IT 21 008
Examensarbete 30 hp
Januari 2021

Homotopy Type Theory and Constraint Programming

Samuel Grahn

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Homotopy Type Theory and Constraint Programming

Samuel Grahn

An important question that arises when reasoning about constraint satisfaction problems or constraint programming models thereof, is the issue of equivalence. In order to prove such equivalences between constraint programming models, we formalise a subset of the MiniZinc constraint programming language in Homotopy Type Theory, through the use of the Cubical Agda library.

Homotopy Type Theory is a new area of mathematical logic, that allows for a simpler translation between objects and propositions, something we can use to model constraints. The univalence axiom states that given an isomorphism between types, there is an equality of types as well. This means that it suffices to prove two types are isomorphic in order to prove their equality, something which allows for less involved proofs and simpler representations of objects.

The formalisation yields a new framework for reasoning about constraint programming, which we use to reason about implied constraints and symmetry breaking constraints. Further, we introduce two different models of the N-queens problem, and prove that they are equivalent.

Through the link between the database query language SQL and constraint programming, we discuss possible avenues to use this formalisation to create an algebraic structure representing constraint models.

Handledare: Justin Pearson
Ämnesgranskare: Lars-Henrik Eriksson
Examinator: Mats Daniels
IT 21 008
Tryckt av: Reprocentralen ITC

Contents

Chapter 1

Introduction

When studying computationally hard problems, solving them somewhat efficiently often reduces to exploring the space of all possible solutions. The size of such a space — in the case of NP-hard problems — often scale exponentially with input size. Constraint programming is an approach to combinatorial problem solving that aims to provide an easy way to search for these solutions. The field of constraint programming has been evolving rapidly since its inception. New solver techniques and faster implementations, as well as use cases in many fields, including biology, electric engineering, numerical analysis, etc [2].

Constraint programming separates any given problem into two parts: modeling and solving. Modeling a problem consists of declaring some set of variables, and through some language — usually a subset of first-order logic — express the constraints; properties of the variables. Once a model has been defined, it can be plugged into a constraint solver; a program that explores the search space in some clever way, until it finds a solution, or finds that there cannot exist a solution[2]. It is particularly effective at solving problems that intuitively translate to — or are initially defined by — restrictions, or rules, that the solution must satisfy. For instance, puzzles like Sudoku, where a number can only occur once in each row, column and subsquare, allow for a simple modeling step. Another problem, perhaps the most famous puzzle in terms of constraint programming examples is the n-queens puzzle, which consists of placing n queens on a n-by-n chess board, without any one queen threatening another.

Mathematically, problems expressed using these methods are referred to as constraint satisfaction problems. Formally, they are defined as triples (X, D, C) where X is a set of variables, D is a set of domains; one for each variable, and C is a set of constraints on the allowed combinations of values of the variables in X. Each domain D_i specifies a set of values that X_i is allowed to take. Each constraint specifies a relation on a subset of the variables of X[11].

However, as is often the case when examining subjects from a purely mathematical standpoint, the question of isomorphisms, equivalences and other types of equalities are raised. The idea is that if we can manage to prove equalities between models, it will be easier to reason about transformations between models, and knowing that certain common rewriting techniques do not change the model.

There have been some prior research into this question, and one method that has arisen as somewhat efficient previously is using a theorem prover to prove equalities between models[4]. Theorem provers are often based on type theory; a form of logic that encompasses data types, which allows for encoding of mathematical statements, and proofs thereof, in a computationally verifiable system. The theorem prover which will be used in this thesis is Agda, which behaves and looks similar to the functional programming language Haskell. What differentiates Agda from other functional programming languages is that it is dependently typed; meaning types can be defined not only using other types, but also using elements of other types. This is important, because dependent types allows for the formulation of logical quantifiers in the interpretation of propositions as types[7], which we will discuss in section 2.12

When working with mathematical structures, it is often considered sufficient to show isomorphisms between objects, and then for all intents and purposes consider them the same. This becomes a problem when formalizing, since they are not in fact equal. This is a problem that Homotopy Type Theory (HoTT) attempts to solve through the axiom of univalence, regarding types as spaces, and equalities as paths between points in the space[12]. This way, should we be able to prove that two types A and B are equivalent, and that type A exhibits a property $P(A)$, we can transport the proof of $P(A)$ along the equality $A \equiv B$, and arrive at a proof of $P(B)$. This path notion stems from homotopy theory; a subset of topology that focuses on the properties that are homotopy transferable[12]. The development of HoTT has yielded successful results in many areas of mathematics through this notion of homotopy equivalence.

