver, we can search for only the first solution, as in

```
solveTest : List (Maybe 𝔹ⁿ or Err)
solveTest = take 1 $ run do
    write (v_{n/2}) true
    solve𝔹LState
        (\v x → write v x >> void (read v))
        assignmentAlg
        exactlyOne true
    getModel
```

In order to keep this example interesting, we are letting the solver search for a solution where not just the first variable can be set to true immerdiately, but some variable $v_{n/2}$ in the middle of the `n` variables is set to true. The output for `n = 10` with `n/2 = 5` looks like

```
left
(just false ,
 just false ,
 just false ,
 just false ,
 just true  ,
 just false ,
 just false ,
 just false ,
 just false ,
 just false )
:: []
```

This runs in under one minute for up to 70 variables for the manually created solver from Section 3.8.3.3 and for up to 30 variables for the automatically created solver from Section 3.8.3.4 on completely unoptimized code. Nedless to say is that the generate and test approach on the pure list monad timouts on these numbers. The reason that the automatically created solver is slower than the manually created one is because it has to generate the images of the given algebra anew for every constructor in the input formula. This can of course be optimized away using inlining.

3.8.4 Summary

In this section, we have achieved the following:

- Created a model for what variables are in a monadic context, espceically what it means to write to them (assign them to a value).
- Formalized and generalised the result in [61], so that in our developed constrained state monad transformer, we can create a monadic variable out of every bi-threshold function.
- Created a SAT-Solver as an example that uses the writes to reduce branching
- Showcased, by the example of Boolean algebras, how every function `f` between algebraic data types can be used to create a solver for the equation `f x = y`

3.9 Creating New Variables

Code-Tag: `ICStateT`

In Section 3.8.3.3 we created a SAT solver over a formula with a fixed number of variables. In general, however, we would like to be more flexible about how many variables we want to create, possibly even add new variables depending on the output of a solver. An example of where this would become useful is solving for general algebraic data types like the lists in Section 3.5. Here, we in general do not know how many constructors and their respective variables we need until we know the exact shape of the object. If the list's length varies between models, it is convenient for each model to have a different number of variables. But even for the SAT case creating new variables during the solving process can be beneficial for techniques that would otherwise run into memory problems depending on the size of the model [49, 51, 56, 87].

Creating a new variable that is independent of all existing variables is, when formalised, not as trivial as it might seem. Take the product construction of the lattice bi-threshold variables from Lemma 3.8:34. To recap its type, we have

```
LBVarProduct :
    (v1 : LBVar S1 L1 X) →
    (v2 : LBVar S2 L2 Y) →
    (LBVar (S1 × S2) (L1 -xBSLx- L2) X × LBVar (S1 × S2) (L1 -xBSLx- L2) Y)
```

Code-Tag: `BiThresholdVariables`

This means that we are getting two variables over the two separate domains `S1` and `S2`, to create two variables over the shared domain `S1 × S2` via the product construction for lattices `L1 -xBSLx- L2`. Therefore, the shape of the state these variables operate on has to change, so if we just want to add a new variable to a state via the product (to ensure independence), we have to add its base lattice to the current state, like

```
LBVarProductLeft :
    (v1 : LBVar S1 L1 X) →
    (LBVar (S1 × S2) (L1 -xBSLx- L2) X)
```

so that the shape of the state we operate on now also has to change from `S1` to `S1 × S2`.

Currently, our constrained state monad transformer cannot change the shape of its state, making the creation of new variables difficult. The general solution for these kinds of problems are indexed monads [88], so we start by creating an indexed version of our constrained state monad transformer.

3.9.1 Indexed Constrained State Monad Transformer

Code-Tag: `ICStateT`

We can create an indexed version of the constrained state monad transformer as follows:

Definition 3.9:1: Indexed Constrained State Monad Transformer

Given an index type `I` with an interpretation `S : I → Set` and a relation `_P_ : {i j : I} → S i → S j → Set`, together with an indexed monad `M : M : J → J → Set → Set` over the index `J`, we define the **Indexed Constrained State Monad Transformer** as the type

```
ICStateT : (I × J) → (I × J) → Set → Set
ICStateT (i , j) (i' , j') X = (s : S i) → M j j' (∃[ s' of S i' ] (s P s') × X)
```

The indices `i i' : I` mark, together with the interpretation `S`, she shape of the current and the next state. The indices `j j' : J` are the indices of the underlying indexed monad, however, they are here only included for generality and will not be used throughout this thesis. To keep things simple, it is important that the indices of type `I` for the states are not changed by the underlying monad (though there would be possible semantics here), so we are using a product to keep the indices of type `I` independent of the indices of type `J`.

This definition continues to form a monad.

Theorem 3.9:2

Given an indexed constrained state monad transformer `ICStateT` from Definition 3.9:1, together with the property that `_P_` forms an indexed pre-order that forms an indexed monoid with reflexivity and transitivity, then `ICStateT` forms an indexed monad.

Proof 3.9:3

Structurally equivalent to the proof of Theorem 3.8:10.

The definition also continues to form an indexed monad transformer for the, slightly non-standard, tuple-based definition of an indexed monad transformer.

Definition 3.9:4: **Indexed Monad Transformer**

Given the index types `I` and `J`, an indexed monad `M : J → J → Set → Set` and another indexed monad `T : (I × J) → (I × J) → Set → Set` together with an operation

```
lift : M j j' A → T (i , j) (i , j') A
```

that satisfies the following laws:

```
lift-return  : lift ∘ return_M  ≡ return_T
lift-bind    : lift (m >>=_M f) ≡ (lift m) >>=_T (lift ∘ f)
```

then T forms an **indexed monad transformer**

Obviously, our indexed constrained state monad transformer also forms an indexed monad transformer

Theorem 3.9:5

Given an indexed constrained state monad transformer `ICStateT` from Definition 3.9:1, together with the property that `_P_` forms an indexed pre-order that forms an indexed monoid with reflexivity and transitivity, then `ICStateT` forms an indexed monad transformer.

Proof 3.9:6

Structurally equivalent to the proof of Theorem 3.8:12.

What does change though is how we lift an error operation. In our definition, the lift is not supposed to change the indices of the monad, however, a left absorbing monadic action (a.k.a. an error) can change the indices arbitrarily. This means that lifting an error action no longer stays left absorbing. We fix this by giving a specific operation to lift error actions

Lemma 3.9:7

Given a left-absorbing monadic action

```
empty : M j j' A
```

Then the monadic action

```
liftErr : ICStateT ij ij' A
liftErr s = empty
```

is also left-absorbing.

Proof 3.9:8

We prove the statement for all left absorbing actions `empty`.

```
preserves-left-absorb :
    ∀ (empty : {A : Set}{j j' : J} → M j j' A)(m' : A → (ICStateT ij ij' B)) →
    (left-absorb : ∀ {A B : Set}{j j' : J} {m' : A → M j j' B} → empty >>= m' ≡ empty) →
    (liftErr empty) >>=c m' ≡ (liftErr empty)
```

We prove this by extensionality over the state `s`, so

```
(liftErr empty >>=C m') s =<>

(do
    (s1 , p1 , a) ← liftErr empty s
    (s2 , p2 , b) ← m' a s1
    return (s2 , transitive p1 p2 , b)
) =<>

(do
    (s1 , p1 , a) ← empty
    (s2 , p2 , b) ← m' a s1
    return (s2 , transitive p1 p2 , b)
) =< left-absorb >

empty =<>

(liftErr empty) s qed
```

We first unpack the monadic bind of the transformer. Then we use the left-absorption property to remove any subsequent computation. Then we apply the definition of `liftErr` backwards.

With this infrastructure in place, we can recreate the error actions, reads and writes from Theorem 3.8:39.

3.9.2 Lattice Based Indexed Constrained Monad Transformers

Code-Tag: `ILatticeStates`

Next, we have to make the indexed constrained monad transformers run on the pre-order induced by lattices again. This time however, it is a bit tricky because there is no unique merge operation between two lattices of a different type and hence no clear induced pre-order. As an example, take the two types

```
(A × B)     (B × C)
```

that we assume have a lattice instance. What type would the merge result in? We could result into a shape

```
(A × B × B × C)
```

where we put both lattices just next to each other, but what if we want both shapes to overlap on the `B`? This would form the type

```
(A × B × C)
```

That we could also find a trivial lattice instance for (given that both `B` use the same lattice instance). It is not clear directly how that merge should occur. We have to be specific about which parts of the lattice should overlap and which parts should just be added. To achieve that specificity, we introduce the concept of a *lattice injection*

3.9.2.1 Lattice Injections

Before we start on what an injection on a lattice is, we quickly get on the same page about our definition of an injection.

Definition 3.9:9: **Computable Injections**

Given two types `A` and `B`, the two operations

```
inf : A → B
outf : B → A
```

form a **computable injection** (or short, injection when clear from the context) iff the following law holds:

```
outfoinf-id : outf ∘ inf ≡ id
```

The main reason we need such a computable injection is to a: automatically inject (and retreive) an old state into a new state with (possibly bigger) shape and b: to ensure that all functions on the old state still return the same value when applied to the new state (after retrieving the portion the function was originally applied to). Thw following trivial lemma holds

Lemma 3.9:10

Given a computable injection between type `A` and `B`, then forall functions `f : A → C` it holds that

```
preserves-function : f ∘ outf ∘ inf ≡ f
```

Proof 3.9:11

By applying `f ∘_` to `outfoinf-id`.

Which means that if we use the transformations from the injection to lift the function on run on the target type of the injection its values stays the same. Further, and this is just useful, computable injections give us a pre-order for types.

Lemma 3.9:12

The type of computable injections given by

```
record Injection (A : Set) (B : Set) : Set where
    field
        inf : A → B
        outf : B → A
        outfoinf-id : outf ∘ inf ≡ id
```

forms a pre-order on types.

Proof 3.9:13

Trivial by proving that injections are reflexive and transitive.

In the context of lattices, we care about injections that preserve the lattice order.

Definition 3.9:14: **Semilattice Injection**

Given two semilattices over the types X and Y, a computable injection from X to Y is a **semilattice injection** iff the following two laws hold:

```
pres-inf  : inf  (a <>ₓ b) ≡ (inf  a <>ᵧ inf  b)
pres-outf : outf (a <>ᵧ b) ≡ (outf a <>ₓ outf b)
```

With a semilattice injection, we make sure that also lattice properties are preserved during the injection. We will soon need this to show that the relation for the indexed constrained state monad transformer remains a pre-order. We prove a few useful properties about semilattice injections

Lemma 3.9:15

Given a semilattice injection from X to Y, then the injection preserves the semilattice order, meaning

```
pres-inf-rel  : a P_X b → (inf a) P_Y (inf b)
pres-outf-rel : a P_Y b → (outf a) P_X (outf b)
```

Proof 3.9:16

As both proofs are analogous, we only prove the first statement. When evaluated, we prove

```
pres-inf-rel  : a <>_X b ≡ b → (inf a) <>_Y (inf b) ≡ (inf b)
pres-inf-rel {a} {b} aPb =
    (inf a) <>_Y (inf b) =< pres-inf >
    inf (a <>_X b)       =< aPb >
    inf b                qed
```

Further, semilattice injections form a pre-order on semilattices

Lemma 3.9:17

The type of semilattice injections given by

```
record SemilatticeInjection
    (bslX : Semilattice X)
    (bslY : Semilattice Y) : Set where
    field
        injection : Injection X Y
        pres-inf  : inf  (a <>_X b) ≡ (inf  a <>_Y inf  b)
        pres-outf : outf (a <>_Y b) ≡ (outf a <>_X outf b)
```

(removing some module-magic) forms a pre-order on semilattices

Proof 3.9:18

Reflexivity is trivial as it follows from the reflexivity of injections, with the preservation contents following from evaluation of the identity function. The transitive case is slightly trickier. It too follows from the transitivity of injections and then just applying the `pres-inf` and `pres-outf` properties of each injection. Spelled out for the injections from `X` to `Y` and `Y` to `Z`, we get

```
inf_YZ (inf_XY (a <>_X b))              =< pres-inf_XY >
inf_YZ (inf_XY a <>_Y inf_XY b)          =< pres-inf_YZ >
inf_YZ (inf_XY a) <>_Z inf_YZ (inf_XY b)   qed
```

with `pres-inf` being proven analogously to `pres-outf`.

In order to use these semilattice injections for products, we need to get one step further and define a semilattice injection for bounded semilattices. Surprisingly, it needs one additional law.

Definition 3.9:19: **Bounded Semilattice Injection**

A semilattice injection from `X` to `Y` is called a **bounded semilattice injection** iff the following law holds:

```
pres-inf-e : inf (e_X) ≡ e_Y
```

Side Note 3.9:20

Surprisingly, this additional condition does not follow from injectivity and the preservation of the lattice operation. When during the injection we always add some constant element `x` with `outf x = e`, all laws for the semilattice injection still hold. This is undesirable when also trying to preserve the semantics of the identity element.

Luckily, `outf` now automatically also preserves the identity element.

Lemma 3.9:21

Given a bounded semilattice injection from X to Y, it holds that

```
pres-outf-e : outf (eᵧ) ≡ eₓ
```

Proof 3.9:22

```
outf (eᵧ)      =< sym pres-inf-e >
outf (inf eₓ) =< outfoinf-id >
eₓ             qed
```

Bounded semilattice injections stay a pre-order.

Lemma 3.9:23

The type of bounded semilattice injections

```
record BoundedSemilatticeInjection (bslX : BoundedSemilattice X) (bslY : BoundedSemilattice Y) : Set where
    field
        semilattice-injection : SemilatticeInjection (semilattice bslX) (semilattice bslY)
        pres-inf-e : inf (eₓ) ≡ eᵧ
```

(removing some module-magic) forms a pre-order.

Proof 3.9:24

Follows trivially through the semilattice injection being a pre-order. Preservation of the identity element follows trivially through evaluation and application of `pres-inf-e` on the sub-elements in the transitivity case.

Semilattice injections are useful to describe the injection into a product,

meaning the injection into a state that now has "one more variable".

Lemma 3.9:25

Given two bounded semilattices over the types `X` and `Y`, the injection given by

```
inf       y  = eₓ , y
outf (_ , y) = y
```

forms a bounded semilattice injection from `Y` into the semilattice product over `X × Y` and the injection given by

```
inf   x      = x , e_Y
outf (x , _) = x
```

forms a bounded semilattice injection from `X` into the semilattice product over `X × Y`. We call these injections `semilatticeProductSndInjection` and `semilatticeProductFstInjection`, respectively.

Proof 3.9:26

Both raw injections trivially form an injection and the preservation proofs follow from evaluation only, with the only exception that the semilattice idempotent rule has to be applied for the `pres-inf` condition.

One really convenient property of bounded semilattice injections is that we can use them to lift the context of a lattice bi-threshold variable.

Lemma 3.9:27

Given a lattice bi-threshold variable `v` into type `X` over a bounded semilattice over type `S`, together with a bounded semilattice injection from said bounded semilattice into another bounded semilattice over type `S'`, then the operations

```
read : S' → VarAsm X
read  = readᵥ v ∘ outf

write : X → S'
write = inf ∘ writeᵥ v
```

form a lattice bi-threshold variable over the bounded semilattice over type `S'` into the type `X`. We call this construction

```
liftLBVar : BoundedSemilatticeInjection bslS bslS' → LBVar S bslS X → LBVar S' bslS' X
```

Proof 3.9:28

We can prove the

```
IsBiThresholdRead (readᵥ v ∘ outf) = ∀ s s' → s P s' → (readᵥ v ∘ outf) s =incAsm= (readᵥ v ∘ outf) s'
```

property using the old variable `v` and the fact that `outf` preserves the lattice order. If we plug in

```
isBiThresholdRead v (outf s) (outfs') (pres-outf-rel sPs') : (readᵥ v ∘ outf) s =incAsm= (readᵥ v ∘ outf) s'
```

we get the desired result. For proving the `write-read` property, we have to use the original `write-read` after using the injective properties of the injection, so

```
write-read : readᵥ v (outf (inf (writeᵥ v x))) ≡ asm x
write-read {x} =
    read v (outf (inf (write v x)))   =<>
    read v ((outf ∘ inf) (write v x)) =< outfoinf-id >
    read v (write v x)                =< write-read v >
    asm x qed
```

The `read-write-read` property follows trivially through evaluation when case splitting by the result of $\texttt{read}_{\texttt{v}}$ `v (outf s)` and applying the `write-read` property in the case that the result is an assignment.

3.9.2.2 Lattice Injection Based State

We can now define the lattice-based relation for our indexed constrained state monad transformer. We define the indices to be arbitrary bounded semilattices, with the interpretation just being the type that the lattices are defined over. We have

```
I = ∃[ X ] (BoundedSemilattice X)
S = fst
```

We define the pre-order between the bounded semilattice states as follows:

Definition 3.9:29: **Semilattice Injection Relation**

Given a bounded semilattice injection between semilattices from the types `X` to `Y`, together with two elements `x : X` and `y : Y`, then `x` relates to `y` by the **semilattice injection relation** iff the following condition holds:

```
inj-directional : inf x P_Y y
```

Again, this forms a pre-order.

Lemma 3.9:30

The type of semilattice injection relations given by

```
record BoundedSemilatticeInjectionRelation
    (i i' : I)
    (x : S i) (y : S i') : Set where
    constructor <_,_>SIR
    field
        semilattice-injection : BoundedSemilatticeInjection (snd i) (snd i')
        inj-directional : inf x P_Y y
```

(removing some module-magic) forms an indexed pre-order on the interpretations S of I.

Proof 3.9:31

In principle, this proof uses the fact that the bounded semilattice injection already forms a pre-order. Proving that the injected x stays directional with y is trivial in the reflexive case, but in the transitive case we have to use the fact that the semilattice injection preserves the semilattice order.
Given a bounded semilattice injection from X to Y and from Y to Z, apart from the injections being transitive, we also have to show that

$$\mathsf{inf}_{YZ}\ (\mathsf{inf}_{XY}\ a)\ P_Z\ c$$

By unrolling the definition of the lattice order relation, we get

```
inf_YZ (inf_XY a) <>_Z c               =< inj-directional_YZ >
inf_YZ (inf_XY a) <>_Z (inf_YZ b <>_Z c) =< associative_Z >
(inf_YZ (inf_XY a) <>_Z inf_YZ b) <>_Z c =< pres-inf_YZ >
inf_YZ (inf_XY a <>_Y b) <>_Z c        =< inj-directional_XY  >
inf_YZ b <>_Z c                        =< inj-directional_YZ >
c                                      qed
```

Further, an injection into a state with a "new variable" also increases the state. We can define

Corollary 3.9:32

Given two indices `i, i'` (two different bounded semilattices), then a state `s : S i` is lower (or equal) to the state `(e_i , s)` or `(s , e_i)`, respectively according to the semilattice injection relation.

Proof 3.9:33

The proof follows trivially from the injection and lattice properties, here given a name because we will refer to the construction.

```
SIRSndInjection : BoundedSemilatticeInjectionRelation i' (i -xi- i') s (e_i , s)
SIRSndInjection = < semilatticeProductSndInjection , reflexive (snd (i -xi- i')) >SIR

SIRFstInjection  : BoundedSemilatticeInjectionRelation i  (i -xi- i') s (s , e_i')
SIRFstInjection  = < semilatticeProductFstInjection  , reflexive (snd (i -xi- i')) >SIR
```

So we are all set to create states with new variables. Now, all we need is to fix our monadic context.

To save ourselves some time, we again **postulate** the indexed monoid instance of `BoundedSemilatticeInjectionRelation` and then we have ourselves a monad as

```
ILState : (i j : I × J) → Set → Set
ILState = ICStateT BoundedSemilatticeInjectionRelation M
```

and a, now indexed, type of variables

```
IV : I → Set → Set
IV (S , bsl) = LBVar S bsl
```

which are the same lattice bi-threshold variables that we have been using in Section 3.8.3. This is good because it means that as long as our indices do not change, all properties from before still hold!

Before we get started with creating a new variable, we define a product on the indices by using the bounded semilattice product

```
_-xi-_ : I → I → I
(X , bslX) -xi- (Y , bslY) = (X × Y) , (bslX -xBSLx- bslY)
```

We can now reap the rewards of all the previous constructions by defining an operation that gives us a new variable that is independent of all previous variables.

```
new : IV i' X → ILState (i , j) ((i' -xi- i) , j) (IV (i' -xi- i) X)
new v s = return ((e bslX , s) , SIRSndInjection , liftLBVar semilatticeProductFstInjection v)
```

Before we show some correctness properties on the newly created variables, we quickly discuss a general problem when dealing with indexed variables in practice. We then formulate the condition that every newly created variables is independent from all previously created ones.

3.9.3 On the Problem of Indexed Variables

The newly introduced index on the variables makes sure that we only use variables in the context that they were created in. This prevens mistakes that can

happen for example in C-style languages when we give a reference to an undefined value into another part of the program that can fill that reference with for example malicious or erroneous content. The downside is that in our formalism, every time we create a new variable, we also change the context, meaning that no previous variables can be used anymore directly. Of course, every previous variable can be lifted into the new context with the function `liftLBVar` from Lemma 3.9:27. As of the current state of this research however, this lifting of variables has to be done manually and is not done automatically, which makes the formalism tedious when actually used in practice. Using a reflection mechanism or a form of free monads [54] of our interface in Section 3.7 could fix that issue, which we will explore in future work.

3.9.4 Independence

Code-Tag: `BiThresholdVariables`

Independence of two variables is the concept that one cannot derive anything about the value of one variable from the value of the other. For solving, this is important when wanting to show completeness. When we have several variables that should for example represent a tuple, we want to make sure that every assignment of the represented tuple can still occur with the variables. If the assignment of one variable removes any possible assignment of the other variables, then our solution space could be missing solutions. We want to have that when we put constraints between two variables, but unless we do so and just get two fresh variables, they should not interfere with each other.

In this thesis, we will **not show completeness**, but in order to make a point that creating a new variable behaves as desired, we at least show that the created variables are, in fact, independent.

To the extent of our scientific knowledge, there is no generally agreed upon definition of what it means for two variables to be independent. There are special formulations in probability theory [57] that only work on numbers, but in category theory, the best formalism just seem to be products. We will create a notion of independence to show that it is preserved when composing two functions to take the product of their domains as an input. Our aim is much more mundane though. We only want to show that a new variable can be assigned without assigning any other previous variable.

Definition 3.9:34: **Independence of Functions**

Two functions `f` `g` with codomains `B` and `C` and shared domain `A` are called **independent** iff the following condition holds

```
Independence : (f : A → B) (g : A → C) → Set
Independence f g = ∀ b c → b ∈ Image f → c ∈ Image g → ∃[ a ] ((f a ≡ b) and (g a ≡ c))
```

Code-Tag: `Independence` So forall two elements `b`

and `c` in the images of the functions, there is an element in the domain such that the functions have `b` and `c` as their outputs, respectively.

This independence is of course induced by composing two functions to run on a shared product domain

Lemma 3.9:35

For all two functions

```
f : A → B
g : C → D
```

the functions

```
f ∘ fst : (A × C) → B
g ∘ snd : (A × C) → D
```

are independent. We call this property
`functionProductIndependence f g`.

Proof 3.9:36

If we are given proofs for

```
(f ∘ fst) (a  , c') ≡ b
(g ∘ snd) (a' , c ) ≡ d
```

We can easily infer that

```
(f ∘ fst) (a , c) ≡ b
(g ∘ snd) (a , c) ≡ d
```

So `(a , c)` is the element creating the two desired images.

Independence is of course not reflexive, but it is commutative. As a result, it is also not transitive.

Lemma 3.9:37

Function independence is commutative.

Proof 3.9:38

Trivial.

This concept of independence can be directly applied to our lattice bi-threshold variables.

Definition 3.9:39: **Lattice Bi-Threshold Variable Indepencence**

Two lattice bi-threshold variables

```
v1 : LBVar S bsl X
v2 : LBVar S bsl Y
```

are called **independent** iff their respective `read` functions are independent and the following two conditions hold:

```
writev1 : read v2 s ≡ read v2 (s <> write v1 x)
writev2 : read v1 s ≡ read v1 (s <> write v2 x)
```

which we will also call the **non-interference of writes**.

We include the extra conditions to make sure that the writes to one variable also do not interfere with the outcome of the other variable. The independence of the reads alone only states that there would be a way to produce any output tuple, but it does not state that the production of the output tuple has to be creatable by the writes. The additional two conditions therefore make sure that the writes do not interfere with the reads of the other variable.

Side Note 3.9:40

It does feel like that the independence of the reads should imply the non-interference of the writes, or vice versa, that the non-interference of the writes implies the independence of the reads. Interestingly enough however, there are counterexamples for both. The independence of the reads just means that there is a way to independently assign the variables, however, a write could assign way more variables and still fulfil the lattice bi-threshold variable laws (which only state that the write should give *any* object that makes a variable output the correct value). This however changes if the write gives us a *minimal* object to assign a variable to a certain value. We will explore this idea for future work, as it will form the foundation of clause learning (hinted in Chapter 5 Section 5.8), where we need to find the smallest thresholds for a variable to have a certain value. This smallest threshold would then be the write that is used to write the desired value into the variable.
What is also interesting is that the non-interference of writes does not imply independence. This is slightly less obvious because if the variables would always return a value, then creating a value that creates the desired output pair for two variables could be obtained by giving the element that writes the two values, if both writes are minimal. However: This does not work when the result is either unassigned or a conflict. In these cases we may know a state that produces the unassigned or conflict output of a variable, but again, we do not know how to minimize that to a state where the other variable can have an arbitrary value. Here, we need an even stronger property: That even given a conflict or unassigned output, we can give a minimal object that creates that result which will then not interfere with the other read. Again, this is explored in Chapter 5 Section 5.8.

It pretty directly follows, that the product construction for lattice bi-threshold variables creates two independent variables.

Theorem 3.9:41

Given a variable product from Lemma 3.8:34

```
< v1' , v2' >LBVar ≡ (v1 , v2)
```

Then `v1` and `v2` are independent.

Proof 3.9:42

Independence of the reads follows from the function product independence of Lemma 3.9:35. Non-interference of writes follows directly from lattice properties and evaluation.

What is further useful is that lifting a variable through a bounded semilattice injection preserves independence.

Theorem 3.9:43

Given a bounded semilattice injection `seminj` from `S` to `S'` with bounded semilattices `bslS` and `bslS'`, together with two independent variables

```
v1 : LBVar S bslS X
v2 : LBVar S bslS Y
```

Then the variables

```
liftLBVar seminj v1
liftLBVar seminj v2
```

stay independent

Proof 3.9:44

We first show the independence of the read functions. From the independence of `v1` and `v2` we know that for any desired output `x` and `y`, there is an `s : S` that makes them create that output. If we now lift the variables and run them on `inf s` in the lifted state, we get that their result stays the same as

```
readv1s=x ∈ (read v1 s ≡ x) [premise]
(read v1 ∘ outf) (inf s)   =<>
(read v1 ∘ (outf ∘ inf)) s =< outfoinf-id >
read v1 s                  =< readv1s=x >
x                          qed

readv2s=y ∈ (read v2 s ≡ y) [premise]
(read v2 ∘ outf) (inf s)   =<>
(read v2 ∘ (outf ∘ inf)) s =< outfoinf-id >
read v2 s                  =< readv2s=y >
y                          qed
```

The non-interference of writes now follows from the non-interference of each variable, paired with the preservation of the lattice operation.

```
read v2 (outf s)                                  =< writev1 >
read v2 (outf s <>S write v1 x)                   =< outfoinf-id >
read v2 (outf s <>S (outf ∘ inf) (write v1 x)) =< pres-outf >
(read v2 ∘ outf) (s <>S' (inf ∘ write v1) x)      qed

read v1 (outf s)                                  =< writev2 >
read v1 (outf s <>S write v2 x)                   =< outfoinf-id >
read v1 (outf s <>S (outf ∘ inf) (write v2 x)) =< pres-outf >
(read v1 ∘ outf) (s <>S' (inf ∘ write v2) x)      qed
```

These theorems now state that the newly created variables are, in fact, independent and stay independent when they are lifted into contexts with even more variables. This however is a property of our lattice bi-threshold variables. We do not yet have a monadic interpretation of independence

3.9.4.1 Towards Dependence and Independence of Monadic Variables

We leave creating an independence criterion for monadic variables to future research. It would formulate that also for two monadic variables, we can assign one without assigning the other. The independence of the monadic variables created by two independent lattice bi-threshold variables would then be implied by the independence of the variables. Likewise, we would create a criterion for dependent monadic variables, we could be used to model sigma-type monadic variables where a different monadic variable could be used to constrain the first value of the monadic variable holding the sigma value. There should be a way to create dependent and independent new variables. This is the final correctness condition we need to properly formalise the monadic interface for references from Section 3.7

3.10 Summary and Future Research

In this chapter we have created a general concept for **monadic variables** that can be used to improve existing monads for search. Monadic variables are two monadic actions `read` and `write` that can read and write to a value, where the `read` is guaranteed to always result in the same value no matter what happens in the monad and a `write` is guaranteed to have the `read` return the written value on later states. We motivated the use of these variables by showcasing how they would improve search on recursive data structures in Section 3.4 and beyond, as they give us the opportunity to only unwrap values (or parts of values) in a monad if we actually need them. This allows searching monads to branch less based on the information of which values are actually needed to be branched on. This is the loose notion of explicit laziness and motivates the concept of **held-in-stasis data types**, which will later be formally defined in Chapter 4 Section 4.3. In order to create these monadic variables, we first introduce the concept of a **constrained state monad transformer** from Section 3.8.1 onwards that constraints the traversed states to have to be related in at least a pre-order. We used this to show that if our variables gave a certain value on one state once, they would not change their value for any greater state according to the pre-order unless a conflict occurred. To construct concrete variables, we enhanced the concept of semilattice based variables from [61, 62, 75]. We develop the concept of a bi-threshold function in Section 3.8.3.1, which is a function that reads from a semilattice, possibly returning a value or conflict, while making sure the value stays the same for an increase in the input unless a conflict occurs (from which on it then always returns a conflict). Conflicts here mean for example that two conflicting values have been assigned to the variables, like assigning a boolean variable to both `true` and `false` at the same time. We then generalised the concept of bi-threshold functions to (semi)lattice bi-threshold variables to (semi)lattice bi-threshold variables. Those variables are bi-threshold functions used for reading a value that also come with a write function. Given an element

of the reading functions codomain, the write function gives an element of the lattice that makes the reading function return the written value (or a conflict). Together with the constrained state monad transformer on the pre-order induced by a semilattice, these (semi)lattice bi-threshold variables can be used to create monadic variables, shown in Theorem 3.8:39.

We showcase the use of monadic variables at the example of a SAT-solver in Section 3.8.3.3. We even give a motivating example of a general theory of solving based on our monadic variables by creating a SAT-solver from just an algebra for evaluating a boolean function in Section 3.8.3.4.

Finally, we develop a mechanism to create new variables in Section 3.9. To do this, we generalise the constrained state monad transformer to an indexed version, allowing us to index variables with a context in which they work. Introducing a new variable also means creating bigger context, which we model through the concept of *semilattice injections* that are injections that form a pre-order on semilattices. Variables can be lifted to be in a bigger context, for example after a variable has been created, by lifting them along a semilattice injection shown in Lemma 3.9:27. The lifts we use for introducing new variables can be shown to create independent variables in Theorem 3.9:41 and 3.9:43.

For future research, the independence concept for bi-threshold variables still needs to be translated into a general independence concept for monadic variables. Further, currently, when creating a new variable, we have to lift the context of all previous variables into the new context by hand, which is undesirable. This could be fixed with a free version of the monadic variable interface presented in Section 3.7 (free as in *free monad* [54]) where we could automatically place the context lifts on variables in the places where we read them. In this case we might be able to make the indices for contexts entirely invisible to a user. In general: finishing the independence condition can give us a general interface for monadic variables as in Section 3.7 that could also lead to more efficient implementations than the one we are providing. An example could be a construction where we are using actual pointers into memory to model the monadic variables.

Further, as mentioned in Chapter 5 Sidenote 5.3:4, we need to also create an indexed version of the constrained state monad transformer so that it would not throw away the state in an error case of `M`, which is necessary to still increase the state after a failed read. If we stop increasing the state just because a computation in the end fails on a read, we might loose the progress in state increase until the read. This is not so much a problem when considering only one computation, but when several computations are run in parallel as is done in Chapter 5 Section 5.3.2, even if one program fails, it might produce information the other program needs to succeed. In this scenario, we cannot throw away the state even on a failed read.

We also need a way to represent more general data types using monadic variables, which we will do in Chapter 3. Of course, the general theory of solving from Chapter 5 can also used to create solvers running on our monadic variables, however we get closer to the idea of actually deriving solvers from evaluation algebras with the concepts in Chapter 4 Section 4.3. We will not

give a construction there, but by showing how folds work on held-in-stasis data types, it is not far to reverse the algebra to create assignments from knowing the output of a function represented as a fold.

Apart from reversing evaluation algebras, there is also a second possibility to use our monadic variables for solving. As of now we have not investigated how these variables would compose for example into a monadic instance. If `V` is our variable type and `M` some monad using `V` to create monadic variables, then it could be that there is a monadic, or maybe just near-monadic instance for the morphism `M∘ V`. This would be computation where we can *write* into the result of come computation `ma : (M∘ V) A` and the monadic variable laws would ensure that the computation then also gives the result we wrote into it. If `ma` was represented by a monadic fold operation as we will create in Chapter 4 Section 4.3, writing to it would mean that the variables that represent the input to the fold would have to be assigned to give the desired result. This would give a general construction for the example we gave in Section 3.8.3.4.

This, for now, concludes the future research on this chapter.

Chapter 4

Data Types for Monadic Solving

4.1 Introduction

In this chapter we show how to automatically transform recursive data to be used together with the concepts from Chapter 3. The main idea is that each data type is transformed into a lattice version of itself that can be accessed via the variables from Chapter 3 Section 3.8.3.1. We define how to prepare a data type to be dissected into multiple variables using a monad and how to both run functions on the dissected data and how to piece it back together afterwards. We will show how to transform data to be *monadically held-in-stasis*, meaning that the arguments to a constructor are exchanged with monadic calls to the arguments instead (which is quite close to how data is represented in a computer anyway). Further, it will be shown how to run a large class of recursive functions over this transformed data without changing the original functions implementation. We will then show how to automatically create lattice instances for recursive data and use it to create lattice bi-threshold variables into recursive data. As a representation for general recursive data, we will use the concept of *Containers* [2]. Our construction impose a few constrains on the containers, however, we will showcase how these constraints are highly likely to not prohibit expressing general type theory by giving a type for the syntax of type theory that expresses all of our wanted formalism while still not violating the constraints.

The results in this chapter can be summarised as follows:

- Creating the notion of *commutable contexts*, where data types can be monadically held-in-stasis and recursive functions can be executed over.
- Showing that all shapely containers (containers where the positions are of a finite type) form commuting contexts with any locally-commutative monad.

- Showing that all containers with decidable equality on the shapes can be transformed into an equally expressive data type with a lattice instance (called a *latticed container*)
- Showing that we can create lattice bi-threshold variables into the arguments of latticed containers, forming the basis of recursive data types being distributed via variables into a lattice based state.
- Enhancing the data type for the syntax of type theory (the "type of type theory") from [8, 24] to additionally allow sigma types and recursive functions
- Showcasing how our improved type of type theory still satisfies all constraints needed to turn it into a lattice type for solving

As this chapter heavily relies on representing general data and data types in terms of *containers* [2], we will start by giving a quick recap of the concept.

4.2 Containers

Code-Tag: `DataTypesExamples`
Code-Tag: `Containers`

Algebraic data types can be represented as fixpoints of so-called *strictly positive functors*. We will explain what that is after an example. Take the definition of lists:

```
data List (A : Set) : Set where
    []   : List A
    _::_ : A → List A → List A
```

We have two constructors `[]` (prononouced *nil*) and `_::_` (pronounced *cons*). The nil constructor constructs the empty list and the cons constructor takes an element of type `A` and a sublist to create a longer list. An alternative but isomorphic definition is to represent the list as a fixpoint to a certain functor.

```
data ListF (A : Set) (R : Set) : Set where
    []F   : ListF A R
    _::F_ : A → R → ListF A R

data ListFix (A : Set) : Set where
    InList : ListF A (ListFix A) → ListFix A
```

Here, we first create a functor that has the shape of a list, here called the *shape-functor*, but instead of the recursive call it just has a placeholder `R` that can be filled with some type. To actually create a recursive data type from this functor, we add a fixpoint type `ListFix` with a constructor that takes an element of the shape functor that (possibly) contains elements of the recursive type to create an element of the recursive type. This type really just uses the shape-functor to contain elements of its own recursive type. We can rewrite the original list constructors as

```
[] : ListFix A
[] = InList []F

_::_ : A → ListFix A → ListFix A
a :: lst = InList (a ::F lst)
```

which look the same as the old constructors, only that for every recursive list application, we have to put the `InList`. This is what makes this so great for solving: data is now represented as a composition of (almost) finitely sized parts that can be searched for individually (we'll get to a few exceptions later).

The problem with the above representation is that it does not compose. The fixpoint type really only works for lists. But what if we for example add a constructor of an unassigned list or list element like we need for the lattices in Chapter 3? We would have to create a new functor and new fixpoint. Luckily, there is a class of functors that all have a fixpoint and that can be composed. This is the class of *strictly positive functors* and they can be represented by so-called *containers* [2].

Definition 4.2:1: **Container**

Given a type `S : Set` called the *shapes*, together with a predicate `P : S → Set` called the *positions*, we call `S ▷ P` a `Container`. The

functor that is induced by a container, called the *semantic extension*, is defined as

```
[[_]] : Container → Set → Set
[[ S ▷ P ]] R = ∃[ s of S ] (P s → R)
```

A container can be roughly imagined as the shapes being the individual constructors and the positions being a type that indexes the holes that, above, where the placeholder for the recursive type `R`. There should be exactly as many positions as there are calles to `R` in the functor.

Let us quickly look at how a list would be represented as a container. We should have two different shapes: One for the nil case and one for the cons. The nil case should not have any holes, so the positions at the nil constructor should be the empty type `Zero`. The cons case has one element of some type, together with one hole, so its positions should be the unit type. Therefore, the list functor from above as a container looks like

```
ListC : Set → Container
ListC A = (Unit or A) ▷ fromSum (\_ → Zero) (\_ → Unit)
```

and we can recreate the functor constructors from above as

```
[]F : [[ ListC A ]] R
[]F = left unit , \()

_::F_ : A → R → [[ ListC A ]] R
a ::F r = right a , \{unit → r}
```

where in the nil case, we give the unit as the shape and do not have to provide any positions because there are no recursive calls in the nil case. In the cons case, we memorise the element `a`, together with giving one recursive call. Before we get to the construction of the fixpoint, we will play around with the definition of a container and show that its extension is indeed a functor and its closure

properties.

Lemma 4.2:2

The semantic extension of a container `C` is a functor with the functor map operation

```
map : (A → B) → [[ C ]] A → [[ C ]] B
map f (s , p) = s , f ∘ p
```

Proof 4.2:3

By extensionality over the element `(s , p) : [[ C ]] A`, it holds that the map preserves the identity

```
map id (s , p) =<>
(s , p ∘ id )  =<>
(s , p)        qed
```

and function composition

```
map (f ∘ g) (s , p)      =<>
(s , f ∘ g ∘ p)          =<>
(map f ∘ map g) (s , p) qed
```

4.2.1 Container Closures

Containers are closed under a range of useful base types. As these constructions are unintuitive at first, we will go through those one by one.

```
ZeroC : Container
ZeroC = Zero ▷ \()
```

The empty container is a container whose functor elements cannot be created. Therefore, it has no shapes and ergo no positions.

```
KC : Set → Container
KC X = X ▷ (\_ → Zero)

UnitC : Container
UnitC = KC Unit
```

The constant container is made up of exactly one element of a given type without any positions. Therefore, its shape is of the given type, but as there are no recursive calls, the type of positions at the one shape should be the empty type. The special case where `X = Unit` is called the unit container. We give this a constructor and destructor for better handling.

```
kc : X → [[ KC X ]] A
kc x = x , \()

kd : [[ KC X ]] A → X
kd (x , p) = x
```

Similar but different to that is the identity container

```
IC : Container
IC = Unit ▷ (\_ → Unit)
```

The identity container creates the identity functor, meaning a functor that is only composed of one position. It has one shape and on that shape, exactly one position. Its constructor and destructor are

```
ic : A → [[ IC ]] A
ic a = unit , \_ → a

idd : [[ IC ]] A → A
idd (s , p) = p unit
```

Now, to some useful closures. Containers are closed under products:

```
_:x:_ : Container → Container → Container
(S1 ▷ P1) :x: (S2 ▷ P2) = (S1 × S2) ▷ \{(s1 , s2) → P1 s1 or P2 s2}
```

Here, the shape is the product of the shape of the two underlying containers, and the positions are the positions of the first shape together with the positions of the second shape. It is slightly unintuitive that we use a disjunction for the positions, but it makes sense: in the prositions, we are creating a function from one position to one specific hole. That means that if we have the positions of both containers, the hole is either from the one container or the other. Again, constructor and destructor are

```
_:,:_ : [[ C ]] A → [[ D ]] A → [[ C :x: D ]] A
(sc , pc) :,: (sd , pd) = (sc , sd) , fromSum pc pd

prodd : [[ C :x: D ]] A → [[ C ]] A × [[ D ]] A
prodd ((sc , sd) , p) = (sc , p ∘ left) , (sd , p ∘ right)
```

Further, containers are closed under sums:

```
_:+:_ : Container → Container → Container
(S1 ▷ P1) :+: (S2 ▷ P2) = (S1 ⊎ S2) ▷ fromSum P1 P2
```

Here, the shape is the disjunction between the original shapes, and the positions are the positions of whichever of the two shapes was selected. Constructors and destructor:

```
leftc : [[ C ]] A → [[ C :+: D ]] A
leftc (s , p) = (left s) , p

rightc : [[ D ]] A → [[ C :+: D ]] A
rightc (s , p) = (right s) , p

sumd : [[ C :+: D ]] A → [[ C ]] A or [[ D ]] A
sumd (left  x , p) = left  (x , p)
sumd (right y , p) = right (y , p)
```

The interesting thing is that containers are also closed under dependent products and dependent sums, also known as functions and existentials. This is needed for the case that we have a function in one of the constructors, or have one argument of a constructor depend on a previous one. We start with the closure under dependent functions:

```
PiC : (A : Set) → (A → Container) → Container
PiC A fC = ((a : A) → S (fC a)) ▷ \f → ∃[ a of A ] P (fC a) (f a)
```

Here, the shape is a function from the input type to the respective shape of the given container family, so it itself is a family of shapes. The positions how have to give the individual holes for this family. So given a family of shapes, then the positions are a point `a` in that family where we give the positions of the container family at the same point `a`. The constructor and destructor is defined as

```
pic : ((a : A) → [[ fC a ]] B) → [[ PiC A fC ]] B
pic f = fst ∘ f , \{(a , p) → snd (f a) p}

pid : [[ PiC A fC ]] B → ((a : A) → [[ fC a ]] B)
pid (s , p) x = s x , p ∘ (x ,_)
```

Similarly, the dependent sum of a container is defined as

```
SigC : (A : Set) → (A → Container) → Container
SigC A fC = (Σ A (S ∘ fC)) ▷ \{(a , s) → P (fC a) s}
```

Where the shape is the expected dependent sum and the positions are derived from the tuple created by the shape. The constructor and destructor are

```
sigc : Σ A (\a → [[ fC a ]] B) → [[ SigC A fC ]] B
sigc (a , (s , p)) = (a , s) , p

sigd : [[ SigC A fC ]] B → Σ A (\a → [[ fC a ]] B)
sigd ((x , s) , p) = x , (s , p)
```

Last but not least, with these two closures, we can compose containers with

```
_:o:_ : Container → Container → Container
(S1 ▷ P1) :o: C2 = SigC S1 \s → PiC (P1 s) \p → C2
```

We say that given a shape of the first container, then forall positions on that shape, we take any form of the second container. The concrete shape and form of this is created by the dependent container sum and product. The constructor and destructor are defined as

```
compc : ([[ C ]] ∘ [[ D ]]) A → [[ C :o: D ]] A
compc (s , f) = (s , fst ∘ f) , \{(pcs , g) → snd (f pcs) g}

compd : [[ C :o: D ]] A → ([[ C ]] ∘ [[ D ]]) A
compd ((s , f) , g) = s , \pcs → f pcs , \pdfpcs → g (pcs , pdfpcs)
```

With all of those constructors and destructors, together with univalence, we can show the correctness of the container constructions.

Lemma 4.2:4

The following equalities for the interpretation of container closures hold

```
container-kc-interp   : [[ KC X ]] A      ≡ X
container-ic-interp   : [[ IC ]] A        ≡ A
container-prod-interp : [[ C :x: D ]] A   ≡ [[ C ]] A × [[ D ]] A
container-sum-interp  : [[ C :+: D ]] A   ≡ [[ C ]] A ⊎ [[ D ]] A
container-pi-interp   : [[ PiC A fC ]] B  ≡ ((a : A) → [[ fC a ]] B)
container-sig-interp  : [[ SigC A fC ]] B ≡ (∃[ a of A ] [[ fC a ]] B)
container-comp-interp : [[ C :o: D ]] A   ≡ ([[ C ]] ∘ [[ D ]]) A
```

proving the correctness of the above closure properties.

Proof 4.2:5

Proof by univalence, showing the (slightly modified with currying on sum destructors so they represent a single function each) constructors and desctructors are inverses of each other. Most just compute or only require trivial case splitting.

At this point it should be worth noting that containers are not closed under holes that are left of a function arrow, so the type morphism

```
toFunc : Set → Set
toFunc A = A → A
```

cannot be expressed via a container. This is on purpose, because if we would allow such a construction as an argument to a functor constructor and then create a fixpoint for that functor, we can create an inconsistency. This property is what makes the functors created by containers so-called *strictly positive functors*. This class of functors actually has a fixpoint that is accepted by the positivity checker.

Side Note 4.2:6

It is a good exercise to try creating an inconsistency by deactivating the positivity checker and writing a type that has a function from itself to itself as the argument to its only constructor. You will simultaneously be able to proof that there always exists an element of that type and that if an element of that type exists, the empty type can be created.

4.2.2 Folds and Fixpoints of Containers Part I

The fixpoint of a functor created by a container can be created as

```
data W (C : Container) : Set where
    In : [[ C ]] (W C) → W C
```

where data types that can be represented by such a fixpoint are called *W-Types* [2]. Here, we get the creation of the recursive data type by a constructor of its functor that has the recursive elements already in it. This is called the *initial algebra*. We also say that `W C` is the *least fixpoint* of the functor created by `C`. We can also get an inverse of it as

```
Out : W C → [[ C ]] (W C)
Out (In f) = f
```

So we can take a some data represented as a fixpoint and return its topmost constructor. We can write a general destructor for these types

no termination check:

```
foldC : ([[ C ]] A → A) → W C → A
foldC alg (In f) = alg (map (foldC alg) f)
```

Side Note 4.2:7

This code can be shown to be terminating when inlining the definition of `map` as:

```
foldC : ([[ C ]] A → A) → W C → A
foldC alg (In (s , p)) = alg (s , foldC alg ∘ p)
```

A fold takes a so-called *algebra*, an element of `[[ C ]] A → A` being a function that takes a functor filled with the results of a recursive call and turns it into the overall result type, and applies it to the fixpoint of the respective container. This can be used as a recursion scheme for the fixpoint of `C` in which most (in practice: all) functions over the data types can be expressed.