The notion of equalities we are looking for are known as extensional, which means objects are equal if they are indistinguishable from each other; in other words, when they share the same external properties. This is in contrast to the more classical notion of equality of being defined the same way — which is known as intensional equality. As HoTT is an extensional type theory, the idea is that if we can manage to formalise models in a way that we cannot distinguish them, we will have an equality of models.

The goal of this thesis is to examine whether we can use HoTT to effectively prove model equivalences. If two models have the same solutions, or the solutions to them are equivalent, there is no way to differentiate the models themselves. This allows us to reason about equality between models without them necessarily being definitionally equal. We can consider them as homotopic spaces, which this thesis attempts to formalise using the theorem prover Agda. We will describe the inner workings of HoTT through the lens of the cubical library for the Agda programming language, which is based on Cubical Type Theory (CTT). CTT is a constructive approach to HoTT[6], something we will discuss in section 2.16. We will assume a basic familiarity with Zermelo-Frankel set theory (ZFC), as this helps draw parallels to clarify concepts which might otherwise be unclear.

The first step is to choose a constraint programming language to model in Agda. We will be using a subset of the MiniZinc constraint modeling language. MiniZinc was created as an attempt at creating a standardized modeling language, and allows for solving the same model in many different solvers[9].

The data types we will consider are the natural numbers, ranges (consecutive subsets of the natural numbers, for instance 3..5 corresponds to the set $\{3,4,5\}$), and arrays with a fixed length.

Since in HoTT, propositions are types, the constraints will be represented as variables of the type representing the proposition.

We will represent each model as a record type; a language construct of Agda which appear similar to structs from C-like languages, but with a type theoretical background which will be explained in section 2.5.

The model record will contain fields for variables and constraints, each of which will need to be defined to create an element of the model — a solution. In other words, in a solution to the model, the variables have been assigned a value, and the constraints have been proven to hold with the values of the variables.

Using this framework for defining models, we will construct two different models of the n-queens puzzle, and prove that they are equivalent.

Chapter 2

Type Theory

Usually, when formalising mathematics, one would use set theory along with first order logic. However, that is not the only way to do it. Other formalisation methods notably include Category Theory, and Type Theory. There are many forms of Type Theory, with different axiomatizations and syntax, but for this thesis we will be using HoTT, which is an extension of Martin-Löf Type Theory (MLTT) through the addition of the Univalence Axiom.

In MLTT, there are types, and their elements. While the elements in set theory can exist without the context of a set containing it, the elements of types in MLTT are bound to its corresponding type. In order to have an element a, we must have that $a : T$, for some type T, which is read as a is an element of type T. A comparison can be made to the set theoretic formulation of the same claim, which is a is an element, and a is in T [12].

2.1 Agda

In order to check the correctness of proofs, we will use a proof assistant. The syntax of Agda closely resembles the syntax of both MLTT and the programming language Haskell. For instance, should we want to declare that we have a natural number with a value of 5, we would write.

```
example₁ : ℕ
example₁ = 5
```

First, we state that example_1 is of type $\mathbb{N}$, and on the second line that its value is 5. Note that Agda allows for the use of unicode index-subscripts in the names of the identifiers. Agda allows any unicode character, excluding spaces, to be used in the names of identifiers.

While the natural numbers are built into Agda, and defined through the Peano axioms, the standard library of Agda is not sufficient for our needs as it does not encode HoTT. Instead, we will be using the cubical library [1] which implements CTT[6]. The difference between these type theories will be presented throughout chapter section 2.14 and section 2.16.

2.2 Introducing Types

When introducing a type in MLTT, one provides a number of properties[12].

1. **Formation Rules**: How to form new types of this kind.
2. **Constructors**: Rules for how to construct an element of the type.
3. **Eliminators**: How to use elements of the type.
4. **Computation Rule**: How eliminators act on constructors.

Often these requirements are automatically handled for us through Agda, but sometimes we will need to explicitly declare instances for these rules ourselves. Some examples will be presented throughout the rest of this chapter.

2.3 Function Types

Functions behave similarly to the functions of set theory, albeit defined differently. In set theory, a function from A to B is a subset $f \subseteq A \times B$ such that for every pair $(x, y), (x, z)$ in f, we have that $y = z$, where we denote $(x, y) \in f$ by $f(x) = y$; that a function only has one value for any given input[12].

In type theory, however, they are defined more closely to their use; returning an output for any given input. The formation rules for function types says that if A is a type, and if B is a type, there is a function type $A \to B$. The constructor for the function type is the lambda abstraction; given a type signature $A \to B$, the expression $\lambda x \to \varphi$ represents a function that, when given an $a : A$, substitutes a for every free occurrence of x in φ[12].

The eliminator of the function type is function application. Given a function $A \to B$, we can apply the function with a value $a : A$, to receive a $b : B$. The computation rule for function types states that $(\lambda x \to \varphi)(a)$ is equal to the substitution of a for every free occurrence of x in φ[12].

In Agda, we can declare a function by stating that it is an element of a function type

```
fact : ℕ → ℕ
```

Now we have declared a function named fact, from ℕ to ℕ, which means that given an a : ℕ, the function will provide a b : ℕ. However, when we postulate the existence of such a function, Agda will notify us that we also need to define it.

```
fact 0 = 1
fact (suc n) = (fact n) * (suc n)
```

Here, we pattern match on the Peano implementation used for the natural numbers, where a natural number is either 0 or the successor (suc) to a natural number[12].

2.4 Universes

Since each element must have a type, and we want a way to reason about higher types, types with types as elements, we introduce the concept of Universes. A Universe is a type whose elements are also types. Further, as in set theory, an equivalent statement to Russels' Paradox[1] can be constructed in naive type theory, which we solve in the same way; by introducing universe levels. Each universe is contained in a larger universe, giving an infinite hierarchy of universes. We give each universe a level ℓ, and denote the universe with Type ℓ[12]. The universes are cumulative, meaning Type ℓ : Type (ℓ-suc ℓ), and for every A : Type ℓ we have a corresponding B : ℓ-suc ℓ, which will be formalised in section 2.7.

In order to simplify declaring types, Agda allows the use of private variables. This means that, as an example, whenever we use the variable m, defined below, Agda knows it is of type ℕ unless otherwise specified.

```
private
  variable
    m n k : ℕ
    ℓ ℓ' ℓ'' : Level
    A B C : Type ℓ
```

When a type does not depend on any other types, as is the case for the natural numbers, one usually declares it as a member of the type with level zero. There is a shorthand for this in Agda, namely Type_0, which is equivalent to Type ℓ-zero.

2.5 Defining Types

Types can be defined in many ways in Agda. For instance, one can write

[1]Russels' paradox defines a set $R = \{x : x \notin x\}$, the set of all sets not containing themselves. If $R \in R$, then it contradicts the definition that it does not contain itself, while if $R \notin R$, then R is a set that does not contain itself, and should be a member of R.[12]

```
ℕfunc : Type₀
ℕfunc = ℕ → ℕ
```

which declares ℕfunc as a synonym for the type ℕ → ℕ. Another way to define types is using the data keyword.

```
data Maybe (A : Type ℓ) : Type ℓ where
  Nothing : Maybe A
  Just : A → Maybe A
```

The formation rules are automatically based on the parameters of the type, i.e. the type A, and states that for any A, there is a type Maybe A. The constructors, Nothing and Just, state how to construct elements of this type. The elimination rule for this type is pattern matching, and the computation rule states that a constructor applied with equal elements produce equal elements in the type.

A third way to define types is using the record syntax.

```
record FunctionRecord (A : Type ℓ) (B : Type ℓ) : Type (ℓ-suc ℓ) where
  constructor _F_
  field
    f : A → B
    g : B → A
```

First, using the constructor declaration, any appearance of A F B is equivalent to FunctionRecord A B, along with some additional definitions

```
data FunctionData
  (A : Type ℓ) (B : Type ℓ')
  : Type (ℓ-suc (ℓ-max ℓ ℓ')) where
  _F_ : (A → B) → (B → A) → FunctionData A B

FunctionData•f : (FunctionData A B) → (A → B)
FunctionData•f (f F _) = f

FunctionData•g : FunctionData A B → (B → A)
FunctionData•g (_ F g) = g
```

Note that all of the above ways of defining types are interchangeable. Thus, we will use whichever is most readable in any given situation.

2.6 Unit Type

The unit type is a type with a single element. The formation rules is that there is just one type of this kind, and the only constructor is tt. The unit type has no eliminators, and thus no computation rule[12]. We define it as

```
data Unit_x : Type_0 where
  tt : Unit_x
```

Note the index subscript $_x$ in the above definition. Whenever we define a type already present in the Cubical library, we use this subscript to avoid the name collision caused by defining the same type twice, but still be able to repeat it for clarity.