In order to express proofs over fixpoint data that depend on the fixpoint data itself (like an equation that gets refined during a case split), we define a dependent fold as:

```
foldPi : ((c : [[ C ]] (Σ (W C) A)) → A (In (map fst c))) → (w : W C) → A w
foldPi alg (In (s , p)) = alg (s , \p' → p p' , foldPi alg (p p'))
```

This fold takes an algebra of the functor containing the induction hypothesis together with the original data to create the statement for the current constructor. This models the behaviour of the statement changing when case splitting on the constructors, so we call it **container induction**. For example, if we want to prove the following statement:

```
not-inv : notB (notB x) ≡ x
not-inv {x = true } = refl ∈ (notB (notB true ) ≡ true )
not-inv {x = false} = refl ∈ (notB (notB false) ≡ false)
```

We have to case split, and in each clause, the statement changes depending on which constructor we have in that clause. Similar with a recursive statement:

```
zero-right-id : x + 0 ≡ x
zero-right-id {x = 0}    = refl                                  ∎ (0 + 0 ≡ 0)
zero-right-id {x = 1+ x} = (zero-right-id {x = x} || 1+_) ∎ ((1+ x) + 0 ≡ (1+ x))
```

Rewritten as a dependent fold, this would look like

```
NatF : Container lzero lzero
NatF = UnitC :+: IC

NatC = W NatF

+-alg : NatC → [[ NatF ]] NatC → NatC
+-alg y (left unit , _)  = y
+-alg y (right unit , p) = In (rightc (ic (p unit)))

_+F_ : NatC → NatC → NatC
x +F y = foldC (+-alg y) x

zero-right-idC : x +F (In $ leftc unitc) ≡ x
zero-right-idC {x} = flip foldPi x
    \{ (left x  , p) → extens (\()) || In ∘ (left x ,_)
    ;  (right y , p) →
        (In (right y , fst ∘ p) +F In (leftc unitc))                                    =<>
        +-alg (In (leftc unitc)) (right y , foldC (+-alg (In (leftc unitc))) ∘ fst ∘ p) =<>
        In (rightc (ic ((foldC (+-alg (In (leftc unitc))) ∘ fst ∘ p) unit)))            =< extens (\_ → snd (p unit)) >
        In (rightc (ic ((fst ∘ p) unit)))                                               =<>
        (In (right y , fst ∘ p))                                                        qed}
```

The proof technique is a bit funny to work with, as the equations from the recursive call are in stored in the second value of the positions. The proof splits by the constructors of `[[ NatF ]] NatC`. In the zero-case (`right y , p`), we dont' do much except show that functions from zero to anything are indeed equal. In the `1+ x` case (`right y , p`), we unwrap the fold as much as possible and then (using extensionality) use the recursive equation from the second part of the tuple stored in the (in this case just one) position. The rest just computes. This will be a common scheme when doing proofs over the folds: simplify the expression until we can apply the recursive equation from the second argument of the positions. We will use this to prove Theorem 4.3:24 (used implicitly), but also need the dependent fold in general to express the ideas on Chapter 5 Section 5.2.2.

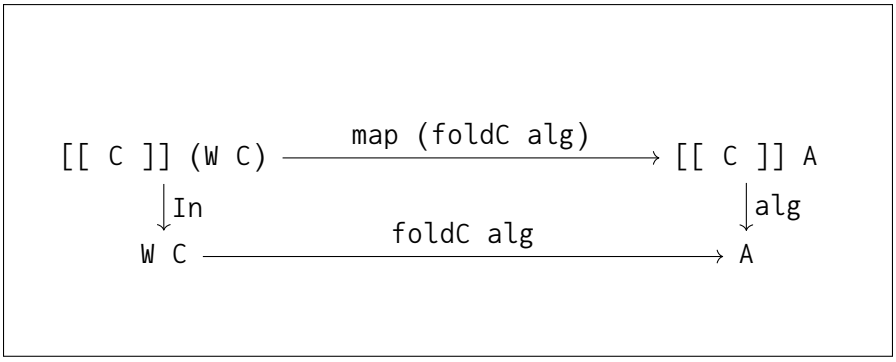
We give a short intuition of why folds are so general when it comes to computation.

4.2.3 Working with Folds

First, the fold satisfies the following property, directly read out from the definition:

```
foldC alg ∘ In ≡ alg ∘ map (foldC alg)
```

As a commuting categorical diagram, this looks as follows:



In categorical terms, this means that a fold is a homomorphism form the initial algebra `In` to any given algebra, but it is better to think of it as creating a recursive function from an (ideally) non-recursive representation of the function (the algebra). Technically, not all terminating functions can be represented using a fold, but in this thesis, we will only look at the class of functions that can be expressed using folds. We will quickly show by examples that this class is plenty big for functions relevant in practise.

The fold represents a function that case splits once on the first constructor and then applies a function to the result of the recursive calls on the holes. We will show this at the example of list concatenation. For readability, we do not use the composed containers here and use the isomorphic list functor from Section 4.2, together with the fold

```
foldList : (ListF A B → B) → ListFix A → B
foldList alg (InList []F) = alg []F
foldList alg (InList (x ::F xs)) = alg (x ::F foldList alg xs)
```

We can now write the concatenation function as

```
concat : List A → List A → List A
concat []       y = y
concat (x :: xs) y = x :: (concat xs y)
```

```
concat : ListFix A → ListFix A → ListFix A
concat x y = foldList (\{ []F       → y
                        ; (x ::F xs) → x :: xs}) x
```

To the left, we make the recursive call on the rest-list explicit. To the right, the recursive call is implicitly done by the fold, so we just put the result of the recursive call `xs` to the position where we previously wrote the recursive call as `concat xs y`.

Writing functions with these folds is elegant for as long as we only have to pattern match on the first constructor. But how do we pattern match on the second one? Say we want to write the function

```
bagsOfTwo : List A → List (List A)
bagsOfTwo  []           = []
bagsOfTwo (a :: [])     = []
bagsOfTwo (a :: b :: xs) = (a :: b :: []) :: bagsOfTwo xs
```

That packages a list into neat packages of two, like

```
bagsOfTwo (1 :: 2 :: 3 :: 4 :: 5 :: []) ≡
    (1 :: 2 :: []) :: (3 :: 4 :: []) :: []
```

Here, the recursive call is not put onto the immediate rest list `b :: xs` but on the rest list `xs` after. We can write a new recursion scheme for that, using a kind of algebra that can look ahead two construcors (here called a *two-lookahead list algebra*), like

```
foldList2' : (ListF A (ListF A B) → B) → ListFix A → B
foldList2' alg (InList []F)                        = alg []F
foldList2' alg (InList (x ::F InList []F))         = alg []F
foldList2' alg (InList (x ::F InList (y ::F ys))) = alg (x ::F y ::F (foldList2' alg ys))

bagsOfTwo : ListFix A → ListFix (ListFix A)
bagsOfTwo = foldList2'
    \{ []F → []
    ; (x ::F []F) → []
    ; (x ::F y ::F ys) → (x :: y :: []) :: ys
    }
```

Here, the recursion scheme can look ahead two constructors and has the recursive call of the rest list after the first two constructors available. How could we represent this as a normal fold?

First, we acknowledge that the results from an algebra can be bound in some context. So we can write

```
foldCtx : (ListF A (M B) → M B) → ListFix A → M B
foldCtx = foldList
```

for some context `M : Set → Set`. This means that the algebra can store additional information, like for example accumulators, that can pass information from deeper levels of the recursion back up. We can now write a special type of context that can be used to implement the `foldList2'` recursion scheme

```
foldList2AlgCtx : {A : Set} → Set → Set
foldList2AlgCtx {A} B = ListF A B × B
```

This context will store the shape of the previously visited constructor, together with the result of the previous recursive call. An algebra to use this context to implement `foldList2'` via a normal fold can look like

```
foldList2Alg : (ListF A (ListF A B) → B) →
    ListF A ((foldList2AlgCtx {A}) B) → ((foldList2AlgCtx {A}) B)
foldList2Alg alg []F                    = []F      , alg []F
foldList2Alg alg (x ::F ([]F , ys))     = x ::F ys , ys
foldList2Alg alg (x ::F (y ::F b , ys)) = x ::F ys , alg (x ::F y ::F b)

foldList2 : (ListF A (ListF A B) → B) → ListFix A → B
foldList2 {A} alg = snd ∘ foldList (foldList2Alg alg)
```

The accumulator now stores the previous constructor for the above recursive call, together with the previous recursive call relative to the above recursion. The algebra is only applied when there are enough values present in the list and the accumulator. Instead of utilising the current recursive call `ys`, we use the previous recursive call `b` from the accumulator. We will quickly show, that `foldList2` and `foldList2'` are equivalent. To do that, we start with a lemma proving that the accumulator does indeed store the correct recursive call.

Lemma 4.2:8

Forall two-lookahead list algebras `alg : ListF A (ListF A B) → B`, list elements `x y : A` and restlists `ys : ListFix A` it holds that

```
foldList2State-tuple :
    foldList (foldList2Alg alg) (InList (x ::F InList (y ::F ys)))
    ≡
    ( x ::F snd (foldList (foldList2Alg alg) (InList (y ::F ys))))
    , alg (x ::F y ::F snd (foldList (foldList2Alg alg) ys))
```

This means that the accumulator holds the correct recursive call and previous element.

Proof 4.2:9

Proof follows trivially from case-splitting over `ys` and, in the case for `ys = InList (z ::F zs)`, case splitting for the result of the recursive call

`foldList2Alg alg (z ::F foldList (foldList2Alg alg) zs)`
where we can see by looking ahead one more computation that indeed, the correct recursive call is stored.

We can now prove that the two recursion schemes are equivalent

Corollary 4.2:10

```
foldList2-correct : foldList2 ≡ foldList2'
```

Proof 4.2:11

We prove this by extensionality over the two-lookahead list algebra `alg : ListF A (ListF A B) → B` and list `klst : FixList A`. We case split over the list `lst`. In the case of the empty and singleton list, the equation evaluates. In the case of `lst = InList (x ::F InList (y ::F ys))` we get

```
foldList2 alg (InList (x ::F InList (y ::F ys)))                    =<>
snd (foldList (foldList2Alg alg) (InList (x ::F InList (y ::F ys)))) =< foldList2State-tuple >
alg (x ::F y ::F snd (foldList (foldList2Alg alg) ys))               =< foldList2-correct >
alg (x ::F y ::F foldList2' alg ys)                                  qed
```

After unfolding the definitions, we apply Lemma 4.2:8 to show that because the recursice call is in the right spot, the algebra is applied to the correct arguments. We then apply the induction hypothesis to derive the statement.

We can now write the `bagsOfTwo` function as before and get

```
bagsOfTwo : ListFix A → ListFix (ListFix A)
bagsOfTwo = foldList2
    \{ []F → []
    ; (x ::F []F) → []
    ; (x ::F y ::F ys) → (x :: y :: []) :: ys
    }

bagsOfTwo (InList (1 ::F InList (2 ::F (InList 3 ::F (InList 4 ::F (InList 5 ::F InList []F))))))
≡
(InList (1 ::F InList (2 ::F InList []F))) ::F InList ((InList (3 ::F InList (4 ::F InList []F))) ::F InList []F)
```

So, written with nicer constructors, we have again

```
bagsOfTwo (1 :: 2 :: 3 :: 4 :: 5 :: []) ≡
    (1 :: 2 :: []) :: (3 :: 4 :: []) :: []
```

This gives a good intuition about how we can derive recursion principles. We define a context that gives us the information we need from subsequent computation, together with (a possible indexed set of) results form (different possible) recursive calls and then use those to compute the final result. In this thesis, we will not create the recursion principle for all functions terminating in Agda, but one should now have a good grasp of how that would be done. Therefore, we assume that all functions we are interested in can be expressed via the normal destructors for the functors constructed by the container compositions together with folds over container fixpoints.

4.2.4 Folds and Fixpoints of Containers Part II

It should at this point be noted that container also have a fixpoint for corecursive types, creating elements of possibly infinite size. For solving, we will soon see how these are an ideal representation of data when we do not know its size yet, as is the case when searching for any element of a recursive data type. We can define the *greatest fixpoint* of a container as

```
record CoW (C : Container) : Set where
    coinductive
    constructor In
    field
        Out : [[ C ]] (CoW C)
```

Side Note 4.2:12

Outside of the container, `Out` has the type

```
Out : CoW C → [[ C ]] (CoW C)
```

Its inverse is the constructor, so we have

```
In : [[ C ]] (CoW C) → CoW C
```

This time, we create the data type from the final coalgebra `Out : CoW C → [[ C ]] (CoW C)`, to create the *greatest fixpoint.*

Just as we can fold over the lest fixpoint of a container, we can "cofold" over the greatest fixpoint as

no termination check:

```
cofold : (A → [[ C ]] A) → A → CoW C
cofold coAlg seed = < map (cofold coAlg) (coAlg seed) >co
```

Side Note 4.2:13

We can make this "terminating" (correct term with corecursive types is *productive*, meaning that the construction of the first constructor always terminates) by writing

```
cofold : (A → [[ C ]] A) → A → CoW C
Out (cofold coAlg seed) with coAlg seed
... | (s , p) = s , (cofold coAlg ∘ p)
```

Cofolding creates the data from an initial seed into the data that represents everything that springs from the seed by recursively applying the coalgebra. This might seem a bit annoying at first because unlike with a fold, we do not get a reduction of a fixpoint to a value. Still, cofolds can be used to describe functions between (possibly infinite) corecursive types. For example, when we use the explicit greatest fixpoints of the list functor

```
record ListCoFix (A : Set) : Set where
    coinductive
    constructor InCoList
    field
        OutCoList : ListF A (ListCoFix A)

cofoldList : (B → ListF A B) → B → ListCoFix A
OutCoList (cofoldList alg b) with alg b
... | []F      = []F
... | x ::F xs = x ::F (cofoldList alg xs)
```

we can write a concatenation function of these possibly infinite lists as

```
coConcat : ListCoFix A → ListCoFix A → ListCoFix A
coConcat {A} xs ys = cofoldList (caseOf OutCoList
    \{[]F → OutCoList ys
    ; (x ::F xs') → x ::F xs'}) xs
```

Side Note 4.2:14

In order to write more readable inline algebras for corecursive data types, we define

```
caseOf : (A → B) → (B → C) → A → C
caseOf f g = g ∘ f
```

so that we can case split a given input on the result of a function `f`

This co-concatenation gives us the first colist until it stops, after which we continue giving the second colist.

Before we go deeper into the properties of greatest fixpoints, we will talk about how to make fixpoints interact with the variables from Chapter 3.

4.3 Fixpoints Interacting with Contexts

Code-Tag: `BiThresholdsAndContainersPtI`

In Chapter 3 Section 3.5 and 3.6, we give an example of how the state monad, together with lenses for references, can be used to distribute recursive data over a state. This can be used to, for example, prevent certain branching or cache results of solving computations. This is close to how data is represented in imperative languages: by using references into the state, we can observe data constructor by constructor without for having to copy the entire value onto the stack every time we call a (lazy) function. In this section, we are automatically transforming data types in a way that we can still use the functions defined on the old data to run on the data that now utilizes references.

In Chapter 3 Section 3.8.3.1 Definition 3.8:29 we have developed a lattice based formalism for variables that can be used for solving. In this section we will discuss how container fixpoints can be used to scatter data across multiple such variables in order to have an arbitrarily partial element of a data type.

A variable functor `V : Set → Set` can itself be seen as a context. This context is not necessarily monadic, but it can wrap values of a certain type. Assume we only want a single variable to hold the topmost constructor, with references to its sub values, so that its overall value is distributed across possibly multiple variables. We could write a special fixpoint

error:

```
data VW (C : Container) (V : Set → Set) : Set where
    InVW : ([[ C ]] ∘ V) (VW C V) → VW C V
```

that states that every recursive call in the data is actually a variable to that value. Agda rightfully complains that this is not strictly positive, as `V` is not necessarily strictly positive. We can write this fixpoint as

```
VW : (C V : Container) → Set
VW C V = W (C :o: V)
```

where we remember from Lemma 4.2:4 that `[[ C :o: V ]] ≡ [[ C ]] ∘ [[ V ]]`. To match with definitions from Chapter 3 Section 3.5, we define

Definition 4.3:1: Held-in-Stasis Data Types

Given two containers `C` and `V`, we define the type `VW C V` to be the type of `(W C)` **held in stasis of V** (short: held in stasis). In the special case where `[[ V ]]` has a monadic instance, we say that `VW C V` is the type of `(W C)` **monadically held in stasis by V** (short: monadically held in stasis).

Analogously, this works for corecursive data types as well. Given the greatest fixpoint

```
CoVW : (C V : Container) → Set
CoVW C V = CoW (C :o: V)
```

we define

Definition 4.3:2: Held-in-Stasis Co-Data Types

Given two containers `C` and `V`, we define the type `CoVW C V` to be the type of `CoW C` **held in (co) stasis of V** (short: held in (co) stasis).

In the special case where `[[ V ]]` has a monadic instance, we say that `CoVW C V` is the type of `W C` **monadically held in (co) stasis by V** (short: monadically held in (co) stasis).

We quickly show that the variables from Chapter 3 Section 3.8.3.1 form possible contexts to hold a data type in stasis.

4.3.1 A Container for Lattice Bi-Threshold Variables

A full lattice bi-threshold variable is not a stricly positive functor, but luckily, its read function

```
read : S → VarAsm X
```

has a type that can, for a given type `S`, be turned into a stricktly positive functor as

```
V : Set → Set
V X = S → VarAsm X
```

or, equivalently, using the container closures:

Lemma 4.3:3

The containers

```
VarAsmC : Container
VarAsmC = UnitC :+: IC :+: UnitC

VC : Container
VC = PiC S (\_ → VarAsmC)
```

have the interpretations

```
VarAsmC=VarAsm : [[ VarAsmC ]] A ≡ VarAsm A
VC=S->VarAsmX  : [[ VC ]] A      ≡ (S → VarAsm A)
```

Proof 4.3:4

The first interpretation is shown by univalence, proving that the functions

```
to (left x , p)          = unassigned
to (right (left x) , p)  = asm (p unit)
to (right (right y) , p) = conflict

from unassigned = left unit , \()
from (asm x)    = right (left unit) , \_ → x
from conflict   = right (right unit) , \()
```

form an isomorphism. The second interpretation is proven with additionally using Lemma 4.2:4, so

```
[[ VC ]] A                    =<>
[[ PiC S (\_ → VarAsmC) ]] A =< container-pi-interp >
(S → [[ VarAsmC ]] A)         =< VarAsmC=VarAsm >
(S → VarAsm A)                qed
```

Of course, not every fixpoint `VW C VC` also forms a data type with variables in the sense of Chapter `MonadicSolving` Section 3.8.3.1, as not every function `S → VarAsm X` can be turned into a lattice bi-threshold variable. We will ensure the `VC` to be a lattice bi-threshold variable by constraining the functor and fixpoint created by `C :o: VC`.

4.3.2 Constraining Held-in-Stasis Functors and Fixpoints

In order to constrain properties that might not be representable by a strictly positive functor, we want to put arbitrary constraints on the positions of a functor so that if we fill them with the functions from Section 4.3.1, we can constrain them to be full lattice bi-threshold variables. We have to leave the full construction of the container based held-in-stasis data types using lattice bi-threshold variables to future research, but the definitions given in this section showcase how easily the step is from the constructions in this chapter to turning every variable used into a lattice bi-threshold variable from Chapter 3 Section 3.8.3.1. Other than that, the main takeaway of this subsection is to show that these constraints can be formulated in Agda.

Definition 4.3:5: **Positionally Constrained Functors**

We say an Element `(s , p) : [[ C ]] A` of a strictly positive functor represented by a container `C` is **positionally constrained** by a constraint `M : A → Set`, iff forall positions `p' : P C s` it holds that `M (p p')`.

Definition 4.3:6: **Inner Constrained Functor**

We say an element `c : [[ C ]] ([[ V ]] A)` is **inner constrained** by a constraint `M : [[ V ]] A → Set` iff `c` is positionally constrained by `M`.

Definition 4.3:7: **Inner Constrained (least) Fixpoint**

We say an element `(In (s , p)) : VW C V` is an **inner constrained (least) fixpoint** by a constraint `M : [[ V ]] (VW C V) → Set`, iff `(s , p)` is inner constrained by `M` and forall positions `p' : P (C :o: V) s`, the element `(p p')` (the recursive fixpoint at the current position) is also an inner constrained (least) fixpoint by `M`.

Side Note 4.3:8

Note how through lemma 4.2:4,

```
[[ C ]] ([[ V ]] A)   =<>
([[ C ]] ∘ [[ V ]]) A =< container-comp-interp >
[[ C :o: V ]] A       qed
```

so a being inner constrained is applicable to the functor to the fixpoint `VW C V`

Luckily, due to the structure of containers, all of these definitions terminate.

These definitions can be used to constrain a fixpoint to be held in stasis by the lattice bi-threshold variables from Chapter 3 Section 3.8.3.1 by using the following constraint:

Definition 4.3:9: has lattice bi-threshold variable

We say a function `read` **has a lattice bi-threshold variable** if there exists a lattice bi-threshold variable on the lattice `lat` with read function `read`.

Definition 4.3:10: lattice bi-threshold variable preservation

Given a container `C`, an element of a functor `c : ([[ C ]] ∘ [[ VC ]]) A` is said to be **lattice bi-threshold variable preserving** (short: variable preserving), iff `c` is inner constrained to have a lattice bi-threshold variable.

Definition 4.3:11: lattice bi-threshold variable preserving (least) fixpoint

Given a container `C`, an element `cvw : VW C VC` is said to be a **lattice bi-threshold variable preserving (least) fixpoint** (short: variable preserving (least) fixpoint) iff `cvw` is an inner constrained (least) fixpoint by the constraint of having a lattice bi-threshold variable.

Of course, something similar works for greatest fixpoints. We define

Definition 4.3:12: **Inner Constrained (greatest) Fixpoint**

We say an element `cvw : VW C V` is an **inner constrained (greatest) fixpoint** by a constraint `M : [[ V ]] (VW C V) → Set`, iff `Out cvw = (s , p)` is inner constrained by `M` and forall positions `p' : P (C :o: V) s`, the element `(p p')` (the recursive fixpoint at the current position) is also an inner constrained (greatest) fixpoint constrained by `M`.

Side Note 4.3:13

As the inner constrained greatest fixpoint is a function from codata to a type, we cannot write it as a normal recursive function, but have to write it as a corecursive data type

```
record InnerConstrainedCoFixpoint (M : [[ V ]] (CoVW C V) → Set) (cvw : CoVW C V) : Set where
    coinductive
    field
        innerConstrainedFunctor : InnerConstrainedFunctor M (Out cvw)
        recursivePreservation : let (s , p) = Out f in
            ∀ (p' : P (C :o: V) s) → InnerConstrainedCoFixpoint M (p p')
```

For full generality, we should express this as a `CoW` fixpoint as well, however, as we can see, the recursive call of the type is indexed by an element. As of now, we cannot express indexed fixpoints but we will do so in future work, using the ideas of Section 4.6.

Again, we can translate this definition to corecursive data types held in stasis by lattice bi-threshold variables

Definition 4.3:14: **lattice bi-threshold variable preserving (greatest) fixpoint**

Given a container `C`, an element `cvw : CoVW C VC` is said to be a **lattice bi-threshold variable preserving (least) fixpoint** (short: variable preserving (greatest) fixpoint) iff `cvw` is an inner constrained (greatest) fixpoint by the constraint of having a lattice bi-threshold variable.

We will now show under which conditions we can reassemble a data type

that is held in stasis by a context V.

4.3.3 Reassembling held-in-stasis data types

In this section we will talk about which contexts in general can be used to reassemble held-in-stasis data types. As we are not necessarily talking about variables here we are using M as a variable for the context (instead of the V from before). We remember that

```
VW C M = W (C :o: M)
```

so reassembling a held-in-stasis data type has to be some action

```
unwrap : VW C M → [[ M ]] (W C)
```

that gives us elements of the out-of-stasis data type in the monadic context. A precursor to defining this operation is the following definition.

Definition 4.3:15: **Commutable Contexts**

Given the endofunctors `F : Set → Set` and `M : Set → Set` and a monad instance on M, together with an action

```
switch : (F ∘ M) A → (M ∘ F) A
```

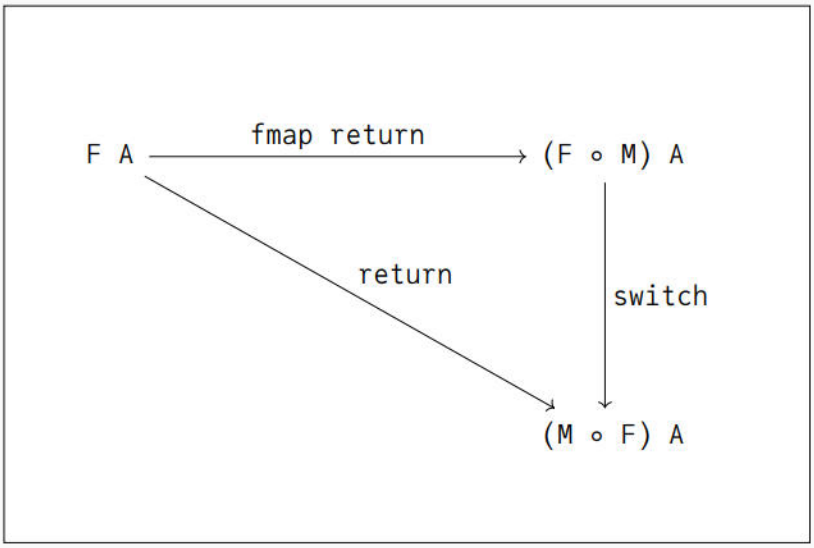
we say F **commutes into** M if the following laws hold:

```
return-prop : ∀ {fa : F A} →
    switch (return <$> fa) ≡ return fa

bind-prop : ∀ {fm : (F ∘ M) A} {f : A → M B} →
    switch ((_>>= f) <$>F fm) ≡ (switch fm >>= switch ∘ fmapF f)
```

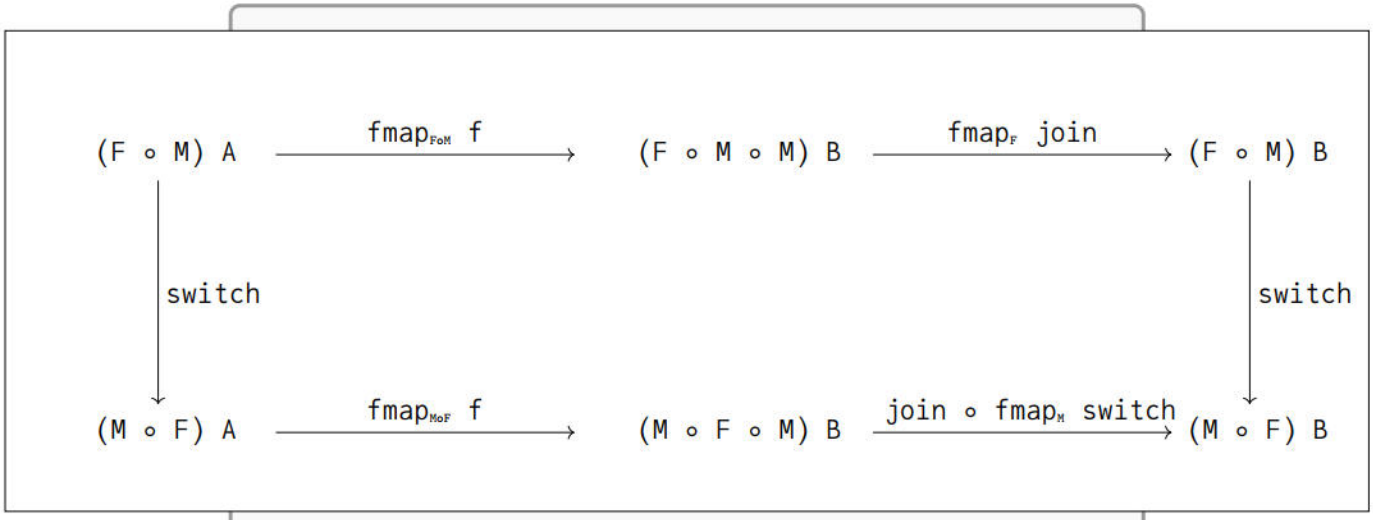
The reson for these laws is described below the definition. More categorically, the `bind-prop` can also equivalently (just by rewriting the definitions of `join` and `_>=>_`) be expressed as

```
bind-prop' :
    switch ∘ fmapF join ∘ FM.fmap f ≡ switch >=> switch ∘ fmapF f
```

resulting in the following two commuting categorical diagrams:



`return-prop`



bind-prop

(where we have unwrapped the definition of `_>=>_` to `fmap ∘ join`)

The definition of commutable contexts ensure that when using `switch` to turn a functor with monadic arguments into a monadic value containing the functor with the now non-monadic values, the properties of the functor are preserved in two ways. First, all values created by `return` should stay their respective values at the respective positions after the switch. Second: When we do some computation inside of the values in the functor, these computations should be preserved when switching, meaning that we can switch the functor before we do the computation and apply the computation on the values inside the functor, so switching again results in the same functor wrapped in the monadic context `M` that we would get if we had done the computation in the positions of the functor directly. Both of these properties ensure that the monadically held-in-stasis functor behaves the same as the unwrapped functor would in the monad `M`. We further emphasize this claim by defining an operator that can apply a function to a monadically held-in-stasis functor:

Lemma 4.3:16

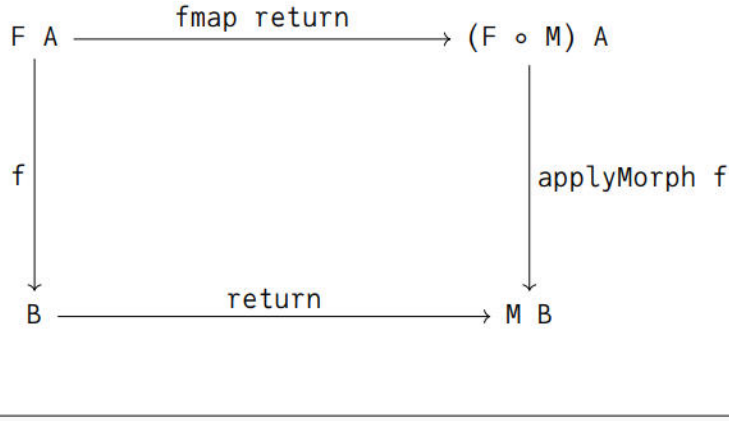
Given the functor `F` commuting into the monad `M` and the operation

```
applyMorph : (F A → B) → (F ∘ M) A → M B
applyMorph f = fmap f ∘ switch
```

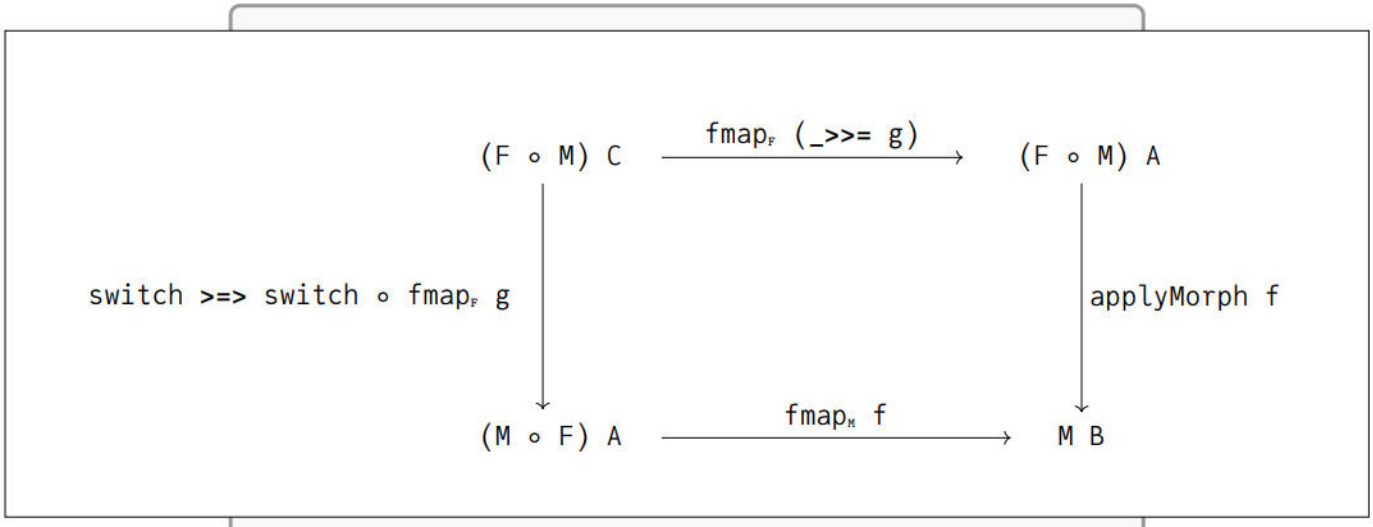
then forall functions `f : F A → B`, `g : C → M A` it holds that:

```
applyMorph-return-prop :
    applyMorph f ∘ fmap_F return ≡ return ∘ f

applyMorph-bind-prop :
    applyMorph f ∘ fmap_F (_>>= g) ≡ fmap_M f ∘ (switch >=> switch ∘ fmap_F g)
```

expressible as the commuting diagrams



applyMorph-return-prop



applyMorph-bind-prop

Proof 4.3:17

We can prove the applyMorph return property as

```
applyMorph f ∘ fmapF return      =<>
fmapM f ∘ switch ∘ fmapF return =< return-prop >
fmapM f ∘ return                 =< fmap-return >
return ∘ f                       qed
```

The applyMorph bin property is just the bind property with `fmap f ∘_` applied to it.

With this property, we know that we can apply a function to a functor value that is held in stasis by a monad, even when the internal values are changed due to monadic computation. We will now cover how to assemble the entire functor fixpoint.

Definition 4.3:18: **Wrapping and Unwrapping Held-in-Stasis Fixpoints**

Given the containers `C` and `M`, where `[[ C ]]` commutes into `[[ M ]]`, together with a monad instance for `[[ M ]]`, we define the two algebras

```
wrapAlg : [[ C ]] (VW C M) → VW C M
wrapAlg = In ∘ compc ∘ map return

unwrapAlg : ([[ C ]] ∘ [[ M ]]) ([[ M ]] (W C)) → [[ M ]] (W C)
unwrapAlg = fmap In ∘ (switch >=> switch)
```

In order for simpler dealing with the held-in-stasis fixpoint, we further define

```
unwrapAlg' : [[ C :o: M ]] ([[ M ]] (W C)) → [[ M ]] (W C)
unwrapAlg' = unwrapAlg ∘ compd
```

We define the wrapping and unwrapping of held-in-stasis fixpoints as

```
unwrap : VW C M → [[ M ]] (W C)
unwrap = foldC unwrapAlg'

wrap : W C → VW C M
wrap = foldC wrapAlg
```

Side Note 4.3:19

We will occasionally colloquially refer to unwrapping as **baking**, because a data that is held-in-stasis is assembled. If at any time, the assembly is only partially completed, we may refer to the data as being **half-baked**.

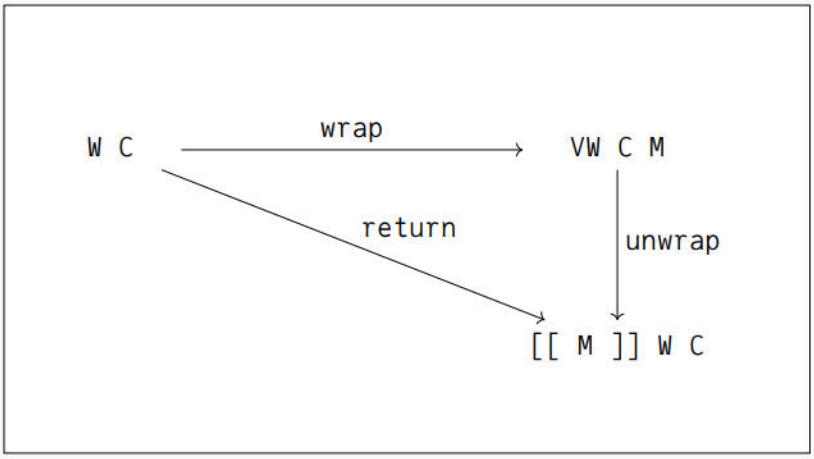
In order to prove that wrapping and unwrapping behave as intended, we prove that `wrap` is what we call a *monadic injection* (where instead of the identity in a normal injection we use `return`, so a non held-in-stasis fixpoint can be wrapped and unwrapped again without changing the value.

Theorem 4.3:20

For all containers `C` and `M` under the conditions of Definition 4.3:18, where it additionally holds that `fmap = map` for `[[ M ]]`, it holds that

```
unwrap-o-wrap : unwrap ∘ wrap ≡ return
```

So, as a commutable diagram:



Proof 4.3:21

We prove this by extensionality, using container induction over the fixpoint data `In (s , p) : W C`. We begin by just unrolling the definitions

```
unwrap (wrap (In (s , fst ∘ p)))                                                    =<>
unwrap (wrapAlg (s , wrap ∘ fst ∘ p))                                               =<>
unwrap ((In ∘' compc ∘' map return) (s , wrap ∘ fst ∘ p))                           =<>
unwrapAlg' ((map unwrap ∘ compc ∘' map return) (s , wrap ∘ fst ∘ p))                =<>
unwrapAlg' ((map unwrap ∘ compc) (s , return ∘ wrap ∘ fst ∘ p))                     =<>
fmap In (switch ((compd ∘ map unwrap ∘ compc) (s , return ∘ wrap ∘ fst ∘ p)) >>= switch) =<>
fmap In (switch (s , map unwrap ∘ return ∘ wrap ∘ fst ∘ p) >>= switch)              =< ... >
```

In order to apply the induction hypothesis, we need the assumption that `fmap = map` for `[[ M ]]` in order to get the original statement into the positions. We then apply the induction hypothesis.

```
fmap In (switch (s , map unwrap ∘ return ∘ wrap ∘ fst ∘ p) >>= switch)    =< map=fmap >
fmap In (switch (s , (fmap unwrap ∘ return) ∘ wrap ∘ fst ∘ p) >>= switch) =< fmap-return >
fmap In (switch (s , return ∘ unwrap ∘ wrap ∘ fst ∘ p) >>= switch)        =< snd ∘ p >
fmap In (switch (s , return ∘ return ∘ fst ∘ p) >>= switch)               =< ... >
```

Finally, we turn this into the target statement using the monad laws:

```
fmap In (return (s , return ∘ fst ∘ p) >>= switch) =< left-identity >
fmap In (switch (s , return ∘ fst ∘ p))            =< return-prop >
fmap In (return (s , fst ∘ p))                     =< fmap-simplified >
return (s , fst ∘ p) >>= return ∘ In               =< left-identity >
return (In (s , fst ∘ p))                          qed
```

The other direction

error:

`fmap wrap ∘ unwrap ≡ return`

does not hold for the simple fact that the monadically held-in-stasis data may contain monadic values `M A` in their positions that have either none or

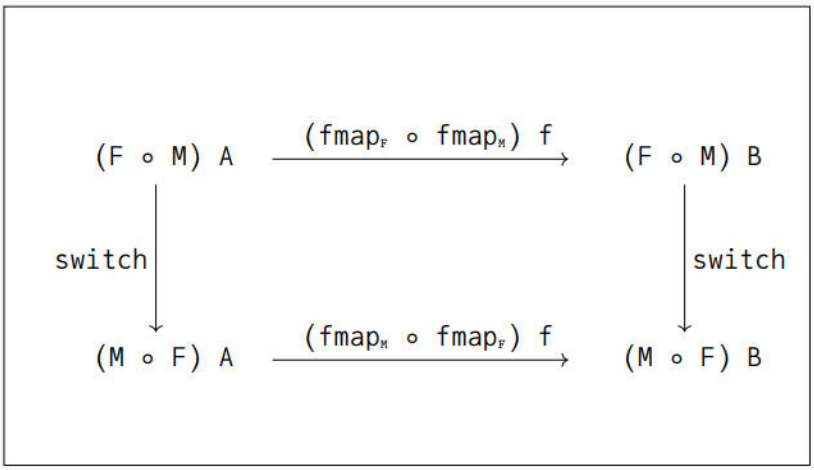
more than one value of `A` and it is not immediately clear what happens during the baking process, meaning we might also get several (or non at all) values of the baked data that are not necessarily expressible using a simple `return`, just as we would expect from a solving procedure. However, we do get a similar property when it comes to applying functions on held-in-stasis data. Before we can prove it, we need to prove a quick lemma.

Lemma 4.3:22

Let `F` commute into `M`, with a monad instance on `M`. Forall functions `f : A → B`, it now holds that

```
switch-pres-fmap : switch ∘ (fmap_F ∘ fmap_M) f ≡ (fmap_M ∘ fmap_F) f ∘ switch
```

So, as a commutable diagram:



Proof 4.3:23

We prove this via extensionality over `fm : (F ∘ M) A`. We mainly use the monad laws, together with the `bind-prop` from the definition of commutable contexts in Definition 4.3:15.

```
switch ((fmap_F ∘ fmap_M) f fm)                    =<>
switch (fmap_F (fmap_M f) fm)                      =< fmap-simplified >
switch (fmap_F (_>>= return ∘ f) fm)                =< bind-prop >
switch fm >>= switch ∘ fmap_F (return ∘ f)          =< fmap-comp >
switch fm >>= switch ∘ (fmap_F return ∘ fmap_F f) =<>
switch fm >>= (switch ∘ fmap_F return) ∘ fmap_F f =< return-prop >
switch fm >>= return ∘ (fmap_F f)                   =< fmap-simplified >
fmap_M (fmap_F f) (switch fm)                      =<>
(fmap_M ∘ fmap_F) f (switch fm)                    =<>
MF.fmap f (switch fm)                              qed
```

We can now show that we can apply functions for non held-in-stasis values on hel-in-stasis data.

Theorem 4.3:24

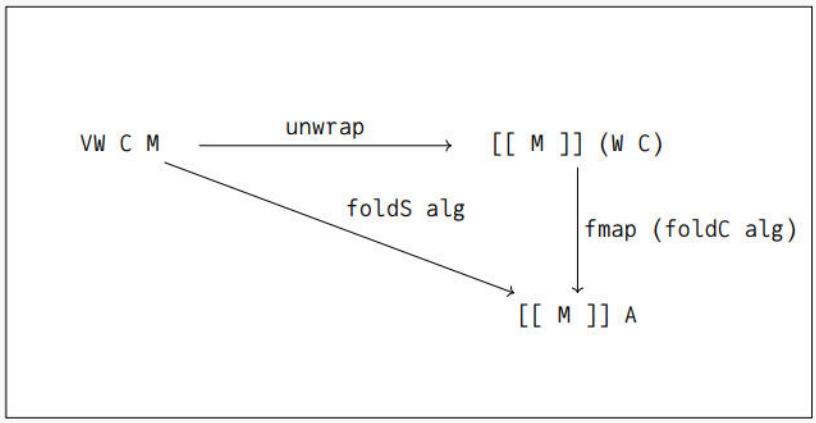
Under the conditions of theorem 4.3:20, with the functors of the containers `C` commuting into `M` with `fmap = map` for `[[ M ]]`, we define a fold over a held-in-stasis data type that uses an algebra for the non-held-in-stasis type as

```
foldS : ([[ C ]] A → A) → VW C M → [[ M ]] A
foldS alg = foldC (switch' >=> applyMorph alg)
```

It now holds that

```
foldS-unwrap : fmap (foldC alg) ∘ unwrap ≡ foldS alg
```

As a commuting diagram:



The significance of this theorem is that we can run functions that are defined without any knowledge of the monadic stasis on held-in-stasis data, achieving the same results, but now being able to potentially add semantics to the program execution.

Proof 4.3:25

We prove this by extensionality over `In (s , p) : VW C M` using dependent container induction. We start by applying the definitions

```
fmap (foldC alg) (unwrap (In ((s , f) , fst ∘ p)))                                          =<>
fmap (foldC alg) ((unwrapAlg' ∘ map unwrap) ((s , f) , fst ∘ p))                            =<>
(fmap (foldC alg) ∘ unwrapAlg' ∘ map unwrap) ((s , f) , fst ∘ p)                            =<>
(fmap (foldC alg) ∘ fmap In ∘ (switch >=> switch) ∘ compd ∘ map unwrap) ((s , f) , fst ∘ p) =< join-prop >
(fmap (foldC alg) ∘ fmap In ∘ switch ∘ map join ∘ compd ∘ map unwrap) ((s , f) , fst ∘ p)   =<>
```

In order to see the effect of the functions on the positions, we have to unwrap the desctructor for the container composition functors `compd`. We then transform the definition with the goal of moving `fmap (foldC alg)` into the positions to apply the induction hypothesis.

```
(fmap (foldC alg) ∘ fmap In ∘ switch ∘ map join ∘ compd ∘ map unwrap) ((s , f) , fst ∘ p)                          =<>
(fmap (foldC alg) ∘ fmap In ∘ switch ∘ map join ∘ compd) ((s , f) , unwrap ∘ fst ∘ p)                              =<>
(fmap (foldC alg) ∘ fmap In ∘ switch ∘ map join) (s , \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs))  =< fmap-comp >
(fmap (foldC alg ∘ In) ∘ switch ∘ map join) (s , \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs))       =<>
(fmap (foldC alg ∘ In) ∘ switch) (s , join ∘ \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs))           =<>
(fmap (alg ∘ map (foldC alg)) ∘ switch) (s , join ∘ \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs))    =< fmap-comp >
(fmap alg ∘ fmap (map (foldC alg)) ∘ switch) (s , join ∘ \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs)) =< ... >
```

to overcome the switch, we use Lemma 4.3:22 and continue transforming until we can apply the induction hypothesis.

```
(fmap alg ∘ fmap (map (foldC alg)) ∘ switch) (s , join ∘ \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs))      =< switch-pres-fmap >
(fmap alg ∘ switch ∘ map (fmap (foldC alg))) (s , join ∘ \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs))      =<>
(fmap alg ∘ switch) (s , (fmap (foldC alg)) ∘ join ∘ \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs))         =< join-fmap >
(fmap alg ∘ switch) (s , join ∘ ((fmap ∘ fmap) (foldC alg)) ∘ \pcs → f pcs , \pdfpcs → (unwrap ∘ fst ∘ p) (pcs , pdfpcs)) =< map=fmap >
(fmap alg ∘ switch) (s , join ∘ \pcs → f pcs , \pdfpcs → (fmap (foldC alg) ∘ unwrap ∘ fst ∘ p) (pcs , pdfpcs))          =< ... >
```

We can now see on which positions the full recursive call occurs and apply the induction hypothesis. We apply the induction hypothesis stored in the positions and transform the term until we hit the desired target.

```
(fmap alg ∘ switch) (s , join ∘ \pcs → f pcs , \pdfpcs → (fmap (foldC alg) ∘ unwrap ∘ fst ∘ p) (pcs , pdfpcs)) =< snd ∘ p >
(fmap alg ∘ switch) (s , join ∘ \pcs → f pcs , \pdfpcs → (foldS alg ∘ fst ∘ p) (pcs , pdfpcs))                 =<>
(fmap alg ∘ switch ∘ map join) (s , \pcs → f pcs , \pdfpcs → (foldS alg ∘ fst ∘ p) (pcs , pdfpcs))            =<>
(fmap alg ∘ switch ∘ map join ∘ compd) ((s , f) , foldS alg ∘ fst ∘ p)                                        =< join-prop >
(fmap alg ∘ (switch >=> switch) ∘ compd) ((s , f) , foldS alg ∘ fst ∘ p)                                     =< fmap-kleisli-switch >
((switch >=> fmap alg ∘ switch) ∘ compd) ((s , f) , foldS alg ∘ fst ∘ p)                                     =<>
(switch' >=> fmap alg ∘ switch) ((s , f) , foldS alg ∘ fst ∘ p)                                              =<>
(switch' >=> applyMorph alg) ((s , f) , foldS alg ∘ fst ∘ p)                                                 =<>
foldS alg (In ((s , f) , fst ∘ p))                                                                           qed
```

One slightly non-trivial monadic transformation is what we call the *kleisli-switch*, which is a monadic property

```
fmap-kleisli-switch : fmap f ∘ (g >=> h) ≡ g >=> fmap f ∘ h
```

that can be proven as

```
fmap f ∘ (g >=> h)                 =<>
fmap f ∘ (\x → g x >>= h)          =<>
(\x → fmap f (g x >>= h))          =< fmap-simplified >
(\x → (g x >>= h) >>= return ∘ f) =< associative >
(\x → g x >>= (h >=> return ∘ f)) =< fmap-simplified >
g >=> fmap f ∘ h                    qed
```

We note at this point that the way we presented to apply a function to held-in-stasis data is not necessarily the most efficient way to do it, precisely because according to Theorem 4.3:24, applying a function is equivalent to baking the entire data. For search, this might mean that it is equivalent to branching on every unassigned variable in the data, regardless of whether the variables value is important for the function evaluation or not. As we have seen in Chapter 3 Section 3.4, this is inefficient. We can write efficient functions using the direct fold for held-in-stasis data types, that does not even require commutable contexts and only a monad instance on `[[ M ]]`:

```
foldM : (([[ C ]] ∘ [[ M ]]) A → [[ M ]] A) → VW C M → [[ M ]] A
foldM alg = foldC (alg [[ M ]] mon ∘ map join ∘ compd)
```

We could somehow reuse algebras for non held-in-stasis data types by using what we call *monadic algebras* of the form

```
alg : ∀ (M : Set → Set) → Monad M → ([[ C ]] ∘ M) A → M A
```

where `(alg id identityMonad)` is equal to the algebra for the non held-in-stasis data. This algebra always exists, however in order to prove it we would need a model for type theory, which we do not do in this thesis. The only problem with this approach is that it is fairly hard to get any kind of correctness criteria concerning the value the algebra gives in the identity context, so we went with the definitions via commutable contexts.

To summarise this Subsection: Data types can be monadically held in stasis in such a way that they can be reassembled in the monadic context. Through the assembly, we are given a way to apply a function to the held in stasis values, where we get the same result in the monadic context as if we had first reassem-

bled the value and then applied the function to it. This direct application of a function is inefficient, but can be shown to be correct. A solution is to use monadic algebras to not force the entire value to be assembled unless needed, so we have a form of explicit lazyness.

4.3.4 Creating Commutable Contexts

In this section we have a look at which containers make up commutable contexts. It is easiest to see how we can commute a container functor with the identity context, using the function

```
switch : ([[ C ]] ∘ id) A → (id ∘ [[ C ]]) A
switch (s , p) = (s , id ∘ p)
```

Something similar works for any monadic context `M` that has an interpretation function `interp : M A → A` for all types `A`, but we will not prove this here because in the context of this thesis, most monads have some form of error state and therefore no universal interpretation function. This means that the above idea does not work here, because the result of the position function of type `P s → M A` can only be retrieved in a monadic context. In general, we would want to move the entire function into the monadic context to be able to perform the switch, so we can get the switched positions as a monadic action `M (P s → A)`. We will show that a container is commutable with a monad if the positions are commutable with the monad.

Lemma 4.3:26

Let `C = S ▷ P` be a container and `M` have a monad instance.
If it holds that forall shapes `s : S`, the function functor `(\A → (P s → A))` of the positions is commutable with `M` for all `s : S` via the switch-function

```
switch' : (P s → M A) → M (P s → A)
```

then also `[[ C ]]` is commutable with `M` with the switch operation

```
switch : ([[ C ]] ∘ M) A → (M ∘ [[ C ]]) A
switch (s , p) = ⟪ (s ,_) (switch' p) ⟫
```

Proof 4.3:27

Let `(s , p) : ([[ C ]] ∘ M) A`. The return property has the type

```
return-prop : ⟪ (s ,_) (switch' (return ∘ p)) ⟫ ≡ return (s , p)
```

It follows from the return property of `switch'` and the applicative laws:

```
⟪ (s ,_) (switch' (return ∘ p)) ⟫ =< return-prop >
⟪ (s ,_) ⟪ p ⟫ ⟫                  =< homomorphism >
return (s , p)                    qed
```

The bind property follows similary, with a lot of monad law applications. It has the type

```
bind-prop : ⟪ (s ,_) (switch' ((_>>= f) ∘ p)) ⟫
          ≡ ⟪ (s ,_) (switch' p) ⟫ >>= (\{(s' , p') → ⟪ (s' ,_) (switch' (f ∘ p')) ⟫})
```

We will not give all of the tedious steps, but just show the most important ones. We start by applying the bind property of the positions:

```
⦇ (s ,_) (switch' ((_>>= f) ∘ p)) ⦈ =< bind-prop >
⦇ (s ,_) (switch' p >>= switch' ∘ Fkt.fmap f) ⦈ =< ... >
```

We now reformulate the expression using mostly associativity and left identity. We eventually end up with a term

```
(do
    (s' , p') ← switch' p >>= return ∘ (s ,_)
    (switch' (f ∘ p')) >>= return ∘ (s' ,_)
) =< ... >
```

Noting that `fmap f m ≡ m >>= return ∘ f`, we now get that this is equal to

```
⦇ (s ,_) (switch' p) ⦈ >>= (\{(s' , p') → ⦇ (s' ,_) (switch' (f ∘ p')) ⦈}) qed
```

proving the statement.

We can use this property to derive a reasonably large class of data types and monads that form commutative contexts. As a basis, we use the following observation: if we want to create a switch function for all positions

```
switch' : (P s → M A) → M (P s → A)
```

Either the monad itself represents some distributions or we have to "bake" every single position in the monad first to reassemble them from the (finite) function graph afterwards. For example, if the position is of type `Unit :+: Unit`, we can write the switch as

```
switch' : (Unit :+: Unit → M A) → M (Unit :+: Unit → A)
switch' p = do
    a ← p (left  unit)
    b ← p (right unit)
    return (
        \{ (left  unit) → a
        ;  (right unit) → b})
```

Such a construction works for all functions from finite domains as long as the monad is locally commutative, meaning that we can switch the order of any two operations so that the resulting function does not change depending on the order that we render the position results in the monad. We will prove this fact by modeling finite number of positions as vertors. The result will be conceptually simple but tedious on the technical side, so we start with a few definitions about finite data types and vectors.

4.3.4.1 Finite Types and Vectors

We use the definition of finite types from the Agda standard library version 2.0 [6].

Definition 4.3:28: **Finite Data Types**

Given a natural number `n : ℕ`, we define the type `Fin n` with exactly `n` many constructors as

```
data Fin : ℕ → Set where
    f0   : Fin (1+ n)
    f1+_ : Fin n → Fin (1+ n)
```

A **finite data type** is a type where there exists a `n : ℕ` so that the type is isomorphic (equal, under univalence) to the type `Fin n`.

The above definition is an indexed data type, which is difficult to use together with the current version of cubical (v0.7), so we cannot naturally case split on the data of the `Fin n` type for more than a few local definitions for which we can use Agda's intensional equality. We therefore use a recursion principle that can be used in a cubical context.

Lemma 4.3:29

Given a type family `T : ℕ → Set`, the induction step `f : ∀ n → T n → T (1+ n)` and the base case `x : ∀ n → T (1+ n)`, then forall `m : ℕ`, given an element `fin : Fin m`, we can derive that `T m` holds. The function computing that statement is called `foldFin`

Proof 4.3:30

In the case of `fin = f0`, we just return `x`. In the inductive case of `fin = f1+ n`, we apply `f` to the recursive call of `foldFin T f x n`.

Similarly, we can define a dependent recursion principle that does not rely on general natural numbers, but ever smaller growing values of finite types. To do that, we define the injection of an element $\texttt{fin}_\texttt{n}$ `: Fin n` into the type `Fin (1+ n)` that preserves the natural number that $\texttt{fin}_\texttt{n}$ represents.

Lemma 4.3:31

Given the interpretation function of a value of a finite type as a natural number:

```
finToNat : Fin n → ℕ
finToNat f0      = 0
finToNat (f1+ n) = 1+ (finToNat n)
```

We can write an injection function

```
injectFin1 : Fin n → Fin (1+ n)
injectFin1 f0      = f0
injectFin1 (f1+ n) = f1+ (injectFin1 n)
```

with the property

```
finToNat-o-injectFin1 : finToNat ∘ injectFin1 ≡ finToNat
```

Proof 4.3:32

Using extensionality, by induction over the finite data.

The dependent recursion principle now looks like

Lemma 4.3:33

Given a natural number `n : ℕ` and a type family `T : Fin (1+ n) → Set`, an induction step `f : ∀ i → T (injectFin1 i) → T (1+ i)` and a base case `x : T f0`, then forall `fin : Fin (1+ n)`, it holds that `T fin`. The function computing this statement is called `foldFin'`

Proof 4.3:34

In the base case of `fin = f0`, we return `x`. In the inductive case, we apply `f` to the recursive case using the injection function as `foldFin' (T ∘ injectFin1) (f ∘ injectFin1) x fin`.

We can now define vectors and their operations.

Definition 4.3:35: **Vectors**

Given a type `A : Set` and a natural number `n : ℕ` we define the type of vectors as

```
Vec : Set → ℕ → Set
Vec A n = Fin n → A
```

so vectors are functions from finite types `Fin n` to some content type `A`, so `n` is the length of the vector and the amount of `A`s present in the vector.

We can define the vector operations as follows

```
[]v : Vec A 0
[]v ()

_::v_ : A → Vec A n → Vec A (1+ n)
(a ::v vec) fin = foldFin' (\_ → A) (\i _ → vec i) a fin

drop1 : Vec A (1+ n) → Vec A n
drop1 vec = vec ∘ f1+_

mapVec : (A → B) → Vec A n → Vec B n
mapVec f vec = f ∘ vec
```

For what we are about to prove, we only need a few correctness properties. First, a lemma with which we can decompose a vector:

Lemma 4.3:36

Given a natural number `n : ℕ` and a vector `vec : Vec A (1+ n)`, then it holds that

```
uncons-uncons : (vec f0 ::v drop1 vec) ≡ vec
```

Proof 4.3:37

We prove this first by extensionality over the finite value `fin` going into the vector, and then by induction over the length `1+ n` of the vector `vec`. In the case of `n = 0`, we use the fact `fin1-eq` that all elements of the type `Fin 1` are equal. We then get

```
(vec f0 ::v drop1 vec) fin =< fin1-eq >
(vec f0 ::v drop1 vec) f0  =<>
vec f0                     =< fin1-eq >
vec fin                    qed
```

In the inductive case of `n = 1+ n'`, we prove the statement by induction over `fin` using `foldFin'`.

Similarly, we can prove what happens when we drop a value from the vector

Lemma 4.3:38

Given a vector `vec : Vec A (1+ n)` and an element `a : A`, it holds that

```
cons-drop-prop : drop1 (a ::v vec) ≡ vec
```

Proof 4.3:39

By extensionality over the finite element `fin : Fin (1+ n)` going into the vector and induction over `n`, where we use induction over `fin` in the recursive case.

Finally, a property about vector mapping:

Lemma 4.3:40

Given a vector `vec : Vec A (1+ n)`, an element `a : A`, and a function `f : A → B`, it holds that

```
mapVec-cons : mapVec f (a ::v vec) ≡ f a ::v mapVec f vec
```

Proof 4.3:41

By induction over `n`, using extensionality for the vector input `fin : Fin (1+ n)`, using equivalence of finite values of type `Fin 1` in the base case and finite induction over `fin` in the recursive case.

We now have the tools to prove that vectors commute into locally commutable monads, but before we prove that, we need one slightly useful lemma.

4.3.4.2 Splitting the Bind Property into a Property for the Functor Map and Join

Depending on taste, it might be useful to separate proofs about the functor map from proofs about the join for a monadic context. When we prove properties about the monadic bind, we always need to interlock these two. We prove a useful lemma that can be used if separating the concepts to create the bind property for commutable contexts is desirable.

Lemma 4.3:42

Given the return property

```
return-prop : ∀ {fa : F A} →
    switch (return <$> fa) ≡ return fa
```

The bind property

```
bind-prop : ∀ {fm : (F ∘ M) A} {f : A → M B} →
    switch ((_>>= f) <$>F fm) ≡ (switch fm >>= switch ∘ fmapF f)
```

Is equivalent to the two properties

```
join-prop : switch ∘ fmapF join ≡ switch >=> switch
fmap-fmap : ∀ {f : A → B} → switch ∘ fmapF (fmapM f) ≡ fmapM (fmapF f) ∘ switch
```

Proof 4.3:43

From the join and fmap property we start with

```
switch ((_>>= f) <$>F fm) =< ... >
```

and use the monadic laws and definition of join to reshape this into

```
((switch ∘ fmap_F join) ∘ fmap_F (fmap_M f)) fm =< ... >
```

we can now use the join property

```
((switch ∘ fmap_F join) ∘ fmap_F (fmap_M f)) fm =< join-prop >
(switch >=> switch) (fmap_F (fmap_M f) fm)          =<>
(switch (fmap_F (fmap_M f) fm) >>= switch)          =< ... >
```

We now apply the fmap-fmap property and apply the monadic laws until we prove the statement.

```
(switch (fmap_F (fmap_M f) fm) >>= switch)                 =< fmap-fmap >
(fmap_M (fmap_F f) (switch fm) >>= switch)                 =< fmap-simplified >
((switch fm) >>= return ∘ fmap_F f >>= switch)             =< associative >
((switch fm) >>= \sfm → return (fmap_F f sfm) >>= switch) =< left-identity >
((switch fm) >>= \sfm → switch (fmap_F f sfm))             =<>
switch fm >>= switch ∘ fmap_F f                            qed
```

In the other direction, we prove the join property as

```
switch ∘ fmap_F join           =<>
switch ∘ fmap_F (_>>= id)      =< bind-prop >
switch >=> switch ∘ fmap_F id  =< fmap_F-id >
switch >=> switch              qed
```

and the fmap-fmap property as

```
switch (FM.fmap f fm)                                =<>
switch ((fmap_F ∘ fmap_M) f fm)                      =<>
switch (fmap_F (fmap_M f) fm)                        =< fmap-simplified >
switch (fmap_F (_>>= return ∘ f) fm)                 =< bind-prop >
switch fm >>= switch ∘ fmap_F (return ∘ f)           =< fmap_F-comp >
switch fm >>= switch ∘ (fmap_F return ∘ fmap_F f)    =<>
switch fm >>= (switch ∘ fmap_F return) ∘ fmap_F f    =< return-prop >
switch fm >>= return ∘ (fmap_F f)                    =< fmap-simplified >
fmap_M (fmap_F f) (switch fm)                        =<>
(fmap_M ∘ fmap_F) f (switch fm)                      =<>
MF.fmap f (switch fm)                                qed
```

4.3.4.3 Commutability of the Vector Functor with a Locally Commutative Monad

We can now prove the important building blocks for commuting the vector context into a locally commutative monad. First, we define what a locally commutative monad is, which we need in order for the monadic result to be unique.

Definition 4.3:44: **Locally Commutative Monad**

Given a monadic instance for `M`, a **locally commutative monad** is given if additionally, the following law holds:

```
commute : ∀ {f : A → B → M C} →
    (ma >>= \a → mb >>= \b → f a b) ≡ (mb >>= \b → ma >>= \a → f a b)
```

Side Note 4.3:45

Note how this is similar to the commutative property for monadic variables from Chapter 3 Section 3.8

This definition assures, that we can, if applicable, exchange two monadic operations without affecting the overall result.

We will now prove the important properties for commuting the vector context into a locally commutative monad.

Definition 4.3:46: **Vector Switch**

We define the switch operation on vectors as

```
switch : Vec (M A) n → M (Vec A n)
switch {n = 0}    vec = return []v
switch {n = 1+ n} vec = ⦇ (vec f0) ::v (switch (drop1 vec)) ⦈
```

Lemma 4.3:47

Given the switch operation from Definition 4.3:46 over a locally commutative monad `M`, it holds that

```
switch-return : switch ∘ mapVec return ≡ return
```

Proof 4.3:48

We prove this by induction over the length `n` of the vector `vec`, using extensionality over `vec`. In the base case of `n = 0` we only use the fact that all vectors of length 0 are equal. In the recursive case of `n = 1+ n`, we prove the statement as

```
(switch ∘ mapVec return) vec                                  =<>
switch (return ∘ vec)                                         =<>
⦇ ((return ∘ vec) f0) ::v (switch (drop1 (return ∘ vec))) ⦈   =<>
⦇ (return (vec f0)) ::v (switch (drop1 (return ∘ vec))) ⦈     =<>
⦇ (return (vec f0)) ::v (switch (return ∘ drop1 vec)) ⦈       =< switch-return >
⦇ (return (vec f0)) ::v (return (drop1 vec)) ⦈                =< homomorphism >
⦇ (vec f0 ::v_) (return (drop1 vec)) ⦈                        =< homomorphism >
return (vec f0 ::v drop1 vec)                                 =< uncons-uncons >
return vec                                                    qed
```

Lemma 4.3:49

Given the switch operation from Definition 4.3:46 over a locally commutative monad `M`, it holds that

```
switch-fmap-fmap : switch ∘ (mapVec ∘ fmapₘ) f ≡ (fmapₘ ∘ mapVec) f ∘ switch
```

Proof 4.3:50

We prove this by induction over the length `n` of the vector `vec`, using extensionality over `vec`. In the base case of `n = 0` we just have to do a few monadic transformations and the fact `vec0-eq` that all vectors of length 0 are equal.

```
return []v                          =< vec0-eq >
return (mapVec f []v)               =< left-identity >
return []v >>= return ∘ mapVec f =< fmap-simplified >
(fmap ∘ mapVec) f (return []v)   qed
```

In the inductive case of `n = 1+ n`, we begin by simplifying the left hand side

```
(switch ∘ (mapVec ∘ fmap) f) vec                                                    =<>
switch (mapVec (fmap f) vec)                                                        =<>

⦇ ((mapVec (fmap f) vec) f0) ::v (switch (drop1 (mapVec (fmap f) vec))) ⦈ =< binOp-simplified >

(do
    v0 ← (mapVec (fmap f) vec) f0
    lst ← switch (drop1 (mapVec (fmap f) vec))
    return (v0 ::v lst)
)                                                                                   =<>
```

We can now apply the induction hypothesis on the rest list

```
(do
    v0 ← (mapVec (fmap f) vec) f0
    lst ← switch (mapVec (fmap f) (drop1 vec))
    return (v0 ::v lst)
) =< switch-fmap-fmap >

(do
    v0  ← fmap f (vec f0)
    lst ← fmap (mapVec f) (switch (drop1 vec))
    return (v0 ::v lst)
) =< ... >
```

After applying the hypothesis, we transform the statement monadically. We often employ the following trivial equality:

```
fmap-bind : (_>>= g) ∘ fmap f ≡ _>>= (g ∘ f)
fmap-bind =
    (_>>= g) ∘ fmap f            =< fmap-simplified >
    (_>>= g) ∘ (_>>= return ∘ f) =< associative >
    (_>>= (return ∘ f >=> g))    =< left-identity >
    _>>= (g ∘ f)                 qed
```

This equation enables us to pull the function from a functor map into the monadic computation directly. Together with the standard monad laws, we end up at the expression

```
(do
    fmap (mapVec f) (do
        v0 ← vec f0
        lst ← switch (drop1 vec)
        return (v0 ::v lst))
) =< binOp-simplified >

fmap (mapVec f) (switch vec) =<>
((fmap ∘ mapVec) f ∘ switch) vec qed
```

and prove the statement.

Lemma 4.3:51

Given the switch operation from Definition 4.3:46 over a locally commutative monad `M`, it holds that

```
switch-join : switch ∘ mapVec join ≡ switch >=> switch
```

Proof 4.3:52

We prove this by induction over the length `n` of the vector `vec`, using extensionality over `vec`. In the base case of `n = 0` we just apply the left identity. In the induction step of `n = 1+ n`, we first transform the left hand side until we can apply the induction hypothesis

```
switch (join ∘ vec)                                          =<>
⦇ ((join ∘ vec) f0) ::v (switch (drop1 (join ∘ vec))) ⦈      =<>
⦇ ((join ∘ vec) f0) ::v (switch (join ∘ drop1 vec)) ⦈        =< switch-join >
⦇ ((join ∘ vec) f0) ::v ((switch >=> switch) (drop1 vec)) ⦈ =< ... >
```

After that, we apply the monad laws to transform this into the individual monadic operations

```
⟪ ((join ∘ vec) f0) ::v ((switch >=> switch) (drop1 vec)) ⟫ =< binOp-simplified >

(do
    v0 ← join (vec f0)
    lst ← (switch >=> switch) (drop1 vec)
    return (v0 ::v lst)
) =< ... >

(do
    v0' ← vec f0
    v0 ← v0'
    lst' ← switch (drop1 vec)
    lst ← switch lst'
    return (v0 ::v lst)
) =< ... >
```

We can now see, that we monadically retrieve the element of the vector at position 0 and then the rest of the switched vector. We now have to reassemble the two switch operations that are supposed to run after one another. To do that, we have to rearrange the monadic operations using the commutative property, like

```
(do
    v0' ← vec f0
    v0 ← v0'
    lst' ← switch (drop1 vec)
    lst ← switch lst'
    return (v0 ::v lst)
) =< commute >

(do
    v0' ← vec f0
    lst' ← switch (drop1 vec)
    v0 ← v0'
    lst ← switch lst'
    return (v0 ::v lst)
) =< cons-drop-prop >

(do
    v0' ← vec f0
    lst' ← switch (drop1 vec)
    v0 ← (v0' ::v lst') f0
    lst ← switch (drop1 (v0' ::v lst'))
    return (v0 ::v lst)
) =< ... >
```

Now, we have both switch operations clearly one after the other. We now repackage this into the idiom brackets

```
(do
    vec' ← do
        v0' ← vec f0
        lst' ← switch (drop1 vec)
        return (v0' ::v lst')
    v0 ← vec' f0
    lst ← switch (drop1 vec')
    return (v0 ::v lst)
) =< binOp-simplified >

(do
    vec' ← ⦇ (vec f0) ::v (switch (drop1 vec)) ⦈
    ⦇ (vec' f0) ::v (switch (drop1 vec')) ⦈
) =< ... >
```

The rest follows basically from the vector properties

```
(⦇ (vec f0) ::v (switch (drop1 vec)) ⦈ >>= switch) =< uncons-uncons >
(switch (vec f0 ::v drop1 vec) >>= switch)         =<>
(switch >=> switch) (vec f0 ::v drop1 vec)         =< uncons-uncons >
(switch >=> switch) vec                            qed
```

proving the statement.

We can now prove an actual theorem with this that allows us to find a huge class of commutable contexts. For that, we first define the container type that makes up the respective functors.

4.3.4.4 Commuting Contexts for Shapely Container Functors

The class of functors that we can commute with the above method are those created by so-called *shapely containers* [3,53]. Those are containers where every shape only has a finite number of positions.

Definition 4.3:53: **Shapely Containers**

A container `S ▷ P` is called **shapely**, iff forall shapes `s : S`, the type of positions `P s` is finite. The data types produced by the fixpoints of shapely containers are called **shapely data types**.

Shapely data types, as a rule of thumb, represent (non dependent) data types where each constructor only has a finite number of recursive arguments, ergo, in general, no functions. We will later show how to circumvent this restriction, but first, we show that this class of functors commutes into locally commutative monads.

We can treat the container positions as a finite type with the following corollary

Corollary 4.3:54

Every shapely container `S ▷ P` has an isomorphic (equal) container

```
vectorContainer = S ▷ (Fin ∘ num ∘ finP)
```

Proof 4.3:55

Using the isomorphism (equivalence under univalence) of positions with some finite type `Fin n`.

We can now show that shapely container functors form commute into locally commutative monads.

Theorem 4.3:56

Given a shapely container `C` and a locally commutative monad instance for `M`, then `[[ C ]]` commutes into `M`.

Proof 4.3:57

Due to Lemma 4.3:26 we only have to show that the functor `\A → (P s → A)` induced by the functions from positions into an element commutes into `M`. As through Corollary 4.3:54, the positions

are isomorphic (equal to) objects of a finite type `Fin n`, the induced functor is the functor of vectors. We then, using Lemma 4.3:42 only need to provide the properties from Lemma 4.3:47, Lemma 4.3:49 and Lemma 4.3:51 to prove that `[[ C ]]` commutes into `M`.

This class of data types is general in the sense that at least the sum-of-product-types can be created with it, which is a decent class of data types. However, in general, we cannot have functions as arguments to these data types. So in the context of solving, how would we solve for functions? There is a data type with which we can represent any kind of function that is almost shapely. We will cover it in Section 4.7. Before we do that however we need to have a look at another class of data types: The data types that have a lattice instance. We need that lattice instance to create a variable that contains a value that is held in stasis. We will create those data types next.

4.4 Lattices for Containers

Code-Tag: `BiThresholdsAndContainersPtII`

A second class that is important in order to create a variable pointing to data represented by containers is the class of containers that have a lattice instance. Here, we will not cover the class of all containers with such an instance, but give large enough classes to cover sum of product data types.

4.4.1 The General Idea

When we want to create a lattice version of a data type, we can add constructors for the `top` and `bot` value and then implement the merge operation similar to the trivial lattice. For example the list type

```
data List (A : Set) : Set where
    [] : List A
    _::_ : A → List A → List A
```

Could have a latticed version like

```
data LatList (A : Set) : Set where
    botList : LatList A
    topList : LatList A
    [] : LatList A
    _::_ : A → LatList A → LatList A
```

where we implement the merge, given a semilattice for A, as

```
_/\_ : LatList A → LatList A → LatList A
topList ∧ b           = b
botList ∧ b           = botList
a ∧ topList           = a
a ∧ botList           = botList
[] ∧ []               = []
(x :: xs) ∧ (y :: ys) = (x ∧A y) :: (xs ∧ ys)
_ ∧ _                 = botList
```

Where we merge the list by comparing the constructors. If they align, the merge is successful, otherwise it results in the botList value. The topList value behaves on merge like a placeholder that takes on the value of the other merged input.

When generalising this construction for containers S⊳ P|, we can start transforming the data type by creating a new container with the trivial lattice of the old shapes $\top_\bot$ as its new shape and transform the positions accordingly. Before we do this construction to create the lattice for a transformed container, we first have a look at how to create lattices for containers in general.

In general, we note that the semantic extension of a container is a sigma type

```
[[_]] : Set → Set
[[ S ▷ P ]] R = ∃[ s of S ] (P s → R)
```

This means that if we find a lattice instance for this sigma type, we find a lattice instance for the functor represented by the container. The main problem here is that in a type Σ A B, the B depends on the first element of type A. Further, assumeing that we will have to find a lattice instance for both A and

all `B a`, all equations from the lattice on `A` that hold on an object `a : A` have to be translated into respectable equations over `B a`. We can formulate this using the concept of an indexed lattice

4.4.2 Indexed Lattice

We begin by defining what an indexed semilattice is.

Definition 4.4:1: Indexed Semilattice

Given a semilattice over a type `L` and a family of types `Q : L → Set`, together with an operation

```
_<>_ : Q a → Q b → Q (a <>L b)
```

this is called an `L-indexed semilattice` (or *semilattice indexed over L*, or just *indexed semilattice*, when L is clear from the context) iff the following laws hold forall `Qi : Q i`:

```
associative : Qx <> (Qy <> Qz) ≡⟨ L.associative || Q ⟩≡ (Qx <> Qy) <> Qz
commutative : Qx <> Qy         ≡⟨ L.commutative || Q ⟩≡ Qy <> Qx
idempotent  : Qx <> Qx         ≡⟨ L.idempotent  || Q ⟩≡ Qx
```

Notice how in the above definition, we have to use the semilattice properties of `L` to define equations that hold over the changing type of the `Qi`. The type of the `Qi` changes with every semilattice operation merging two `Qx : Q x` and `Qy : Q y` together to form an element of type `Q (x <>`L` y)`. Therefore, the equations have to account for this type change. For example, in the idempotent case, we have to prove an equation where the left hand side is of type `Q (x <>L x)` and the right hand side of type `Q x`, so we have to specify that the type change is performed by using the original equation from the semilattice `L`. An explanation of how this works in some more detail can be found in Chapter 2 Section 2.4.5.

We can now redefine the order relation for these lattices.

Definition 4.4:2: **Order of an Indexed Lattice**

Given a lattice indexed over a lattice `L` with type family `Q : L → Set`, we define its order relation as

```
_P_ : Q a → Q b → Set
_P_ {a} {b} Qa Qb = ∃[ aPb of (a Pₗ b) ] (Qa <> Qb ≡⟨ aPb || Q ⟩≡ Qb)
```

This defines a pre-order where if the indices are in their respective order, then the indexed elements are in order by the same equation, only that we change the type according to the proof of order of the indices.

Lemma 4.4:3

The relation defined in Definition 4.4:2 defines an indexed pre-order

Proof 4.4:4

As the proofs for the pre-order properties have to mimic the respective proofs of the pre-order of the indices, we will put the proofs side by side. This will also show the type change of the type family `Q`. Reflexivity follows directly from idempotency. For transitivity, we have

```
(ab=b , QaQb=Qb) : ∃[ aPb of a Pₗ b ] (Qa <> Qb ≡⟨ aPb || Q ⟩≡ Qb)
(bc=c , QbQc=Qc) : ∃[ bPc of b Pₗ c ] (Qb <> Qc ≡⟨ bPc || Q ⟩≡ Qc)
```

```
a <>ₗ c             =< bc=c >
(a <>ₗ b <>ₗ c)     =< L.associative >
(a <>ₗ b) <>ₗ c     =< ab=b >
b <>ₗ c             =< bc=c >
c                   qed
```

```
Qa <> Qc            =< QbQc=Qc >
(Qa <> Qb <> Qc)    =< associative >
(Qa <> Qb) <> Qc    =< QaQb=Qb >
Qb <> Qc            =< QbQc=Qc >
Qc                  qed
```

Next, we create indexed bounded semilattices:

Definition 4.4:5: **Indexed Bounded Semilattice**

Given a semilattice indexed over `L`, where `L` is a bounded semilattice with $\mathtt{e}_{\mathtt{L}}$, together with an element

```
e : Q eL
```

we call it a `bounded semilattice indexed over L` (or short: *indexed bounded semilattice*) iff the following laws hold:

```
identity-left : e <> Qx ≡⟨ L.identity-left || Q ⟩≡ Qx
```

We can prove that the identity element is also a right identity due to commutativity.

Lemma 4.4:6

Given a bounded semilattice indexed over `L`, then it also holds that

```
identity-right : Qx <> e ≡⟨ L.identity-right || Q ⟩≡ Qx
```

Proof 4.4:7

```
x <> e =< commutative >
e <> x =< identity-left >
x      qed
```

```
Qx <> e =< commutative >
e <> Qx =< identity-left >
Qx      qed
```

We can now define indexed lattices after some naming conventions.

Definition 4.4:8: **Indexed Bounded Meet Semilattice**

An indexed bounded semilattice where `_<>_` is called `_∧_` (pron. "meet") and `e` is called `top` is called an **indexed bounded meet semilattice**

Definition 4.4:9: **Indexed Bounded Join Semilattice**

An indexed bounded semilattice where `_<>_` is called `_∨_` (pron. "join") and `e` is called `bot` is called an **indexed bounded join semilattice**

Definition 4.4:10: **Indexed Lattice**

An indexed bounded meet and indexed bounded join semilattice, both indexed over the lattice `L` is called and **indexed lattice** iff the following laws hold forall `Qi : Q i`:

```
absorb-∧ : Qx ∧ (Qx ∨ Qy) ≡⟨ L.absorb-∧ || Q ⟩≡ Qx
absorb-∨ : Qx ∨ (Qx ∧ Qy) ≡⟨ L.absorb-∨ || Q ⟩≡ Qx
```

We can now create lattices that are indexed by another lattice. We will use this to define lattices for sigma types and, as an extent, data types defined by containers.

4.4.3 Lattices for Sigma Types and Container Extensions

We remember again how the container extension is a sigma type

```
[[_]] : Set → Set
[[ S ▷ P ]] R = ∃[ s of S ] (P s → R)
```

So in order to create a lattice instance for a container extension, we have to create a lattice instance for a sigma type. This can be done with this simple theorem:

Theorem 4.4:11

Given a lattice instance for a type A and a lattice instance of B indexed over A. Then there is a lattice instance for Σ A B with the lattice operations

```
top = topA , topB
bot = botA , botB
(a1 , b1) ∧ (a2 , b2) = (a1 ∧A a2) , (b1 LatB.∧B b2)
(a1 , b1) ∨ (a2 , b2) = (a1 ∨A a2) , (b1 LatB.∨B b2)
```

Proof 4.4:12

Going through all properties, applying the respecitve lattice properties for A and B simultaneously.

Therefore, in order to create a lattice instance for a container extension, we have to create a lattice for the shapes S and a lattice indexed over the lattice of S for the type `s-> P s → A`, for some type A. In the next section, we give a concrete construction for one possible lattice instance that only has a small number of prerequisites.

4.4.4 Lattices for Containers with Decidable Equality on Shapes

We remember that there is a trivial lattice instance for types that have decidable equality. We quickly repeat the definition from Chapter 2 Section 2.6.5. Every type can be equipped with extra constructors for `top` and `bot` using the data type

```
data _⊤⊥ (A : Set) : Set where
    topTB : A⊤⊥
    botTB : A⊤⊥
    valTB : A → A⊤⊥
```

This type has a lattice instance when there is a decidable equality instance on A, using the operators

```
_∧_ : X⊤⊥ → X⊤⊥ → X⊤⊥
topTB     ∧ x          = x
botTB     ∧ _          = botTB
x         ∧ topTB      = x
_         ∧ botTB      = botTB
(valTB x) ∧ (valTB y) = if x == y then valTB x else botTB

_∨_ : X⊤⊥ → X⊤⊥ → X⊤⊥
topTB     ∨ _          = topTB
botTB     ∨ x          = x
_         ∨ topTB      = topTB
x         ∨ botTB      = x
(valTB x) ∨ (valTB y) = if x == y then valTB x else topTB
```

We will not repeat the proof for this forming a lattice here because it will become visible when doing proofs on lattices indexed over a trivial lattice. There is one equation that comes in handy when making the proofs. We will not show their implementation, as it is a bit technical, but they are useful to have around:

```
meetVal : x ≡ y → (valTB x) ∧ (valTB y) ≡ valTB x
meetBot : x ≢ y → (valTB x) ∧ (valTB y) ≡ botTB
joinVal : x ≡ y → (valTB x) ∨ (valTB y) ≡ valTB x
joinTop : x ≢ y → (valTB x) ∨ (valTB y) ≡ topTB
```

At this point we note that decidable equality is a bit more difficult under cubical type theory. The reason is that when two objects `a b : A` are equal and an equality is returned, is it not immediately clear which equality. It could be an equality that is realised by an isomorphism that potentially changes the shape of the data when used in coercions. Further, when switching the elements to check the equality for `b` and `a`, a completely different equality could be returned that is not even symmetric to the equation obtained when checking `a` and `b` for equality. All this is of course nonsensical in most cases when dealing with a form of computable equality. Therefore, we supply a definition of decidable equality that at least forbids univalence within the equalities it provides, effectively locally turning off univalence.

Definition 4.4:13: **Non-Univalent Decidable Equality**

Given a type A and an operation

```
_==_ : (a : A) → (b : A) → Dec (a ≡ b)
```

is called a **non-univalent decidable equality** iff the following laws hold:

```
decEq-refl   : a == a ≡ yes refl
decEq-sym    : a == b ≡ yes a=b → b == a ≡ yes (sym a=b)
decEq-sym-no : a == b ≡ no ¬a=b → b == a ≡ no (¬a=b ∘ sym)
decEq-trans  : a == b ≡ yes a=b → b == c ≡ yes b=c → a == c ≡ yes (trans a=b b=c)
```

Side Note 4.4:14

Please do not confuse the non-univalent decidable equality with turning off univalence entirely. Univalence can still be used on the described object and the entire proof stays computable. Only the computed equalities inside of the construction just do not use univalence.

We will now prove that every functor of a container where the shapes S have decidable equality can be turned into a functor with a lattice instance, if the type A wrapped in the functor has a lattice instance as well. The main idea of this proof is that we can use the trivial lattice over the shapes S to turn it into a lattice and then use the pointwise meet and join of the position function from the semantic extension to create the lattice. As an intuition: This corresponds to merging data types by first checking whether they were created by the same constructor and then merging the holes at the same position with the underlying lattice A. An example would be merging two lists by checking whether they have the same length first and then zipping them with the meet or join operation of A.

Though looking like a simple statement, this is by far the biggest proof in this thesis yet. The reason is that the construction often does not compute, so the proof steps have to be done manually. To the extent of our experience, this

proof cannot be done unless cubical Agda is empolyed because we need manual control over which term is evaluated how far at every point. We will not show the complete code of the proof here as it is fairly unreadable and use cubical visualisations to show what is going on.

We start by showing the important lemma needed to prove the theorem that every functor of a container where the shapes have decidable equality can be turned into a functor with a lattice instance. According to theorem 4.4:11, we need to turn the shapes `S` into a lattice and then supply an indexed lattice instance on the lattice indexed type family `\s → P s → A`, where `P` are the positions adapted to the trivial lattice. For as long as `S` has decidable equality, we can turn it into a trivial lattice. Adapting `P` to fit that change is a bit tricky. We will start by defining the construction of the lattice version of the container.

Definition 4.4:15: **Trivially Latticed Container**

Given a container `S ▷ P`, we can create a so-called **trivially latticed container** (short: *latticed container*, denoted as $\texttt{latticedContainer}_{\texttt{S▷P}}$) as

```
S⊤⊥ ▷ (top: Zero bot: Zero val: P)
```

where `(top:_bot:_val:_)` is the destructor of the trivial lattice. For shorthand notation, we write

```
P⊤⊥ : S⊤⊥ → Set
P⊤⊥ = top: Zero bot: Zero val: P
```

(Not to be confused with the $\_^{\top}_{\bot}$ type. `P` itself does not require a decidable equality instance).

When defining the merge operation for the semilattices, given a lattice instance for `A`, we would like to use the pointwise merge of two position functions, like

```
-- this is wrong --
pw-meet : ∀ {s} → (f g : P s → A) → (P s → A)
pw-meet f g ps = f ps ∧ g ps
```

Sadly, this does not work when `s` is supposed to be an object of a lattice. Let us try to define this pointwise merge when the indices `s` of the type family also have to merge.

```
pw-meet : ∀ {s1 s2} → (f : P⊤⊥ s1 → A) → (g : P⊤⊥ s2 → A) → (P (s1 ∧ s2) → A)
pw-meet f g ps = {!!} ∧ {!!}
```

The problem now is, that `f` is a function indexed by `s1` and `g` is a function indexed by `s2`, neither of which fit the output index `s1 ∧ s2`. Therefore, the lemma for the semilattice instance uses a slightly more complicated construction.

Lemma 4.4:16

Given the trivially latticed container $\mathsf{S}^{\top}_{\bot} \triangleright \mathsf{P}^{\top}_{\bot}$, where `S` has a non-univalent decidable equality instance and the type `A` with a lattice instance, then there is an indexed meet semilattice instance using the meet operation

```
_/\P_ : {a : S tb} → (P⊤⊥ a → A) → {b : S tb} → (P⊤⊥ b → A) → (P⊤⊥ (a ∧S⊤ b) → A)
_/\P_ {topTB}   pa {b}       pb pm = pb pm
_/\P_ {valTB x} pa {topTB}   pb pm = pa pm
_/\P_ {valTB x} pa {valTB y} pb pm =
    ifDec x == y
    then (\  x=y → pa (coerce (meetVal x=y || P⊤⊥) pm) ∧A pb (coerce ((x=y || valTB) || P⊤⊥) (coerce (meetVal x=y || P⊤⊥)  pm)))
    else (\ ¬x=y → absurd (coerce (meetBot ¬x=y || P⊤⊥) pm))
```

For better overview, here is the same definition without the coercions, as could be written if all coercions could be done implicitly by the proof system:

```
-- idealised --
_/\P_ : {a : S tb} → (P⊤⊥ a → A) → {b : S tb} → (P⊤⊥ b → A) → (P⊤⊥ (a Λs⊤⊥ b) → A)
_/\P_ {topTB}   pa {b}       pb pm = pb pm
_/\P_ {valTB x} pa {topTB}   pb pm = pa pm
_/\P_ {valTB x} pa {valTB y} pb pm =
    if x == y
    then (pa pm ΛA pb pm)
    else (absurd pm)
```

To make proofs more readable, we use the notation

```
syntax _/\P_ {x} Qx {y} Qy = Qx [ x ]/\P[ y ] Qy
```

to make the indices for the functions explicit.

Side Note 4.4:17

The coercions in the last clause of the position function meet are necessary because `pm : P⊤⊥ (a Λs⊤⊥ b)`, so it has to be coerced into

```
                                coerce (meetVal x=y || P⊤⊥) pm  : P⊤⊥ a
coerce ((x=y || valTB) || P⊤⊥) (coerce (meetVal x=y || P⊤⊥) pm) : P⊤⊥ b
```

Proving Lemma 4.4:16 is a massive proof, so we will chop it down into some individual sublemmata.

Lemma 4.4:18

The operation from Lemma 4.4:16 is idempotent, so forall `x : S⊤⊥`

```
Qx [ x ]/\P[ x ] Qx ≡⟨ idempotent_{s⊤⊥} ⟩≡ Qx
```

Proof 4.4:19

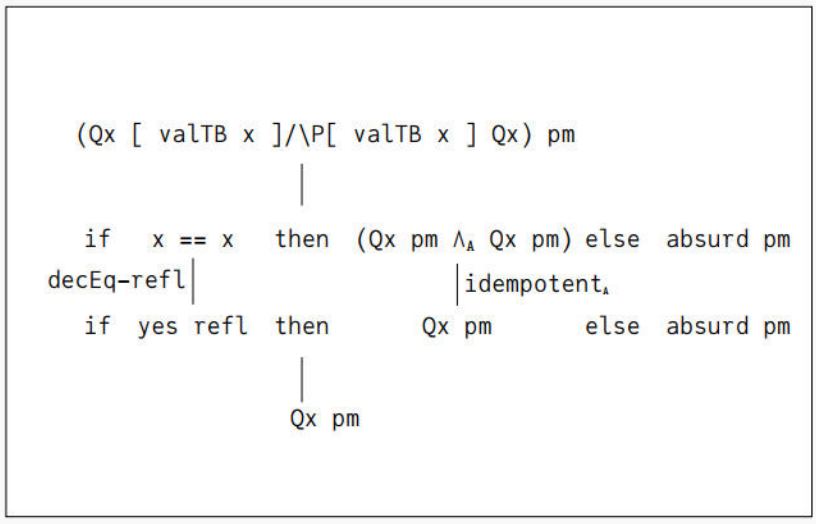
We case split on the index `x`. The `topTB` and `botTB` cases are trivial, as there the positions are just the absurd function. In the case of `x = valTB x`, we have to show that

```
Qx [ valTB x ]/\P[ valTB x ] Qx ≡⟨ idempotent_{s⊤⊥} ⟩≡ Qx
```

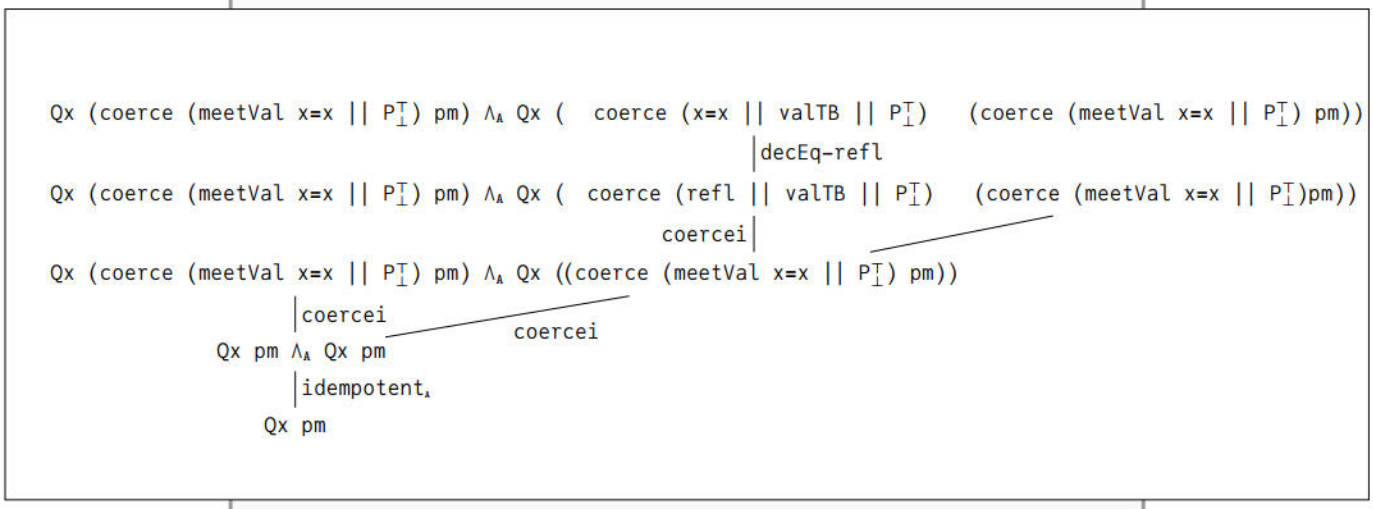
So the type changes along:

```
(pm : P⊤⊥ (valTB x Λ_{s⊤⊥} valTB x)) → A
                  |idempotent_{s⊤⊥}
      (pm : P⊤⊥ (valTB x)) → A
```

We start by showing the proof without coercions:



The coercions do not fit on a page, so we only show the most important argument. In order to apply the idempotency of A, we have to do the following transformation:



Lemma 4.4:20

The operation from Lemma 4.4:16 is commutative, so

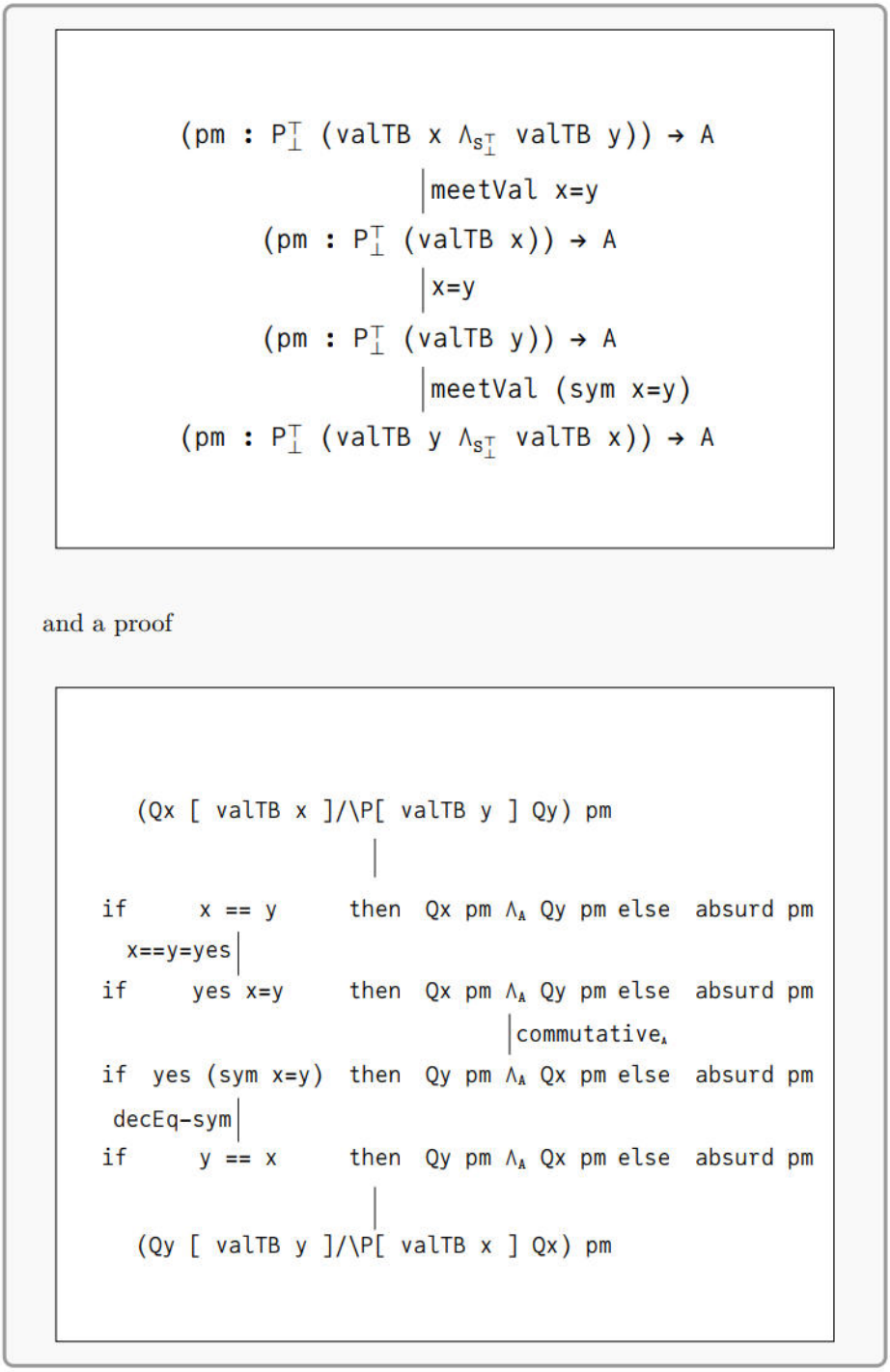
```
Qx [ x ]/\P[ y ] Qy ≡⟨ S⊤⊥.commutative ⟩≡ Qy [ y ]/\P[ x ] Qx
```

Proof 4.4:21

We case split on the indices `x` and `y`. All but one of the cases is trivial as there are no positions on `topTB` and `botTB`. In the two cases where the indices merge to one computable value, the statement follows through simple reduction. Therefore, we look at the case where `x = valTB x` and `y = valTB y`. In general, the type change here is

```
(pm : P⊤⊥ (valTB x Λs⊤⊥ valTB y)) → A
                  |commutative s⊤⊥
(pm : P⊤⊥ (valTB y Λs⊤⊥ valTB x)) → A
```

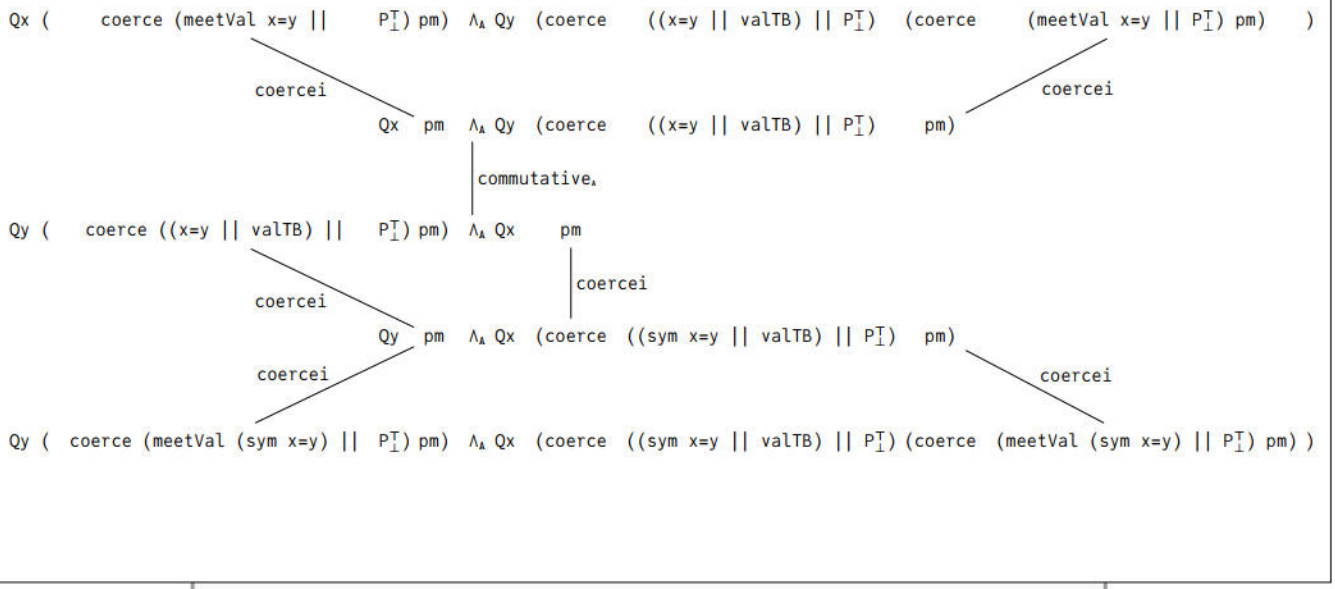
We furthermore split for the result of `x == y`. In the case of `x==y=yes : x == y ≡ yes x=y`, without coercions, we get a type change of


(pm : P⊤⊥ (valTB x Λs⊤⊥ valTB y)) → A
meetVal x=y
(pm : P⊤⊥ (valTB x)) → A
x=y
(pm : P⊤⊥ (valTB y)) → A
meetVal (sym x=y)
(pm : P⊤⊥ (valTB y Λs⊤⊥ valTB x)) → A

and a proof

(Qx [valTB x]/\P[valTB y] Qy) pm
if x == y then Qx pm ΛA Qy pm else absurd pm
x==y=yes
if yes x=y then Qx pm ΛA Qy pm else absurd pm
commutativeA
if yes (sym x=y) then Qy pm ΛA Qx pm else absurd pm
decEq-sym
if y == x then Qy pm ΛA Qx pm else absurd pm
(Qy [valTB y]/\P[valTB x] Qx) pm

We, again, look at the important part of the coercions



Now, we have to look at the case of
`x==y=yes : x == y ≡ no ¬x=y`, where the meet results in a `botTB`. It has the simple type change

```
(pm : P⊤⊥ (valTB x Λs⊤ valTB y)) → A
                 |meetBot ¬x=y
        (pm : P⊤⊥ botTB) → A
                 |meetBot (¬x=y ∘ sym)
(pm : P⊤⊥ (valTB y Λs⊤ valTB x)) → A
```

Without coercions, the proofs goes as

```
            (Qx [ valTB x ]/\P[ valTB y ] Qy) pm
                                  │
          if      x == y      then  Qx pm Λ_A Qy pm else  absurd pm
             x==y=no │
          if     no ¬x=y      then  Qx pm Λ_A Qy pm else  absurd pm
                                                          │
          if  no (¬x=y ∘ sym) then  Qy pm Λ_A Qx pm else  absurd pm
 decEq-sym-no x==y=no │
          if      y == x      then  Qy pm Λ_A Qx pm else  absurd pm
                                  │
            (Qy [ valTB y ]/\P[ valTB x ] Qx) pm
```

The coercions in this case are basically trivial, as we are only proving that all functions from the empty type `absurd : Zero → A` are equal.