2.7 Universe Lifting

Since the universes are cumulative, for any type A : Type ℓ there is a type Lift B : Type (ℓ-suc ℓ) that is equivalent to A. In Agda this operation of lifting to a higher universe is represented by the Lift type, defined as

```
record Lift_x {i j} (A : Type i) : Type (ℓ-max i j) where
  constructor lift
  field
    lower : A
```

where i and j are the universe levels in question. Note that they are surrounded by curly braces, which means they are implicit parameters. An implicit parameter is automatically inferred by Agda compiler at compile time. When such an inference is not possible, we may opt to manually supply the argument by writing {i = ℓ} when applying the function, if we want i to take the value ℓ.

Note that when using this definition, if i is a higher level than j, Lift does not actually lift, and instead returns a type of the same universe level as the input type. This is because the cumulativity of the universes ensures every type can be lifted, they cannot neccessarily be lowered.

We use this universe lifting to represent a unit type in any given universe, which we define as

```
Unit*_x : {ℓ : Level} → Type ℓ
Unit*_x = Lift Unit

pattern tt*_x = lift tt
```

In this application of Lift, note that the implicit parameters which are automatically inferred have been assigned values. Namely, i is automatically infered to have the value ℓ-zero, since that is the level of the type Unit. Meanwhile, the value of j is inferred to be the value of the output level ℓ, which itself is a type parameter, and will be inferred whenever the Unit* type is used.

Moreover, note the pattern keyword, which defines a pattern synonym, causing the agda compiler to treat the identifier tt*$_x$ as lift tt.

2.8 Product Types

Given two types, A and B, we can construct the type $A \times B$ of pairs. The constructor for product types is the _,_ operator which has type A → B → (A × B), i.e. takes a value of type A, and a value of type B, and gives us a value of type $A \times B$.

```
record _×_ (A : Type ℓ) (B : Type ℓ') : Type (ℓ-max ℓ ℓ') where
  field
    fst : A
    snd : B
```

The eliminators for the product type are the two projection functions; fst and snd, which yields the first and second element of a pair, respectively. Finally, we have the computation rule, which states that given x : A × B, we have x ≡ (fst x , snd x).

The rules are usually implicitly defined in Agda, so we will not be explicitly mentioning the formation- and elimination rules in the future, but it is good to have an idea of the basic principles for defining types.

2.9 Families

Families are defined the same way as they are in set theory. A Type Family is a function from a type to a universe. As an example we will provide a family defined on the natural numbers

```
finFamily : ℕ → Type ℓ-zero
finFamily n = Fin n
```

When provided a natural number n as an argument, the function provides a type with exactly n elements.

2.10 Dependent Functions

A dependent function is a function whose output type depends on the value of its input. Given a type A and a family B : A → Type ℓ, we can create the dependent function type $\Pi_{x:A} Bx$. However, during this thesis we will use the notation of Agda, which is (x : A) → B x, or ∀ x → B x, depending on context.

Note that if B x = C is a constant family, the dependent function has constant return type.

2.11 Dependent Product

A dependent product is like a pair, except the type of the second element depends on the value of the first. Given a type A and a family B, we have the dependent pair Σ A B, with projections fst : Σ A B → A and snd : (x : Σ A B) → B (fst x).

```
record Σₓ (A : Type ℓ) (B : A → Type ℓ') : Type (ℓ-max ℓ ℓ') where
  field
    fstₓ : A
    sndₓ : B fstₓ
```

The cubical library has some syntax declarations for dependent products, allowing us to define them more easily. As an example, we can define

```
example₂ : Σ[ a ∈ ℕ ] a ≡ 2
example₂ .fst = 2
example₂ .snd = refl
```

2.12 Propositions

In set theory, the logic and proofs of propositions exist on a separate layer from the sets themselves. When we claim, for instance, that $\forall a.\phi(a)$, for some formula $\phi(a)$, there is an underlying interpretation of the quantifier ∀, as well as interpretation for any symbols in ϕ. This means that the language of first order logic lies on a separate layer from the sets of which the logic speaks.

In MLTT, however, propositions are themselves also types. Any type can be considered as the proposition that there exists an object of that type. This leads to a translation between the familiar methods of formulating propositions in set theory, to the analogous types in type theory. For instance, a statement $\exists x \in A, P(x)$ in first order logic corresponds to the dependent product Σ[x ∈ A], P x. As an example, we will define the types that describe the ordering of natural numbers. We can represent ≤ as

```
_≤ₓ_ : ℕ → ℕ → Type₀
x ≤ₓ y = Σ[ n ∈ ℕ ] n + x ≡ y
```

which is a function that, given x y : ℕ, returns the dependent product consisting of a natural number n, and a proof that n + x ≡ y. This corresponds to the statement $\exists n \in \mathbb{N},\ n+x = y$ from first order logic. An element of the type a ≤ b represents a proof of the statement a is smaller than b. In other words, when a type correspoding to a proposition is inhabited by some element, we consider that proposition to be true, and if the type is not inhabited we consider the proposition to be false. For completeness, a similar definition can be made for <

```
_<ₓ_ : ℕ → ℕ → Type₀
x <ₓ y = suc x ≤ y
```

In the same way that dependent products represent the existential quantifier, we can use dependent functions to express the universal quantifier. Further, we can also use dependent types to define theorems. Throughout this thesis we will be using several simple theorems regarding integers. For instance, the following will be used heavily

```
+-zeroₓ : ∀{x} → (x + 0) ≡ x
+-zeroₓ {x} = +-comm x zero

zero-≤ₓ : ∀{x} → 0 ≤ x
zero-≤ₓ {x} = x , (+-zero x)

≤-reflₓ : ∀{x} → x ≤ x
≤-reflₓ = 0 , refl
```

where +-comm is a proof of commutativity of the addition function.

2.13 Empty Type

The empty type is a type with no elements. It is used to create contradictions and to describe logical negation[12]. We define it as follows

```
data ⊥ₓ : Type₀ where
```

If we have $A \wedge \neg A$ in first order logic, any statement is provable. The same is true in type theory, though it is represented differently. The empty type corresponds to uninhabitance, which in the case of propositions as types correspond to falsehood. However, if the uninhabited type is inhabited, any type is inhabited, and thus every statement is provable[12].

```
recₓ : ∀ {A : Type ℓ} → ⊥ₓ → A
recₓ ()
```

Note the absence of an =-symbol in the second line, the definition of rec_x. The () pattern is the absurdity match in Agda, and represents the fact that there are no constructors that fit the current parameter, in this case the empty type. This allows us to construct contradictions. Note, however, that in HoTT, we do not have the law of excluded middle, i.e. it is not necessarily the case that ¬¬A → A, thus this will mainly be used to disregard certain invalid inputs to functions we define.

As there is a fair amount of overlap in the naming convention of the cubical library, it is common to import modules by assigning them shortcuts. In the case of the empty type, we import the module as ⊥. The empty type itself is then reached by ⊥.⊥, and likewise for the recursor rec which is reached as ⊥.rec.

2.14 Equality Types

When considering elements *a* and *b* of a type *A*, we need a way to state whether or not *a* and *b* are equal. This equality is represented by a dependent type Path : (A : Type ℓ) → A → A → Type ℓ, with Path A a b, denoted a ≡ b consisting of proofs that *a* and *b* are equal. However, the name Path hints at a slightly different meaning for this type. In HoTT, an element x : (a ≡ b) represents a path between the points *a* and *b*[12]. This topological interpretation of types as spaces will yield some interesting properties later. For any point a : A, there is an element refl : a ≡ a, representing the constant path. However, there can exist many paths between elements. Paths x y : (a ≡ a) can represent the two independent loops on a torus, for instance. We will mostly deal with types known as sets. A set in type theory is a type where all paths are constant; in other words, every path between two points is the same path.

```
isSetₓ : (A : Type ℓ) → Type ℓ
isSetₓ A = ∀ (a b : A) → ∀ (x y : a ≡ b) → x ≡ y
```

The main difference between HoTT and CTT is that in CTT, the path type is encoded using an interval type. The interval type can take two values; i0 and i1, called the endpoints.

A dependent path is represented in Agda as a type PathP : (A : I → Type ℓ) → A i0 → A i1 → Type ℓ, meaning that for any family A indexed by the interval type, there is a type of paths between elements of the two types A i0 and A i1. This means we can present equalities between elements of definitionally different types, as long as the types are equal.