Lemma 4.4:22

The operation from Lemma 4.4:16 is associative, so

```
Qx [ x ]/\P[ y Λ_S̲ z ] (Qy [ y ]/\P[ z ] Qz) ≡⟨ S̲.associative ⟩≡ (Qx [ x ]/\P[ y ] Qy) [ x Λ_S̲ y ]/\P[ z ] Qz
```

Proof 4.4:23

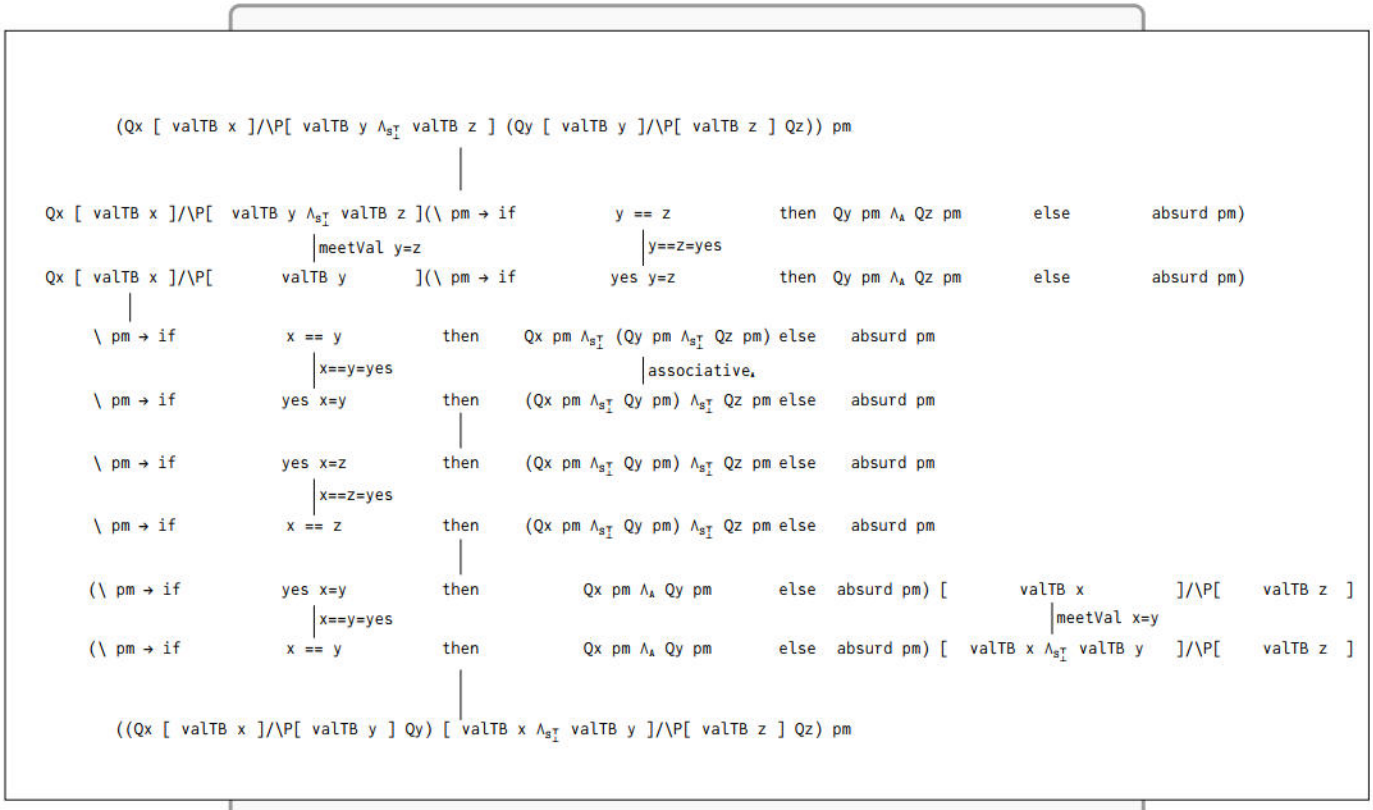
This proof is again trivial for all cases except `x = valTB x`, `y = valTB y` and `z = valTB z`. Here, we have to distinguish the consistent outcomes of `x == y`, `y == z` and `x == z`. We begin with the case of a successful meet, so

```
x==y=yes : x == y ≡ yes x=y
y==z=yes : y == z ≡ yes y=z
x==z=yes : x == z ≡ yes x=z
```

The type change in this case is

```
(pm : P⊤⊥ (valTB x Λs⊤⊥ (valTB y Λs⊤⊥ valTB z))) → A
                          |meetVal y=z
        (pm : P⊤⊥ (valTB x Λs⊤⊥ valTB y)) → A
                          |meetVal x=y
             (pm : P⊤⊥ (valTB x)) → A
                          |meetVal x=z
        (pm : P⊤⊥ (valTB x Λs⊤⊥ valTB z)) → A
                          |meetVal x=y
(pm : P⊤⊥ ((valTB x Λs⊤⊥ valTB y) Λs⊤⊥ valTB z)) → A
```

Without coercions, the proofs looks like



The coercions in this case are indeed tricky. The reason is that during the type change from

```
(pm : P⊤⊥ (valTB x Λs⊤⊥ valTB y)) → A
                  |meetVal x=y
      (pm : P⊤⊥ (valTB x)) → A
                  |meetVal x=z
(pm : P⊤⊥ (valTB x Λs⊤⊥ valTB z)) → A
```

The type of the `pm` parameter changes between the results of the meet value of `valTB x Λs⊤⊥ valTB y` and `valTB x Λs⊤⊥ valTB y`. This type change also needs to be reflexted in the coercions, especially in the input parameter of the `Qz` function, where `pm` has to be coerced into the type `P⊤⊥ (valTB z)` throughout this entire transformation. This requires a few equations on coercions that have been proven in the code but would be too technical to explain here, so we just show the important parts of the proof. At first, the input `pm` to `Qz` is transformed via the `meetVal x=y` equation to `P⊤⊥ (valTB x)`, so we start with the coercions

```
Qz (coerce ((y=z || valTB) || P⊤⊥) (coerce ((x=y || valTB) || P⊤⊥) pm))
```

We then apply a law, proven in the cubical standard library [4] that states that two consecutive coercions (especially over a congruence) can be expressed as one coercion over the transitive equation, so

```
Qz (coerce ((y=z || valTB) || P⊤⊥) (coerce ((x=y || valTB) || P⊤⊥) pm)) =< coerce-trans-subst >
Qz (coerce (((trans x=y y=z) || valTB) || P⊤⊥) pm)
```

We now use the property that our decidable equality preserves transitivity, so we get

```
yes x=z =< x==z=yes>
x == y =< decEq-trans x==y=yes y==z=yes >
yes (trans x=y y=z)
```

Now, through the injectivity of the decidables type constructors, we get the fact that

```
x=z=transx=yy=z : x=z ≡ trans x=y y=z
```

We can use this to acquire that

```
Qz (coerce (((trans x=y y=z) || valTB) || P⊤⊥) pm) =< x=z=transx=yy=z >
Qz (coerce ((x=z || valTB) || P⊤⊥) pm)
```

finally, we have to make the entire term after the application of associativity fit the definition of `... [ valTB x ]/\P[ valTB z ] Qz` again. We remember the definition from Lemma 4.4:16

```
(Qx' [ valTB x ]/\P[ valTB y ] Qz) pm =
    ifDec x == z
    then (\  x=z → Qx' (coerce (meetVal x=z || P⊤⊥) pm) Λ₄ Qz (coerce ((x=z || valTB) || P⊤⊥) (coerce (meetVal x=z || P⊤⊥)  pm)))
    else (\ ¬x=z → absurd (coerce (meetBot ¬x=z || P⊤⊥) pm))
```

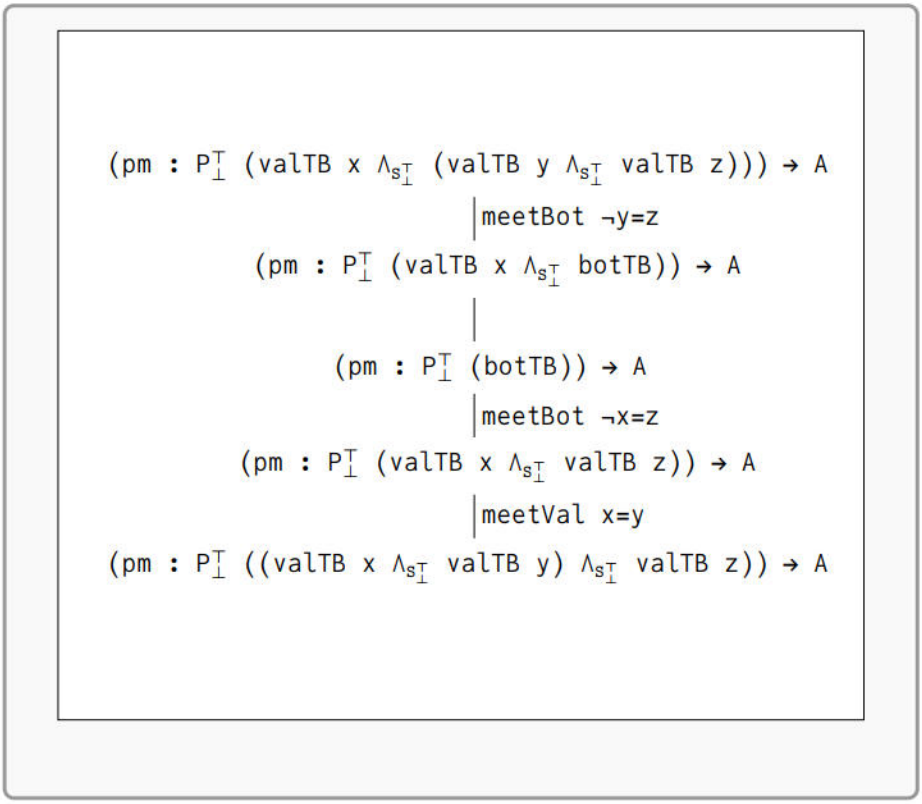
Here we see, that the `Qz` has to get the `pm` through two coercions. We add this extra coercion by applying a law (that is a bit too technical, so we only prove it in the actual code) that coercing a value back and forth via `sym` does not change the value, so we get

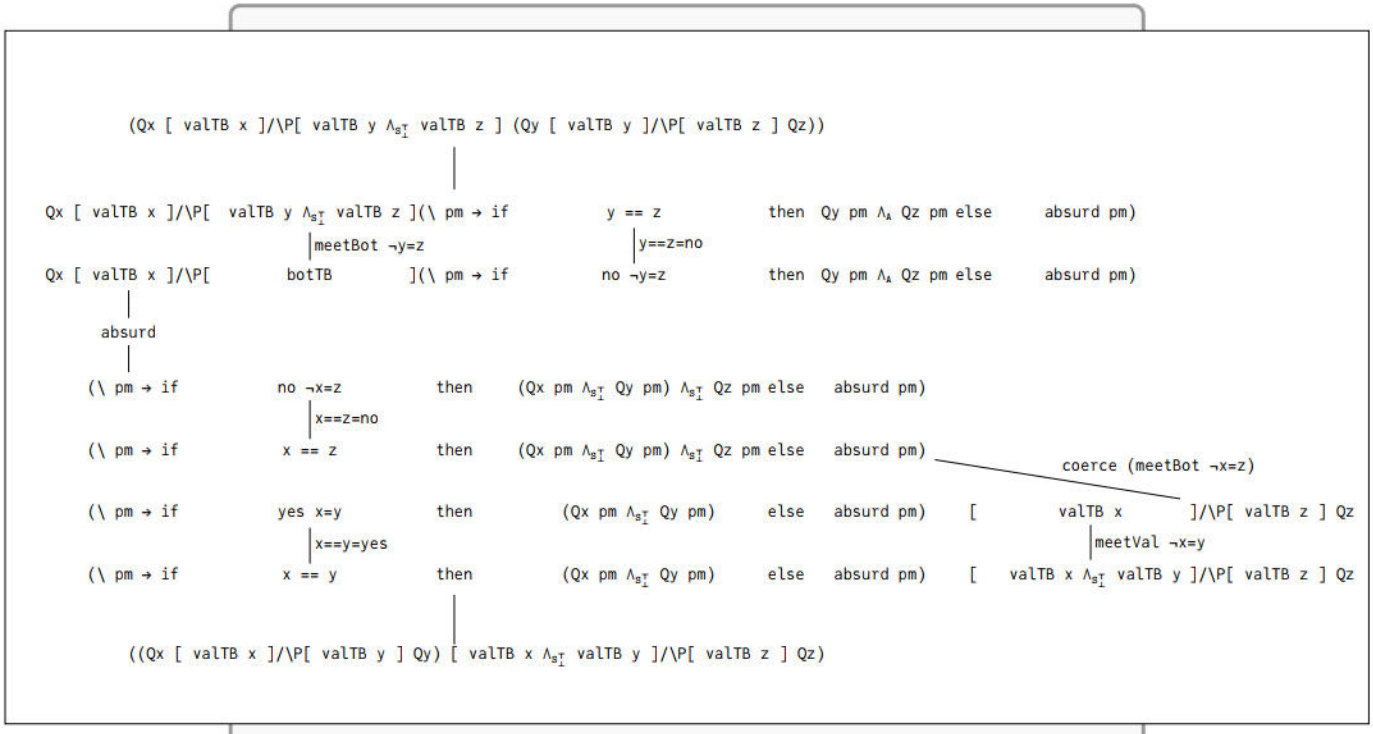
```
Qz (coerce ((x=z || valTB) || P⊤⊥) pm) =< coerce-sym-subst >
Qz (coerce (x=z || valTB || P⊤⊥) (coerce (meetVal x=z || P⊤⊥) (coerce (sym (meetVal x=z) || P⊤⊥) pm)))
```

All other coercions are straightforwardly resolved similar to the proof of Lemma 4.4:20.
There are three more consistent results of the decidable equality.
The first two are analogous, as they both describe one of the equality checks to fail (only two of those three cases produce consistent equalities). We showcase this at the case of

```
x==y=yes : x == y ≡ yes x=y
y==z=no  : y == z ≡ no ¬y=z
x==z=no  : x == z ≡ no ¬x=z
```

The type change in this case is


(pm : P⊤⊥ (valTB x Λs⊤⊥ (valTB y Λs⊤⊥ valTB z))) → A
meetBot ¬y=z
(pm : P⊤⊥ (valTB x Λs⊤⊥ botTB)) → A
(pm : P⊤⊥ (botTB)) → A
meetBot ¬x=z
(pm : P⊤⊥ (valTB x Λs⊤⊥ valTB z)) → A
meetVal x=y
(pm : P⊤⊥ ((valTB x Λs⊤⊥ valTB y) Λs⊤⊥ valTB z)) → A



The crucial shortcut for this and the analogous case is, that there is no case for `... [ botTB ]/P[ ... ] ...` or `... [ ... ]/P[ botTB ] ...`, so we can derive the equivalence faster due to the trivial equivalence of any functions `absurd : Zero → A`. In the above case however, we afterwards have to be careful about the coercions. When we apply the `coerce (meetBot ¬x=z)` to the index of the type of `pm`, we have to transform the non-existant `pm` over the result of a failed meet into a `pm` that is defined for the value from the successful meet between `valTB x` and `valTB y`. We will show by example how these cases are handled. We can fill the following coercion:

```
Qz (coerce (y=z || valTB || P⊤) (coerce (x=y || valTB || P⊤) (coerce (meetVal x=y || P⊤)(coerce (meetBot ¬x=y || P⊤) pm) )))
                                                                                                  |coercei
Qz (coerce (y=z || valTB || P⊤) (coerce (x=y || valTB || P⊤) (coerce (meetVal x=y || P⊤)            pm                )))
```

To end up with a term that only uses successful meets as is used in `_[_]/\P[_]_`. The type change for the `pm` throughout the above coercions now goes as

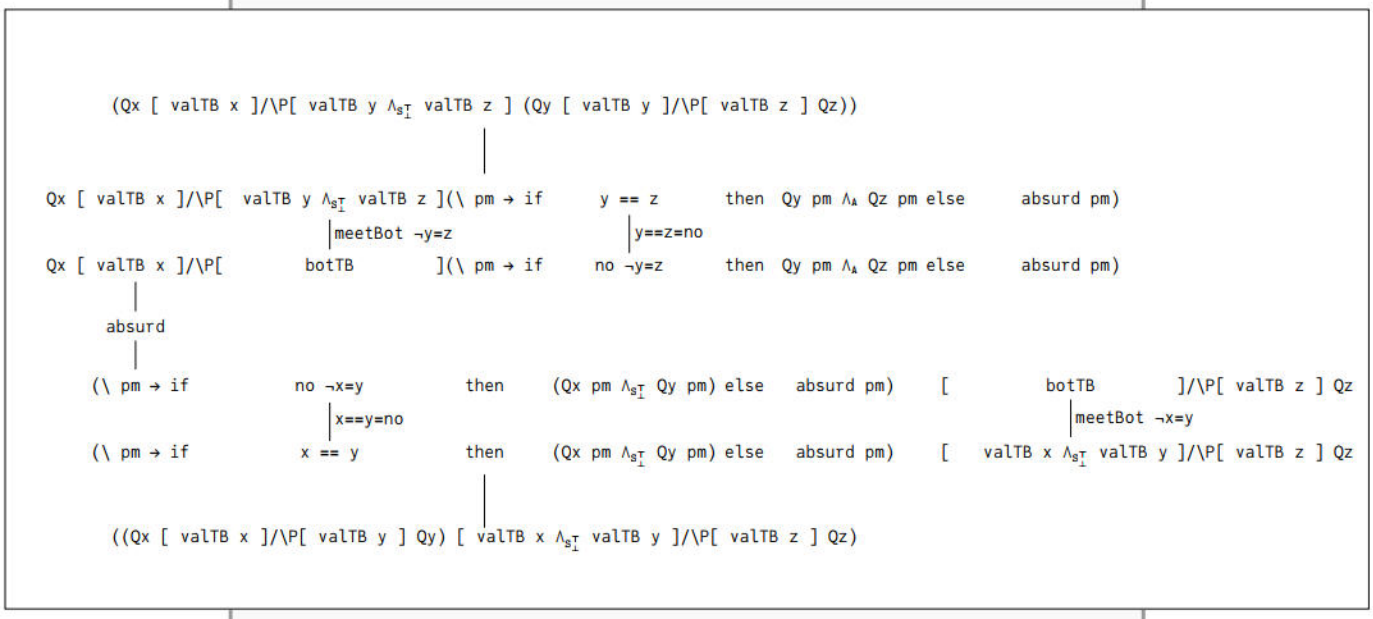
```
                (pm : P⊤ (botTB))
                        |meetBot ¬x=y
            (pm : P⊤ (valTB x ∧s⊤ valTB y))
                        |meetVal x=y
              (pm : P⊤ (valTB x))
                        |x=y
              (pm : P⊤ (valTB y))
                        |y=z
              (pm : P⊤ (valTB z))
```

This technique is applied whenever we have to switch from unsuccsessful meets to successful ones. Further we note, that those cases are inconsistent because for example both a `x=y` and `¬x=z` can exists (where the positive equality comes from the definition of `_[_]/\P[_]_`), meaning that these cases cannot occur, but we still have to give the

transformation in order for the proof to compile.
In the final case of

```
x==y=yes : x == y ≡ no ¬x=y
y==z=no  : y == z ≡ no ¬y=z
```

we can make even stronger use of equivalence of any functions `absurd : Zero → A`, as no meet in the computation is successful:



In all of those cases, all coercions again resolve straight forwardly like in the proof of Lemma 4.4:20, together with the above technique to change unsuccsessful meets into successful ones.

We now show, that the construction from Lemma 4.4:16 are not just an indexed semilattice, but a bounded indexed semilattice.

Lemma 4.4:24

The indexed semilattice from Lemma 4.4:16 forms a bounded indexed meet semilattice with the identity element

```
top = absurd
```

Proof 4.4:25

The `identity-left` property follows trivially from evaluation.

The interesting thing is, that we can now create a bounded meet semilattice completely analogously:

Lemma 4.4:26

Given the trivially latticed container $\mathsf{S}^{\top}_{\bot} \triangleright \mathsf{P}^{\top}_{\bot}$, where S has a non-univalent decidable equality instance and the type A with a lattice instance, then there is an indexed join semilattice instance using the join operation

```
_\/P_ : {a : S tb} → (P⊤⊥ a → A) → {b : S tb} → (P⊤⊥ b → A) → (P⊤⊥ (a V_S⊤ b) → A)
_\/P_ {topTB}   pa {b}       pb pm = pb pm
_\/P_ {valTB x} pa {topTB}   pb pm = pa pm
_\/P_ {valTB x} pa {valTB y} pb pm =
    ifDec x == y
    then (\  x=y → pa (coerce (joinVal x=y || P⊤⊥) pm) V_A pb (coerce ((x=y || valTB) || P⊤⊥) (coerce (joinVal x=y || P⊤⊥)  pm)))
    else (\ ¬x=y → absurd (coerce (joinTop ¬x=y || P⊤⊥) pm))
```

For better overview, here is the same definition without the coercions, as could be written if all coercions could be done implicitly by the proof system:

```
-- idealised --
_\/P_ : {a : S tb} → (P⊤ a → A) → {b : S tb} → (P⊤ b → A) → (P⊤ (a V_S⊤ b) → A)
_\/P_ {botTB}   pa {b}       pb pm = pb pm
_\/P_ {valTB x} pa {botTB}   pb pm = pa pm
_\/P_ {valTB x} pa {valTB y} pb pm =
    if x == y
    then (pa pm V_A pb pm)
    else (absurd pm)
```

To make proofs more readable, we use the notation

```
syntax _\/P_ {x} Qx {y} Qy = Qx [ x ]\/P[ y ] Qy
```

to make the indices for the functions explicit.

Proof 4.4:27

Completely analogously to Lemma 4.4:16

Lemma 4.4:28

The indexed semilattice from Lemma 4.4:26 forms a bounded indexed meet semilattice with the identity element

```
bot = absurd
```

Proof 4.4:29

The `identity-left` property follows trivially from evaluation.

We now have the fundamental building block to create a lattice instance for the types represented by containers where the shapes have decidable equality.

Theorem 4.4:30

The bounded semilattices from Lemma 4.4:16, 4.4:24, 4.4:26 and 4.4:28 together form a lattice.

Proof 4.4:31

We have already shown in Lemma 4.4:24 and Lemma 4.4:28 that the structures in question are bounded semilattices, so all we need to show are their interacting absorption laws.
As both absorption laws are analogous, we only prove that

```
Qx [ x ]/\P[ x ∨ₛ y ] (Qx [ x ]\/P[ y ] Qy) ≡⟨ absorb-∧ₐ || (\s → P⊤⊥ s → A) ⟩≡ Qx
```

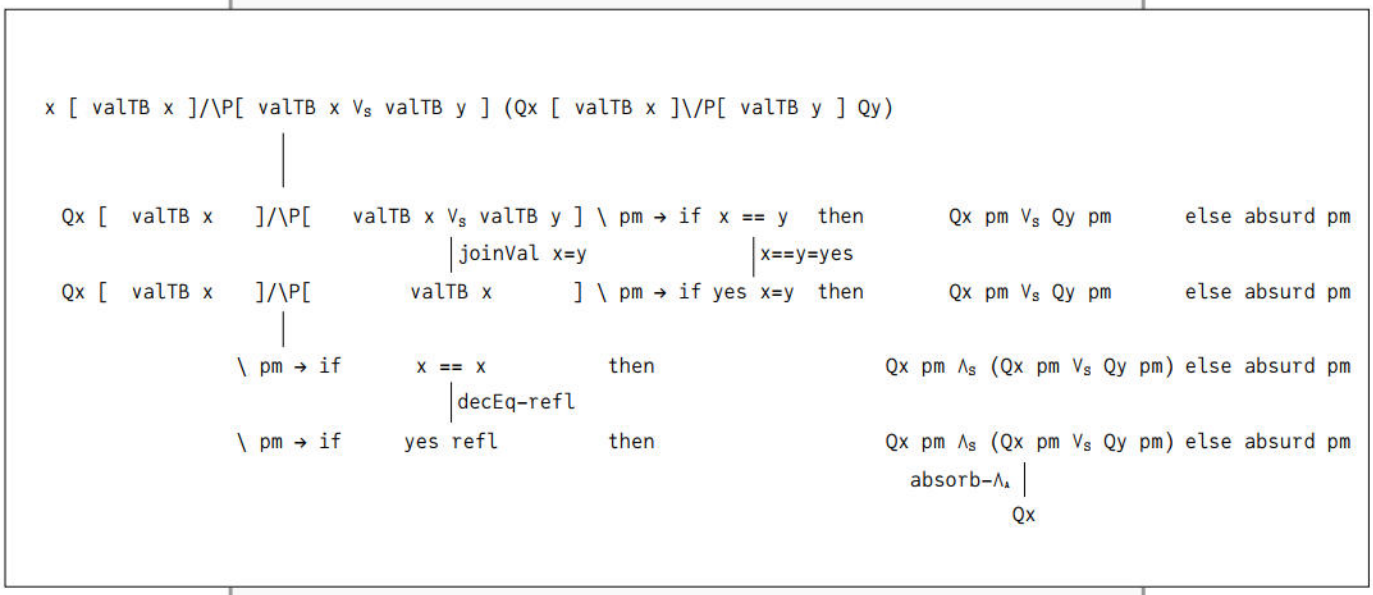
The only non-trivial case is again with `x = valTB x` and `y = valTB y`. We again distinguish the cases for `x == y`. In the case of

```
x==y=yes : x == y ≡ yes x=y
```

the type change goes as follows:

```
(pm : P⊤⊥ (valTB x ∧ₛ (valTB x ∨ₛ valTB y))) → A
                    |joinVal x=y
      (pm : P⊤⊥ (valTB x ∧ₛ valTB x)) → A
                    |meetVal refl
           (pm : P⊤⊥ (valTB x)) → A
```

Without coercions, we have



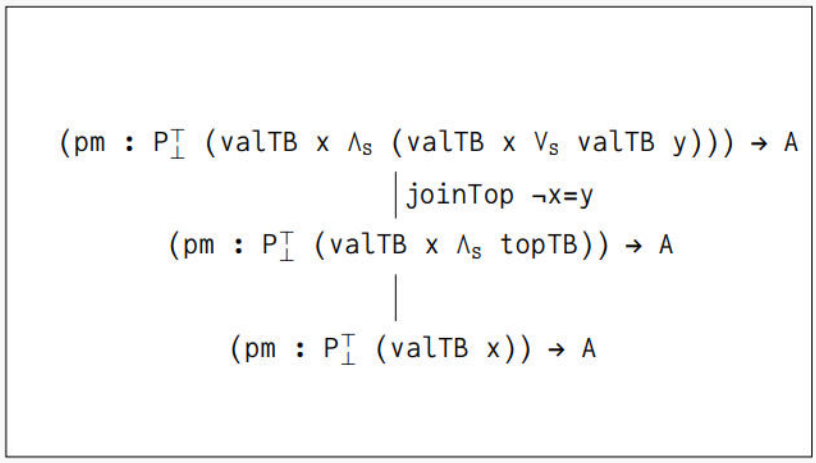
Resolving the coercions is similar to Lemma 4.4:22, only that we use one additional rule, where we don't show the proof because it is too technical:

```
coerce-refl : coerce refl x ≡ x
```

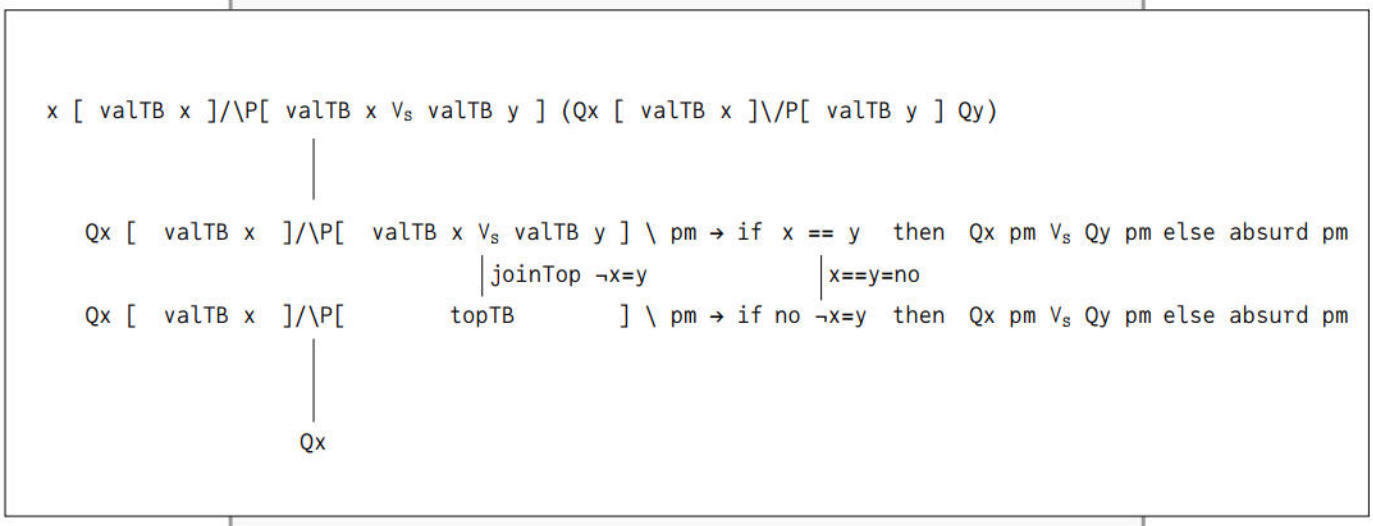
In the case of

```
x==y=no : x == y ≡ no ¬x=y
```

the type change goes as follows:



Without coercions, we have



where the coercions are removed with the same techniques as above.

This concludes, that we can derive a lattice instance for data defined via a container where the shapes have decidable equality. Together with the results form Section 4.3.4.4, if the container is additionally shapely, we can put the type monadically in stasis. The only property required to show how this can be used for solving is to create a lattice bi-threshold variable from the latticed containers into the original container.

4.5 Lattice Bi-Threshold Variables for Latticed Containers

Code-Tag: `BiThresholdsAndContainersPtIII`

We would now like to have a variable from our latticed containers into the original container. Naïvely, we could try to write a function like

```
--naive, not working
trivialVariable : LBVar ([[ latticedContainer ]] A) ([[ C ]] A)
```

Sadly, this does not directly work. The reason is the lattice bi-threshold read property from Chapter 3 Definition 3.8:29, by which a read value must not change if the underlying lattice grows. In our example however, we would read out the value for the lattice `A` for every position directly, meaning that the read out value would change if the positions grow. This would therefore not be a lattice bi-threshold variable. There is however an alternative when we are given a variable `varB : LBVar A B` for the positions. Now, we can use `varB` to read out the individual positions whilst retaining the lattice bi-threshold read property.

We begin with a simple variable that can read out the shape of the latticed container.

Lemma 4.5:1

Given the container `C` with decidable equality for the shapes and a bounded semilattice instance for type `A`, then the following two operations make a lattice bi-threshold variable from the latticed container of `C` from Definition 4.4:15 into its semantic extension `[[ C ]] Unit` with the positions being filled with unit types

```
read : [[ latticedContainer_C ]] A → VarAsm ([[ C ]] A)
read (topTB , p)   = unassigned
read (botTB , p)   = conflict
read (valTB s , p) = asm (s , \_ → unit)

write : [[ C ]] A → [[ latticedContainer_C ]] A
write (s , p) = (valTB s , \_ → e_A)
```

Proof 4.5:2

The proof is straight forward by going through all cases and computing the equalities, remove absurd cases or applying the respective rules, so we just quickly recap the conditions to be proven.
First, we have to prove that `read` is a bi-threshold read.

```
isBiThresholdRead : ∀ s s' → s P s' → read s =incAsm= read s'
```

where `P` is the pre-order of the semilattice. The `_=incAsm=_` type is an indexed type that can simplified be expressed as

pseudocode:

```
data _=incAsm=_ (a b : VarAsm X) : Set where
    unas-to-anything :  unassigned =incAsm= a
    asm-eq           :  asm x      =incAsm= asm x
    asm-conf         :  asm x      =incAsm= conflict
    conf-conf        :  conflict   =incAsm= conflict
```

To prevent it from being an indexed data type to make it work in cubical Agda, it is truly expressed as

```
data _=incAsm=_ (a b : VarAsm X) : Set where
    unas-to-anything :  a ≡ unassigned →                a =incAsm= b
    asm-eq           :  a ≡ asm x      → b ≡ asm x     → a =incAsm= b
    asm-conf         :  a ≡ asm x      → b ≡ conflict → a =incAsm= b
    conf-conf        :  a ≡ conflict   → b ≡ conflict → a =incAsm= b
```

In order to prove the `isBiThresholdRead` property, we go through all cases of the shapes of the input and apply the respective rule to get the `_=incAsm=_` property.
After that, we have to prove the `write-read` and `read-write-read` property, which can be trivially done by case splitting on the shapes of the latticed container. We repeat the conditions for overview:

```
write-read      : read (write x) ≡ asm x
read-write-read : ⦇ write (read s) ⦈ >>= read ≡ read s
```

where we use the standard monad instance for variable assignments.

Just being able to read the shape of a container however is insufficient when trying to get a variable into a recursive data type. Our goal is here to create a construction where we not only read the shape of the container, but fill the positions with variables accessing the possibly recursive content of the functor. Each variable can read and write from and into its current position, so together with the monadic `read` from Chapter 3 Section 3.8.3.2, this gives a monadically held-in-stasis data type that can be used for monadic computation using the switching contexts from Section 4.3.

Lemma 4.5:3

Let there be a bounded semilattice instance for the type `A`, and `varB : LBVar A B` be a lattice bi-threshold variable form `A` to `B`. Given a container `S ▷ P` with decidable equality on `S`, together with decidable equality for all positions `P s`, then for every shape `s : S` and position `pm : P s`, together with another condition stated below, there exists a lattice bi-threshold variable from the latticed container extension `[[ latticedContainer C ]] A` into `B` with the operations

```
read : [[ latticedContainer_C ]] A → VarAsm B
read (topTB    , p') = unassigned
read (botTB    , p') = conflict
read (valTB s' , p') with s == s'
... | yes s=s' = read_varB (p' (coerce (s=s' || P) pm))
... | no ¬s=s' = unassigned

write : B → [[ latticedContainer_C ]] A
write (s , p) = (valTB s , \pm' → if pm == pm' then LBVar.write varB b else e_A)
```

The additional condition is that the order relation on the semilattice for $S_\perp^\top$, which is implemented as an equality `x <>_S⊤⊥ y ≡ y`, has to

coincide with the equality given by `x == y`, meaning basically that no univalence can be used to determine whether one value is greater than another. Ignoring coercions, this is formulated as

```
--naive, without coercions
decEqAxiom : ∀ {x y x=y} → (lq : valTB x P_{S⊤⊥} valTB y) →
    (x==y=yes : x == y ≡ yes x=y) → lq ≡ (x=y || valTB)
```

And with coercions as

```
decEqAxiom : ∀ {x y x=y} → (lq : valTB x P_{S⊤⊥} valTB y) → (x==y=yes : x == y ≡ yes x=y) →
    (coerce (\i → if x==y=yes i then valTB x else botTB ≡ valTB y) lq) ≡ (x=y || valTB)
```

where P is the order relation of the respective, bounded semilattice for $S^{\top}_{\bot}$.

Side Note 4.5:4

This lemma needs two additional conditions to be met, that however still fit in nicely with our data type context. First, also the positions need to have decidable equality. As for our recursively latticeable (commuting contexts with monadic variables) we need to have shapely containers anyway, we also directly get decidable equality for the positions. The second condition is that the order relation has to coincide with the equation provided by the decidable equality. As our decidable equality effectively does not allow for any meaningful use of univalence, this is a bit of a stronger constraint, but again, not necessarily one that is hard to fulfill. We have not yet checked with this Agda, but so far all of our data types only use constructs where the implementation of the order relation does not require univalence.

Proof 4.5:5

This proof is not difficult by concept, but tricky when accounting for the coercions. The `isBiThresholdRead` property can be proven by going through all cases for the shapes of the two related containers. The cases can be either resolved by conflict or by straight forward application of the `_=incAsm=_` constructors. The non-trivial case is the one where `s = (valTB x , p1)`, `s' = (valTB y , p2)` and `sPs' : (valTB x , p1) P_latticedCon (valTB y , p2)`, where all decidable equalities resolve positively, so

```
s==x=yes : s == x ≡ yes s=x
s==y=yes : s == y ≡ yes s=y
x==y=yes : x == y ≡ yes x=y
```

We can prove the statement by using the `isBiThresholdRead` property of `varB`. This states that we get an element of `read_varB s =incAsm= read_varB s'` for all `s P s'`. If we provide the two values `s` and `s'` from which the variable reads, we only need to prove that they are related by the pre-order `P` the get the `_=incAsm=_`. Without coercions, we simply provide the read positions, `p1 pm` and `p2 pm` and prove that

```
p1 pm <>_A p2 pm ≡ p2 pm
```

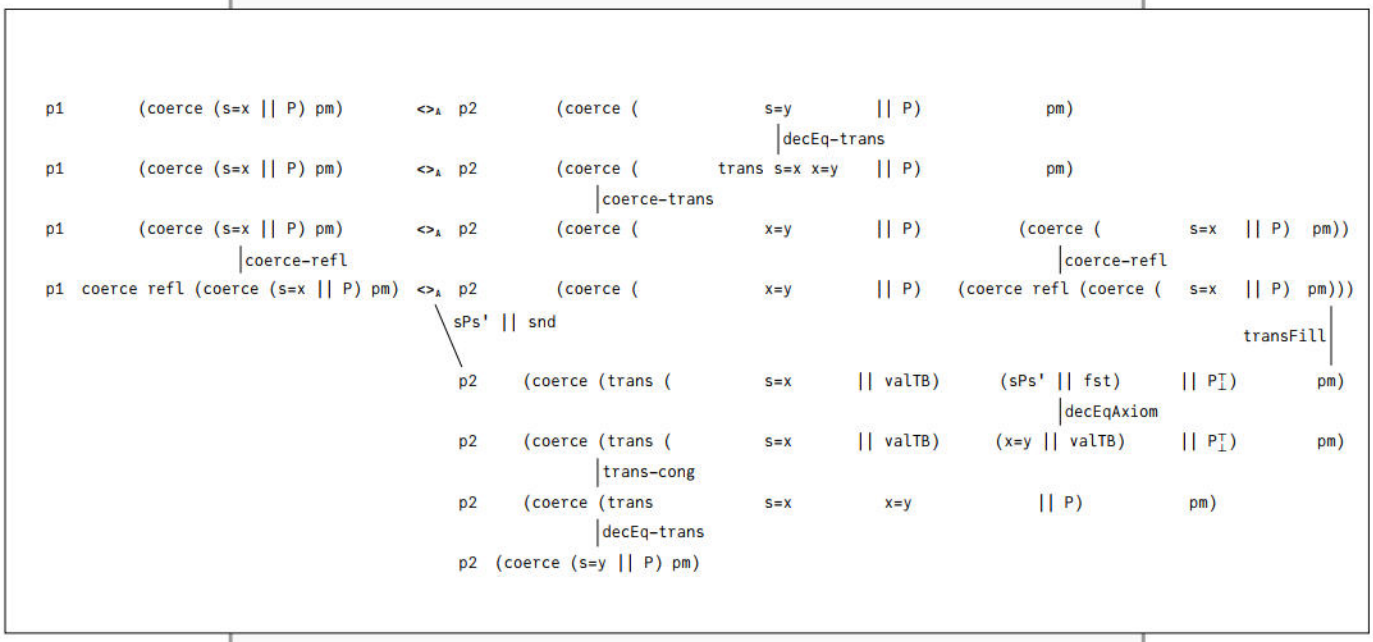
which we can easily do because from `sPs' : (valTB x , p1) P_latticedCon (valTB y , p2)`, we can, colloquially, derive that the pointwise merge of `p1` and `p2` increase, hence, they increase at any given position, so also `pm`. Without coercions, the type of `sPs'` computes to

```
(valTB x , \pm → p1 pm <>_A p2 pm) ≡ (valTB y , \pm → p2 pm)
```

So, without coercions, we can prove the statement as

```
sPs' || snd || (_$ pm)
```

where we first take the second part of the equation and then apply `pm` to the function on each side.
With coercions, we give the elements
`p1 (coerce (s=x || P) pm)` and
`p2 (coerce (s=y || P) pm)` as input to the `isBiThresholdRead` property of `varB`. The proof looks as follows:



We note that here, especially, a lot of complexity of the proof has been omitted for readability, but the main arguments are all contained. We have also used another equivalence that we will not show here because it is very technical, but it has the signature

```
trans-cong : ∀ {a b c : A} {a=b : a ≡ b} {b=c : b ≡ c} {f : A → B} →
    trans (a=b || f) (b=c || f) ≡ trans a=b b=c || f
```

Both the `write-read` and `read-write-read` property follow straight forwardly from the `write-read` property of `varB` and the evaluation of decidable equality using the `decEq-refl` property.

With this lemma, we can fill the shape that we can read out of the variable from Lemma 4.5:1 with variables that can access the content of the positions of the latticed container. We can create a function

```
subvariables : [[ C ]] Unit → [[ C ]] (LBVar ([[ latticedContainer ]] A) B)
```

that fills the shape, and together with the function `read` from the monadic variable of Chapter 3 Theorem 3.8:39 that can monadically read a lattice bi-threshold variable, we can create a function

```
stasisVars :  [[ C ]] Unit → [[ C ]] (M B)
stasisVars = map read ∘ subvariables
```

that can put the data type monadically in stasis and get it ready for solving procedures while enabling us to define conventional functions over it.

Sadly, this PhD has time, money and food constraints, so we finish this section with two conjectures for future work that would need to be proven in order to have the complete correctness for this construction. We note, however, that neither of those should be hard proofs by concept or wrong as a statement, but they require some engineering that this PhD has no further resources for.