Further, we represent non-dependent path equality, i.e. when the type family A is constant, as [13]

```
_≡ₓ_ : {A : Type ℓ} → A → A → Type ℓ
_≡ₓ_ {A = A} x y = PathP (λ i → A) x y
```

using these paths to represent equality, we can prove the properties of equalities, for instance reflexivity, as

```
reflₓ : {x : A} → x ≡ x
reflₓ {x = x} = λ i → x
```

In other words, we prove that for any x : A, there is a path x ≡ x.

Notably, there are several other properties of paths we will be using. Namely, we can prove that for any a b : A with p : a ≡ b, along with a family B : A → Type ℓ and a dependent function f : (x : A) → B x, we can prove that there is a path between f a and f b

```
congₓ : ∀ {ℓ ℓ'} {A : Type ℓ} {B : A → Type ℓ'} {a b : A}
  (f : (x : A) → B x)
  (p : a ≡ b) →
  PathP (λ i → B (p i)) (f a) (f b)
congₓ f p = λ i → f (p i)
```

which is a significant help when proving statements later in this thesis.

2.15 Type Equivalence

There are multiple equivalent ways to define type equivalences[12]. The definition used in the cubical library uses contractibility of fibers, two concepts which we must define.

A contractible type is a type *A* where there is an element x : A such that for any y : A, there is a path x ≡ y.

```
isContrₓ : Type ℓ → Type ℓ
isContrₓ A = Σ[ x ∈ A ] (∀ y → x ≡ y)
```

Using these paths we can take any point in the space, and transport it along the paths to the point *x*, i.e. we can contract it to a single point.

A fiber over a function f : A → B at a point y : B is the type of all x : A such that f x ≡ y, in other words, the type

```
fiberₓ : (f : A → B) (y : B) → Type _
fiberₓ {A = A} f y = Σ[ x ∈ A ] (f x ≡ y)
```

With those concepts defined, we can proceed to define type equivalence. An equivalence between types *A* and *B* is a function f : A →B, along with a proof that the fibers of *f* are contractible. We denote this as A ≃ B.[12].

```
record isEquivₓ
  {ℓ ℓ' : Level}{A : Type ℓ}{B : Type ℓ'} (f : A → B)
  : Type (ℓ-max ℓ ℓ') where
  field
    equiv-proof : (y : B) → isContr (fiber f y)
```

We can note that any equality $A \equiv B$ yields a trivial equivalence A ≃ B; i.e. that there exists an f : (A ≡B) →(A ≃ B).

The cubical library also defines a type called Iso, for isomorphisms between types. Reasoning with isomorphisms rather than with contractible fibers will be more familiar to those without a background in topology. An isomorphism consists of two functions and proofs that they are left and right inverses of each other.

```
record Isoₓ {ℓ ℓ'} (A : Type ℓ) (B : Type ℓ') : Type (ℓ-max ℓ ℓ') where
  constructor iso
  field
    fun : A → B
    inv : B → A
    rightInv : ∀ b → fun (inv b) ≡ b
    leftInv : ∀ a → inv (fun a) ≡ a
```

Further, these two representations of equivalence are proven equivalent in the cubical libray, through isoToIsEquiv. The type of isomorphisms is then proven equivalent to equality through isoToPath, which will be used later.

2.16 Univalence

In MLTT, it is impossible to define a predicate that distinguishes isomorphic types[12]. In other words, given A ≃ B, we cannot prove that A ≢B, but it is not necessarily true that A ≡B. However, mathematicians often use isomorphic objects and informally assume that if one can prove a statement for a mathematical object, all objects isomorphic to that object has that same property. In fact is consistent with MLTT to assume that if we have A ≃ B, we also have A ≡B. This assumption is called the univalence axiom, and is expressed in cubical Agda as ua_x : A ≃ B → A ≡ B.

In Cubical Agda, the interval types used for paths allows for the direct proof of this statement, turning it from an axiom to a theorem. In this way, CTT is a constructive approach to homotopy type theory. The proof of this involves the operation of glueing, which expresses that to be extensible is invariant by equivalence, which allows for the definition of a composition operator for universes, with which one can prove the univalence axiom[6].

2.17 Intensional and Extensional Equalities

In logic, multiple types of equalities can be considered. Notably, we can consider two objects to be the same if they are defined identically, this type of equality is called intensional equality. However, we often want to consider objects equal if they have the same external properties. For instance, any two functions can be considered equal if they produce the same output for any given input. This type of equality is called extensional. HoTT and CTT are extensional type theories, as we will discuss in section 2.16.

2.18 Function Extensionality

Consider the functions

```
f_ex g_ex : ℕ → ℕ

f_ex n = (n + 5)
g_ex n = (5 + n)
```

For any number n it is obvious the two functions always return the same result. However, they are not defined the same way. In order to consider these functions equal, we must use a consequence of the axiom of univalence; function extensionality. If two functions always produce the same value for any given input, they are extensionally equal, and in HoTT, this indeed implies equality. Thus we can define

```
example₃ : f_ex ≡ g_ex
example₃ = funExt λ x → +-comm x 5
```

In the above example, we use function extensionality, funExt, to prove the equality. It transports the proof of the equality after function application to an equality proof before said application; i.e. on the functions themselves. Function extensionality is proved in the Cubical library through direct use of the interval types from CTT.

```
funExt_x : {B : A → I → Type ℓ}
  {f : (x : A) → B x i0}
  {g : (x : A) → B x i1}
```

There are three implicit parameters; B is an type family dependent on A and I, the interval type. This type is needed to specify that the functions f and g may have different range types. The functions f and g are declared as functions dependent on this family.

```
→ ((x : A) → PathP (B x) (f x) (g x))
```

It also requires a proof that given any x, there is a path betwen f x and g x. And finally, it will provide a proof that there is a path between f and g

```
→ PathP (λ i → (x : A) → B x i) f g
```

And finally, it needs a proof.

```
funExtₓ x i p = x p i
```

2.19 Mere Propositions

When we consider the usual rules from first-order logic, we want any two proofs of a proposition to be equivalent, a notion which is known as proof-irrelevance. This enables us to easier work with the rules known to us from first-order logic. Types that satisfy this equivalence are called mere propositions. This is defined using a type

```
isPropₓ : (A : Type ℓ) → Type ℓ
isPropₓ A = (a b : A) → a ≡ b
```

We can use this to define certain logical consequences known from first order logic

```
equivalence : (P : isProp A) → (Q : isProp B)
             → (A → B) → (B → A) → A ≡ B
equivalence P Q f g = isoToPath (iso f g
  (λ b → Q (f (g b)) b)
  (λ a → P (g (f a)) a))
```

When forming other types using mere propositions, the property of being a mere proposition is often inherited. For instance, we can define

```
isPropPair : {ϕ : isProp A} {ψ : isProp B}
           → isProp (A × B)
isPropPair {ϕ = ϕ} {ψ} (a1 , a2) (b1 , b2)
  = cong (λ a → (a , a2)) (ϕ a1 b1)
  • cong (λ a → (b1 , a)) (ψ a2 b2)
```

When using mere propositions to form a dependent product, it is enough to prove equivalence between the elements of the first type, since we have

```
Σ≡Propₓ : {B : A → Type ℓ}
          → ((x : A)
          → isProp (B x))
          → {u v : Σ A B}
          → (p : u .fst ≡ v .fst)
          → u ≡ v

Σ≡Propₓ pB {u} {v} p i .fst = (p i)
Σ≡Propₓ pB {u} {v} p i .snd
  = isProp→PathP (λ i → pB (p i)) (u .snd) (v .snd) i
```

When interpreting the dependent product type as the existential quantifier, we note that the above theorem equates to the fact that any two proofs of the existence of an element satisfying a proposition are equivalent, if they instantiate this existential quantifier with the same element.

2.20 Decidability

For some propositions, one can construct algorithms to find whether they hold or not. When regarding propositions as types, such an algorithm would yield either a proof of inhabitance, or truth, or of emptiness, falsehood. A type is considered decidable if there is an algorithm that terminates in finite time for deciding whether it is inhabited. The type representing decidability is

```
data Decₓ (P : Type ℓ) : Type ℓ where
  yesₓ : ( p : P) → Decₓ P
  noₓ : (¬p : ¬ P) → Decₓ P
```

For instance, we can prove that if we have two natural numbers m, n, it is decidable whether m ≤ n, by providing an algorithm for calculating it.

```
decidable≤ : (m n : ℕ) → Dec (m ≤ n)
```

The type zero-≤ states that any natural number is greater than or equal to zero.

```
decidable≤ zero n = yes zero-≤
```