Conjecture 4.5:6

The variable from Lemma 4.5:1 can, together with the construction from Lemma 4.5:3, be used to create a lattice bi-threshold variable that directly points to the shape with the variables to the subparts contained in it.

Proof 4.5:7

Future work. Main argument is that the construction from Lemma 4.5:3 should result in a bijective function from the shapes and positions of the `[[ C ]] Unit` object, which should result in a new lattice bi-

threshold variable by composition with the bijection.

Conjecture 4.5:8

The subvariable construction can be used to create variables into container fixpoints of shapely containers with decidable equality on shapes and positions.

Proof 4.5:9

Future work. Creating the fixpoint itself is trivial, but we have to prove that it is reassembled properly by showing the conditions for commutable contexts are met with the lattice bi-threshold variables and that the created variables stay independent if desired.

Before we finish this chapter, we want to give an even better intuition for how strong the classes of data types are that we can automatically reshape to fit solving procedures. To do that, we showcase how to type of type theory fits all required criteria to be automatically reshaped. Before we can completely show the type, we need one more intuition on a class of data types that we can already express, even if we have not yet done a respective construction.

4.6 An Intuition for Monadic Indexed Data

Code-Tag: `MonadicIndexingExample`

Sometimes, and especially in solving, we want constraints on data. In type theory, this is usually done using indexed data types. In general, containers can be used to express indexed data, for example using so-called ornaments [30, 70] or just indexed containers [9], but our containers have to many constraints on them that these constructions might no longer work. We can still express the same things as we could however, as we will showcase in this intuition.

In our example, we look at lists.

```
data List (A : Set) : Set where
    [] : List A
    _::_ : A → List A → List A
```

If we want to transform the list type into the type of vectors, we add an index on the length of the list.

```
data Vec (A : Set) : ℕ → Set where
    []   : Vec A 0
    _::_ : A → Vec A n → Vec A (1+ n)
```

Even if our theory at first only supports normal lists, but we are in a monad `M` with a left absorbing failure operation `fail : ∀ {A} → M A`, we can get a monadic operation that returns a vector of a certain length that has the same shape as a given list as

```
toVec : (n : ℕ) → List A → M (Vec A n)
toVec 0      []         = ❪ [] ❫
toVec (1+ n) (x :: lst) = ❪ (x ::_) (toVec n lst) ❫
toVec _      _          = fail
```

In this transformation, we return the vector of given length with the same shapes and positions as the list iff the list has the correct length (we are not going to prove this for this example, it should be intuitively obvious). If `toVec` is further used to create a monadic variable then we have a variable of the indexed type. In general, indexed data is really just the sigma space of the non-indexed data type together with a filter function.

```
Σ : (A : Set) → (B : A → Set) → Set
```

where `A` is the non-indexed type, but an element can only be given if we have a proof that it is in the set `B`. So vectors would be the type

```
Vec : (A : Set) → (n : ℕ) → Set
Vec A n = Σ (List A) (\lst → length lst ≡ n)
```

If the second argument of the sigma type `B` is decidable forall `A`, we can create such a filter as a monadic action. So if we have

```
B? : ∀ (a : A) → Dec (B a)
```

We can write the monadic filter to constrain the non-indexed type to be indexed as

```
filterM : A → M (Σ A B)
filterM a with B? a
... | yes Ba = return (a , Ba)
... | no ¬Ba = fail
```

And if we can show that for a monadic variable `m` the expression `read m >>= filterM` is the read and (fst (filterM a) |) »= write m |)| is again a monadic variable, we created a monadic variable for an indexed type. As how this is done for sufficiently large classes of indexed data should be obvious to any skilled computer scientist we will not give concrete examples here. The point of this section is that how to deal with indexed data comes from the same mechanic as dealing with failing monads, so we do not need to provide extra structure for monadic variables on indexed data types. It is still important to remind ourselves of this fact because the type of type theory is a heavily indexed type, so in order to get a clear understanding of why this research will lead to solvers for this general type, we need to understand that we have almost all building blocks to express it in terms of monadic variables and the solving structures we developed.

4.7 Towards Encoding the Type of Type Theory

Code-Tag: `TypeOfTypeTheory`

In this section we have a look at how general our data encodings for solving are. So far the constraints, being that the containers of the functions used have to be shapely and have to have decidable equality on its shapes and positions, prevent us from expressing functions as data when done naïvely. Here, we showcase that we can express functions as data while still sticking to the shapely and decidable equality constraints. We do so by refurbishing the type of type theory created by [8, 24] to fit with our current universe of expressible types. This type of type theory does not contain any function arrows while still being able to express the syntax (and semantics) of functions. While we do not have the resources anymore to mechanically show that it can be expressed in our presented formalism, the possibility will become clear enough to the educated reader.

A quick overview over the results in this section:

- We explain the working of the type of type theory presented in [24]
- We add additional constructors, so that our type can additionally express sigma types, universe polymorphism, rudimentary equality types, well formed types (W-types) and recursive functions.
- We informally check whether the type meets the constraints necessary to express it as a type for our solving formalism.

We want to stress how important the result of adding equality, sigma and W-types is: This makes the meta theory strong enough to express a large class of recursive, dependent functions; not only allowing for recursive function synthesis when used in a solving procedure but further opening up the possibility that the entire theory of this PhD might be formalisable in the meta theory. We will further stress in Chapter 5 why this is important for a general theory of solving.

For future work, we leave the following:

- Create a full, mechanically verified interpretation of the type (We have done so enough to have a proof of concept but have not fully finished the interpretation for every single one of the many constructors)
- Create a data type with a proof of isomorphism to the type of type theory using shapely containers with decidable equality on shapes and positions that can be held in stasis of a monad (we will only informally check that the preconditions hold to create such type).
- Create a normalisation procedure (has been done in [24] but would need to be adapted to our new constructors). We note that the equivalencies presented here might not be enough to define a normalisation procedure, but it should become clear how they would be added.

4.7.1 The Type of Type Theory

This section reiterates the results of [8, 24] and adds some functionality to showcase how type theory can be expressed within (an only slightly larger) type theory. As this is not a thesis on cubical type theory, we will limit the type to express intensional type theory and just point out the positions where further axioms for equality can be added.

The main idea of the type of type theory is to express the syntax of type theory as a type and to include equality types and coercions to allow correct transformations between equal expressions. Later, this type can be given an interpretation in general type theory, but we will leave doing this to future work.

The presented type of type theory does not contain any function arrows and the syntax is filtered to only represent correct terms by indexing the data. The way this is done is by defining the type as a mutual definition of four subtypes.

Mutual in this case means that the types can be used to index each other and use functions defined on them for a more complex type structure. Technically, this could make this a data type inexpressible by conventional data types [10], however, for as long as we do not use recursive functions, it can always be desugared into a conventional indexed data type. In theory, mutually defined functions can turn recursive by calling each other (meaning we would not see it from a single definition calling itself), however for as long as we do not case-split anywhere, no such function would pass the termination checker.

To turn the here presented indexed type of type theory into a sum of products data type where the indices are defined by a separate function, we can use the technique from Section 4.6.

We begin with the signatures of the mutually defined data types. First, we have the data types needed to express terms in type theory:

```
data Ctx : ℕ → Set                          -- Type for contexts
data Tp  : Ctx n → ℕ → Set                  -- Type for types
data Tm  : (Γ : Ctx n) → Tp Γ n' → Set      -- Type for terms
data Tms : (Γ : Ctx n) (Δ : Ctx n) → Set    -- Type for context transformations
```

The first type `Ctx` will be used to express the context that a term or type is in. It can be pictured as a list of variables with certain types that are available to the creation of the type or term, without containing the actual contents of the variables. The natural number given is the universe level in which the context lives.

The type `Tp` is the type for types. Every type lives in a certain context and has a universe level.

The `Tm` type represents the terms of the theory. Every term, again, lives in a certain context and has a type from the same context.

Last, the `Tms` type (standing for *term substitution*, but we will also refer to it as a `context change`) is a type that represents a change in contexts. For example, a term that still requires a value to create its output, meaning its context contains a variable of a certain type, can be turned into a term of its output by giving it the value of the desired variable, therefore shrinking the context of values it still needs. The same principle applied for types. To not redefine these context transformations for types and terms, they get their extra type definition used by types and terms, respectively.

Just having contexts, terms and types alone however is not enough to define meaningful transformations between terms however, as would be needed for normalisation (evaluation). Additionally, we need equality types that terms, types, contexts and substitutions can be coerced between. These equality types have the signature

```
data _=Ctx=_ : Ctx n   → Ctx n'    → Set
data _=Tp=_  : Tp Γ n  → Tp Δ n'   → Set
data _=Tm=_  : Tm Γ A  → Tm Δ B    → Set
data _=Tms=_ : Tms Γ Δ → Tms Γ' Δ' → Set
```

We will now go through each of the constructors, but first a small technical side note.

Side Note 4.7:1

In Agda, when defining mutually recursive data types (also called inductive-inductive data types [10]), the constructors of the data types are not mutually brought into context, meaning that when earlier defined types use a constructor that is defined later it is not in scope. To circumvent this, we can define a function of (basically) the same name that just evaluates to the constructor needed as the mutually defined functions are in scope for all data type declarations. To avoid clutter here, we will express the data types in a way as if constructor names are in scope for all data types regardless of definition order.

4.7.1.1 Constructors for Contexts, Types, Terms and Term Substitutions

We start with the constructors for contexts. Contexts are basically lists of types, but every new type we add to the context depends on the entire previous context as well.

```
data Ctx where
    []ctx : Ctx 0
    _::ctx_ : (Γ : Ctx n) → Tp Γ n' → Ctx (max n n')
```

There is an empty context and a context that attaches a type to defined over the previous context to form a larger context. The universe level is the maximum of the previous context and the universe level the types lives on.

Next we define our possible types. In both [8, 24], only a universe type and pi-types are considered, which is enough as a proof of concept but does not form a particularly rich type theory. We therefore add constructors to have all base

types of type theory at our disposal.

```
data Tp where
    SetS : (n : ℕ) → Tp Γ (1+ n)

    ZERO : Tp Γ n
    UNIT : Tp Γ n
    BOOL : Tp Γ n

    Pi  : (A : Tp Γ n) → Tp (Γ ::ctx A) n' → Tp Γ (max n n')
    Sig : (A : Tp Γ n) → Tp (Γ ::ctx A) n' → Tp Γ (max n n')
    EQ  : {A B : Tp Γ n} → (a : Tm Γ A) → (b : Tm Γ B) → Tp Γ n
    ...
```

(please not that a few more constructors will follow soon)

First, we have a universe type called `SetS` that represents the `Set` type (Agda's type for types) at a certain universe level. It itself is a type of the sucecssive level. Next up, we have the base types `ZERO`, `UNIT` and `BOOL` (sometimes in the literature referred to as *zero*, *one* and `two`). We chose not to use non-dependent sums and products as base types as we can express them with the given base types and sigma/pi-types. The sigma and pi types both have two types are arguments, where the second type has an element of the first type in its context. The universe level is the maximum of the two sublevels. At last, there is a type for equalities that needs to be included to actually prove any meaningful statements in the theory itself.

We have also added a constructor for `W` types, but in order to understand its definition we have to look at the definition of all other parts of the type of type theory first.

We now conclude the constructors for types with a few constructors that are important for the meta theory.

```
    _[_]TpTms : Tp Δ n → Tms Γ Δ → Tp Γ n

    toTp      : Tm Γ (SetS n) → Tp Γ n

    coerce-Tp : Γ =Ctx= Δ → Tp Γ n → Tp Δ n
```

The `_[_]TpTms` constructor allows us to change the context of a type along

a term substitution. It might be a bit unituitive to read in which direction the term substitution goes, but context of the type (and also later term) is put to the right (front) of the term substitution, which transforms the context to the context on the left (back). This is exactly opposite to how a function does it, but is the preferred notation in [8,24]. It can also be pictured as wanting a type in the context `Γ`, and getting it by transforming `Γ` into `Δ` and providing a type in context `Δ`. The `toTp` constructor allows us to change a term into a type and the `coerce-Tp` constructor allows us to use the type in any equivalent context.

Before we define the terms (they have a lot of constructors), we will define the term substitutions; the type that allows us to change contexts when we either do not use a variable or have the concrete value of a variable.

```
data Tms where

    idtms      : Tms Γ Γ

    v1         : Tms (Γ ::ctx B) Γ

    _addTm_    : (tms : Tms Γ Δ) → Tm Γ (A [ tms ]TpTms) → Tms Γ (Δ ::ctx A)

    _otms_     : Tms Δ Γ' → Tms Γ Δ → Tms Γ Γ'

    coerce-Tms : Γ1 =Ctx= Γ2 → Δ1 =Ctx= Δ2 → Tms Γ1 Δ1 → Tms Γ2 Δ2
```

Of course, we can 'change' contexts by not changing contexts using `idtms`. We can add a variable to the context (that we are not using) using `v1` (adding the variable to the back, being the output of the transformation).

Side Note 4.7:2

The names `v1` and later `v0` stem from the intuition of deBrujin indices, where if you want a specific value in the context you drop the context (meaning you declare that you do not need the topmost variable) using `v1` until you get the value you want using the later defined `v0` (will be defined in the terms). This is how the type deals with variables, which makes its terms quite often a bit hard to read, but it is the easiest way to ensure that every value used is actually contained in the context.

If we have a concrete value encoded as a term, we can apply it to the first element of a context transformation using `_addTm_`, resulting in a context

transformation into a smaller context.

With `_otms_` (where the ∘ part is reminiscent of function composition) we can compose two context transformations. `coerce-Tms` can coerce the transformation between equivalent contexts.

We can now almost define some classical application operators. First, we can define a context transformation that just applies a term to a context

```
apply : Tm Γ A → Tms Γ (Γ ::ctx A)
apply a = idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) a
```

We apply the term to the identity context and use an equation (that we will define later) that says that we can apply the identity transformation to a type without changing the type. We can now define the application of a term to a type as

```
_$Tp_ : Tp (Γ ::ctx A) n → Tm Γ A → Tp Γ n
_$Tp_ B a = B [ apply a ]TpTms
```

We will now get to the definition of the terms. The terms (as far as possible) give constructors and destructors for each type. We start with the constructors and start with the base types.

```
data Tm where
    unitTm : Tm Γ UNIT

    trueTm  : Tm Γ BOOL
    falseTm : Tm Γ BOOL
    ...
```

The `UNIT` types has its obvious one constructor, and the Booleans have their classical two constructors. All of these constructors work on an arbitrary universe level. The `ZERO` type does not have a constructor for obvious reasons.

The constructor for pi-types is a lambda expression.

```
lam : Tm (Γ ::ctx A) B → Tm Γ (Pi A B)
```

Here, we can give a term in a context that still requires a value for `A` to produce a `B` to create a term of type `Pi A B` (which represents the dependent function type `(a : A) → B a`) in the context wihtout the `A`.

For sigma types (dependent tuples), we do not have to change the overall context, as we can directly lower the context for the second argument:

```
sigma : (a : Tm Γ A) → Tm Γ (B $Tp a) → Tm Γ (Sig A B)
```

Here, we are given a term `(a : A)` and apply it to the type of the second term, making it have the type `B a`. It should be noted that the second term can also access the value of `a` because it lives in the same context, so we do not need to put it into the context of the second term.

We introduce a constructor for equivalences by just stating that any equivalence of the meta theory can be turned into a term for the equivalence of two terms. As we are only dealing with the equivalence of terms here, the context of the terms in the equivalence have to stay equal (so our meta theory does not deal with type, substitution or context equalities)

```
eqC : {A B : Tp Γ n} {a : Tm Γ A} {b : Tm Γ B} →
    a =Tm= b → Tm Γ (EQ a b)
```

We have a final constructor for `W` types that we will explain later.
We now get to the destructors. We start again with the base types.

```
absurdTm : Tm Γ ZERO → Tm Γ A

ifTm_then_else_ : Tm Γ BOOL →
    Tm Γ A → Tm Γ A → Tm Γ A
```

The `ZERO` type has a destructor for the ex falso quodlibet rule, stating that

if there is an inconsistency, anything follows. This is important when proving a case that cannot happen. In this case we often cannot derive the goal statement constructively, so we use the `absurd` function to prove anything because the current case simply cannot happen anyway.

The Booleans have their straight-forward if-then-else destructor. We here just give a non dependent variant; it can be made dependent by wrapping it into a dependent function.

Dependent functions also have a destructor. To abide by the categorical notation, this is not directly function application, but a precursor

```
app : Tm Γ (Pi A B) → Tm (Γ ::ctx A) B
```

This just states that a dependent function can be turned into its result in a context that still requires the function input. Together with the term substitutions, we can now define function application as

```
_$Tm_ : Tm Γ (Pi A B) → (a : Tm Γ A) → Tm Γ (B $Tp a)
_$Tm_ f a = app f [ apply a ]TmTms
```

Sigma types could be defined using a single destructor, but we figured it might be easier to define it via the two projections

```
fstTm : Tm Γ (Sig A B) → Tm Γ A
sndTm : (a : Tm Γ (Sig A B)) → Tm Γ (B $Tp (fstTm a))
```

The first projection just retrieves the first element. The second projection retrieves the second element, but not before applying its type to the first element.

Again, there is also a destructor for `W` types that we will define later.

Terms have one constructor that is used as a variable. We have a placeholder for values in the context called

```
v0 : Tm (Γ ::ctx A) (A [ v1 ]TpTms)
```

Where the name `v0` stems from the deBrujin indices mentioned in Sidenote 4.7:2, so `v0` is the variable with deBrujin index 0 (where we can use `v1` from the term substitutions to create variables with higher index). As the context of the term's result has to be the same context as the one of the term itself, we have to enlarge the context of `A` by itself using `v1` (meaning it has a variable with type of itself in its context without being able to use it).

There are a few miscillaneous constructors for terms similar to the ones in types. We have:

```
_[_]TmTms  : Tm Δ A → (tms : Tms Γ Δ) → Tm Γ (A [ tms ]TpTms)

toTm       : (A : Tp Γ n) → Tm Γ (SetS n)

coerce-Tm  : A =Tp= B → Tm Γ A → Tm Δ B
```

The `_[_]TmTms` applies a term substitution to the context of a term, just like with types. Also just like types, there is a conversion from a type into a term. Finally, terms can be coerced between equivalent types.

4.7.1.2 Constructors for Equalities

As for equalities, we can technically add as many equalities between the base terms of the meta theory for as long as we dot not create any inconsistencies when writing an interpretation. This makes this type so versatile: We have a few terms that we need in order to define what computation is, but we can attach any meaning we want by choosing the right equations. The equations we show here are mainly those of [24], together with additional compute rules for our new constructors. They might be incomplete however as we put writing a normaliser onto future work.

It is worth noticing that we could define this type of type theory as a higher order inductive type, meaning that we would give constructors for equality to make it easier for the type to interact with the cubical notion of equality. This, however, would make the argument of the type having only a finite number of constructors and positions harder, so we will refer to the classical definition of equality predicates via indexed data types.

We repeat the type signatures of the equalities. Note that we can define qualities between any two arguments of the same "kind" (context, type, term or term substitution), so for example we can even define equalities of terms with different types.

```
data _=Ctx=_ : Ctx n   → Ctx n'    → Set
data _=Tp=_  : Tp Γ n  → Tp Δ n'   → Set
data _=Tm=_  : Tm Γ A  → Tm Δ B    → Set
data _=Tms=_ : Tms Γ Δ → Tms Γ' Δ' → Set
```

Equalities for Contexts We begin with the equalities for contexts. They are fairly straight forward. We begin with the equalities for when the individual elements are equal:

```
data _=Ctx=_ where
    []ctx-eq : []ctx =Ctx= []ctx
    ::ctx-eq : Γ =Ctx= Δ → A =Tp= B → (Γ ::ctx A) =Ctx= (Δ ::ctx B)
    ...
```

Empty contexts are of course equal, and if we have two equal contexts, adding equal types to the front continues to make them equal.

Further, we need a few equalities that all of our equality types have that turn the relation into an equivalence relation:

```
    refl-Ctx  : A =Ctx= A
    sym-Ctx   : A =Ctx= B → B =Ctx= A
    trans-Ctx : A =Ctx= B → B =Ctx= C → A =Ctx= C
```

At last, we add one equation that should actually be a consequence of our definition, but due to the resource constraints of the thesis we just assume it (it has been proven for the type given in [24] but would have to be adjusted for our type). For some constructions we need context equality to follow from type equality. (similar should hold for terms and term substitutions, but we ended up not needing those equalities).

```
    -- this should be a consequence, not an axiom/constructor
    ctx-Eq : A =Tp= B → Γ =Ctx= Δ
```

Equalities for Types Next, we look at the equations for types. First, there are some general equations for interactions with term substitutions:

```
tms-idtms-Tp :             A [ idtms ]TpTms =Tp= A
tms-assoc-Tp : A [ tms ]TpTms [ tms' ]TpTms =Tp= A [ tms otms tms' ]TpTms
```

The identity substitution leaves the type as is, and applying two substitutions composes the the substitutions.

We continue with the equations of the base types. Those state that all base types stay the same under term substitutions:

```
tms-SetS : {tms : Tms Γ Δ} → SetS {Δ} n   [ tms ]TpTms  =Tp= SetS {Γ} n
tms-ZERO : {tms : Tms Γ Δ} → ZERO {Δ} {n} [ tms ]TpTms  =Tp= ZERO {Γ} {n}
tms-UNIT : {tms : Tms Γ Δ} → UNIT {Δ} {n} [ tms ]TpTms  =Tp= UNIT {Γ} {n}
tms-BOOL : {tms : Tms Γ Δ} → BOOL {Δ} {n} [ tms ]TpTms  =Tp= BOOL {Γ} {n}
```

As contexts and or universe level are implicit for these types we will make them explicit here to see the specific context changes while universe levels stay the same.

Term substitutions should further interact with pi and sigma types as well:

```
tms-Pi  : Pi A B  [ tms ]TpTms =Tp= Pi  (A [ tms ]TpTms) (B [ tms transformTp A ]TpTms)
tms-Sig : Sig A B [ tms ]TpTms =Tp= Sig (A [ tms ]TpTms) (B [ tms transformTp A ]TpTms)
```

These equations apply the term substitutions to the individual subtypes `A` and `B`. The problem is, that this now slightly changes the context for `B`. Before, it needed an element of `A`, but after the application of the term substitution, it is given an element of `A [ tms ]TpTms`. To resolve this, we need to transform the context for `B` to now hold the substitution-transformed type `A [ tms ]TpTms` using

```
_transformTp_ : (tms : Tms Γ Δ) → (A : Tp Δ n) → Tms (Γ ::ctx (A [ tms ]TpTms)) (Δ ::ctx A)
_transformTp_ tms A = (tms otms v1) addTm (coerce-Tm tms-assoc-Tp v0)
```

where we first enlarge the context of the original term substitution using `v1` to then add a type on top that is changed with the term substitution (which is what `addTm` does).

As we have defined the equality type to only be between terms of the same context, it interacts with term substitutions straight forwardly, switching from transforming the context of types to transforming the context of terms:

```
tms-Eq  : EQ a b [ tms ]TpTms =Tp= EQ (a [ tms ]TmTms) (b [ tms ]TmTms)
```

The last interaction with term substitutions is with the transformation of a type into a term.

```
toTp-tms : (toTp a) [ tms ]TpTms =Tp= toTp (coerce-Tm tms-SetS (a [ tms ]TmTms))
```

As `toTp` requires a term of type `SetS n` and not `SetS n [ ... ]TmTms` we have to apply the `tms-SetS` rule to get rid of the term substitution in the type.

Neither [24] nor [8] give an equation for the interaction of term substitutions with coercions, so we will not do so here either, however, coercions are always treated with a coherence condition:

```
coerce-Tp-coh : coerce-Tp Γ=Δ A =Tp= A
```

Note that this does not change the context of `A`, so if `A` is used in a different context, this rule cannot be magically used to just remove all coercions.

Lastly, we still have the equations to turn this into an equivalence relation. Furthermore, we add some congruencies:

```
refl-Tp  : A =Tp= A
sym-Tp   : A =Tp= B → B =Tp= A
trans-Tp : A =Tp= B → B =Tp= C → A =Tp= C

Pi-cong  : (A=Tp=B : A =Tp= B) → (C=Tp=D : C =Tp= D) → Pi  A C =Tp= Pi  B D
Sig-cong : (A=Tp=B : A =Tp= B) → (C=Tp=D : C =Tp= D) → Sig A C =Tp= Sig B D
tms-cong-Tp  : A =Tp= B → tms =Tms= tms' → A [ tms ]TpTms =Tp= B [ tms' ]TpTms
toTp-cong : A =Tm= B → toTp A =Tp= toTp B
coerce-Tp-cong : A =Tp= C → coerce-Tp Γ=Δ A =Tp= coerce-Tp Γ=Δ C
```

Two useful congruences we can derive are

```
Pi-cong-fst : (A=Tp=C : A =Tp= C) → Pi A B =Tp= Pi C (coerce-Tp (ctx-Tp-eq A=Tp=C) B)
Pi-cong-fst eq = Pi-cong eq (sym-Tp coerce-Tp-coh)

Pi-cong-snd : B =Tp= C → Pi A B =Tp= Pi A C
Pi-cong-snd eq = Pi-cong refl-Tp eq
```

If we want to apply an equation to the second subtype of the pi type we can freely do so, however, if we apply it to the first type, we have to change the context of the second type accordingly using the induced equality of contexts from the equality of types, wrapped as

```
ctx-Tp-eq : A =Tp= B → (Γ ::ctx A) =Ctx= (Δ ::ctx B)
ctx-Tp-eq A=Tp=B = ::ctx-eq (ctx-Eq A=Tp=B) A=Tp=B
```

Equalities for Terms We will now switch to the equalities for terms. In [24], only three non-trivial equations are given for types, namely:

```
data _=Tm=_ where
    tms-v0-addTm-Tm  :      v0 [ tms addTm a ]TmTms =Tm= a
    beta : app (lam t) =Tm= t
    eta  : lam (app t) =Tm= t
    ...
```

The first equation states, that a variable turns into the value it is given in the term substitution. The next equation is the one for beta-equality, which is

usually expressed as `(\ x → t) a = t a` but looks slightly different when dealing with terms in a context. The equation states that applying a lambda term just gives the term in the lambda (that is already the correct value as it lives in a certain context with its values cahnged as soon as the term substitution is applied). This is the equation needed for substitution along the evaluation of terms, which further only consists of normalising the term substitution with the first equation `tms-v0-addTm-Tm` that fills in the actual values. The third equation is the one for eta-equality, which is usually expressed as `f = (\ x → f x)` and states that functions stay the same when wrapped in a lambda and are just being given the argument. This equation is used in normalisation to get to the head normal form (though we will not do that in this thesis though).

As we have our additional types, we also have to add the compute rules for our other destructors (if they have any):

```
ite-true  : ifTm trueTm  then a else b =Tm= a
ite-false : ifTm falseTm then a else b =Tm= b

fstTm-sigma : fstTm (sigma a b) =Tm= a
sndTm-sigma : sndTm (sigma a b) =Tm= b
```

Because of these compute rules, we also need to define how term substitutions interact with the `iftm_them_else_` (ite) and `sigma` constructor. With the ite constructor this is straight forward:

```
tms-ite : (ifTm b then a else c) [ tms ]TmTms =Tm= ifTm (coerce-Tm tms-BOOL (b [ tms ]TmTms)) then (a [ tms ]TmTms) else (c [ tms ]TmTms)
```

It is more difficult to define this for the sigma constructor. When applying a term substitution for sigma types, the context of both entries changes accordingly, however, the type for `B` is applied to the first argument, which is now applied to the term substitution, which might technically be a different type. We account for this change using an extra equation as follows:

```
tms-sigma : (sigma a b) [ tms ]TmTms =Tm= sigma (a [ tms ]TmTms) (coerce-Tm tms-apply-Tp (b [ tms ]TmTms))
```

The additional equation can be derived, but needs a larger proof, so we will

give it its own lemma

Lemma 4.7:3

```
tms-apply-Tp : (B $Tp a) [ tms ]TpTms =Tp= (B [ tms transformTp A ]TpTms) $Tp (a [ tms ]TmTms)
```

Proof 4.7:4

We begin by "unrolling the definition" of the original statement through our equations:

```
(B $Tp a) [ tms ]TpTms                                                                                    =<Tp tms-assoc-Tp >
B [ apply a otms tms ]TpTms                                                                               =<Tp tms-cong-Tp refl-Tp otms-addTm >
B [ (idtms otms tms) addTm coerce-Tm tms-assoc-Tp (coerce-Tm (sym-Tp tms-idtms-Tp) a [ tms ]TmTms) ]TpTms =<Tp tms-cong-Tp ... >
```

If twe do the same starting from the end, we get

```
...
B [ ((tms otms v1) addTm coerce-Tm tms-assoc-Tp v0) otms (idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms)) ]TpTms =<Tp>
B [ (tms transformTp A) otms apply (a [ tms ]TmTms) ]TpTms                                                                    =<Tp sym-Tp tms-assoc-Tp >
(B [ tms transformTp A ]TpTms) $Tp (a [ tms ]TmTms)                                                                           qedTp
```

We now have to prove through congruence that these two term substitutions are equal. We start by simplifying from the beginning:

```
(idtms otms tms) addTm coerce-Tm tms-assoc-Tp (coerce-Tm (sym-Tp tms-idtms-Tp) a [ tms ]TmTms) =<Tms addTm-cong refl-Tms (

    coerce-Tm tms-assoc-Tp (coerce-Tm (sym-Tp tms-idtms-Tp) a [ tms ]TmTms) =<Tm coerce-Tm-coh >
    coerce-Tm (sym-Tp tms-idtms-Tp) a [ tms ]TmTms                          =<Tm tms-cong-fst-Tm coerce-Tm-coh >
    a [ tms ]TmTms                                                          =<Tm tms-cong-Tm refl-Tm (sym-Tms idtms-left-id) >
    a [ idtms otms tms ]TmTms qedTm ) >

(idtms otms tms) addTm (a [ idtms otms tms ]TmTms)                                  =<Tms addTm-cong idtms-left-id (tms-cong-Tm refl-Tm idtms-left-id) >
tms addTm (a [ tms ]TmTms)                                                          =<Tms addTm-cong ... >
```

We mainly use the coercion coherence and term substitution identity rules to simplify the statement.
Next, we have to unsimplify the statement again to fit the term substi-

tution of the other end. We will do so by applying equations to the left and right argument of `tms addTm (a [ tms ]TmTms`, so we provide an equality for the term substitution and one for the term:

```
-- tms eq, applied to tms
tms                                                                                                                  =<Tms sym-Tms idtms-right-id >
tms otms idtms                                                                                                       =<Tms otms-cong refl-Tms (sym-Tms v1-otms) >
tms otms (v1 otms (idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms)))                                    =<Tms sym-Tms otms-assoc >
(tms otms v1) otms (idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms))                                    qedTms

-- tm eq, appled to a [ tms ]TmTms
a [ tms ]TmTms                                                                                                                                                  =<Tm sym-Tm coerce-Tm-coh >
coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms)                                                                                                                =<Tm sym-Tm tms-v0-addTm-Tm >
v0 [ idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms) ]TmTms                                                                                        =<Tm tms-cong-fst-Tm (sym-Tm coerce-Tm-coh) >
coerce-Tm tms-assoc-Tp v0 [ idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms)]TmTms                                                                  =<Tm sym-Tm coerce-Tm-coh >
coerce-Tm tms-assoc-Tp (coerce-Tm tms-assoc-Tp v0 [ idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms)]TmTms) qedTm
```

This leaves us with the following overall term that fits into the end of our proof, where we only need to apply one equation for term substitution composition:

```
((tms otms v1) otms (idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms)))
    addTm
    (coerce-Tm tms-assoc-Tp (coerce-Tm tms-assoc-Tp v0 [ idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms)]TmTms)) =<Tms sym-Tms otms-addTm >
((tms otms v1) addTm coerce-Tm tms-assoc-Tp v0) otms (idtms addTm coerce-Tm (sym-Tp tms-idtms-Tp) (a [ tms ]TmTms))          qedTms
```

So pluggin the initial two parts together using the respective congruence rule gives the overall proof.

The other term equations are similar to the ones for types, so we will just give them here for the sake of completeness:

```
    refl-Tm  : A =Tm= A
    sym-Tm   : A =Tm= B → B =Tm= A
    trans-Tm : A =Tm= B → B =Tm= C → A =Tm= C

    coerce-Tm-cong : a =Tm= c  →                                  coerce-Tm A=B a =Tm= coerce-Tm A=B c
    tms-cong-Tm    : a =Tm= b  → tms =Tms= tms' →                 a [ tms ]TmTms =Tm= b [ tms' ]TmTms
    ite-cong       : b =Tm= b' → a =Tm= a' → c =Tm= c' → (ifTm b then a else c) =Tm= (ifTm b' then a' else c')
    toTm-cong      : A =Tp= B  →                                          toTm A =Tm= toTm B

    coerce-Tm-coh : coerce-Tm A=B A =Tm= A

    tms-idtms-Tm     :             a [ idtms ]TmTms =Tm= a
    tms-assoc-Tm     : a [ tms ]TmTms [ tms' ]TmTms =Tm= a [ tms otms tms' ]TmTms

    toTm-tms : (toTm A) [ tms ]TmTms =Tm= toTm (A [ tms ]TpTms)

    toTm-o-toTp-id :          toTm (toTp a) =Tm= a
```

For a full theory, we are obviously still missing a few congruence equations but it should be straight forward how to add them.

Equalities for Term Substitutions Finally, we give the equalities for term substitutions. In [24], only four non trivial equations are given for term substitutions (that we make five to have the identity subsitution be a left and right identity):

```
data Tms where
    id-left-id  :          idtms otms tms =Tms= tms
    id-right-id :          tms otms idtms =Tms= tms
    v1-addTm-id :             v1 addTm v0 =Tms= idtms {Γ ::ctx B}
    v1-otms     :   v1 otms (tms addTm a) =Tms= tms
    otms-addTm  : (tms addTm a) otms tms' =Tms= ((tms otms tms') addTm (coerce-Tm tms-assoc-Tp (a [ tms' ]TmTms)))
    ...
```

The identity substitutions should of course be identities for composition. If we add a term to a context in which we are not using it, that should not do anything at all. Same if we add the term in a composition. Finally, adding a term is preserved by composition (coercing the composition into the type as well).

For our theory, we added associativity for term substitution composition, as was also done in [8]:

```
otms-assoc : (a otms b) otms c =Tms= a otms (b otms c)
```

The remaining equations for substitutions are similar to the equations before:

```
refl-Tms  : A =Tms= A
sym-Tms   : A =Tms= B → B =Tms= A
trans-Tms : A =Tms= B → B =Tms= C → A =Tms= C

coerce-Tms-coh : coerce-Tms A B tms =Tms= tms
```

Some extra Syntax for Equalities of the Meta Theory In order to write equality proofs in the meta theory the same way as we usually write equations, we introduce the analog notation

```
A =<Tp  A=B > B=C
a =<Tm  a=b > b=c
a =<Tms a=b > b=c
_qedTp
_qedTm
_qedTms
```

to define chains of equations in the meta theory.

4.7.1.3 Some Simple Consequences

As we are not giving an interpretation function from our type of type theory into Agda's type of types `Set` we want to give a few examples to show that the type still behaves as expected. We start by expressing how non-dependent functions work in the meta theory.

Non-Dependent Functions The type of non-dependent functions can be given as

```
_=>_ : Tp Γ n → Tp Γ n' → Tp Γ (max n n')
A ⇛ B = Pi A (B [ v1 ]TpTms)
```

where we are just dropping the argument from the context of the output type. There is one equation that can now help us to remove the complexity added by the dependent types for the case that we are in a non-dependent setting:

```
apply-irr-Tp : (A [ v1 ]TpTms) $Tp a =Tp= A
apply-irr-Tp =
    (A [ v1 ]TpTms) $Tp a =<Tp tms-assoc-Tp >
    A [ v1 otms apply a ]TpTms =<Tp tms-cong-Tp refl-Tp v1-otms >
    A [ idtms ]TpTms =<Tp tms-idtms-Tp >
    A qedTp
```

This equation states that on the type level, if we apply a term that we are not using, we stay equal to the original type. We can use this to write a function application for non-dependent functions:

```
_$Tm'_ : Tm Γ (A ⇛ B) → (a : Tm Γ A) → Tm Γ B
_$Tm'_ f a = coerce-Tm apply-irr-Tp (f $Tm a)
```

where we just have to drop the irrelevant type application from the dependent function. Next, we want to define non-dependent function composition. For this, we need the compute rule for the interaction of non-dependent functions with term substitutions:

```
tms-=> : (A ⇛ B) [ tms ]TpTms =Tp= (A [ tms ]TpTms) ⇛ (B [ tms ]TpTms)
tms-=> =
    Pi A (B [ v1 ]TpTms) [ tms ]TpTms                                  =<Tp tms-Pi >
    Pi (A [ tms ]TpTms) ((B [ v1 ]TpTms) [ tms transformTp A ]TpTms) =<Tp Pi-cong-snd tms-assoc-Tp >
    Pi (A [ tms ]TpTms) (B [ v1 otms tms transformTp A ]TpTms)       =<Tp Pi-cong-snd (tms-cong-Tp refl-Tp v1-otms) >
    Pi (A [ tms ]TpTms) (B [ tms otms v1 ]TpTms)                     =<Tp Pi-cong-snd (sym-Tp tms-assoc-Tp) >
    Pi (A [ tms ]TpTms) ((B [ tms ]TpTms) [ v1 ]TpTms)               qedTp
```

We basically use the `tms-Pi` rule for the interaction of pi types with term substitutions and then eliminate the `v1` and their respective applications.

We can now formulate non-dependent function composition with only one small helper function:

```
_oTm_ : Tm Γ (B ⇛ C) → Tm Γ (A ⇛ B) → Tm Γ (A ⇛ C)
f oTm g = lam ((lift-func f) $Tm' app g)
```

We start this expression by using the meta-lambda to get the function argument into the context. We can write the composition now seemingly point-free (here meaning that we do not need `v0` as an explicit variable. We obviously are still using a lambda expression). This is because when using `app` on a function, it automatically gets its argument from the context, turning it into a value of its result. We can now apply `f` that value of `g`. We need a helper function for this, because the argument to `g` is currently in the context, but the context of `f` is still the one without the argument, so we need to lift the function to ignore the argument with

```
lift-func : Tm Γ (A ⇛ B) → Tm (Γ ::ctx C) (A [ v1 ]TpTms ⇛ B [ v1 ]TpTms)
lift-func f = coerce-Tm tms-=> (f [ v1 ]TmTms)
```

Next up, we show some other non-dependent data types

Non-Dependent Tuples We continue to expressing non-dependent tuples. We can define their type as

```
_:x:_ : Tp Γ n → Tp Γ n' → Tp Γ (max n n')
_:x:_ A B = Sig A (B [ v1 ]TpTms)
```

This is just a sigma type where the type of the second argument ignores the value of the first.

We can define a constructor for non-dependent tuples as:

```
mkTup : Tm Γ A → Tm Γ B → Tm Γ (A :x: B)
mkTup a b = sigma a (coerce-Tm (sym-Tp apply-irr-Tp) b)
```

which just straight forwardly uses the constructor for a sigma type, so we know well enough that it works as intended. The projections are also straight forward:

```
fstTup : Tm Γ (A :x: B) → Tm Γ A
fstTup axb = fstTm axb

sndTup : Tm Γ (A :x: B) → Tm Γ B
sndTup {B} axb = coerce-Tm apply-irr-Tp (sndTm axb)
```

Where the only non-trivial part is to drop the type argument for the second type.

Dependent Sums It is not as easy when it comes to non-dependent sums. Technically, we can express a type isomorphic to a non dependent sum as

pseudocode:

```
A :+: B = Σ 𝔹 (\b → if b then A else B)
```

The problem here is that this does not work well in the presence of universe levels because A and B could potentially be at different levels, but the if-then-else has to result in one uniform universe level. So fix this, we define a lifting operation for lifting universe levels:

```
liftTpL : (n : ℕ) → Tp Γ n' → Tp Γ (max n n')
liftTpL n A = UNIT {n = n} :x: A

liftTpR : (n : ℕ) → Tp Γ n' → Tp Γ (max n' n)
liftTpR {Gamma} {n'} n A = A :x: UNIT {n = n}
```

These meta-types use the fact that sigma-types merge the universe levels of their respective subtypes, so adding a unit-type of an arbitrary universe merges the overall universe level to at least the level provided. We can now express a non-dependent sum as

```
_:+:_ : Tp Γ n → Tp Γ n' → Tp Γ (max n n')
_:+:_ A B = Sig BOOL (toTp (ifTm (coerce-Tm tms-BOOL v0)
                            then toTm (liftTpR n' (A [ v1 ]TpTms))
                            else toTm (liftTpL n (B [ v1 ]TpTms))))
```

where the Boolean argument in the beginning determins whether we have an Element of the left type `A` or the right type `B`.

As working with this type is quite cumbersome, we only show how to create the constructor for the left element (the one for the right is analogous). Before we can do that, we need two lemmata. The first one states that the unit type (and, analogously every non-dependent unary base type) is equal under all equal contexts.

Lemma 4.7:5

```
UNIT-Ctx-Eq : Γ =Ctx= Δ → UNIT {Gamma = Gamma} {n = n} =Tp= UNIT {Gamma = Delta} {n = n}
```

This works analogously for `SetS`, `ZERO` and `BOOL`.

Proof 4.7:6

```
UNIT {Gamma = Gamma} {n = n} =<Tp sym-Tp tms-UNIT >
UNIT {Gamma = Gamma} {n = n} [ idtms ]TpTms =<Tp tms-cong-Tp refl-Tp (sym-Tms coerce-Tms-coh) >
UNIT {Gamma = Gamma} {n = n} [ coerce-Tms eq refl-Ctx idtms ]TpTms =<Tp tms-UNIT >
UNIT {Gamma = Delta} {n = n} qedTp
```

We use the fact that we can coerce the identity term substitution to substitute between any two equal contexts. As the above mentioned unary types all stay equal under term substitution application, we can use this to convert them between equal contexts.

The second lemma is that when we evaluate the non-dependent sum type by giving it the `trueTm`, we get the left lifted term. This holds analogously for the right term on `falseTm`.

Lemma 4.7:7

```
eval-:+:-left :
    toTp (ifTm coerce-Tm tms-BOOL v0
        then toTm (liftTpR n' (A [ v1 ]TpTms))
        else toTm (liftTpL n (B [ v1 ]TpTms)))
        $Tp trueTm
    =Tp= A :x: (UNIT {n = n'})
```

Proof 4.7:8

We start by moving the term substitution into the type conversion

```
toTp (ifTm coerce-Tm tms-BOOL v0
    then toTm (liftTpR n' (A [ v1 ]TpTms))
    else toTm (liftTpL n (B [ v1 ]TpTms)))
    $Tp trueTm =<Tp toTp-tms >

toTp (coerce-Tm tms-SetS ((
    ifTm coerce-Tm tms-BOOL v0
    then toTm (liftTpR n' (A [ v1 ]TpTms))
    else toTm (liftTpL n (B [ v1 ]TpTms)))
    [ apply trueTm ]TmTms)) =<Tp toTp-cong ... >
```

We now use the congruence to evaluate the type inside the type conversion.

```
coerce-Tm tms-SetS ((
    ifTm coerce-Tm tms-BOOL v0
    then toTm (liftTpR n' (A [ v1 ]TpTms))
    else toTm (liftTpL n (B [ v1 ]TpTms)))
    [ apply trueTm ]TmTms)                                    =<Tm coerce-Tm-coh >

(ifTm coerce-Tm tms-BOOL v0
then toTm (liftTpR n' (A [ v1 ]TpTms))
else toTm (liftTpL n (B [ v1 ]TpTms)))
[ apply trueTm ]TmTms                                         =<Tm tms-ite >

ifTm coerce-Tm tms-BOOL (coerce-Tm tms-BOOL v0 [ apply trueTm ]TmTms)
then toTm (liftTpR n' (A [ v1 ]TpTms)) [ apply trueTm ]TmTms
else (toTm (liftTpL n (B [ v1 ]TpTms)) [ apply trueTm ]TmTms)
                                                              =<Tm ite-cong
                                                                  (trans-Tm coerce-Tm-coh
                                                                  (trans-Tm (tms-cong-fst-Tm coerce-Tm-coh)
                                                                  (trans-Tm tms-v0-addTm-Tm
                                                                  coerce-Tm-coh)))
                                                                  refl-Tm refl-Tm >

ifTm trueTm
then toTm (liftTpR n' (A [ v1 ]TpTms)) [ apply trueTm ]TmTms
else (toTm (liftTpL n (B [ v1 ]TpTms)) [ apply trueTm ]TmTms) =<Tm ite-true >

toTm (liftTpR n' (A [ v1 ]TpTms)) [ apply trueTm ]TmTms       =<Tm toTm-tms >
toTm (liftTpR n' (A [ v1 ]TpTms) [ apply trueTm ]TpTms)       =<Tm toTm-cong ... >
```

We use the term substitution rule for ite to evaluate that the `v0` variable evaluates to true and then use the computation rule of `ite` to actually evaluate the expression.
Next, we have to show that applying the `trueTm` to both elements of the sigma type does not change anything. We start by unwrapping the sigma type and propagating the term substitutions:

```
toTm (liftTpR n' (A [ v1 ]TpTms) [ apply trueTm ]TpTms)                                                    =<Tm toTm-cong tms-Sig >
toTm (Sig ((A [ v1 ]TpTms) [ apply trueTm ]TpTms) ((UNIT [ v1 ]TpTms) [ apply trueTm transformTp (A [ v1 ]TpTms) ]TpTms)) =<Tm toTm-cong (Sig-cong apply-irr-Tp ...) >
```

To show that the type application does not change anything by using the `Sig-cong` rule, using `apply-irr-Tp` on the first argument and then show that the second argument stays unaffected through that change as well:

```
(UNIT [ v1 ]TpTms) [ apply trueTm transformTp (A [ v1 ]TpTms) ]TpTms =<Tp tms-assoc-Tp >
UNIT [ v1 otms (apply trueTm) transformTp (A [ v1 ]TpTms) ]TpTms     =<Tp tms-UNIT >
UNIT                                                                 =<Tp UNIT-Ctx-Eq (::ctx-eq refl-Ctx apply-irr-Tp) >
UNIT                                                                 =<Tp sym-Tp tms-UNIT >
UNIT [ v1 ]TpTms                                                     qedTp
```

As the context of the `UNIT` type changes here, we need to apply the `UNIT-Ctx-Eq` rule from Lemma 4.7:7. This makes us end up with the correct type

```
toTm (Sig A (UNIT [ v1 ]TpTms)) =<Tm>
toTm (liftTpR n' A) qedTm
```

and we can remove the term-type conversion at the end to prove the statement.

```
toTp (toTm (liftTpR n' A)) =<Tp toTp-o-toTm-id >
A :x: UNIT qedTp
```

With this lemma, we can actually write the left (and right) constructor for non-dependent sums as

```
leftTm : Tm Γ A → Tm Γ (A :+: B)
leftTm a = sigma trueTm (coerce-Tm (sym-Tp eval-:+:-left) (mkTup a unitTm))

rightTm : Tm Γ B → Tm Γ (A :+: B)
rightTm b = sigma falseTm (coerce-Tm (sym-Tp eval-:+:-right) (mkTup unitTm b))
```

4.7.2 Well Formed Types in the Type of Type Theory

Next to now having sigma types and some consequences for non-dependent types in the type of type theory, our biggest contribution to this type are the constructors for W-types and folds over them as their computation rules. This

enables the type of type theory to express recursive functions, which it was not able to in [8,24]. This means that if we can continue to transform this type into our solving formalism we can potentially do program synthesis for terminating recursive functions expressible via folds over W-Types, which is a fairly large class as has been noted in Section 4.2.3.

As a data type declaration, the type of containers looks like:

```
record Container i j : Set (lsuc i ⊔ lsuc j) where
    constructor _|>_
    field
        S : Set i
        P : S → Set j
```

Of course, our meta theory does not support data type or record definitions, but we can define an isomorphic type as

```
Container : (i j : Level) → Set (lsuc i ⊔ lsuc j)
Container i j = Σ (Set i) (\S → (S → Set j))
```

To define this in our meta language, we first define a small helper function to pull types from the context more easily:

```
v0Tp : Tm (Γ ::ctx SetS n) (SetS n)
v0Tp = coerce-Tm tms-SetS v0
```

We can now define the container type in the meta theory as

```
ContainerTp : (n n' : ℕ) → Tp Γ (max (1+ n) (max n (1+ n')))
ContainerTp n n' = Sig (SetS n) ((toTp v0Tp) ⇛ (SetS n'))
```

We leave the universe level directly as is from the definition to not require coercions from cubical agda. In contrast to the alternatice container definition

using Σ types above, the meta theory definition does not use explicit lambda. The reason is that the sigma type already puts the value of the first entry into the context of the second type where it can be used for example using `v0`.

We can now add the type of W-types into our type of type theory

```
data Tp where
    ...
    W : Tm Γ (ContainerTp n n') → Tp Γ (max (1+ n) (max n (1+ n')))
```

This adds a a constructor for well formed types (W-types) to our theory.

Next, we need a constructor for this type. Therefore, we need the semantic extension of a container in our meta theory. To recap: This is how the semantic extension looks like in standard Agda:

```
[[_]] : Container i j → Set k → Set (i ⊔ j ⊔ k)
[[ S ▷ P]] R = ∃[ s of S ] (P s → R)
```

This is a sigma type that consists of a shape and a function from the positions on that shape into some desired value. This stays the same in our meta-theory, just looks a bit odd:

```
[[_]] : Tm Γ (ContainerTp n n') → Tp Γ n'' → Tp Γ (max n (max n' n''))
[[_]] S|>P A = Sig (toTp (fstTm S|>P)) ((coerce-Tp (::ctx-eq refl-Ctx eq) ∘ toTpTms2 ∘ app ∘ coerce-Tm tms-=>) (sndTm S|>P) ⇛ A [ v1 ]TpTms)
    where
        eq : (toTp v0Tp [ apply (fstTm S|>P) ]TpTms) =Tp= toTp (fstTm S|>P)
        eq =
            toTp v0Tp [ apply (fstTm S|>P) ]TpTms =<Tp toTp-tms >
            toTp (coerce-Tm tms-SetS (v0Tp [ apply (fstTm S|>P) ]TmTms)) =<Tp toTp-cong
                                                                           (trans-Tm coerce-Tm-coh
                                                                           (trans-Tm (tms-cong-fst-Tm coerce-Tm-coh)
                                                                           (trans-Tm tms-v0-addTm-Tm
                                                                           coerce-Tm-coh))) >
            toTp (fstTm S|>P) qedTp
```

Basically, we have a sigma type where the first element `S` is of the type of shapes (the first element of the container), and the second element gets this shape into its context. It can then use the positions (the second element of the container) to create the positions at the current shape and create the type of the non-dependent function into the desired type stored in the functor. The entire rest is just context transformations to show that, indeed, we can use the

second element of the container in the current context. We will not go into the tedious, technical details here because no new concepts for transformations are used and just included it for the sake of completeness.

With this semantic extension, we can create the constructor for W-types in the terms as

```
data Tm where
    ...
    In : Tm Γ ([[ C ]] (W C)) → Tm Γ (W C)
```

This is analogous to the W-types in Agda:

```
data W (C : Container i j) : Set (i ⊔ j) where
    In : [[ C ]] (W C) → W C
```

We can also add a destructor, which is our "constructor" for recursive functions over W-types. In Agda, this would be a fold (which is just a defined function, not a constructor):

```
foldC : ([[ C ]] A → A) → W C → A
```

In our meta theory, this looks like

```
data Tm where
    ...
    foldC : Tm (Γ ::ctx [[ C ]] A) (A [ v1 ]TpTms) → Tm (Γ ::ctx W C) (A [ v1 ]TpTms)
```

where the first term is the algebra that results in a term that can take a W-type and turn it into the result. Writing it like this is also convenient for the computation rule that we will soon introduce.

In Agda, we can define the fold as

no termination check:

```
foldC : ([[ C ]] A → A) → W C → A
foldC alg (In f) = alg (map (foldC alg) f)
```

(which can be made terminating by inlining or sized types). In order to express this here, we have to express the map function first. In Agda, this looks like

```
map : (A → B) → [[ C ]] A → [[ C ]] B
map f (s , p) = (s , f ∘ p)
```

Again, this stays pretty much the same in the meta theory, just with the explicit context transformations:

```
mapC : Tm (Γ ::ctx A) (B [ v1 ]TpTms) → Tm Γ ([[ C ]] A) → Tm Γ ([[ C ]] B)
mapC f s,p = sigma (fstTm s,p) ((lift-func (lam f) [ apply (fstTm s,p) ]TmTms) oTm-tms (sndTm s,p))
```

In order to write the function composition without too many context transformations, we provice the helper function

```
_oTm-tms_ :
    Tm Γ ((B ⇛ C) [ tms ]TpTms) →
    Tm Γ ((A ⇛ B) [ tms ]TpTms) →
    Tm Γ ((A ⇛ C) [ tms ]TpTms)
f oTm-tms g = coerce-Tm (sym-Tp tms-=>) (coerce-Tm tms-=> f oTm coerce-Tm tms-=> g)
```

We can now express the computation rule for folds in the term equations as

```
foldC-In : ∀
    {C : Tm Γ (ContainerTp n n')}
    {CW : Tm Γ ([[ C ]] (W C))}
    {alg : Tm (Γ ::ctx [[ C ]] A) (A [ v1 ]TpTms)}  →

    (foldC alg [ apply (In CW) ]TmTms) =Tm= mapC (foldC alg) CW
```

which can be used to evaluate the folds in the meta theory. We will leave the exact term substitution interaction and congruence rules to future work for when we build the actual normaliser, but the above formulation already showcases that the basic container definitions work in our meta theory.

4.8 Summary and Future Research

In this Chapter, we achieve three main results. In Section 4.3, we create the concept of **held-in-stasis** data types, which are data types where the recursive calls (meaning, positions in its functor) are wrapped in a monadic action. This represents the data of the type to have to be individually unwrapped by a monad constructor by constructor, allowing us the use the concept of explicit laziness motivated in Chapter 3 Section 3.5. In order to be able to unwrap a data that is held-in-stasis by a monad, we need the concept of **Commuting Contexts**, which are an endofunctor `F` and a monad `M` together with an operation `switch : F (M A) → M (F A)` that allows us to monadically unwrap a single constructor from data of a well-formed data type. Recursively applied, this correctly recreates the data type within the monad `M`, as shown in Theorem 4.3:20 and 4.3:24. Theorem 4.3:24 further gives us insights as to how to run a function on held-in-stasis data, as it is a statement on the preservation of results from a fold operation for held-in-stasis data types that produces the same values as a container fold on the unwrapped data.

To automatically create commuting contexts, we prove in Theorem 4.3:56 that all shapely data types (meaning their containers have only finitely many positions for each shape) can form commuting contexts with locally commutative monads. Locally commutative monads are monads where two adjacent actions can always be swapped. They are a special case of the commutativity property of the monadic variables from Chapter 3 Definition 3.8:1.

The next big result is on creating lattice instances for constructors of well-formed data types. In Theorem 4.4:30, we show that every data type represented by a container `C` with decidable equality on its shapes has a special container, called its *latticed container*, has a lattice instance on its container extension `[[ latticedContainer ]] A` if `A` has a lattice instance. In Lemma 4.5:3, we can show that we can create lattice bi-threshold variables pointing at the positions of this latticed data type. Even though it is obvious from the con-

struction we do not show that they point to the correct values yet.

In Section 4.7, we recreate the data type representing the syntax of type theory from [8, 24] and add extra constructors to allow it to represent sigma types, well-formed types and recursive functions as the eliminators of well-formed types. During that construction, we informally verify that the constraints for our automatic construction of data types still apply to this general type of type theory. We can see that it has a finite amount of constructors and therefore decidable equality on its shapes, as well as only finite positions in each constructor (as it does not take functions as arguments), so it is shapely. The data type is indexed, which is however not necessarily a problem when the data type is bound in a possibly failing monad. We can filter out non-indexed data by whether the data would have proper indices. Therefore, it is highly likely that it can be represented in our framework.

In future research, we have to first construct the concrete locally commutative contexts from variables that we needed to automatically create the commuting contexts with the shapely data. Next, we would need to run the lattice construction for constructors of data types with decidable equality on shapes from Theorem 4.4:30 on the fixpoint of the well formed data types to create the full lattice instance for the entire data type. Same goes for the construction of Lemma 4.5:3, so that we can use variable reads (and possibly writes) as a context for the commuting context construction. We would then create a lattice of at least a non-indexed version of the type of type theory that is monadically held-in-stasis by the monadic variables from Chapter 3. We then write a monadic filter function that can filter our non-indexed type of type theory only for values that actually would have a proper index. This would create a monadic representation for the type of type theory. If during our constructions, already some solving capabilities are added like are foreshadowed in the future research of Chapter 3 Section 3.10, we would have created a solver for type theory. Now, together with the ideas from Chapter 5, we could create a self-improving solver.

Chapter 5

Towards A General Theory of Solving

5.1 Introduction

This Chapter gives all the explanations of how the individual results of this thesis hold together. It is mainly about creating a general theory of solving that in the future could support a notion of what an optimal or at the least steadily performance increasing solver would be. We talk about why having an expressive input language to a solver like in Chapter 4 Section 4.7 (the type of type theory) is not just a nice gimmick, but the actual key to achieving optimal solving. For that, we will start to define what a general solver is, together with giving an algorithm that can find an optimal solver given some metric. Such an algorithm has been created in [84], however its implementation does not run on type theory and leaves a few questions on runtime performance open that we will investigate. Further, we will elaborate how lattice based states can be used to parallelise solvers and give some analysis formalism about their performance, giving a theorem for the mechanism of how parallelisation can speed up solving. At last, we will showcase how search trees, branching and clause learning can be implemented using our formalism, with a theorem about the mechanism by which clause learning can speed up the solving process in our formalism.

One should not be fooled by the work-in-progress nature of this Chapter. While without the full universe of types and terms begun in Chapter 4 Section 4.7 this theory cannot be made precise, its ideas are important to understand how this research is on the right track towards achieving a general theory of solving. Without this chapter, no other result in this thesis would truly make sense pursuing. Further, some ideas that are usually inherent to solving (e.g. concrete search strategies) do not have a particularly high focus in this thesis because they are subsumed by our general theory, so without understanding this chapter, the thesis could feel incomplete to some.

The results of this chapter can be summarised as follows:

- We create a general framework of how solver for any theory would be expressed in type theory, using a simple coinductive type `Perhaps` to model a solvers behaviour under semi decidability.
- Translate the ideas of self-improving solvers from [84] into our formalism
- Give evidence how a solvers implementation would be change during execution in two possible ways: Exchanging the algebra for its creation mid execution or having the search state be expressive enough to change the solvers behaviour by merely changing into a different state.
- Showing how to run solvers in parallel by modeling them to run on an increasing, lattice based state
- Showing how increasing the states information may decrease runtime (leaving dealing with the possible overhead of a bigger state to future research).
- Showcasing how search strategies are part of the state change that would be modeled through lattices and therefore do not need to be made explicit yet until they can be properly incorporated together with the ideas from [84]
- Explaining unit propagation, branching and clause learning in the lattice based solving theory
- Giving a concrete working mechanism of branches within a search state
- Modelling clause learning as caching solving computation together with a construction of simple clauses that can cause a speedup.

Things this chapter cannot show unless the results on Chapter 4 Section 4.7 (the type of type theory) are complete and weaved into the lattice based theory:

- A solver is always created for a concrete universe. Therefore, the fully running solver for type theory requires the type of type theory to be fully completed
- The concrete choice which branches are opened and how they are combined cannot be made precise without the precise complexity notion arising from for example counting normalisation steps in the type of type theory. The solver reasoning about the solving process itself can only be made concrete on the concrete universe.
- The concrete choice of clauses to learn. This would be done using the same technology as choosing the correct branches which, again, requires the concrete universe.

All of these results can be achieved but require additional engineering that, due to the resource constraints on this PhD, have to be left to future work.

5.2 General Solvers

Code-Tag: `GeneralSolving`

In an inconsistent world, we can create a solver as a function

```
solve : (X : Set) → X
```

where we are given a type and create an inhabitant of that type. Of course, the existence of such a function would imply that we could find a proof for `Zero`, which would lead to an inconsistency. In an ideal consistent world, we would want to have a function

```
solve : (X : Set) → Dec X
```

Where we either get an inhabitant of `X` or a proof that there is no such inhabitant. Sadly, because `Set` contains recursive functions and can express for example the lambda calculus, such an ideal `solve` function cannot be created due to undecidability [12]. What we can create however is a solver that may create a value, but we will never know if it ever does so. We model that using a corecursive data type.

Definition 5.2:1: **Perhaps**

Given a type `A`, we define the corecursive data type `Perhaps` as the greatest fixpoint of the algebra

```
PC-alg : A ⊎ B → B
```

defined in Agda as the corecursive record (with two renames for the disjunction constructors):

```
record Perhaps (A : Set i) : Set i where
    constructor PC
    coinductive
    field
        val? : A ⊎ Perhaps A
```

We furthermore rename the constructors to `_⊎_` as

```
val x = left  x
ctd x = right x
```

so that a `Perhaps A` value always breaks down to either a value or a continued search for a value.

For proof purposes, we can show that bisimilarity holds for the `Perhaps` type

Lemma 5.2:2

Bisimilarity holds for the perhaps type, meaning that forall `ph ph' : Perhaps A`, it holds that

```
bisim : val? ph ≡ val? ph' → ph ≡ ph'
```

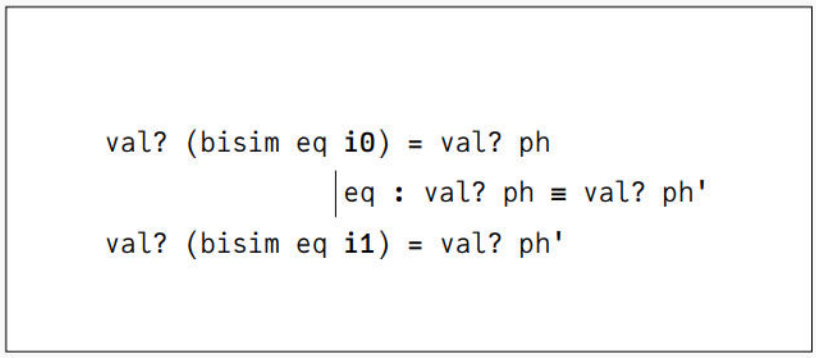
So if the next value of two `Perhaps` values are equal, then so are the two perhaps values.

Proof 5.2:3

Using cubical the cubical proof

```
bisim : val? ph ≡ val? ph' → ph ≡ ph'
val? (bisim eq i) = eq i
```

expressed through the diagram



stating that if we unpack the `Perhaps` values during the equation, they have to end up at the same endpoints as the equation between their next iteration given, so also the original values have to be the same (similar to extensionality).

With the `Perhaps` type, we have a value that may exist, however we might not know whether it ever will. We will define a few convenience functions

```
extract : ℕ → Perhaps A → Maybe A
extract 0 _      = nothing
extract (1+ n) p with val? p
... | val  d = just d
... | ctd ph = extract n ph

hasAnyVal : Perhaps A → Set
hasAnyVal = ∃[ n of ℕ ] ∃[ a of A ] (extract n ph ≡ just a)
```

The first function `extract` can extract a value if it occurs in a depth of at most `n` continuing calls `ctd`. With `hasAnyVal`, we state that at some point, the `Perhaps` value gives a value.

The best realistic, general solver that could now be built has the type

```
solve : (X : Set) → Perhaps X
```

where it may or may not give a value of the given type.

Example 5.2:4

As an example, we can use such a solver to create a SAT-Solver. First, we define some infrastructure for n-ary Boolean functions:

```
funN : (n : ℕ) (A : Set) (B : Set) → Set
funN 0      A B = B
funN (1+ n) A B = A → (funN n A B)

_^_ : Set k → ℕ → Set
A ^ 0      = Unit
A ^ (1+ n) = A × A ^ n

_$n_ : funN n A B → A ^ n → B
_$n_ {n = 0}    b unit      = b
_$n_ {n = 1+ n} f (a , atp) = f a $n atp
```

Where `funN` describes an n-ary function with a given output type, `_^_` descrines n-ary tuples and `_$n_` the uncurried function application giving an n-ary tuple to an n-ary function. The SAT-Solver would now simply be

```
SAT-Solve : (f : funN n 𝔹 𝔹) → Perhaps (∃[ b of 𝔹 ^ n] (f $n b ≡ true))
SAT-Solve f = solve (∃[ b of 𝔹 ^ n] (f $n b ≡ true))
```

We will later also ensure that this solver would be complete, meaning that it finds a solution iff one exists. Also, defining SAT-solvers this generally actually gives rise to solvers beyond classical SAT. In

classical SAT, we are given a Boolean formula that has to have a certain shape, created with the Boolean functions `not``𝔹`, `and``𝔹` and `or``𝔹`, but a general Boolean function has no such limitation, meaning that we have even already created a solver for Quantified Boolean Formulas (QBF) and beyond.

Of course in practice, a solver of the exact type `solve : (X : Set) → Perhaps X` cannot be created while also doing anything meaningful because `Set` does not have any destructors. We cannot case split over an element of an arbitrary type to know which specific type it is. This issue can be fixed by using a universe of possible types

Definition 5.2:5: **Universe**

A **universe** is a type `U` together with an interpretation function

```
[_]U : U → Set
```

that turns an element of `U` into a type.

A universe is actually isomorphic to the definition of a container. With a type of type theory like the one in Chapter 4 Section 4.7 basically every type can be described, so constraining ourselves to universes as objects for types is not a loss of generality. A solver now has the type

```
solve : (X : U) → Perhaps [ X ]U
```

Side Note 5.2:6

Using universes additionally has the upside that the type for solvers in general stays predicative. If we define the type of all solvers via the `Set` definition, we get

```
Solver : Set (lsuc i)
Solver = (X : Set i) → Perhaps X
```

which means that a solver could not create another solver, as the type of all solvers lived in a higher universe. Given however a universe with an interpretation

```
[_]U : U → Set i
```

for a fixed level `i` resolves the issue and the type for all possible solvers becomes

```
Solver : Set i
Solver = (X : U) → Perhaps [ X ]U
```

and this type might be within the universe. So for a sufficiently expressible universe, it holds that

```
Solver ∈ Image [_]U
```

As with the universe created in Chapter 4 Section 4.7 we have a strong enough universe to express any type of interest that itself lives in `Set0`, we will introduce some notation so that we do not always have to make the existence of the universe explicit. Instead of writing

```
module _ (XinU : X ∈ Image [_]U) where
    XU = fst XinU
    find : Perhaps [ XU ]U
    find = solve XU
```

we just directly write

```
find : Perhaps X
find = solve X
```

We note that this will (directly) not compile in Agda in some relevant cases due to the remarks in Sidenote 5.2:6. The actual code has to be written using the proofs of inclusivity of types in the interpretation of U.

5.2.1 Complete and Self Improving Solvers

We can now state what it means for a solver to be what we call *semi complete*. This means that for as long as an inhabitant of the given type exists, it is eventually found. This is theoretically not arbitrary completeness because classical mathematics assumes that there exist things that cannot be proven, for constructive logics however we can always enumerate all possible proofs so we can semi-decide whether a proof exists, completely for all possible statements.

Definition 5.2:7: **Completeness of a Solver**

A solver over a universe `solve : (X : U) → Perhaps [ X ]U` is **semi-complete** iff it holds that

```
∀ {X : U} (x : [ X ]U) → hasAnyVal (solver X)
```

This means that for as long as there is any proof of `[ X ]U`, the solver finds a proof (not necessarily the same one).
This general property can be expressed as

```
semi-complete : Solver → Set i
semi-complete solver = ∀ {X : U} (x : [ X ]U) → hasAnyVal (solver X)
```

or, for the impredicative definition of a solver as an object of type `(X : Set i) → Perhaps X`:

```
semi-complete : Solver → Set (lsuc i)
semi-complete solver = ∀ {X} (x : X) → hasAnyVal (solver X)
```

The interesting thing is that a solver can now search for other solvers. If that searching solver is complete, it is even guaranteed to find one if it exists. For exmaple, we can write (omitting universe notation)

```
createSolver : Σ Solver semi-complete → Perhaps (Σ Solver semi-complete)
createSolver (solve , _) = solve (Σ Solver semi-complete)
```

We are not using the completeness property here, but this makes a point that a solver can, with en expressive enough universe, create other solvers. This becomes especially important when we measure the performance of a solver using a function

```
measure : Solver → R
```

Where `R` is some type with a (partial) order relation defined over it. We note that in general type theory, most useful measures cannot be expressed this way because they would rely on the concrete implementation of the solver. This issue can be solved however by defining the measure over the respective solver term in the universe.

Side Note 5.2:8

We do not need to give a concrete measure function for this construction to work, however, it can help to have some intuitions on what that measure could be. The most obvious measure would be using the term within the $\mathcal{O}$-notation. The notation $\mathtt{a_n} \in \mathcal{O}(\mathtt{f(n)})$ states that given a sequence $\mathtt{a_n}$ (representing the amount of steps an algorithm takes on in input of size $\mathtt{n}$, modeled as a function $\mathtt{a} : \mathbb{N} \to \mathbb{N}$) and a function $\mathtt{f} : \mathbb{N} \to \mathbb{R}$, there exists an input size $\mathtt{n_0}$, such that forall $\mathtt{n'} \geq \mathtt{n_0}$ it holds that $\mathtt{f(n')} \geq \mathtt{a'_n}$, meaning that for sufficiently large inputs, we get an upper bound in the amount of steps that our solver does. A term $\mathtt{a}$ for the $\mathcal{O}$-notation is now considered larger than a term $\mathtt{b}$ iff it holds that $\mathtt{a_n} \in \mathcal{O}(\mathtt{b(n)})$ (where $\mathtt{a_n}$ is the sequence we get from the function $\mathtt{a}$).
As the $\mathcal{O}$-notation only deals with worst-case complexity, it might also be useful to consider the expected value of solver runtime given a distribution of problem inputs. The problem here is minor, but as we do not know if the solver terminates or not, we can often at best give a lower bound on the exprected value. If we however include the allowed runtime for the solver on the problem in the problem input (meaning that the problem input would be a $\mathtt{Set} \times \mathbb{N}$, with the second number stating the maximum steps a solver can make), we can (possibly) get the expected value of the solvers chance of success. This would mean a solver increases its chances of success by running faster (or, if we also add the memory consumption as a constraint, by saving memory).

We can now have a solver improve itself (or, technically, any solver) by writing (omitting universe notation):

```
improveSolver : (solve : Solver) → Perhaps (∃[ solve' of Solver ] (measure solve < measure solve'))
improveSolver solve = solve (∃[ solve' of Solver ] (measure solve ≤ measure solve'))
```

Therefore, as soon as we have found a better solver, we can just use the better solver for solving. Under certain choices for the concrete type $\mathtt{R}$, a continued improvement can lead us to a global maximum for a solver according to the measure.

Side Note 5.2:9

The continued improvement only get us to the maximum of a sequence if for the pre-order defined on R every (strongly) monotone growing sequence actually reaches its maximum (if it exists), so

```
∀ (a : ℕ → R) →
    -- if a maximum in the sequence exists
    (∃[ n of ℕ ] (∀ n' → a n' ≤ a n)) →
    -- and the sequence grows strongly monotone
    (∀ n n' → n < n' → a n < a n') →
    -- it also reaches its maximum
    (∃[ n of ℕ ] (∀ n' → a n' ≤ a n))
```

Of course, this, in general, does not work on the real numbers and possibly not even on terms for the $\mathcal{O}$ notation. There are a few tricks. For example for the real numbers, we can ask the improvement to be at least the size of a fixed ε, as local maxima would eventually not allow for arbitrarily large improvements, giving a minimum of improvement would find the global optimum within ε as the margin of error, if it exists.
But more generally, if there is a global maximum, we can actually search for it by specifying we want the solver for the global optimum. However, not only would that be potentially slow if we do not improve the solver during the search for the global optimum, there also might not be a global optimum. In this case, we would want to maximise the improvements that we get in between solver improvements. But even that maximum might not exist and might not be findable through the hillclimbing approach, having the same problems as before. What we can do however is to search for a solver where the improvement speed is attempted to be increased by order or magnitudes, meaning that it starts maximizing more and more derivatives and derivatives of those derivatives and so on until we find a proof of global maximality and maximum improvements. If there were a solver with better improvement, its improvement term would eventually show up in this recursively improved sequence. Making this idea precise is probably a whole course on calculus and ordinals, so we will not go into detail about this in this thesis.
To keep things simple, we assume that improving a solver, or at least improving the improvements recursively, will suffice for reasonable optimal solvers.

The author in [84] stops at this point because this construction gives us a way to search for optimal solvers according to the metric (though we note that our type theoretic formulation remains novel, as well as our input on possible measures beyond the $\mathcal{O}$-notation. The paper uses an older formalism).

For us, however, two more things are important. First, of course, the initial complete solver to start the self improvement majorly decides about how fast the improvement is actually going. That is why we will lift existing, well established search techniques into the general form needed for the theory in this thesis in order to have this construction have a good starting point. But as soon as we have rebuilt the existing techniques the next interesting question arises: does this theory of self improving solvers give us insides as to how a concrete, optimal solver would operate?

5.2.2 Towards Locally Improving Solvers

If we stop a solver mid evaluation, we get a state where there are a collection of unresolved recursive calls to the solver in an only partially assembled result value. Interestingly enough, we can use this partially assembled result value in order to exchange the algebra with which the value is being created. We showcase this at the example of non-dependent container folds before we go to the dependent container folds necessary to create a solver.

We can define a fold operation that stops using an algebra after a certain computation depth, from which on it continues with a different algebra.

```
foldCN : (n : ℕ) → ([[ C ]] A → A) → W C → (([[ C ]] A → A) → A)
foldCN 0      alg (In (s , p)) alg' = foldC alg' (In (s , p))
foldCN (1+ n) alg (In (s , p)) alg' = alg (s , (\w → foldCN n alg w alg') ∘ p)
```

We can read this function as `foldCN n alg w` producing a partial value `A` for `n` steps that still needs an algebra `alg'` in order to be fully created. In the case where there are no steps left, we switch to the algebra `alg'` and just continue using a normal fold. If we have steps left, we use the original algebra to assemble the overall value, using the values from the switched out algebra. For a simple correctness check, we prove that this works as intended if we stick to the same algebra

Lemma 5.2:10

```
∀ {n alg w} → foldCN n alg w alg ≡ foldC alg w
```

Proof 5.2:11

By induction over `n`

This partial fold showcases the principle of exchanging a program during its execution. There is a slight flaw in this that we will ignore for now, but technically this is still not running the improved solver everywhere possible because the final construction of values is still done using the old algebra (and at the point where the better solvers have given their values we could also use the better solvers to put those values together). We will ignore this for now to keep the implementation simple.

This principle of exchanging the algebra mid computation also works for dependent types with the dependent fold. We quickly recap the definition of the dependent container fold:

```
foldPi : ((c : [[ C ]] (Σ (W C) A)) → A (In (map fst c))) → (w : W C) → A w
foldPi alg (In (s , p)) = alg (s , \p' → p p' , foldPi alg (p p'))
```

We are given a special kind of algebra that also contains the original value before the computation, so that it can be used as an index to the result (by mapping `fst` to the functor, so that the original value is restored without changing the shape of the current constructor). This way, we can define dependent functions using folds, as is needed for the correct implementation of the solver. We can create the same, algebra switching computation here as

```
foldPiN : (n : ℕ) →
    ((c : [[ C ]] (Σ (W C) A)) → A (In (map fst c))) →
    (w : W C) →
    ((c : [[ C ]] (Σ (W C) A)) → A (In (map fst c))) → A w
foldPiN 0      alg w            alg' = foldPi alg' w
foldPiN (1+ n) alg (In (s , p)) alg' = alg (s , \p' → p p' , foldPiN n alg (p p') alg')
```

and still have the same correctness property

Lemma 5.2:12

```
∀ {n alg w} → foldPiN n alg w alg ≡ foldPi alg w
```

Proof 5.2:13

By induction over `n`

As solvers result in the `Perhaps` functor which is dependent on the input to produce the correct type, the solver implementation will automatically be correct. It is highly likely that as long as both algebras result in semi-complete solvers, also the solver resulting from the algebra change will be semi-complete. This will likely result from implementation of the standard `Perhaps` monad. The only thing that should change when exchanging the solver algebra is that the desired values will be computed faster, but we have to leave this to be done in future work. The point is: There is strong indication that a solver can improve itself even while it is running, meaning that we do not have to rely on the improvement solving call to actually finish and give an improvement before we can run another solving query. Both the improvement and other solving query can run in parallel. Technically, even the improvement query can occasionally change its own solvers implementation mid run if it finds better solver implementations for special cases. As interesting as this construction is, we have to leave it for future work, but one point should be clear: A solver will benefit from parallelisation. With parallelisation it can run its own improvement alongside its other search queries and speed up during the computation. Therefore, we will continue at this point to show that the theory developed in this thesis can be used to run a parallelisable solver and also show that other often used solving techniques can be implemented here.

5.3 On Constructing a Solver

Code-Tag: `GeneralSolving` Code-Tag: `LatticeTheory`

In this section, we will have a look at the possible construction techniques for a solver to see if we can derive some behaviour of an optimal solver from its construction.

5.3.1 Constructing Perhaps-Values

First, we note how a solver for a universe `U` is a function into a `Perhaps` type:

```
solve : (X : U) → Perhaps [ X ]U
```

As `Perhaps` is a coinductive type of a strictly positive functor, we can create a co-fold operation over a co-algerba into the type, creating its value from a seed

```
coFold : (A → A ⊎ A) → A → Perhaps A
val? (coFold coalg a) with coalg a
... | val x  = val x
... | ctd a' = ctd (coFold coalg a')
```

We assume that the productive functions definable into `Perhaps` are definable using such a co-fold and a co-algebra when additionally using some context `M` (resulting in a co-algebra of type `M A → M A ⊎ M A`) like showcased in Chapter 4 Section 4.2.3. Similarly, we can also map over a `Perhaps` value as

```
mapPerhaps : (A → B) → Perhaps A → Perhaps B
val? (mapPerhaps f ph) with val? ph
... | val x = val (f x)
... | ctd c = ctd (mapPerhaps f c)
```

Side Note 5.3:1

We are sufficiently certain that there is a monad instance for `Perhaps` using the above functions as a basis, however, due to the time, food and money constraints on this thesis we will not formally prove its existence at this point.

Therefore, in order to create a solver, we have to find a coalgebra `coalg : ∀ X → M [ X ]U → M [ X ]U ⊎ M [ X ]U`, using some context `M : Set → Set` and a seed (initial state) `seed : ∀ X → M X`, possibly extracting some additional values from the context `M` with a function `interp : ∀ X → M [ X ]U → [ X ]U` to implement the solver via a co-

fold as

```
solve : (X : U) → Perhaps [ X ]U
solver X = mapPerhaps interp (coFold coalg seed)
```

We note at this point that through the use of the co-algebra, the solver is basically always created using a state transition. If we look at the definition of the co-fold

```
coFold : (A → A ⊎ A) → A → Perhaps A
val? (coFold coalg a) with coalg a
... | val x  = val x
... | ctd a' = ctd (coFold coalg a')
```

We either give a value or continue with a state change. Therefore, the algebra in use is similar to using a state monad to create a state change until some value can be deduced. It is not isomorphic however because the state monad always creates a successive state, whereas the perhaps co-algebra discards the state when a value has been found. That being said, we can always turn a stateful computation resulting in maybe a state value as an output into a coalgebra for `Perhaps` as

```
coAlgFromState : (A → A × Maybe A) → (A → A ⊎ A)
coAlgFromState mMa a with mMa a
... | a' , just x  = val x
... | a' , nothing = ctd a'

stateFromCoAlg : (A → A ⊎ A) → (A → A × Maybe A)
stateFromCoAlg coAlg a with coAlg a
... | val x  = a , just x
... | ctd a' = a' , nothing
```

and, vice versa, turn a coalgebra for `Perhaps` into a stateful computation. The only reason they are not isomorphic is because `coAlgFromState` throws away the state on successful value creation. This can be fixed with a slightly

different definition of `Perhaps`, which we will do later in this section, but for now it can already be shown that these two functions form an injection from `State A (Maybe A)` to the co-algebras `A → A ⊎ A`

Lemma 5.3:2

```
(coAlgFromState ∘ stateFromCoAlg) ≡ id
```

Proof 5.3:3

Proof by extensionality over the co-algebra `f : A → A ⊎ A` and seed value `a : A`, by case splitting on `f a`.

It should be noted that the connection between creating a `Perhaps` value from a stateful computation is **the main connection between Chapter 3 and generalised solving**.

Side Note 5.3:4

There is a slight design flaw in Chapter 3, in that we *do not* use a state monad isomorphic to `State S (M X)` as a state transformer, but rather use the generally used state monad transformer `StateT S M X = (S → M (S × X))`, which means that we also possibly throw away the state during the state change. This is a problem, because we are throwing away the state on failure to compute a value, which is exactly when we would need the state to redo the computation on a bigger state, as is done in the cofold. More concrete, if we created the coalgebra from a state change of the generally used state monad transformer on `Maybe`, extracting the coalgebra would look like

```
-- this would create perhaps values where if no value can be created form the initial state
-- no value will ever be created.
coAlgFromStateT : (A → Maybe (A × A)) → (A → A ⊎ A)
coAlgFromStateT mMa a with mMa a
... | just (a' , x) = val x
... | nothing       = ctd a
```

where we continue the construction on the input state `a` in the case of failure and not some next state.
Fixing this is simple, but we would have to redo all results from Chapter 3 in the `State A (M X)` monad transformer which we will have to push to future research. The results from Chapter 3 can still be used however by using a monad `M X = A × M' X`, which will always preserve the state. The problem why this cannot be done directly is because as `M` is in the current formulation not indexed by the previous state, we lose the ability to know that the state we are getting out of the monad is actually bigger than the initial state we started from. We can get this behaviour by using `id` for `M` for the results of Chapter 3 and redo the variable extraction on the `LState (VarAsm X)` monad.

5.3.2 Running Solvers in Parallel

5.3.2.1 Naive Parallelisation

There is a simple way to make two solvers run in parallel by just taking two `Perhaps` values and using the first value that comes up, as

```
choose : Perhaps A → Perhaps A → Perhaps A
val? (choose a b) with val? a
... | val x  = val x
... | ctd a' = ctd (choose b a')
```

If the first `Perhaps` has a value we return it, otherwise we continue bo checking the other perhaps value first. We can show that using this method, we are getting one of the values (if exists) in double the amount of `ctd` calls than it would have needed to just query the successful `Perhaps` value. However, as we have mentioned in Section 5.3.1, `Perhaps` values are always created with some form of state. Therefore, we will investigate what can (potentially) happen to the complexity if two `Perhaps` values share their state, potentially sharing results that both constructions need which will then only need to be computed once.

We begin by proving that the naïve `choose` construction actually doubles the time that we need in order to create one value. To do that, we define a predicate that pinpoints exactly the amount of `ctd` calls before a value is created. As we are using cubical Agda, we will not describe it as an indexed type but make use of equality types as

```
_hasVal_at_ : Perhaps A → A → ℕ → Set i
ph hasVal a at 0 = val? ph ≡ val a
ph hasVal a at (1+ n) = ∃[ a' ] ((val? ph ≡ ctd a') and (a' hasVal a at n))
```

If the value is at index `0`, we state that the `Perhaps` value has to result in a value when asked for the next iteration. In the recursive case, we state that the next iteration has to be a `ctd` where the result reaches the value in one less steps. We can now prove the following lemma:

Lemma 5.3:5

Given values `a b : Perhaps A`, `x y : A` and natural numbers `n n' : ℕ`, it holds that

```
chooseVal : a hasVal x at n → b hasVal y at n' →
    (choose a b hasVal x at (2 * min n n')) or (choose a b hasVal y at (1+ (2 * min n n')))
```

so the resulting value of `choose` is either on an even index when it is being the left value `x` or an odd index when it is the right value `y`. Both values need at least the double of the minimum between the two original indices to reach the value.

Proof 5.3:6

We do the prove by induction over `n` and `n'`. In the case of `n = 0`, the first value `a` has to have `val? a ≡ val x` because `a hasVal x at 0 = val? a ≡ val x`, which implies that `val? (choose a b) ≡ val? a`, we can deduce that `choose a b hasVal x at 0`. Similar holds in the case where both `n = 1+ n` and `n' = 0`, where through

```
a hasVal x at (1+ n) = ∃[ a' ] ((val? a ≡ ctd a') × (a' hasVal a at n))
```

it holds that `val? a ≡ ctd a'`, wherefore `val? (choose a b) = ctd (choose b a)`. Further, as `b hasVal y at 0 = val? b ≡ val y` we can de-

duce that `val? (choose b a') ≡ val y` and therefore `choose a b hasVal y at (1+ 0)`.
In the recursive case, where through `n = 1+ n` and `n' = 1+ n'` it holds that `val? a = ctd a'` and `val? b = ctd b'`, we get one of two values. It either holds that

```
even : choose a' b' hasVal x at (2 * min n n')
```

or that it holds that

```
odd : choose a' b' hasVal y at (1+ (2 * min n n'))
```

In order to transform these into the goal statements, we need a small lemma, stating that if we have to `ctd` values, the resulting value takes two `ctd` values more to get to the result.

Lemma 5.3:7

chooseHasValSuc : choose a b hasVal x at n -> choose (PC (ctd a)) (PC (ctd b)) hasVal x at (1+ 1+ n)

Proof 5.3:8

By induction over `n`

We apply this statement to `even` or `odd`, respectively and only have to transform the integer terms respectively through coercions to get the resired output of either

```
chooseHasValSuc even : choose a b hasVal x at (2 * min (1+ n) (1+ n'))
```

or of

```
chooseHasValSuc odd : choose a b hasVal y at (1+ (2 * min (1+ n) (1+ n')))
```

Side Note 5.3:9

We do not show the entire proof code here because it is quite techincal, but we note that there is one construction that is often needed. When only the proof `val? a ≡ val x` exists, the result of `choose a b` can still not be computed for technical reasons. However, the value `choose (PC (val x)) b` can be computed (where `PC` is the constructor for `Perhaps` values). In order to transform those, we can use bisimilarity to get

```
valEq : val? a ≡ val x → a ≡ PC (val x)
valEq eq = bisim {ph = a} {ph' = PC (val x)} eq

ctdEq : val? a ≡ ctd a' → a ≡ PC (ctd a')
ctdEq eq = bisim {ph = a} {ph' = PC (ctd a')} eq
```

and coerce the terms that need to compute, respectively.

5.3.2.2 Lattice Based State Change

In order to improve upon the search speed of Lemma 5.3:5 we will have the states of a `Perhaps` value construction (states that are always existing according to Section 5.3.1) be shared between two `Perhaps` values. We will start by making the state change in the construction of a `Perhaps` value visible, and make sure that the result of the `Perhaps` value does not change if additional information is added to the state, exactly like the constructions from Chapter 3 (though we will not use the monadic construction due to Sidenote 5.3:4). We will then proceed to have two statefully created `Perhaps` values share their state to, potentially, speed up their computation.

Definition 5.3:10: Perhaps Semilattice State

Given a type `S` with a semilattice instance, a type `X` and two functions

```
read : S → VarAsm X
δ    : S → S
```

then this structure is called a `perhaps semilattice state` (or short: perhaps state) iff it holds that `read` is a semilattice bi-threshold function over the semilattice of `S`, `δ` only increases its input and `δ` preserves the semilattice pre-order `_P_`, so

```
read-threshold : IsBiThresholdRead read
δ-dir  : ∀ {s} → s P (δ s)
δ-pres-P : ∀ {s} {s'} → s P s' → δ s P δ s'
```

In order to understand the components and laws of this definition we have a look at how a perhaps semilattice state is turned into a `Perhaps` value

Definition 5.3:11: Perhaps Value of a Perhaps Semilattice State

Given a perhaps semilattice state from Definition 5.3:10, we define its `Perhaps` value as

```
toPerhaps : S → Perhaps X
val? (toPerhaps s) with read s
... | unassigned = ctd (toPerhaps (δ s))
... | asm x      = val x
... | conflict   = val? noValue
```

The idea of the perhaps semilattice state construction is, that the `Perhaps`

value is created from a semilattice state, with a semilattice bi-threshold function (precursor of a variable from Chapter 3) to read out the value as soon as it exists in the state. The state should only increase, so that the `read` function can, during computation, only jump from `unassigned` to having a fixed value and, possibly, eventually to become a `conflict`. For that, we have, much like in Chapter 3, constrained the state transition function δ to be a function that only increases in value. Futhermore, in order to guarantee that the state also increases when halted and being resumed at a later, bigger starting point (we need that for parallelisation), we state that δ also has to preserve the semilattice pre-order. This means that if we transition on a bigger state, we get at least the same information increase we would have had on the smaller state. This does not follow from δ increasing alone, because the information that we are adding on a smaller state could be vastly larger than the information we derive from the bigger state.

Before we jump into the parallelisation and how it might affect complexity, we need to fix on a complexity measure. We will eventually use the same metric as with `Perhaps`, however, it turned out to be beneficial to start with a different metric that we will later show to be equivalent. To begin, we show that we can turn a perhaps semilattice state into a bi-threshold variable. We will then show that the point at which the variable turns into a value only ever occurs earlier when we start from a bigger state, showing a possible runtime improvement. We then show that the point at which the variable turns into a value is the same point at which the generated `Perhaps` value creates its output.

To begin showing that we can turn a perhaps semilattice state into a bi-threshold variable, we give a (family of) functions that will make up its `read` function(s).

Definition 5.3:12: **Perhaps Semilattice State Read (Assignment Extraction)**

Given a perhaps semilattice state from Definition 5.3:10 on the bi-threshold function `read : S → VarAsm X` and state transition function `δ : S → S`, we define the extraction of its assignment after `n` steps as

```
extractAsm : ℕ → S → VarAsm X
extractAsm 0      s = read s
extractAsm (1+ n) s = extractAsm n (δ s)
```

This value extraction gives the value of the `read` bi-threshold function after increasing the state `n` times using `δ`. This can also be pic-

tured as getting the value of `read` after `n` steps. We will shot that every `extractAsm n : S → VarAsm X` continues to be a bi-threshold function. To create these proofs, we quickly reiterate a few properties of the `_=incAsm=_` relation from Chapter 3 Section 3.8.3.1, which determines for a bi-threshold function which results of a read can successively follow after each other according to the pre-order `_P_` of the semilattice. In pseudocode, it is defined as

pseudocode:

```
data _=incAsm=_ (a b : VarAsm X) : Set where
    unas-to-anything :  unassigned =incAsm= a
    asm-eq           :  asm x      =incAsm= asm x
    asm-conf         :  asm x      =incAsm= conflict
    conf-conf        :  conflict   =incAsm= conflict
```

which we rewrite to have the equivalences explicit (and be able to use cubical and homotopy type theory for proofs):

```
data _=incAsm=_ (a b : VarAsm X) : Set where
    unas-to-anything :  a ≡ unassigned →                 a =incAsm= b
    asm-eq           :  a ≡ asm x      → b ≡ asm x     → a =incAsm= b
    asm-conf         :  a ≡ asm x      → b ≡ conflict → a =incAsm= b
    conf-conf        :  a ≡ conflict   → b ≡ conflict → a =incAsm= b
```

A function `read` is a bi-threshold function iff it holds that

```
isBiThresholdRead : ∀ (s s' : S) → s P s' → read s =incAsm= read s'
```

where we call the output of this function at states `s` and `s'` the **threshold of the function `read` at states `s` and `s'`**.

There are two main properties we are interested in:

Lemma 5.3:13

```
asm-asm-+-conf : ∀ {a} {x} → (asm x) =incAsm= a → (a ≡ asm x) or (a ≡ conflict)
```

Proof 5.3:14

Only the `asm-eq` and `asm-conf` cases are still possible according to the premise.

Lemma 5.3:15

```
asm-stays-conf : ∀ {a} → conflict =incAsm= a → (a ≡ conflict)
```

Proof 5.3:16

Only the `conf-conf` case is still possible according to the premise.

If the first value is `unassigned`, we have no information about the second value. That means that a threshold read value can turn from `unassigned` to `asm x`, then stay on that `asm x` to possibly eventually turn into a `conflict`. We will now recreate this property for our `extractAsm n` reads.

Lemma 5.3:17

Given a perhaps semilattice state from Definition 5.3:10 with type `S` having the respective semilattice instance and pre-order relation `_P_`, then it holds that

```
extractAsm-conf-conf : ∀ {s : S} {s' : S} {n : Nat} →
    s P s' → extractAsm n s ≡ conflict → extractAsm n s' ≡ conflict
```

meaning that if we read a conflict after `n` steps starting on `s`, we will continue to read a conflict after `n` steps on every bigger state `s'`.

Proof 5.3:18

By induction over `n`, using the `asm-stays-conf` property form Lemma 5.3:15 on the `read` threshold at `n=0`.

Lemma 5.3:19

Given a perhaps semilattice state from Definition 5.3:10 with type `S` having the respective semilattice instance and pre-order relation `_P_` and `X` being the codomain of the `read` function, then it holds that

```
extractAsm-asm-inc : ∀ {s : S} {s' : S} {n : Nat} {x : X} →
    s P s' → extractAsm n s ≡ asm x → (extractAsm n s' ≡ asm x) or (extractAsm n s' ≡ conflict)
```

meaning that if we read an assignment after `n` steps starting on `s`, we will later on every bigger state `s'` after `n` steps, either read the same value or a conflict.

Proof 5.3:20

By induction over `n`, using the `asm-asm-+-conf` property form Lemma 5.3:13 on the `read` threshold at `n=0`.

With these two lemmata, we can prove that, indeed, `extractAsm n` is a bi-threshold function for every `n`

Theorem 5.3:21

Given a perhaps semilattice state from Definition 5.3:10 with threshold function `read : S → VarAsm X` and the semilattice pre-order `?P?` on `S`, then forall `n`, `extractAsm n` is a bi-threshold function.

Proof 5.3:22

Given two states `s` and `s'` with `s P s'`, we case split by `extractAsm n s` and use Lemma 5.3:17 and 5.3:19 to create the respective constructors for the `_=incAsm=_` predicate.

For solvers, this means that we now have a method to create `Perhaps` values based on a state that runs on semilattices. Those solvers have a stable behaviour, in that we do not get a different outcome by just running them for longer and they do not suddenly stop giving a value even though they previously could have. Further, these solvers can be halted mid computation and be later resumed on more (or less) information.

5.3.2.3 Speedup through Increase in Information

There is a different upside to using semilattices as states. We can show that adding information can only speed up the creation of our semilattice state `Perhaps` values. To do that, we first define the first iteration after which a value has been created

Definition 5.3:23: Perhaps Semilattice State Having First Value After n Steps

Given a perhaps semilattice state from Definition 5.3:10, we define that it has a value `x : X` after `n : ℕ` steps on state `s : S` as

```
hasFirstAsmValue_at_on_ : X → ℕ → S → Set
hasFirstAsmValue x at 0      on s =  extractAsm 0 s ≡ asm x
hasFirstAsmValue x at (1+ n) on s = (extractAsm n s ≡ unassigned) and (extractAsm (1+ n) s ≡ asm x)
```

Meaning that it either starts with the `read` function giving a value or the `read` function was previously unassigned and now turns into an assignment.

We will later show in Lemma 5.3:31 and Lemma 5.3:33 that this is equivalent to asking the respective `Perhaps` value whether it has a value `x` after `n` iterations. We can now show that increasing the state `s` to a bigger state `s'` can only speed up the computation (or keep it unaffected).

Theorem 5.3:24

Given a perhaps semilattice state from Definition 5.3:10 with state type `S` having the semilattice pre-order relation `_P_` and `read` having the codomain `X`, then forall states `s : S` and `s' : S` with `s P s'` (meaning `s'` is "bigger" than `s`), it holds that the first value `x : X` on `s'` occurs sooner or equally soon as it would on `s` (if both produce a value). Formally, this is expressed as

```
speedup : ∀ {n n' s s' x} →
    hasFirstAsmValue x at n on s →
    hasFirstAsmValue x at n' on s' →
    s P s' → n' ≤ n
```

Proof 5.3:25

Side Note 5.3:26

Often, we want to have the final value of `extractAsm` after `n` steps on value `s` just based on the information that `read` already has a certain value of `s` that can only change marginally when applying `δ` to `s` for any `n` times. Therefore, there are the following helpful corollaries

Corollary 5.3:27

```
extractAsm-asm-read : ∀ {s n x} → read s ≡ asm x → (extractAsm n s ≡ asm x) or (extractAsm n s ≡ conflict)
extractAsm-conflict-read : read s ≡ conflict → extractAsm n s ≡ conflict
```

Proof 5.3:28

Both by induction over `n`, using the respective Lemmas 5.3:17 and 5.3:19.

These corollaries give us the possibility to determine the possible outcomes of `extractAsm` after `n` steps when the `read` function already gives some information on the initial state `s`.

We prove this statement by induction over `n` and `n'`. In the case where both `n = 0` and `n' = 0`, it obviously holds that `0 ≤ 0`. Next, we have to show that the case for `n = 0` and `n' = 1+ n'`, meaning that `n'` would definitely be larger than `n`, cannot occur. As the first value of `s` is at the current iteration, we know that the result of `read` has to be an assignment, so it holds that `read s ≡ asm x`. This also means that at no later states `s'`, `read` could turn to be unassigned again. But precisely that would have to happen if going through the state change starting from `s'` we would first have to hit a state that results in an unassignment and then turns into an assignment. Especially, if starting from `s'` the value change were at `1+ n'`, then the value of `s'` at `n'` would have to be unassigned, which it cannot be when from the threshold read onf `read s ≡ asm x` it follows that either `read s' ≡ asm x` or `read s' ≡ conflict`.
In the opposite case where `n = 1+ n` and `n' = 0`, it obviously holds

that `0 ≤ 1+ 0`.
Finally, in the recursive case `n = 1+ n` and `n' = 1+ n'`, we use the fact that `n' ≤ n → 1+ n' ≤ 1+ n` on the recursive call. Using a small lemma, we can transform the premises into a recursive call on `δ s` and `δ s'` as the next states, using the property that `δ` preserves the lattice order. The small Lemma is simply

Lemma 5.3:29

```
firstValPred : ∀ {x s n} → hasFirstAsmValue x at (1+ n) on s → hasFirstAsmValue x at n on δ s
```

Proof 5.3:30

By case splitting on n

We can use this lemma to transform the premises given for `n = 1+ n` and `n' = 1+ n'` are turned into statements on the smaller numbers `n` and `n'` for the recursive call (induction hypothesis) of the proof. This concludes the proof.

To conclude, we quickly show the equivalence (in the propositional sense, not the type equivalence sense) of the statements that a perhaps semilattice state has the first value `x` at `n` iterations on state `s` and that the respective `Perhaps` value creates its value on the `n`th step.

Lemma 5.3:31

Given a perhaps semilattice state from Definition 5.3:10, it holds that

```
firstVal=>Perhaps : ∀ {n x s} → hasFirstAsmValue x at n on s → toPerhaps s hasVal x at n
```

Proof 5.3:32

We prove this by induction over `n`. In the case of `n = 0`, it follows from the premise that the first value is at `0` that `read s ≡ asm x`, which will also result in the `Perhaps` value to create `val x` next. In the recursive case of `n = 1+ n`, it follows that `read s ≡ unassigned` by routing out all other cases through Corollary 5.3:27. Therefore, we can show that currently, the `Perhaps` value would give a `ctd` output, so we use the recursive call (again, using Lemma 5.3:29 to lower the `1+ n` of the premise) to prove that `toPerhaps s` as the output at the same index and therefore `toPerhaps (δ s)` has the output at the next index `1+ n`.

Lemma 5.3:33

Given a perhaps semilattice state from Definition 5.3:10, it holds that

```
Perhaps=>firstVal : ∀ {n x s} → toPerhaps s hasVal x at n → hasFirstAsmValue x at n on s
```

Proof 5.3:34

In the case of `n = 0`, it follows that `val? (Perhaps s) ≡ val x`, which, through case splitting on `read s`, also implies that `read s ≡ asm x`, showing that we have our first value at `0`. In the recursive case of `n = 1+ n`, we can again show that `read s ≡ unassigned`, because otherwise the value flip would not occur. We can show that `read s ≢ asm x` because that would mean that the value for the `Perhaps` value would have already been created. We also know that `read s ≢ conflict`, as in this case there would not be any `Perhaps` value being created, conflicting the premise. To lift the recursive call, we can use a lemma that can only be applied at `read s ≡ unassigned`

Lemma 5.3:35

```
firstValSucc : ∀ {n x s} →
    (read s ≡ unassigned) →
    hasFirstAsmValue x at n on (δ s) → hasFirstAsmValue x at (1+ n) on s
```

Proof 5.3:36

By case splitting on n

We apply this to the recursive call (which needs to have its inputs recreated from reassembling the parts given in the premise through coercions, but that is a bit technical) to prove the lemma.

With these two lemmata we have shown that the creation of a `Perhaps` value from a lattice-based state is indeed possibly sped up by increasing the information content of the initial state. We will now show how this is useful when having several solvers run in parallel.

5.3.2.4 Parallelising Perhaps Semilattice States (and therefore Solvers)

We can now show how two solvers created using perhaps semilattice states can run in parallel, possibly speeding up each others computation by increasing each others states. This can be pictured as having several solvers run in parallel that check individual constraints on the model. If one solver determines that in order to create its output, other values have to be assigned a certain way, then adding this knowledge to the state can speedup the search for another goal that might depend on these values. Sadly, due to the time, money and food constraints on this thesis we cannot give a full framework to create arbitrary, interworking constraints yet, but we can give the construction that runs two solvers in parallel.

Theorem 5.3:37

Given two perhaps semilattice states `p1` and `p2` on semilattice state type `S` that both share the same `read` threshold function, we get a merged perhaps semilattice state using the pointwise merged state stransition function

```
δ : S → S
δ s = δ_p1 s <> δ_p2 s
```

where `_<>_` is the merge operation of the semilattice on `S`.

Proof 5.3:38

As boths semilattice state share the same `read` function, it is trivial why it would stay a bi-threshold function. Therefore, we are only left to show that the merged `δ` is increasing and preserving the semilattice pre-order `_P_`.
That the function stays increasing can be shown through a general lemma on the pre-orders of semilattices

Lemma 5.3:39

```
mergeP : ∀ {x} → x P y → x' P y' → (x <> x') P (y <> y')
```

Proof 5.3:40

As `_P_` is defined as

```
x P y = x <> y ≡ y
```

we can show this, given `xPy : x P y` and `x'Py' : x' P y'` using the lattice properties as

```
(x <> x') <> y <> y'   =< commutative >
(x' <> x) <> y <> y'   =< associative >
((x' <> x) <> y) <> y' =< associative >
(x' <> (x <> y)) <> y' =< xPy >
(x' <> y) <> y'        =< commutative >
(y <> x') <> y'        =< associative >
y <> (x' <> y')        =< x'Py' >
y <> y'                qed
```

From this, we can create the statement

```
mergeP' : x P y → x P y' → x P (y <> y')
```

by applying idempotency on the result of `mergeP`. We then use `mergeP' (δ-dir`$_{p1}$`) (δ-dir`$_{p2}$`)` to prove that the merge stays increasing (or decreasing, depending on the semantics of the semilattice). We now have to show that the merge preserves the pre-order. We do so using the semilattice properties

```
((δp1 s <> δp2 s) <> δp1 s' <> δp2 s')   =< associative >
((δp1 s <> δp2 s) <> δp1 s') <> δp2 s'   =< associative >
(δp1 s <> δp2 s <> δp1 s') <> δp2 s'     =< commutative >
((δp1 s <> (δp1 s' <> δp2 s)) <> δp2 s') =< associative >
((δp1 s <> δp1 s') <> δp2 s) <> δp2 s'   =< associative >
(δp1 s <> δp1 s') <> (δp2 s <> δp2 s')   =< δ-pres-Pp1/p2 >
(δp1 s' <> δp2 s')                       qed
```

which concludes the proof.

With more resources, we could quickly proof that the merge continues to only ever speed up the solving process, but we leave this as an exercise to the enthusiastic reader. Of course, in future work, this can be extended into a whole monadic structure that can express constraints, possibly generated from representations of evaluation functions like we started to do in Chapter 4 that gives us a framework to turn any written program into a constraint solver, but for now it just gives us a framework to use lattices in order to reason about the complexity of solving (or running programs in general) based in the information available at a certain state during the computation.

5.4 On the Generality of Lattice Based States

Using states that have a semilattice instance gives us the power to reason about computation based on the knowledge at a certain state. In fact, we can define whether certain knowledge exists on a state by giving a bi-threshold function `f : S → VarAsm X` for some statement `X` (which we will, colloqially, also just call a variable here), and if it gives an output then the knowledge is present at that state. Interestingly enough, if there is a full lattice on `S`, we could define that the meet-semilattice describes information increase and the join-semilattice information decrease. That means variables on the meet semilattice represent knowledge that *is* present on a state, whereas variables on the join-semilattice represent which information *is not* present. Technically, requiring computation to only go in one direction (usually increasing information) seems a bit restrictive because programs are very much allowed to also delete information, which is important for space complexity. However, a program running can have one part that increases information whereas another parts decreases it again, using the same theory. For solvers, we usually want only increasing information because it often leads to guaranteeing that we find a solution, but of course the theory of solving can, at this point, easily be generalised to also include forgetting information again. We put doing that to future work. Further: Using lattices is not a restriction on expressibility, as we have come close to showing that all interesting data types can be encoded as lattices and variables on them in

Chapter 4.

Further, having a formalism of what any algorithm could possibly know on a state gives rise to, potentially making use of the algorithm for creating self improving solvers mentioned earlier in Section 5.2.1. Here, if there is a proof that an optimal or at least improved solver should jump to a certain state of knowledge, then it has an obligation to do so. That means if at the current state the information of which successive state gives an improvement is available, we can act upon in, giving a nicer formalism of what it means to improve a solver mid solving. Sadly, due to the time, money and food constraints on this PhD, we cannot continue to make this idea fully precise, however, we can still use the idea to explain what happens during branching and clause learning

5.5 Search States

Code-Tag: `GeneralSolving`

So far, we have actually only discussed the theory of running a program, and it makes sense: solvers are programs. And as the constraints we are solving for are also just programs it is no surprise that most of the theory of solving is really the theory of computation in general. One thing however that predominantly comes up in solving while having few mentions only in general program execution is the notion of search, often through a tree of possible values. A tree here is just a way to express a collection of possible values or solutions that we have to traverse in order to get the desired outcome. Therefore, solving might just as well be described as traversing a tree, filtering the values of a tree or composing trees until they are easy enough to traverse. With our definition, this looks as follows.

We start by defining possibly infinite trees that also contain subnodes that might only perhaps exist. This construction on the long run will make it easier to define a monad instance for search trees together with solvers, however in this thesis we will just quickly showcase how search over trees would be expressed using a state that can, with the constructions of Chapter 4, be turned into a lattice in future work.

Definition 5.5:1: **Perhaps Trees**

Perhaps trees are define to be the greatest fixpoint over the following functor, called the **perhaps tree functor**:

```
data PTreeF (A : Set) (B : Set) : Set where
    ptval : A → PTreeF A B
    ptctd : B → PTreeF A B
    _<,>_ : (a : B) → (b : B) → PTreeF A B
```

In Agda, expressed as the coinductive record

```
record CoPTree (A : Set i) : Set i where
    coinductive
    field
         next : PTreeF A (CoPTree A)
```

These trees are normal, possibly infinite trees, but their functor additionally contains the constructors of the functor for `Perhaps`. This means that, in theory, we can even create new nodes as the result of a solve. A solver that searches for solutions in a given tree now has the signature

```
chooseFromTree : CoPTree X → Perhaps X
```

This is a common pattern in solving. One side creates the tree of possible values through tree combinators and the other searches the tree for results. The tree combinators are usually the monadic operators plus a disjunction operator, like the `MonadPlus` interface mentioned in Chapter 3 Section 3.3. We can quickly create these combinators, though we will leave proving that they form a monad and that the disjunctions does not remove values as an exercise to the enthusiastic reader.

```
_<|>_ : CoPTree A → CoPTree A → CoPTree A
next (a <|> b) = a <,> b

map : (A → B) → CoPTree A → CoPTree B
next (map f t) with next t
... | ptval x  = ptval (f x)
... | ptctd t' = ptctd (map f t')
... | a <,> b  = (map f a) <,> (map f b)

join : CoPTree (CoPTree A) → CoPTree A
next (join t) with next t
... | ptval x  = next x
... | ptctd t' = ptctd (join t')
... | a <,> b  = (join a) <,> (join b)

_>>=_ : CoPTree A → (A → CoPTree B) → CoPTree B
ma >>= fmb = join (map fmb ma)
```

When it comes to defining the tree of a data type, it can help to define how sigma types can be created, as they form the basis for non-function data types, plus they are important for solvers as they are used to solve for values in the context of certain values existing. A combinator to create a tree for a sigma type is just a generalised version of the monadic bind (using a generalised, dependent version of a map):

```
mapPi : ((a : A) → B a) → CoPTree A → CoPTree (Σ A B)
mapPi f = mapCoPTree (\x → x , f x)

_>>=Sigma_ : CoPTree A → ((a : A) → CoPTree (B a)) → CoPTree (Σ A B)
ma >>=Σ fmb = mapPi fmb ma >>= \{(a , mb) → map (a ,_) mb }
```

These monadic combinators work on all monads and can be used to create an arbitrarily constrained value monadically.

We can also define a search strategy, which is just a state change used to create the `Perhaps` value for the solver. We are not giving any properties for it at this point because those should be common folklore.

Definition 5.5:2: **Raw Search Strategy**

A **raw search strategy** (or short: search strategy) consists of any three functions

```
initialNode : CoPTree X → S
δ           : S → S
hasVal?     : S → Maybe X
```

Where we can turn the initial node into a state, have a state transition and can possibly retrieve a solution from a state. We can turn a raw search strategy into a perhaps value as

```
chooseStratState : S → Perhaps X
val? (chooseStratState s) with hasVal? s
... | just x = val x
... | nothing = ctd (chooseStratState (δ s))

chooseStrat : CoPTree X → Perhaps X
chooseStrat t = chooseStratState (initialNode t)
```

A search strategy therefore always creates a correct solver, so the only properties left to prove about this is whether the strategy is complete or fast. It is also common folklore that search strategies only differ by how the container for the search nodes is implemented. Using a stack gives depth-first-search, a queue breadth-first search and we can weight the subnodes to use heuristics. Before we go into detail on why this is important, a quick example of how breadth first search can be implemented (proof of completeness left as an exercise)

```
BFS-strategy : StateStrategy (List (CoPTree X)) X
initialNode BFS-strategy a = a :: []
δ           BFS-strategy   = getSuccNodes
hasVal?     BFS-strategy   = head ∘ getValsCoPT
```

With the auxillary functions for getting all successive nodes from a list (`getSuccNodes`) and getting all not yet computed nodes (`getValsCoPT`) being

```
getValsCoPT : List (CoPTree A) → List A
getValsCoPT = concatMap (help ∘ next)
    where
        help : PTreeF A (CoPTree A) → List A
        help (ptval x) = [ x ]
        help _         = []

getSuccNodes : List (CoPTree A) → List (CoPTree A)
getSuccNodes = concatMap (help ∘ next)
    where
        help : PTreeF A (CoPTree A) → List (CoPTree A)
        help (ptval x) = []
        help (ptctd t) = [ t ]
        help (x <,> y) = x :: y :: []
```

What is more important here is more of an intuition: Unless a provably correct solver has found a solution, it has to have a representation of all nodes that have not yet been shown solutionless in its state; otherwise, we could not prove that it is complete because we are lacking the proof to discard nodes with potential solutions. Each node represents part of a recursive call to the solver that can only differ by the additional information of a state. This means that changing the behaviour of a solver on a node means changing the initial state that it enters the node at. This also means that we can create a proof of optimality or increased performance just for a certain node and current state. This would make precise of how to improve a solver during its execution based in the ideas in Section 5.2.1. Sadly, in order to make this idea precise and turn it into a running solver, we would have to have a proper universe of all types that can produce executable functions. We got close to that in Chapter 4 Section 4.7, but the technology has to be pushed over the finish line in future work.

One last mention though: If the search state is a collection of nodes, then increasing the state would mean lowering the amount of nodes that have to be searched, as on a state where we give a result, there should only be one possible node left (If this one shall be just the first found one even though there still are other nodes with potential solutions then this means that solutions can be discarded if finding them takes longer than finding an easier solution). This gives an intuition of what happens during the solving procedure: we decrease the amount of possible solutions by increase of knowledge on the search. Two parallel solvers therefore can decrease each others possible solutions, possibly

together faster than alone.

Side Note 5.5:3

In future research, a search state would be expressed as a finite product (tree) of initial solving states using the constructions from Chapter 3 Section 3.9, by using our concept later from Section 5.7 to create branches like a new variable.

5.6 Unit Propagation

Code-Tag: `LatticeTheory`

Using this general theory, we want to coin a broader meaning to the (Boolean satisfiability) SAT-technique of *unit propagation*. In SAT, the basic solving procedure (called the DPLL algorithm [31, 32]) is roughly expressed as follows: A SAT-problem is a Boolean formula. This formula is expressed in conjunctive normal form (CNF), which is just a finite conjunction of disjunctive constraints on finite sets of Boolean variables with polarity (so they may or may not be negated). Those variables with polarity are called *atoms*, and the disjunction constraints on a finite set of atoms is called a *clause*. If the variable of an atom is assigned in a way that makes the atom evaluate to `true` (so assigning the variabe to `true` for no negation and `false` for negation), we call the atom *positively assigned*.

Finding a solution means finding an assignment to the variables so that forall constraints, at least one atom in each of the finite conjunctions evaluates to true. This translates to the described Boolean formula evaluating to true on the same assignment. In our theory, such an assignment is expressed through the product lattice, which we used when creating new variables in Chapter 3 Section 3.9, where each sub-lattice of the n-product lattice represents the assignment of the respective variable. During the SAT unit propagation, variables that we know have to be true or false (because they are the only non-positive-assigned atom in a clause) are set to their necessary values. This process is repeated recursively until no more variables are assigned through being the only ones in their clause. In our theory, assigning a variable means increasing the information of the search state. Therefore, unit propagation is nothing but the recursive increase of information of a state until a final value arises. This is why the perhaps semilattice state values have the δ function: to increase the state's information until, hopefully, the desired value can be read. We therefore define:

Definition 5.6:1: **Order Preserving Increasing Function (OPIFunc)**

Given a semilattice on a type S with pre-order _P_, we call a function δ : S → S an **order preserving increasing function** (OPIFunc) iff the following two laws hold:

```
δ-dir    : ∀ {s} → s P δ s
δ-pres-P : ∀ {s} {s'} → s P s' → δ s P δ s'
```

Definition 5.6:2: **Unit Propagation**

Given an order preserving increasing function δ : S → S, we call the n-th application δ^n s on an initial state s : S the **n-th unit propagation of δ at state s**. The general process of recursively applying δ any number of times is just called **unit propagation**.

Running several unit propagations in parallel can be done by merging order preserving increasing functions as

Corollary 5.6:3

Two order preserving increasing functions δ_1 δ_2 : S → S on the same semilattice for S can be merged into one order preserving increasing function on the same semilattice using the function

```
δ : S → S
δ s = δ ₁ s <> δ ₂ s
```

Proof 5.6:4

Analogous to Theorem 5.3:37.

What is interesting to note here is that unit propagation just describes any lattice based state increasion, which means every computation that can be per-

formed by solvers created using perhaps semilattice state values. Therefore, if the search state is part of this state as well as is described in Section 5.5, then also performing search strategies is part of unit propagation. Therefore, with a sufficiently expressive lattice universe, search tactics can be described with the same formalism as unit propagation. This is the reason why we created structure around the lattice bi-threshold variables. If we look at `S` being our universe, then the lattice bi-threshold variables are our interpretation of that universe in the category of types (`Set`). This means that our search states can hold arbitrary data and we are not just forced to use Boolean variables but can use any structure from type theory. Thinking in those terms also gives rise to the idea that solving is really just solving for a path through a tree (or graph) that is "fast" by some criteria, meaning that solvers can be used to improve solvers.

Now, where unit propagation and search techniques fall together, we only need to formalise which parts of the state behave like a possible solution and which part of the state behaves like a branch to test out different solutions. Distinguishing these will, for future research, be important to define the exact mechanism with which we choose the best available search strategy for sub-goals and we will see how are are already being provided a mechanic of how to "change" a solvers implementation while it is still running.

5.7 Branches

Code-Tag: `LatticeTheory`

5.7.1 General Definition

In this section we will briefly describe what branching means in the general theory. Often when we describe solving as a lattice, we picture the lattice state as a single partial solution that is completed as the state grows. This intuition is not wrong, but as we saw in Section 5.5, the solving state can also be a collection of solutions. Depending on how this range of solutions looks like, we pick one overall solution. In this section we will describe how, by the same mechanism that we use to describe to complete one solution, we can describe how to complete several solutions at the same time. As all of those solutions will exist within one solving state, they can potentially interfere with each other. This also will make unit propagation and branching to be expressed with the same concept in the general theory, as unit propagation not unly uncreases several branched solutions at the same time, but even creates new branches.

Side Note 5.7:1

We note that there is a lattice based theory for solving that has been used to describe SAT-solving and clause learning in [33], however, our theory is slightly more general. While [33] mentions that it is as well not just constrained to SAT-solving and could express more general data types, they are not as rigorous as we are in Chapter 4 whith what they can express. Further, while their branching structure is also contained in the state, they do not give a general mechanism do identify branches and strick with a concrete powerset construction. Finally, in contrast to [33], our state potentially also includes the possibility to have the search strategy be part of the search state.

The first thing we note that when creating perhaps values via perhaps semilattice states is that the `read` function is just a way to read out information from the state. All the computation, even the trying out of different solutions, happens within that state. If we choose the value from several different solutions of a problem that are all encoded within the state, we still choose the value from the overall state. Therefore, the existence of a branch is also really just a special property of a state.

Now to branches in the semilattice based setting. Given an order preserving increasing function, a branch on a state is just a part of the state that behaves like the overall computation, but on a different starting value. To make this precise, we quickly reiterate the definition of semilattice injections from Definition 3.9:14:

Definition 5.7:2: **Semilattice Injection**

Given two semilattices over the types `X` and `Y` and a computable injection from `X` to `Y` defined via the two functions

```
inf : A → B
outf : B → A
```

with the law that

```
outfoinf-id : outf ∘ inf ≡ id
```

This injection is a **semilattice injection** iff the following two laws hold:

```
pres-inf  : inf  (a <>ₓ b) ≡ (inf  a <>ᵧ inf  b)
pres-outf : outf (a <>ᵧ b) ≡ (outf a <>ₓ outf b)
```

A branch is now defined as follows. We first give a textual explanation of the axioms and later a visual one.

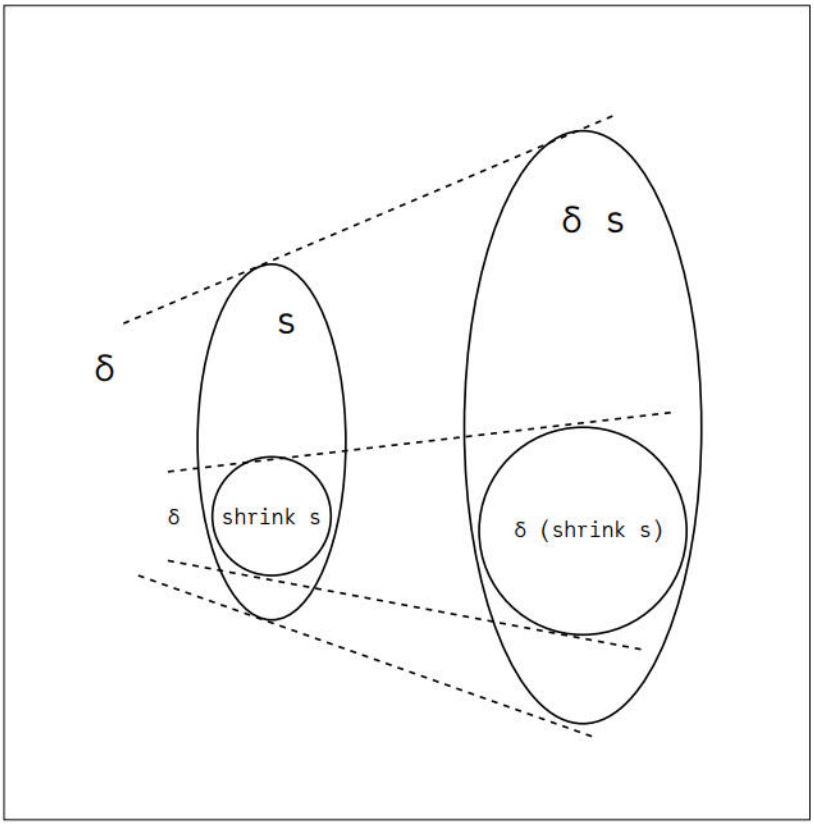
Definition 5.7:3: **Semilattice Branch**

Given an order preserving increasing function `δ : S → S` on a semilattice for `S`, then a **semilattice branch** (or short: branch) is a semilattice injection (where `inf` is called `shrink` and `outf` is called `unshrink`) where the following two laws hold:

```
shrink-unshrink-revdir : shrink (unshrink s) P s
δ-similar : shrink ∘ δ ∘ unshrink ≡ δ ∘ shrink ∘ unshrink
```

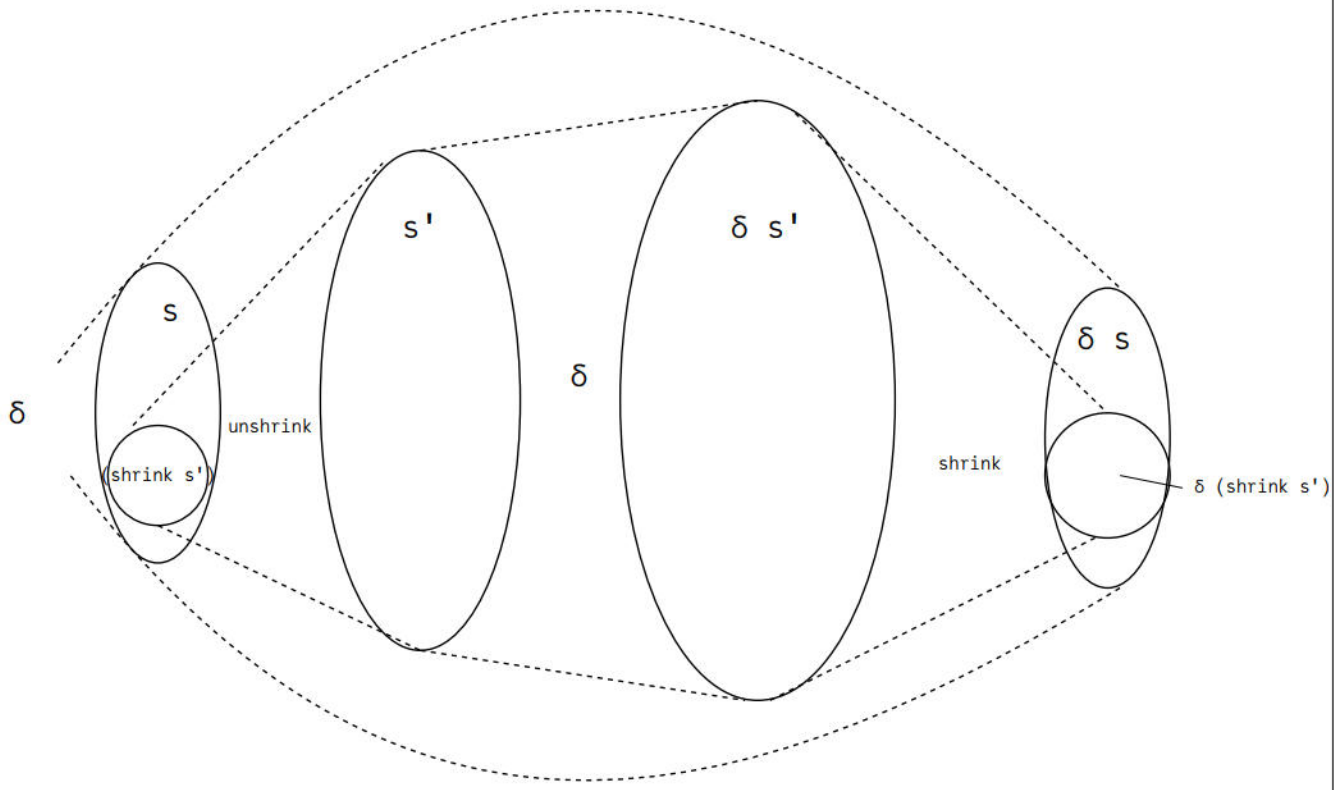
A branch is a sub-part of the overall state where a state transition on the overall state also means a state transition by the same `δ` on the subpart that behaves the same as if `δ` was applied on the subpart. This is what the `δ-similar` condition is for: `δ` propagating the the subpart (retrieved by first unshrinking and then shrinking back so that we get the value of the branch without everything else) should behave just the same as `δ` applied only on the subpart (retrieved via `unshrink`).

Branches Working Principle



The working principle of branches can be visualised by having a sub-state `(shrink s)` in the overall state `s`, where propagating `s` results in the same propagation to also happen in `shrink s`.

Delta Similar Condition



The `δ''-similarity` condition expresses that when propagating the overall state, the shrinked substate `s'` (branch) is propagated as well. If we unshrink the small subsection, propagate it and shrink it back into place, we get the same result as propagating the whole state with only the information from `shrink s'` cut out (proven shortly in

Theorem 5.7:8). Due to δ preserving the order relation and through `shrink-unshrink-revdir` stating that the substate is actually contained in the original state, we even get that the propagation of the branch happens inside the entire propagation of δ. We will prove this later in Theorem 5.7:8.

If we cut out `s` to only contain `shrink s'` and apply delta on it, we get the same computation as running δ on `s'` and then unshrinking it, as we can prove by the following lemma:

Lemma 5.7:4

Given a branch from Definition 5.7:3, it holds that

```
δ-similar' : δ ≡ unshrink ∘ δ ∘ shrink
```

Proof 5.7:5

We rename the `outfoinf-id` property to `injection-prop`.

```
δ                                        =< inject-prop >
δ ∘ unshrink ∘ shrink                    =< inject-prop >
unshrink ∘ shrink ∘ δ ∘ unshrink ∘ shrink =< δ-similar >
unshrink ∘ δ ∘ shrink ∘ unshrink ∘ shrink =< inject-prop >
unshrink ∘ δ ∘ shrink                    qed
```

Therefore, we truly apply δ additionally on a different state when we have a branch contained in δ. This means that if we create a function that really only computes the propagation of the branch, we again get an order preserving increasing function that can be parallelised with other solvers

Lemma 5.7:6

Given a branch on the semilattice S, then the following function:

```
δsub : S → S
δsub s = s <> (shrink ∘ δ ∘ unshrink) s
```

is also an order preserving increasing function

Proof 5.7:7

First, we have to prove that the function stays increasing,
so

```
s P (δsub s)
```

We can simply prove this as

```
s <> s <> shrink (δ (unshrink s))   =< associative >
(s <> s) <> shrink (δ (unshrink s)) =< idempotent >
s <> shrink (δ (unshrink s))        qed
```

Next, we have to show that it preserves the lattice order,
so

```
∀ {s} {s'} → s P s' → (δsub s) P (δsub s')
```

To prove this, we use the `δ-similar` condition to bring `subδ` into a more convenient shape.

```
s <> (shrink ∘ δ ∘ unshrink) s = s <> (δ ∘ shrink ∘ unshrink) s
```

As all of δ, shrink and unshrink preserve the order relation, we can chain those order preservations and add s with Lemma 5.3:39 to prove the statement.

With the above lemma, we get the information increase that only concerns the given branch as an order preserving increasing function. Therefore, when parallelising solvers, we can choose whether we want to fully encorporate a branch on our current state or keep running only the branch as a branch on a smaller part of the state. This will be useful for future constructions when actually running branches in parallel. We can show that the computation of the subbranch is indeed contain in the overall computation (which is the useful statement for this thesis):

Theorem 5.7:8

Given the order preserving increasing function from Lemma 5.7:6, all computation on the branch is contained in the overall computation, so

```
δ-contained : (δ_sub s) P (δ s)
```

Proof 5.7:9

```
(s <> shrink (δ (unshrink s))) <> δ s =< δ-similar >
(s <> δ (shrink (unshrink s))) <> δ s =< commutative >
(δ (shrink (unshrink s)) <> s) <> δ s =< associative >
δ (shrink (unshrink s)) <> s <> δ s   =< δ-dir >
δ (shrink (unshrink s)) <> δ s        =< δ-pres-P (shrink-unshrink-revdir) >
δ s                                   qed
```

5.7.2 On Constructing Branches

In Chapter 3 Section 3.9 we used semilattice injections to create new variables for a lattice based computation. The same mechanism would be applied to create new branches. As a branch would also return the result of a solver (through a semilattice bi-threshold variable), creating a new branch is literally

like creating a new variable, only that it has to be made sure that the overall computation continues to run on the lattice within that variable. Sadly, due to the time, money and food constraints on this thesis, we will leaving making this construction concrete to future work.

5.7.3 On Search Strategies (Branch Combinators)

A search strategy would be expressed as using two branches for a choice and then combining these two branches to choose the correct value. We have discussed in Section 5.2.1 how to possibly formally verify which choice is the correct one, but unless we have finished the results from Chapter 4 Section 4.7 we cannot make the optimal choice combinator concrete, and we are not going to reiterate old uninformed and well known search techniques, but they would look similar to the showcase of Section 5.5. In an old version of this thesis we also expreimented on the so-called *andorra principle* [52], where, from a choice of solvers, if all but one solver fail we use its results. This technique makes the solver give a value for as long as it is unique, which is great for parallelism because then different subgoals can work together and merge information on the main state as soon as they are absolutely certain that the information does not change. However, as soon as the choice is not unique anymore, we still need the formalism from Section 5.2.1 to find the best possible solution candidate which, again, for optimal solvers, requires a universe strong enough to extress the type of type theory, so we will have to complete the results of Chapter 4 Section 4.7 first to create a truly usable search strategy that is better than the status quo.

5.8 Clause Learning

Code-Tag: `LatticeTheory`

Our solving formalism relies on a state that increases until we can find a certain result. Of course, if we already had knowledge on where this state would increase to we could take a shortcut. Due the deterministic time hierarchy [12] we know that in general, we cannot have this information on where the computation will go before we have actually done the computational steps, however, that of course only holds true if we do a computation only once. In search, especially when heavily using branches, we recursively call the solver on all kinds of problems to solve. Therefore, in order to avoid computing a result twice between branches, we can cache the state change a branch creates and use it whenever we find ourselves in the same initial state as the branch once was in. This is especially important when the results of the solver depends on the co-search-problem, meaning that it depends on a range of branches to *not* having a solution, which is often times even more expensive and can lead to huge performance increases when caching the computation state that caused the conflict, which is usually done via clause learning [17].

In Section 5.7 we talked about how we can have multiple recursive calls on different inputs be contained in the current state. This means that the branches

constantly create information on where the state change is going to go. In this section, we will create a mechanism of how the state change of one branch can speed up the state change in another.

We begin by defining a complexity measure. We use an upper bound on when a certain threshold has been reached by the state, which would be the latest state that for example a bi-threshold variable would give a value if that value is given on that threshold.

Definition 5.8:1: **Reaching a Threshold**

Given an order preserving increasing function δ, a threshold st and an initial state s and natural number n, we say that δ **has reached threshold st from s at n** iff `st P (δⁿ) s`. To make proofs easier, we express it as the predicate

```
_reaches-threshold_from_at_ : OPIFunc S sl → S → S → ℕ → Set
δ reaches-threshold st from s at 0 = st P s
δ reaches-threshold st from s at (1+ n) = opi reaches-threshold st from (δ s) at n
```

(Note how in the actual Agda code, δ still has to be unwrapped to get the function without the laws)

The reason we have to use upper bounds is because we often don't know whether there is additional information introduced to the state that would speed up the propagation, as it might exist as shown in Theorem 5.3:24 (or, at least, the theorem does not rule out the possibility and there can be simple examples found where adding information actually speed up computation).

We can start by proving that reaching thresholds is transitive.

Lemma 5.8:2

If an order preserving function δ reaches s from s' in n' steps and reaches st from s in n steps, then it reaches st from s' in n + n' steps

```
reaches-transitivity : ∀ {s s' st n n'} →
    opi reaches-threshold st from s at n →
    opi reaches-threshold s from s' at n' →
    opi reaches-threshold st from s' at (n + n')
```

Proof 5.8:3

We prove this statement by induction over `n` and `n'`.
In the case of `n = 0` and `n' = 0`, we just have to prove that `st P s'` from `st P s` and `s P s'`, which holds due to transitivity.
In the case of `n = 1+ n` and `n' = 0`, we have to prove that `δ` reaches st from `delts s'` at `n + 0`, as we have made one computational step on the left. From `n' = 0`, we have the information available that `s P s'` and due to `δ` preserving `_P_`, we can lift this for the recursive to `(δ s) P (δ s')`.
In the case of `n = 0` and `n' = 1+ n'`, we have the opposite case, but here the recursion just computes without any changes to the premises.
In the case of `n = 1+ n` and `n' = 1+ n'`, we have to transform the second premise according to the following lemma:

Lemma 5.8:4

```
reaches-next-threshold :
    opi reaches-threshold st from s at n →
    opi reaches-threshold (δ st) from s at (1+ n)
```

Proof 5.8:5

By recursion over `n`, using the fact that `δ` preserves `_P_` at `n = 0`.

To apply the recursion with the second premise transformed we only have to do a coercion to make the natural numbers align.

The transitivity gives the obvious upper bound of just running both computations after one another. In general, this is not a lower bound though because the first computation could add information that makes the second computation run faster, it is however always the worst case scenario due to undecidability and the deterministic time hierarchy.

We will now create an order preserving increasing function that can be used to lower the upper bound if additional information is available from earlier computation. It is going to be a function that waits on a certain threshold in order to create new information.

Lemma 5.8:6

Given that the semilattice pre-order over S named _P_ is a decidable predicate with the function

```
_hasReached?_ : ∀ s st → Dec (st P s)
```

determining whether a threshold st has been reached on s, then the function

```
δ[s⊢st] : S → S
δ[s⊢st] s' with s' hasReached? s
... | yes s'Ps = s' <> st
... | no ¬s'Ps = s'
```

forms an order preserving increasing function. This function is called a **semilattice clause**.

Proof 5.8:7

From _P_ begin directional (stating that `s P (s <> x)`) and idempotence, we can derive that this function is increasing. In order to show that is preserves `sPs' : s P s'`, we have to case split on whether the threshold has been reached at s and s'. If neither threshold has been reached, no information has been created and the statement holds from just sPs'. If s reaches the threshold but s' does not we have a constradition because once a threshold has been reached, due to _P_ being a pre-order, it cannot be unreached again. If s has not reached the threshold but s has, the statement follows from the transitivity of sPs' with directionality, as st was added to the state after s'. If both have reached the threshold, the statement follows from the small lemma

Lemma 5.8:8

addP : forall x y : S -> (z : S) -> x P y -> (x <> z) P (y <> z)

Proof 5.8:9

by using Lemma 5.3:39 on reflexivity

by adding `st` to `sPs'`

These semilattice clauses are the generalisation of what a clause is in SAT, especially in the context of clause learning. A clause just waits for a certain threshold in the assignment (for example a subset of variables being assigned) to then give some additional information (like that there has been a conflict reached). The only minor differences to the clauses learned in SAT is that a learned SAT clause can do several different deductions on different assignments, but that can be modeled by just placing several clauses. Through our lattice generalisation though, we are not bound to just deducing Boolean or similar assignments. We can deduce any information gathered through the search process and just have a general way to cache the recursive call of our solvers.

We will now prove that these clauses do in fact give a possible speedup once they exist. This theorem is fairly obvious and does not prove in general that clause learning works, but from Theorem 5.3:24 we know that it is likely that there is a way to make clauses speed up computation by avoiding computation to be done twice.

Theorem 5.8:10

If an order preserving function δ reaches `s` from `s'` in `n'` steps, then the merge of δ with the clause $\delta_{s \vdash st}$ reaches `st` from `s'` in `1 + n'` steps

```
add-clause-prop : ∀ {s s' st n} →
    δ reaches-threshold s from s' at n →
    (mergeOPI δ δ_{s⊢st}) reaches-threshold st from s' at (1+ n)
```

Proof 5.8:11

We make this proof by induction over `n`. In the case of `n = 0`, we know from the premise that we have reached s, so the clause fires and produces the `st` value in the next step. We now have to show that `st P (δ_sl-st s')`. As the merge reintroduces a few state values, we prove this order as

```
st <> δ s' <> s' <> st   =< associative >
(st <> δ s') <> s' <> st =< commutative >
(δ s' <> st) <> s' <> st =< associative >
δ s' <> st <> s' <> st   =< associative >
δ s' <> (st <> s') <> st =< commutative >
δ s' <> (s' <> st) <> st =< associative >
δ s' <> s' <> st <> st   =< idempotent >
δ s' <> s' <> st         qed
```

The reintroduction of previous state values is a common theme in this proof, so it is actually recommended to use a solver for associative, commutative structures to keep the proof small (though we did not use that here).
In the recursive case of `n = 1+ n`, we use a small lemma to apply the recursive proof.

Lemma 5.8:12

```
reaches-higher-threshold :
    s P s' →
    opi reaches-threshold st from s at n →
    opi reaches-threshold st from s' at n
```

Proof 5.8:13

By induction over `n`, using transitivity at `n = 0`.

This lemma says that if we reach a threshold `st` from `s`, we also reach it from any larger `s'`. To apply this lemma, we case split of

whether `s'`, `δ s'` and or `δ (δ s')` have reached `s`, which determines whether the clause has fired at which point during the computation. We then use the lemma to apply to the recursive case, but we still have to prove a range of identities to show that the state of the recursion is lower than the desired initial state. All of those identities follow from basic semilattice laws and are nothing but tedious, so we will not show all of those proofs here. We will however make a list of necessary identities:

```
eq1 : (δ (δ s') <> δ s' <> st) <> δ (δ s' <> s' <> st) <> (δ s' <> s' <> st) <> st
    ≡ δ (δ s' <> s' <> st) <> (δ s' <> s' <> st) <> st

eq2 : (δ (δ s') <> δ s' <> st) <> δ (δ s' <> s' <> st) <> (δ s' <> s' <> st)
    ≡ δ (δ s' <> s' <> st) <> (δ s' <> s' <> st)

eq3 : (δ (δ s') <> δ s') <> δ (δ s' <> s' <> st) <> (δ s' <> s' <> st) <> st
    ≡ δ (δ s' <> s' <> st) <> (δ s' <> s' <> st) <> st

eq4 : (δ (δ s') <> δ s') <> δ (δ s' <> s' <> st) <> (δ s' <> s' <> st)
    ≡ δ (δ s' <> s' <> st) <> (δ s' <> s' <> st)

eq5 : (δ (δ s') <> δ s' <> st) <> δ (δ s' <> s') <> (δ s' <> s') <> st
    ≡ δ (δ s' <> s') <> (δ s' <> s') <> st

eq7 : (δ (δ s') <> δ s') <> δ (δ s' <> s') <> (δ s' <> s') <> st
    ≡ δ (δ s' <> s') <> (δ s' <> s') <> st

eq8 : (δ (δ s') <> δ s') <> δ (δ s' <> s') <> (δ s' <> s')
    ≡ δ (δ s' <> s') <> (δ s' <> s')
```

Notice how there is no `eq6`. The identity required here actually cannot be deduced because `st` is not contained often enough in the equation to be discarded using idempotence, however, this is in a case that is impossible because the threshold has not been reached after a state that the threshold has been reached at. In all other cases, the necessary equalities for the desired order relation can be deduced.

With this theorem, we know that all branches can be connected via clauses, for as long as they are obtained via actually performed computation (otherwise the clause would deduce a value that would not have been deduced). Every branch can be merged with clauses created from computation of other branches

to allow them to shortcut their computation (that this is correct and does not produce any unwanted deductions can be shown with Lemma 5.8:2). Because no matter where a different branch starts from, if it is merged with a clause created from computation on another state and it reaches this other initial state, it can make use of the clause (and in all other cases just stays the same, but we leave proving that to the enthusiastic reader). This build the foundation of clause learning, which we will create an entire automatic construction for in future work.

5.8.1 On why this is Clause Learning

We remember from Chapter 2 Section 2.7.3 that SAT-solvers speed up by remembering on which partial assignments we found a conflict, which is often an expensive exhaustive proof showing that a subset of variables has no satisfying assignment. We then remember to avoid this partial assignment in the future (meaning, other branches in the search). Our clauses can do the same (once automatically created). The result of a solver may depend on a lot of branches not having a solution, similar to showing that there is no satisfying assignment of a set of variables that have, potentially, exponentially many assignments. If a deduction is made as a result of all of those branches failing, we also know all of thos branches have been created from our initial state on. Therefore, we can cache the computation result so that if another branch is on our current initial state will immediately fail (or make any of the deductions we did after all of the failing branches have failed). This means that other branches do not have to redo the expensive proof all over again. Our concept is more general however, because we allow to cache any (hopefully expensive) computation, not just the one from expensive failure. What is important however to maximise the change of a clause being used is that we can minimise the threshold deeded in order to create our result.

5.8.2 On Creating Smaller Clauses

Of course, just creating clauses from computation where we reached a threshold `st` from some initial state `s` is not necessarily efficient, because if we always memorise only the exact recursive call of a solver and the exact final state, we might not maximise our chances of ever using that clause. Ideally, we want clauses on smaller states where the threshold is hit more often. We conjecture that there is a construction to create clauses for smaller initial states even if the computation was done on a bigger state if the state transition function δ has an additional, computable function

```
δ-low : ∀ (st s : S) →
    st P δ s →
    ∃[ s' of S ] ((s' P s) and (st P δ s'))
```

This function gives us, for any given threshold that is actually reached in one propagation step on state `s`, a smaller (or equal, but we will not repeat that) state `s'` that still reaches the threshold. We will use this to find smaller states that result in the same faster information increase that we want for the clause learning. Ideally, `s'` is minimal with respect to the pre-order `P`, but we skip modelling that to streamline this explanation. Further, the smallest input would not be unique as we already discussed in Chapter 2 Section 2.7.3, so which exact object is chosen for a smaller input is still up to the concrete implementation.

We could now recursively create smaller inputs not just for one propagation step, but several propagation steps after each other. For that, we need the following small lemma

Lemma 5.8:14

```
δⁿ-pres-P : ∀ {s s' n} → s P s' → (δⁿ s) P (δⁿ s')
```

Proof 5.8:15

By induction over `n` using the `δ-pres-P` property.

With the property that delta continues to preserve the pre-order even over multiple state transitions, we can recursively create a smaller input to reach a threshold after `n` transition steps as

```
δ-low-rec : ∀ {n} → (st s : S) →
    st P ((δ ^ n) s) →
    ∃[ s' of S ] ((s' P s) and (st P (δ ^ n) s'))
δ-low-rec {n = 0}    st s dPst = s , reflexive , dPst
δ-low-rec {n = 1+ n} st s dPst =
    let
        (s' ,  s'Pds , stPdns') = δ-low-rec {n = n} st (δ s) dPst
        (s'' , s''Ps , s'Pds'') = δ-low s' s s'Pds
    in s''  , s''Ps , transitive stPdns' (δ^n-pres-P {n = n} s'Pds'')
```

Here, in the recursive case, we first create a smaller reason for the `n` steps after the next state `δ s`, which result in a smaller input `s'`. Next, we create a smaller input `s'` that gets us from `s` to `s'` in one step of `δ`. We now went from the original input `s` hitting the threshold `st` after `1+ n` steps to hitting the same threshold `st` on a smaller state `s''`, which was created by giving a smaller initial state that `δ s` to hit the threshold `st` after n steps and then finding a smaller initial state to only hit that smaller next step `s'`. This way, we can recursively create smaller inputs that cause a certain threshold and use it to create clauses for clause learning.

There is an important connection to this basic theory and the variables we have created in Chapter 3 Section 3.8.3.1 that we will explore in future work and have already started to investigate in our paper [38]

Side Note 5.8:16

IMPORTANT: When expressing computation by the means of lattice based variables as is described in Chapter 3, we can get the smallest lattice part that was read in order to create a result by using variables that provide that information. If for every value a lattice bi-threshold variable produces during a read we can give the smallest lattice element it needed to turn into a value, (and the read of the variable was actually necessary), we know which parts of a lattice actually caused which writes to the lattice. Something similar has been done in [82] and we have also already used the idea in [38] to automatically produce the information needed to create clauses for clause learning in general type theory. Further, in Chapter 3 Sidenote 3.9:40 we already provided a concrete position where to use the variables in order to know which part of the lattice was relevant to cause a certain threshold.

5.9 Summary and Future Research

In this Chapter we have talked about an general theory of solving for type theory. We have described solvers as functions from a representation of a type into an object of the represented type, wrapped in a possibly failing context if no such object exists. The solutions were wrapped in a coinductive context to account for semidecidability, so naively, a solver would have the type

```
solve : (X : Set) → Perhaps X
```

and using a universe U with interpretation [_]U (so that it can actually be built), it has the type

```
solve (X : U) → Perhaps [ X ]U
```

In Section 5.2.1, we used the ideas from [84] to showcase how solvers can be used to create new (and improved) solvers

```
improveSolver : (solve : Solver) → Perhaps (∃[ solve' of Solver ] (measure solve < measure solve'))
improveSolver solve = solve (∃[ solve' of Solver ] (measure solve ≤ measure solve'))
```

To showcase how improving a solver could already improve the solving process while still in execution, we give a possible construction in Section 5.2.2 to exchange the algebra for the solver mid execution.

Another way to change the solvers behaviour is to change the state that it currently runs on. As the Perhaps value is always created by some sort of state transition, any solver would have a solving state. In order to get some properties on what actually happens to the solving state we model solvers that run on a (semi)lattice based state in Section 5.3.2.2 and argue in Section 5.4 that when using full lattices, this almost comes without loss of generality. When the solving state is an only ever increasing state according to a semilattice, we can show that it also induces a family of threshold functions in Theorem 5.3:21. Further, we can prove in Theorem 5.3:24, that increasing the information on a state can only equal or improve the number of recursive calls until a solution (equal to the solver's speed when ignoring memory overhead). This means the solver is

not getting slower, but possibly faster with increased information on its state when not accounting for memory overhead. We use this to create a construction for solvers running in parallel in Section 5.3.2.4 and prove that the construction results in a new solver in Theorem 5.3:37. Together with Theorem 5.3:24, this means that without accounting for memory overhead, solvers running in parallel are potentially faster (or just as fast) as each solver individually. We later also use the parallelisation to give a construction on how to add clauses to the solving process in Section 5.8, where we can prove in Theorem 5.8:10 that a clause added in parallel speeds up the search substantially when activated (though we should point out that this is *not* a proof that clause learning would give any exponential speedup as was done in [17], we just prove that, without accounting for memory overhead, clauses give a speedup when run in parallel to a solver).

Before (learned) clauses can be properly introduced, we present a concept to model branches on a semilattice based state in Section 5.7. We model those branches as parts of the current state that behave like the state transition function, meaning that the overall state contains substates that create the same state increasion as they would if they were the main state. We can show in Theorem 5.7:8 that these branches are propagated (meaning their state is increased according to the transition function) by the overall state transition function. We conjecture that these branches can be constructed with the same principle as new variables were introduced in Chapter 3 Section 3.9 for future research. We can also show that branches can interchange clauses, which we model as state transition functions that wait for a certain threshold in the state to create new information. The main idea is that we model clause learning as caching the computation of a solver, so if we visit (part of a) solving state twice, we can use the first time to remember which deductions have been made and use it as a clause in every other branch. Once those other branches reach the same state (or bigger), the cached computation triggers and increases their solving speed. The reason that this is clause learning is because we also cache computation that is the result of branching, so our clauses would prevent that branching in another state as also discussed in Section 5.8.1. Finally, in section 5.8.2 we discuss how smaller clauses can be generated if the state transition function were to give smaller explanations for given output thresholds.

For future research, we obviously have to start completely finishing the results from Chapter 5 Section 4.7 and build a `Perhaps` based solver for the type of type theory. To do this more easily, we would create a monad for the `Perhaps` values so that we can modularly create a solver. This solver would have a monad-plus instance that relies on branches, whose automatic construction we would implement using the concepts in Chapter 3 Section 3.9, together with a configurable branch combinator. With this solver for general type theory, we would fully implement the ideas of Section 5.2.1 of self improving solvers in the meta theory, already letting our slightly weaker solver for type theory aid in the construction of the proofs.

Further, we would use the ideas from Chapter 3 Section 5.8:16 to use the writes and reads from variables to gain more information on a smaller clause we could learn. Every time we read a value (and presumably case split on it),

we know that the part of the state that it read from was relevant to our current computation. The relevant part it read from the state was the (minimal) write value that caused the current assignment. When we write to possibly other parts of the state, we can store that this write was a result of the reads we made before, creating an implication graph. From this implication graph, we can pinpoint the part of our initial state that was responsible for a threshold passed.

Finally, we just note how future research would definitely make more use of category theory in order to modularly express solvers over a universe without the contextual overhead through monadic notation or universe interpretations. We already mentioned this in Chapter 1 Section 1.5.2. We strongly assume that every type in type theory has an own, simply explained way to be solved for, accounting to all the optimisations we talked about in this thesis, so in the grand scheme of things this research aims to find an executable categorical interpretation of a self improving solving system for type theory.

Appendix A

Towards Clause Learning à la Carte through VarMonads (Paper)

Informal Proceeding at 32nd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2022)

Paper presented at the 32nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2022), Tbilisi, Georgia, and Virtual, September 22-23, 2022 (arXiv:2208.04235).

Towards Clause Learning à la Carte through VarMonads

Arved Friedemann[1][0000−0001−7252−264X] and
Oliver Keszocze[2][0000−0003−2033−6153]

[1] Swansea University, UK
a.r.h.friedemann@swansea.ac.uk
[2] Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany
oliver.keszoecze@fau.de

Abstract. More and more languages have a need for constraint solving capabilities for features like error detection or automatic code generation. Imagine a dependently typed language that can immediately implement a program as soon as its type is given. In SAT-solving, there have been several techniques to speed up a search process for satisfying assignments to variables that could be used for program synthesis. One of these techniques is clause learning where, if a search branch runs into a conflict, the cause of the conflict is analysed and used to create a new clause that lets a branch fail earlier if the conflict arises again. We provide a framework with which this technique can come for free not just for Boolean solvers, but for any constraint solver running on recursive algebraic data types. We achieve this by tracking the read operations that happen before a variable is assigned and use this information to create the dependency graph needed for conflict analysis.
Our results are implemented in Agda for best readability, but they transfer to other functional languages as well. For brevity, we do not provide an entire search system utilizing the clause learning, but it will become clear from the formalisms that our technique indeed enables a clause learning search system to be built.

Keywords: Clause Learning · Meta Theory a la Carte · Dependency Analysis

1 Introduction

Search is probably the most reoccurring problem in computer science, since any problem can be formulated as a search problem from the input to the desired output. One of the most general search problems is program synthesis, where we search for a general recursive program that fits a set of constraints or that has a certain type. In fact, in type theory, proof search and program synthesis are the same thing [6,19]. If we could create a general search procedure for program synthesis, we would have a widely reusable tool for problem solving in computer science. To move towards a general solution to program synthesis we provide a type theoretical framework that makes search techniques from search or solving

engines available to general programs. Specifically, we provide the basis to allow for clause learning in any search that can be described by a general recursive program.

The high performance solving engines available to us thus far [21] are for primitive languages like Boolean satisfiability (SAT) [4] or Sat-Modulo-Theories (SMT) [3]. The technique they use that is relevant for this paper is the Conflict-Driven-Clause-Learning, short clause learning, where an analysis of what caused a conflict is used to speed up the search process.

There have been functional approaches to solving, especially in the constraint solving field. A center piece of this development is [15], where a highly adaptable framework for monadic constraint solving is implemented. This framework supports a version of constraint propagation (comparable to unit propagation used in SAT and SMT solving) and exchangeable search strategies through search nodes as first class objects. It features a variety of (mostly uninformed) search strategies such as (bounded) depth-first- or (bounded) breadth-first-search, but no clause learning. Because the approach in [15] is so highly adaptable, we will not present a complete solving system in this paper, but rather only the center piece that is needed for the clause learning.

The approach in [15] has been extended to allow for an easier implementation of heuristics [16], but clause learning is still left to SAT and SMT search engines [18,2,9]. These constraint satisfaction engines rely on variables that are filled with values to construct a model or to check consistency of constraints, which is the motivation for our paper. We use the fact that variables are the key component to make constraints communicate to implement a version of clause learning. We also call this variable-based constraint solving, which comes with a lot more benefits that will be compatible with our formalism. In [12,13,14], a system to make propagation via variables inherently concurrent was developed, so adding our principle to it even yields a possibility to do clause learning in parallel. What we add in this paper is a means to use the variables in any constraint solver to get clause learning for free.

Contributions

The main contribution of this paper is to create a structure for a free dependency graph for the evaluation of functions on recursive data types. In order to achieve this and to prove some correctness criteria like termination of the results, we implement the approach in Agda and provide an application of the Meta Theory à la Carte [7] approach to datatypes that are distributed over memory via pointers.

2 Preliminaries

2.1 Solving and Clause Learning

We give a short summary of how solving engines work for SAT [4] and hint at where these ideas generalise beyond the realm of Boolean satisfiability.

In SAT, we are given a Boolean formula in Conjunctive Normal Form (CNF). Given a set of variables VAR, let $C \subset \mathfrak{P}(\pm VAR)$, where both C and $c \in C$ are finite. An element $pc_i \in c$ is a variable $c_i \in VAR$ with a polarity $p \in \{-, +\}$ that indicates whether the variable in that clause is negated or not. The CNF has the form:

$$\bigwedge_{c \in C} \bigvee_{pc_i \in c} pc_i$$

The search is for a satisfying assignment $\delta : VAR \to \mathbb{B}$.

The central solving part comes from the DPLL procedure [5] and is called the unit propagation. The idea is that if there is an unfullfilled clause with a single unsassigned literal c_i, the value of c_i is assigned *true* or *false* depending on its polarity, so that the clause evaluates to *true*. This assignment can cause other clauses to have only a single unassigned variable and the process continues. For each variable, we can memorise the reason why it was assigned. This reason is, that within the clause that triggered the assignment of the variable, all other variables have been assigned to the 'wrong' values. The representation of the reason is the list of variables and their assignments in the direct clause that led to the new assignment. This information is called the *dependency graph*. If a conflict arises, meaning that a variable has been assigned conflicting values, it can be tracked what caused the conflict. This information is used for two things. First, it can deduce the earliest decision that led to the conflict, so the backtracking can go further back than it usually would, and second it is used to create a new clause that propagates the conflict earlier if it arises again in a different branch of the search. This last part is what's referred to as *clause learning*.

When we generalise this idea, we don't need variables to only contain Boolean values, but they can contain any value that can be compared by equality (or values that can be merged until the merge causes a conflict, as in [12,13,14]). The reason for an assignment is just the list of variables and their values that have been read until a new assignment was made. Obtaining this is what we refer to as *dependency tracking*. A learned clause in the general setting is just a program that checks for the learned variable assignments and produces a conflict if necessary.

2.2 Agda and Dependent Type Theory

The formalism for this paper uses literal Agda code. Agda is a functional language just like Haskell, but with a few additional features. In this paper, most of the difference are only of cosmetic nature, like mixfix operators (e.g. `if_then_else_`), but we use three important concepts of dependent type theory.

The first concept is, that the type of a function can depend on its arguments. We mainly use this to give types as arguments to a function, e.g. written as `(A : Set) -> List A`, which is the type of a function taking a type and returning in this case a List that is parametrised by that type (note that

Agda's type for types is called `Set`). If an argument to a function should be clear from the result, we write `{A : Set} -> List A`, where `A` is deduced based on how the function is used and `A` does not need to be provided explicitly, or `{{Eq A}} -> List A -> Bool` when the element of the type class `Eq A` should be uniquely deducible by search. These implicit arguments from the signature can, down in the function definition, be brought into scope by writing e.g. `{A = B}`, which takes the implicit variable `A` from the function signature and renames it to `B` (where quite often we don't want to rename the variable but just bring it into scope, so we just write `{A = A}`).

The second concept is, that we can apply functions on types within the type signature. This is the generalised version of a (dependent) type alias. So we can write `Cumbersome A B = A -x- A -x- B -x- B` (Where `-x-` is our ascii tuple type)

The third important concept is that of a sigma type. The type `Sigma A P` is a generalised version of a tuple where the first entry is an element `a` of `A` and the second entry is an element of `P a`, so `P` yields the type of the second entry. Imagine we have a pointer type `V`, where `V A` denotes a pointer that contains a value of `A`. Then, the tuple that contains a pointer and its value has the type `Sigma A V`. If we don't know the type of the pointer, we can have a sigma type that takes the type of the pointer as the first entry to the tuple, such as `Sigma Set (\ A -> A -x- V A)`. A possible object of this tuple type could look like `(Bool, true, ptr 23)`.

We ignore one safety feature of Agda called universe-polymorphism. The idea is that `Set` itself needs a type, but if it were `Set : Set` then contradictions can happen. Agda solves this by writing `Set l : Set (suc l)`, but this results in additional boilerplate that we omit here for brevity.

Type classes in Agda are implemented via modules. Especially records (that can be automatically searched for by putting them into `{{...}}` brackets) serve as modules that can be opened with `open <record name> <record object>` to put their functions into the current scope.

A few mentions of types and standard functions we are using. The string `-x-` was already mentioned to be the tuple type, with `_,_` being its constructor. `T` is the unit type with `tt` being its only, argumentless constructor. `_$_` is the usual function application operator with lowest binding strength and `_o_` is function composition.

3 Motivating Example

Let us look at the simple function

```
any : List Bool -> Bool
any = foldr (_||_) false
```

that computes whether at least one value in the list is true. When evaluating this function, we get an answer like

```
any [ false , true , false ] = true
```

We would now like to know why the function returned `true`. The reason is the part of the input that caused `any` to evaluate to its value. This could look like

```
any [ false , true , false ] = true because
        any [ false , true , ... ] = true
```

On this specific example, this is the smallest reason for the result. If the first value had been `true`, the reason would have been shorter. The second value was the final reason for the result being `true`. For search problems, these reasons are essential to guide the search process. Let us imagine a function

```
sudoku : (sfield : Mat 9 x 9 Nat) -> Bool
```

which checks whether a given Sudoku-field filled with numbers is valid. To solve a sudoku, we might have to check several fields whether they are valid. Let's assume that we have a reason that

```
sudoku sfield = false because
        sfield at (2 , 3) = 5
        sfield at (5 , 3) = 5
```

then, we no longer need to test fields that have a 5 at position (2,3) and (5,3), ruling out plenty of fields that no longer need to be branched on. We could retrieve these reasons by checking an entire (partial) field, but then returning only the portion of the fields that actually causes the field to be invalid, the same way as we did with the list in the first example. This is the idea of clause learning, generalised to a functional context.

We mention here that this clause learning can become quite complex. The above example only provides the clause to a very direct conflict, but the most interesting conflicts would occur if there was one variable that cannot be assigned anything because of a sub-assignment. Imagine an assignment with

$$\forall x, x' \in \{1, ..., 8\}.\ \texttt{sfield at (x , 9)} \neq \texttt{sfield at (x' , 9)}$$
$$\forall y, y' \in \{2, ..., 9\}.\ \texttt{sfield at (9 , y)} \neq \texttt{sfield at (9 , y')}$$

Then the field (9,9) could not be assigned because it always violates one of the row or column constraints. If the search algorithm runs within the tracking of read variables, the above reason would be marked as the reason why the solver failed to assign (9,9) and therefore be giving an interesting clause.

Achieving this directly as shown is not possible in (pure) functional languages. This is because of the principle that the result of a function cannot depend on its evaluation. If a function could tell which values it read during evaluation, its results could change depending on how much of a value has been evaluated. There is however a way to achieve something similar in a functional language. Let us rewrite the any-function as follows

```
any lst = case lst of
        [] -> false
        (x :: xs) -> case x of
                        true -> true
                        false -> any xs
```

Here we can see a bit more precisely how much of the value we actually read. Each case statement looks for the uppermost constructor to adjust its behavior. Assuming the code was written with as little case distinctions as necessary, this means that in each branch of the case-statement we know on which part of the input we currently depend on. Therefore, if our environment keeps track on which values we read on each case-statement, it is easy to acquire the dependency graph needed for clause learning.

4 VarMonads

We will now define a type class that we can use to change the behaviour of the program depending on the variables that it read or wrote to, to add features like the creation of a dependency graph (this even allows for adding further features, such as instant parallelism or variable sharing...). This approach is inspired by a Haskell-Package called "monad-var" [1] that offers type classes for pointer-like structures.

In this paper we lift the monad-var approach to allow for simple clause learning and will discuss how it can be used to get even more pointer features for free.

We start off with a type class that imitates Haskell's atomic IORef pointer model. In some monad M with a pointer type V, we have three instructions:

```
record BaseVarMonad M V where
  field
        new : A -> M (V A)
        get : V A -> M A
        write : V A -> A -> M T
```

`new a` creates a new pointer with the value `a`, `get p` returns the current value of the pointer `p` and `write p v` writes the given value into the pointer. This creates a basic structure for a VarMonad.

The semantics of these VarMonads are generally as expected of a pointer interface, but the details can change based on the underlying monad. In [12,13,14] e.g., the information content in a pointer can only ever increase, which is actually an important constraint needed for solving that we will handle later. We will use the VarMonads to track the `get` operations of a program to retrieve the reasons for assignments. Before we go into constructions on VarMonads, we will show how datatypes can be encoded in a way that can be handeled well in this pointer setting.

5 Data Types and Meta Theory à la Carte

Data types à la Carte (DTC) [17] is an approach for having composable data structures. It was first introduced for Haskell and later ported to functional proof assistants like Coq and Agda [7,8,11], now called Meta Theory à la Carte (MTC) [7]. The key idea is the following:

A data type is represented by a functor that takes the type of its "recursive call". E.g., if we look at the following definition of a list:

```
data ListF (A : Set) : (B : Set) -> Set where
  nil : ListF A B
  lcons : A -> B -> ListF A B
```

We can see that at the position where there would usually be the recursive call to `ListF A`, there is a `B` instead. This way, the recursive call can be exchanged to e.g. be a pointer to the sublist, instead of the sublist itself. This approach can be used for other features like data type composability, but here we only need the possibility to switch the recursive call to being a pointer. We now first explore how to use the above functor to create a list in the classical sense, and how to define a function over it.

If we want to have a list in the classical sense we need to have an operator that gives this list-functor itself as an argument to the recursive call. Naively, this type would be `ListF A (ListF A (ListF A (...)))`. In Haskell this is achieved with the following operator

```
data Fix f = In f (Fix f)
```

The type of A-lists can be obtained as `Fix (ListF A)`, which is constructed using the `In` constructor together with the list functor constructors that take another `Fix (ListF A)` as the recursive argument. The problem is that this `Fix` operator could potentially be used to express infinite data structures and is not allowed in languages like Agda or Coq. To fix this, a different, non-recursive operator is used that creates a church-encoding of the data type, effectively freezing the constructors with the function that reduces them during a fold (also called an "algebra"). These are the operators used in Meta Theory à la Carte (MTC) [7] (and have later been refined [10,11], but we will stick with the original definitions here for simplicity). In this paper, we will use both approaches, but will for now focus most on the MTC approach. Using the approach provides additional guarantees like termination of our constructions. It also makes it possible for future research to have the correctness of the constructions verified. The first Idea of MTC looks as follows:

```
Algebra : (F : Set -> Set) -> (A : Set) -> Set
Algebra F A = F A -> A

Fix : (F : Set -> Set) -> Set
Fix F = forall {A} -> Algebra F A -> A
```

```
foldF : Algebra F A -> Fix F -> A
foldF alg fa = fa alg
```

The fixpoint of a functor is now a function that takes an arbitrary algebra to reduce the value to some type `A`. This freezes the constructors of that type until the function evaluating it is given. If we now construct the fixpoint of one of our data-functors, we can fold over it via `foldF`. The usage of this is best shown by the example of how to create the any-function for our list-functor:

```
anyFL : Fix (ListF Bool) -> Bool
anyFL = foldF \ {
  nil -> false;
  (lcons x xs) -> x || xs}
```

The important part for our VarMonads is now: Instead of the direct recursive call, we can put a pointer to the recursive substructure. That way, even arbitrarily large data structures are inherently distributed over several variables (for each of which we can do the dependency tracking separately). With a structure distributed over a state with pointers, a naive implementation of the any-function could look like

```
anyVM : {{bvm : BaseVarMonad M V}} -> Fix (ListF Bool o V) -> M Bool
anyVM = foldF \ {
    nil -> return false;
    (lcons x xs) -> (x ||_) <$> (join $ get xs)}
```

If we would directly apply the clause learning to this however, this function would now always read the entire list and there would be no information on which parts had been important. To solve this issue, we use the Mendler-Style algebras as proposed in [7,20]. These algebras make explicit when the recusive value is used.

```
Algebra : (F : Set -> Set) -> (A : Set) -> Set
Algebra F A = forall R -> ([[_]] : R -> A) -> F R -> A
```

This is the algebra that we will use for the fixpoints in the context of VarMonads. The way that this works is that the functor `F` contains a value of an unknown type `R` that can only be extracted when the extra function given is applied (the brackets `[[ ]]`). In practise, this looks as follows. Let's say we want to write the any-function with the Mendler-style algebras. They look just like before, only that we wrap the `[[ ]]`-call around the result from the recursive call.

```
anyFL : Fix (ListF Bool) -> Bool
anyFL = foldF \ {
  _ [[_]] nil -> false;
  _ [[_]] (lcons x xs) -> x || [[ xs ]]}
```

This way, we can get information about whether the recursive call is used or not without having to rely on information from the lazyness. Our any-function now looks like:

```
anyDTC : {{bvm : BaseVarMonad M V}} -> Fix (ListF Bool o V) -> M Bool
anyDTC = foldF \ {
    _ [[_]] nil -> return false;
    _ [[_]] (lcons true xs) -> return true;
    _ [[_]] (lcons false xs) -> get xs >>= [[_]] }
```

The `[[_]]` function is only ever applied to the recursive call in the last case where we have to look at the rest of the list, so this function will only read exactly what is necessary to retrieve the information of the dependency graph and there are no side effects to the state when the recursive value is not used by the function.

We can now almost ignore the explicit pointers themselves completely by formulating a special fold

```
foldBVM :
  {{bvm : BaseVarMonad M V}} ->
  Algebra F (M A) -> Fix (F o V) -> M A
foldBVM {{bvm}} alg = foldF \ _ [[_]] -> alg _ (get >=> [[_]])
```

Here, we give an algebra that just acts on the original functor `F` whose recursive calls are fed with the result from the `get` action. The only downside is that the result `A` needs to be wrapped into the monadic context because retrieving the recursive value has a side effect within the monad. We can now write the any-function as

```
anyDTC : {{bvm : BaseVarMonad M V}} -> Fix (ListF Bool o V) -> M Bool
anyDTC = foldBVM {F = ListF Bool} \ {
    _ [[_]] nil -> return false;
    _ [[_]] (lcons true xs) -> return true;
    _ [[_]] (lcons false xs) -> [[ xs ]] }
```

Where the `[[ ]]` brackets retrieve the result of the recursive call in a monadic context.

There is one more upside to using Mendler-style algebras. When constructing a fix-value, we no longer rely on a functor instance. Instead we can just write

```
In : F (Fix F) -> Fix F
In f A alg = alg _ (foldF alg) f
```

However, we cannot get rid of the functor instance when extracting values. This looks as

```
Ex : {{Functor F}} -> Fix F -> F (Fix F)
Ex = foldF \ _ [[_]] f -> In o [[_]] <$> f
```

So, when we want to look at the uppermost constructor (for a show instance or merging operations or something) we need a functor instance.

Pointers cannot have a functor instance for the simple fact they have to be writeable, which would mean that we could also write any value of the new type after the map. This cannot be done as not all functions used in a map are surjective. With the `Ex`-function however, we have a special case where the type nor value of the pointer are actually changed. As we can safely assume all `Fix`-values were created with `In`, the `Ex`-function recurses over `Fix`-values that have been extracted with `Ex` and are being packed back in using `In`, which is just like applying `id`.

Therefore, it is fine to create a read-only variant for pointers that has a functor instance for the `Ex`-function. We call our variant "Lensed Variables" (which is related to but not the same as general functional lenses).

A lensed variable has an original variable and type stored, but also additionally a function that can transform the original value into any other value. This way, two parts of the program can look at the same variable through different "lenses":

```
data LensPtr (V : Set -> Set) (A : Set) where
       constructor LP
  field
       origType : Set
       origPtr : V origType
       t : origType -> A
```

We can give such a Pointer a Functor instance as follows:

```
LensPtrFunctor : Functor (LensPtr V)
LensPtrFunctor = record { _<$>_ = \ f (LP T p t) -> LP T p (f o t)}
```

Where the pointer has an additional transformation to its output. Now we can use the normal MTC constructions (like `Ex`) that require functor instances with pointer structures as long as we only apply the `id` functor or only read from the pointer.

6 Variable Tracking

Now that we have lifted all functions into our VarMonad, we want to track the values that have been read. This can be achieved with a simple state transformer. Before that, we first settle on the type of data that will be tracked.

```
AsmCont : (C : Set -> Set) -> (V : Set -> Set) -> Set
AsmCont C V = C (Sigma Set \A -> (A -x- V A))
```

We define an assignment-container `AsmCont` to be a functor that contains tuple variables with the value they had when read. The `Sigma` type is a dependent

tuple, in this case meaning that we actually store a triple with first the type of the pointer value, then the value and then the pointer itself. We define a special VarMonad, a `TrackVarMonad` that has a BaseVarMonad `bvm` and an additional action `getCurrAssignments` to retrieve the currently read values.

```
record TrackVarMonad ...
    getCurrAssignments : M (AsmCont C V)
```

We can create an instance of this monad via the following construction where we add the read values to the state. Here we assume the containter `C` to have a `MonadPlus` instance, so that they can behave like sets with a merge and a singleton operation.

```
BaseVarMonad=>TrackVarMonad : {{mpc : MonadPlus C}} ->
  BaseVarMonad M V ->
  TrackVarMonad C (StateT (AsmCont C V) M) V
BaseVarMonad=>TrackVarMonad {C = C} bbvm = record {
    bvm = record {
      new = liftT o new ;
      get = \ {A = A} p -> do
        v <- liftT (get p)
        modifyS (_<|> return (A , v , p))
        return v
        ;
      write = \ p -> liftT o write p } ;
    getCurrAssignments = getS }
  where
    open BaseVarMonad bbvm
```

This construction is straightforward, as it accumulates the read values into the state. A welcome side effect is that now, through the alternative-instance of the state monad, parallelism or different programs reading different variables comes for free (though we do not use it in the scope of this paper).

7 Product VarMonads

Ultimately, we want to write the list of reads into every variable. We want to hide this extra information behind a VarMonad that can only retrieve the extra value, so we define a new specialised VarMonad

```
record SpecVarMonad ...
    get : V A -> M B
    write : V A -> B -> M T
```

This VarMonad has an extra `get` and `write` operation for the extra attached value of every pointer. A naïve implementation to create the tuple of a corresponding BaseVarMonad and SpecVarMonad pair would be a construction

```
naiveProd : BaseVarMonad M V -> (B : Set) ->
                    BaseVarMonad M (\ A -> V (A -x- B))
                 -x- SpecVarMonad M (\ A -> V (A -x- B))
```

This constructs a pair of VarMonads that only operate on pointers of tuples with the main value and the additional value. This however is not expressive enough. If we look at the type of the reasons from Section 6, we notice that the type B depends on the general variable type of the monad. This is similar to having a pointer that eventually points to itself, like a type `V (A -x- V (A -x-...))`. This means that we need a pointer type

```
FullAsmPtr M V C A =
        V (A -x- Fix (\ R -> AsmCont C (\ B -> V (B -x- R)) ) )
```

where the `AsmCont` container stores the tuple based variables. We generalise this type for our product construction

```
RecTupPtr M V F A = V ( A -x- Fix (\R -> F (\ B -> V (B -x- R) ) ) )
```

where we allow a general container `F` to take an arbitrary pointer type that is exchanged with the recursive tuple pointers. Our assignment-container pointer will later look like

```
AsmPtr M V C = RecTupPtr M V (AsmCont C)
```

We now formulate the product construction as

```
recProdVarMonad : BaseVarMonad M V ->
        {B : Set} -> {F : (Set -> Set) -> Set} ->
        {{func : Functor (\ R -> F (\B -> V (B -x- R)))}} ->
        (forall {V'} -> F V') ->
            BaseVarMonad M (RecTupPtr M V F)
        -x- SpecVarMonad M (RecTupPtr M V F) (F (RecTupPtr M V F))
recProdVarMonad bvm mpty = (record {
      new = new o (_, In mpty) ;
      get = (fst <$>_) o get ;
      write = \ p v -> snd <$> get p >>= \b -> write p (v , b) }
    ) , (record {
      get = \ p -> snd <$> get p >>= Ex ;
      write = \ p v -> fst <$> get p >>= \ a -> write p (a , In v) })
  where open BaseVarMonad bvm
```

First, we need the container `F` to be a Functor, at least for tuple based pointers, so that it can be deconstructed via `Ex`. Then we need it to have an empty element `mpty`, so that it can be created alongside a value without further information. The `BaseVarMonad` is now created in a straightforward manner, where the original value is just wrapped into the tuple and if one value of the tuple is updated, the other one is read and untouched.

We notice how, in this construction, we always override the old values with either the variables or the reasons. For a correctly working system, old values have to be preserved somehow (e.g. by using lattices for merging values), but we omit that detail here for brevity.

The `SpecVarMonad` now acts on the second element of the tuple in each pointer. It is being implemented analogously otherwise. We note that as soon as one constrains all values to be (bounded) lattices, the boilerplate for reading old values and constructing default values can be removed. This however makes the type signatures more complicated than currently necessary.

8 Dependency Graph

We now combine the previous two constructions to track the reasons for every assignment and writing it into the values themselves. We create a Clause-Learning VarMonad for the purpose. It, again, has a BaseVarMonad `bvm` and has a special operation `getReasons`

```
record CLVarMonad ...
    getReasons : V A -> M $ C (AsmCont C V)
```

This VarMonad has an extra operation to retrieve the reasons from a pointer. There can be several reasons, because the pointer might be assigned several times. The construction for this VarMonad looks as follows:

```
BaseVarMonad=>CLVarMonad : BaseVarMonad M V ->
  (forall {A} -> C A) ->
  {{func : Functor (\ R -> C $ AsmCont C (\B -> V (B -x- R)))}} ->
  {{mplus : MonadPlus C}} ->
  CLVarMonad (StateT (AsmCont C (AsmPtr M V C)) M) (AsmPtr M V C) C
BaseVarMonad=>CLVarMonad {M} {V = V} {C = C} bvm mpty = record {
    bvm = record {
      new = \ x -> new x >>= putAssignments ;
      get = get ;
      write = \ p v -> putAssignments p >> write p v };
    getReasons = getR }
  where
    vmtup = recProdVarMonad bvm
            {B = C $ AsmCont C (AsmPtr M V C)}
            {F = C o AsmCont C} mpty
    trackM = BaseVarMonad=>TrackVarMonad (fst vmtup)
    lspec = liftSpecVarMonad (snd vmtup)
    open BaseVarMonad bvm using (mon)
    open TrackVarMonad trackM
    open SpecVarMonad lspec renaming (get to getR; write to writeR)
    putAssignments : AsmPtr M V C A ->
            StateT (AsmCont C (AsmPtr M V C)) M (AsmPtr M V C A)
```

```
putAssignments p =
        getCurrAssignments >>= writeR p o return >> return p
```

The important parts are the constructions around the `bvm`. First, we apply the product construction to create pointers that hold the reasons as well. We give the type of pointer content explicitly to make it easier for Agda to figure out the types. Then, we apply the tracking to just the BaseVarMonad from the product, using the SpecVarMonad to retrieve the reasons from a pointer. The SpecVarMonad needs to be lifted to still work within the StateT transformer. We use the TackVarMonad as our main VarMonad and then just put the current remembered assignments into the modified pointers.

With this construction, we now get a VarMonad that can tell for every value why it has been assigned (if we only perform minimal reads).

9 Constrained Values

One last feature that should be mentioned is that, with a bit of extra code, the values used in the VarMonad can be constraint to all have certain properties. This can be useful to give all values additional features like a `Show`, `Eq` or even a `Lattice` instance. As the interface of the `BaseVarMonad` changes, the above constructions have to be redone, but we decided to give the raw definitions first to abstract from unnecessary detail. This is the definition of a constrained VarMonad, that takes an extra parameter `K` that serves as a predicate on the values within the pointers.

```
record ConstrVarMonad ...
    new : {{k : K A}} -> A -> M (V A)
    get : {{k : K A}} -> V A -> M A
    write : {{k : K A}} -> V A -> A -> M T
```

Let e.g. `K = Show`, and all values stored in variables can be converted into text. There can also be several constraints, e.g. by choosing

```
K = \ A -> Show A -x- Eq A -x- Lattice A
```

The constructions only change marginally, however, some constructions become simpler with certain constraints. If `K` implies a (bounded) lattice, the `write` function can be implemented as

```
write p v = get p >>= \v' -> write p (v /\ v')
```

And then the product construction does not need to handle retrieving the old value (or constructing a default value).

We even note that the lattice constraint is essential for the constructions to have the desired semantics. If a variable can be arbitrarily reassigned, any bookkeeping of its value gets invalidated. This does not happen when the value can only grow in information content.

In this paper, we are only using these constraints for show instances, but e.g. for the recursive construction of an overall reason (by traversing the dependency graph), all values need to have an `Eq` instance to ensure termination.

10 Running the Constructions

We run our constructions with a default implementation. The default BaseVarMonad runs on a state with a map, mapping integers to values. From this BaseVarMonad, we derive a CLVarMonad with our constructions and let the initial list example run. We adapted the list constructors in a way so that they directly create a list with a pointer at the recursive call.

```
anyTest : Bool
anyTest = runDefCLVarMonad $ do
  anyDTC =<< false :: true :: false :: []
```

Note how on the top level, we use the DTC implementation of `any` in order to only read necessary values.

For simplicity, we omit the `runDefCLVarMonad` boilerplate from now on. We can now extract the reasons as follows:

```
do
  res <- new =<< anyDTC =<< false :: true :: false :: []
  getReasons res
```

Together with the show constraints, we get the list of reasons

```
"(p3 = lcons false p2) ^ (p2 = lcons true p1)"
```

Where `p<i>` are the pointers in which the values are stored in. This is precisely the result that we wanted. As we have used the DTC approach, this now works for all functions on DTC data types.

11 Summary and Future Work

In this paper, we have created a mechanism to retrieve the reason as to why a value has been assigned. This approach works with all DTC and MTC data types and can be used in a solving system to implement clause learning. The underlying implementation was created using Agda, giving safety features like type safety and a guarantee of termination. Only checking consistency with universe polymorphism is left to future work. We lifted the MTC approach to work within our monadic structures and showed that all DTC (and MTC) data types and functions over them can be represented in our system. As our constructions use a state monad for the current assignment and dependency tracking, we get search features like branching for free. Therefore, for future work, our approach only needs to be combined with [15] to create a full solving system. Furthermore, it needs to be investigated how to combine the results with [12,13,14] so that the constraint solving and clause learning come for free even in a concurrent setting. For now, we have built the center piece for clause learning and we are one step closer to building a general purpose, functional solving system.

References

1. monad-var: Generic operations over variables. https://hackage.haskell.org/package/monad-var-0.2.2.0, accessed: 2022-05-19
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of sat/smt solvers to coq through proof witnesses. In: International Conference on Certified Programs and Proofs. pp. 135–150. Springer (2011)
3. Barrett, C., Tinelli, C.: Satisfiability Modulo Theories, pp. 305–343. Springer International Publishing, Cham (2018)
4. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)
5. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (jul 1960). https://doi.org/10.1145/321033.321034, https://doi.org/10.1145/321033.321034
6. De Groote, P.: The Curry-Howard Isomorphism. Cahiers du Centre de logique, Academia (1995), https://books.google.de/books?id=8uLuAAAAMAAJ
7. Delaware, B., d. S. Oliveira, B.C., Schrijvers, T.: Meta-theory à la carte. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 207–218 (2013)
8. Forster, Y., Stark, K.: Coq à la carte: a practical approach to modular syntax with binders. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 186–200 (2020)
9. Hallahan, W.T., Xue, A., Piskac, R.: G2q: Haskell constraint solving. In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell. pp. 44–57 (2019)
10. Keuchel, S.: Reusability for Mechanized Meta-Theory. Ph.D. thesis, Faculty of Sciences, Ghent University (2018)
11. Keuchel, S., Schrijvers, T.: Generic datatypes à la carte. In: Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming. pp. 13–24 (2013)
12. Kuper, L., Newton, R.R.: Lvars: lattice-based data structures for deterministic parallelism. In: Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing. pp. 71–84 (2013)
13. Kuper, L., Turon, A., Krishnaswami, N.R., Newton, R.R.: Freeze after writing: Quasi-deterministic parallel programming with lvars. ACM SIGPLAN Notices **49**(1), 257–270 (2014)
14. Newton, R.R., Ağacan, Ö.S., Fogg, P., Tobin-Hochstadt, S.: Parallel type-checking with haskell using saturating lvars and stream generators. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–12 (2016)
15. Schrijvers, T., Stuckey, P., Wadler, P.: Monadic constraint programming. Journal of Functional Programming **19**(6), 663–697 (2009)
16. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.J.: Search combinators. Constraints **18**(2), 269–305 (2013)
17. Swierstra, W.: Data types à la carte. Journal of functional programming **18**(4), 423–436 (2008)
18. Uhler, R., Dave, N.: Smten with satisfiability-based search. ACM SIGPLAN Notices **49**(10), 157–176 (2014)
19. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)

20. Uustalu, T., Vene, V.: Coding recursion a la mendler. In: Department of Computer Science, Utrecht University. Citeseer (2000)
21. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015-2018. J. Satisf. Boolean Model. Comput. **11**(1), 221–259 (2019), https://doi.org/10.3233/SAT190123

Bibliography

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample Guided Inductive Synthesis Modulo Theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 270–288, Cham, 2018. Springer International Publishing.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[3] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications: 6th International Conference, TLCA 2003 Valencia, Spain, June 10–12, 2003 Proceedings 6*, pages 16–30. Springer, 2003.

[4] AgdaCubical. A standard library for Cubical Agda. `https://github.com/agda/cubical`. Accessed: 2024-14-11.

[5] AgdaManual. Agda's Documentation. `https://agda.readthedocs.io/en/v2.7.0.1/`. Accessed: 2024-12-11.

[6] AgdaStdLib. Agda's Standard Library. `https://agda.github.io/agda-stdlib/`. Accessed: 2024-13-11.

[7] Thorsten Altenkirch. Computer Aided Formal Reasoning, Lecture 20: Russell's paradox. `http://www.cs.nott.ac.uk/~psztxa/g53cfr/l20.html/l20.html`. Accessed: 2022-11-29.

[8] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, 2016.

[9] Thorsten Altenkirch and Peter Morris. Indexed containers. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 277–285, 2009.

[10] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In *Algebra and Coalgebra in Computer Science: 4th International Conference,*

CALCO 2011, Winchester, UK, August 30–September 2, 2011. Proceedings 4, pages 70–84. Springer, 2011.

[11] Liao Amelia, Mullanix Reed, and Camille Favier Naïm. 1Lab. `https://1lab.dev/`, 2025. Accessed: 2024-11-11.

[12] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[13] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. Extending SMT solvers to Higher-Order Logic. In Pascal Fontaine, editor, *Automated Deduction – CADE 27*, pages 35–54, Cham, 2019. Springer International Publishing.

[14] Michael Barr and Charles Wells. *Category theory for computing science*, volume 1. Prentice Hall New York, 1990.

[15] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[16] Roberto J Bayardo Jr and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Aaai/iaai*, pages 203–208. Providence, RI, 1997.

[17] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of artificial intelligence research*, 22:319–351, 2004.

[18] Andrew Becker, David Novo, and Paolo Ienne. Automated circuit elaboration from incomplete architectural descriptions. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 391–395. IEEE, 2013.

[19] Christoph Benzmüller, Nik Sultana, Lawrence C Paulson, and Frank Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.

[20] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[21] Paul Maximilian Bittner, Thomas Thüm, and Ina Schaefer. SAT encodings of the at-most-k constraint. In *International Conference on Software Engineering and Formal Methods*, pages 127–144. Springer, 2019.

[22] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems*, pages 12–27, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[23] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. KiCS2: A new compiler from Curry to Haskell. In *International Workshop on Functional and Constraint Logic Programming*, pages 1–18. Springer, 2011.

[24] James Chapman. Type theory should eat itself. *Electronic notes in theoretical computer science*, 228:21–36, 2009.

[25] Jingchao Chen. A new SAT encoding of the at-most-one constraint. *Proceedings Constraint Modelling and Reformulation*, 2010.

[26] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[27] Stephen Cook. The P versus NP problem. *Clay Mathematics Institute*, 2, 2000.

[28] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[29] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1):423–453, 2018.

[30] Pierre-Evariste Dagand and Conor McBride. A categorical treatment of ornaments. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, page 530–539, USA, 2013. IEEE Computer Society.

[31] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[32] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[33] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract Conflict Driven Learning. *SIGPLAN Not.*, 48(1):143–154, jan 2013.

[34] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. Smtcoq: A plug-in for integrating smt solvers into coq. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, pages 126–133. Springer, 2017.

[35] Mathias Fleury. Optimizing a verified sat solver. In *NASA Formal Methods: 11th International Symposium, NFM 2019, Houston, TX, USA, May 7–9, 2019, Proceedings 11*, pages 148–165. Springer, 2019.

[36] John Nathan Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009.

[37] Arved Friedemann. Code for PhD Thesis: Towards a General Theory of Solving for Dependent Type Theory - Formalisations of Monadic Constructions for the use in Solving for Recursive Data. `https://github.com/ArvedFriedemann/PhDCode`, March 2025.

[38] Arved Friedemann and Oliver Keszocze. Towards Clause Learning\a la Carte through VarMonads. *arXiv preprint arXiv:2208.10460*, 2022.

[39] Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for treecut width and treedepth. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 117–129. SIAM, 2019.

[40] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[41] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.

[42] Michael Hanus. Functional logic programming: From theory to Curry. *Programming Logics*, pages 123–168, 2013.

[43] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. Curry: A truly functional logic language. In *Proceedings ILPS*, volume 95, pages 95–107. Citeseer, 1995.

[44] Tom Harding. holmes: Tools and combinators for solving constraint problems. `https://hackage.haskell.org/package/holmes`, 2021. Accessed: 2022-09-28.

[45] Steffen Hölldobler, Norbert Manthey, Julian Stecklina, Peter Steinke, et al. A short overview on modern parallel SAT-solvers. In *2011 International Conference on Advanced Computer Science and Information Systems*, pages 201–206. IEEE, 2011.

[46] Holger H Hoos. SAT-encodings, search space structure, and local search performance. In *IJCAI*, volume 99, pages 296–303. Citeseer, 1999.

[47] William A Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[48] Jason ZS Hu and Jacques Carette. Formalizing category theory in agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 327–342, 2021.

[49] Mikoláš Janota. On Q-resolution and CDCL QBF solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 402–418. Springer, 2016.

[50] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–128. Springer, 2012.

[51] Mikolás Janota and Joao Marques-Silva. Solving QBF by clause selection. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[52] Sverker Janson and Seif Haridi. Kernal Andorra Prolog and its computation model. In *Proceedings of the 7th International Conference*, pages 31–46. Citeseer, 1992.

[53] C Barry Jay and J Robin B Cockett. Shapely types and shape polymorphism. In *European Symposium on Programming*, pages 302–316. Springer, 1994.

[54] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *ACM SIGPLAN Notices*, 50(12):94–105, 2015.

[55] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *ACM SIGPLAN Notices*, 40(9):192–203, 2005.

[56] Hans Kleine Büning and Uwe Bubeck. Theory of quantified boolean formulas. In *Handbook of satisfiability*, pages 735–760. IOS Press, 2009.

[57] Achim Klenke. *Probability theory: a comprehensive course*. Springer Science & Business Media, 2013.

[58] Edward Kmett. guanxi: An exploration of relational programming in haskell. `https://github.com/ekmett/guanxi`, 2020. Accessed: 2022-09-28.

[59] Pepijn Kokke and Wouter Swierstra. Auto in agda. In *International Conference on Mathematics of Program Construction*, pages 276–301. Springer, 2015.

[60] Sava Krstić and Amit Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In *International Symposium on Frontiers of Combining Systems*, pages 1–27. Springer, 2007.

[61] Lindsey Kuper and Ryan R Newton. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84, 2013.

[62] Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. Freeze after writing: Quasi-deterministic parallel programming with LVars. *ACM SIGPLAN Notices*, 49(1):257–270, 2014.

[63] Andrew Lawrence, Ulrich Berger, and Monika Seisenberger. Extracting a DPLL Algorithm. *Electronic Notes in Theoretical Computer Science*, 286:243–256, 2012.

[64] Chu Min Li, Felip Manya, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound max-sat solvers. In *International conference on principles and practice of constraint programming*, pages 403–414. Springer, 2005.

[65] Chu-Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 128–133, 2010.

[66] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995.

[67] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[68] J.P. Marques Silva and K.A. Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.

[69] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[70] Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of functional programming*, 47, 2010.

[71] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[72] Edward J McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[73] Agustín Mista and Alejandro Russo. Generating random structurally rich algebraic data type values. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 48–54, 2019.

[74] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[75] Ryan R Newton, Ömer S Ağacan, Peter Fogg, and Sam Tobin-Hochstadt. Parallel type-checking with haskell using saturating LVars and stream generators. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[76] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.

[77] Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. versat: A verified modern SAT solver. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 363–378. Springer, 2012.

[78] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

[79] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[80] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[81] Alexey Radul. *Propagation networks: A flexible and expressive substrate for computation*. PhD thesis, Massachusetts Institute of Technology, 2009.

[82] Alexey Radul and Gerald Jay Sussman. The art of the propagator. In *Proceedings of the 2009 international lisp conference*, pages 1–10, 2009.

[83] Bertrand Russell. *The principles of mathematics*. Routledge, 2020.

[84] Jürgen Schmidhuber. Completely self-referential optimal reinforcement learners. In *Artificial Neural Networks: Formal Models and Their Applications–ICANN 2005: 15th International Conference, Warsaw, Poland, September 11-15, 2005. Proceedings, Part II 15*, pages 223–233. Springer, 2005.

[85] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663–697, 2009.

[86] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013.

[87] Irfansha Shaik and Jaco van de Pol. Classical planning as qbf without grounding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, pages 329–337, 2022.

[88] Mark Snyder and Perry Alexander. Monad factory: Type-indexed monads. In *Trends in Functional Programming: 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers 11*, pages 198–213. Springer, 2011.

[89] Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. In *International Joint Conference on Automated Reasoning*, pages 108–116. Springer, 2018.

[90] B.A. Trakhtenbrot. A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms. *Annals of the History of Computing*, 6(4):384–400, 1984.

[91] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[92] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[93] Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for Arithmetic Constraints. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 671–686, Cham, 2017. Springer International Publishing.