No number is less than zero.

```
decidable≤ (suc m) (zero) = no (¬-<-zero)
```

Finally, (suc m) ≤ (suc n) if m ≤ n. The with syntax, followed by a number of lines starting with ... | — representing the same function name and parameters as on the line above — allows for pattern matching by providing separate implementations for each possible value of whatever is applied. In the below example, we match decidable≤ m n with its two constructors, yes and no.

```
decidable≤ (suc m) (suc n) with decidable≤ m n
... | yes p = yes (suc-≤-suc p)
... | no ¬p = no (λ x → ¬p (pred-≤-pred x))
```

where suc-≤-suc : m ≤ n → (suc m) ≤ (suc n) and pred-≤-pred : (suc m) ≤ (suc n) → m ≤ n[1]. A common use case for decidability is on equality types. A type with decidable equality types is called discrete, a property we will be using later to declare certain functions and types.

```
Discrete_x : Type ℓ → Type ℓ
Discrete_x A = (x y : A) → Dec (x ≡ y)
```

As an example, we will prove that the natural numbers are discrete.

```
discreteℕ_x : Discrete ℕ
discreteℕ_x zero zero = yes refl
discreteℕ_x zero (suc y) = no ℕ.znots
discreteℕ_x (suc x) zero = no ℕ.snotz
discreteℕ_x (suc x) (suc y) with discreteℕ_x x y
... | yes p = yes (cong suc p)
... | no ¬p = no λ x_1 → ¬p (cong predℕ x_1)
```

where ℕ.znots : ∀n → ¬ (0 ≡ suc n) and ℕ.snotz : ∀n → ¬ (suc n ≡ 0)[1].

Chapter 3

Constraint Programming

Constraint programming is a method to define models for mathematical satisfaction or optimization problems, and let generalized solvers find solutions to these models[2]. We will focus on a subset of the MiniZinc constraint programming language[9], consisting of fixed-size arrays, integers, ranges and a few of the built in functions.

A MiniZinc model consists of variables, parameters and constraints. The parameters are declared symbols, which when defined present an instance of the model. As an example, we model the existence of n natural numbers whose product is lesser than or equal to their sum, first in MiniZinc, and then represent it in Agda.

```
int: n;
array[1..n] of var int: v;

constraint (forall(x in v)(x >= 0));
constraint (sum(x in v)(x) <= product(x in v)(x));
```

Listing 3.1: plustimes.mzn

In order to use HoTT to prove anything about our selected subset of MiniZinc, we need to represent MiniZinc models in HoTT.

3.1 Data Types

The data types from MiniZinc we will be representing in HoTT are the natural numbers, ranges, and arrays. The natural numbers will simply be represented by the natural numbers in HoTT.

Ranges, where $a..b$ in MiniZinc represents the set of natural numbers $\{ a, a+1, \dots, b \}$[9], will be reduced in scope to ranges with $a = 0$. This is most easily represented by the type Fin, defined in the cubical library.

```
data Finₓ : ℕ → Type₀ where
  zeroₓ : {n : ℕ} → Finₓ (suc n)
  sucₓ : {n : ℕ} (i : Finₓ n) → Finₓ (suc n)
```

Fin is a type family, and for any n : ℕ, Fin n is a type with exactly n elements.

Arrays will be represented by the vector type Vec defined in the cubical library.

```
data Vecₓ (A : Type ℓ) : ℕ → Type ℓ where
  [] : Vecₓ A zero
  _∷_ : ∀ {n} (x : A) (xs : Vecₓ A n) → Vecₓ A (suc n)
```

Which states that a vector is either the empty vector [], or a ∷ v, where a : A and v is a vector. This definition is in effect a linked list with its length encoded in its type.

In order to easier represent calculations to be done later, we also define some well known functions representing common usages on lists. The head of the list is the first element of the list.

```
headₓ : Vec A (suc n) → A
headₓ (x ∷ xs) = x
```

The tail is everything except the first element

```
tailₓ : Vec A (suc n) → Vec A n
tailₓ (x ∷ xs) = xs
```

Map applies a function to each element of a list

```
mapₓ : (A → B) → Vec A n → Vec B n
mapₓ f [] = []
mapₓ f (x ∷ v) = f x ∷ mapₓ f v
```

Foldr applies a function with an accumulator to each element of the list.

```
foldr : (A → B → B) → B → Vec A n → B
foldr f b [] = b
foldr f b (x ∷ xs) = f x (foldr f b xs)
```

Zip takes elements from two vectors and produces a vector of pairs.

```
zip : Vec A n → Vec B n → Vec (A × B) n
zip [] [] = []
zip (x ∷ xs) (y ∷ ys) = (x , y) ∷ (zip xs ys)
```

Indices is the array $[0, 1, 2, ..., n]$ for any n.

```
indices : ∀{n} → Vec (Fin n) n
indices {zero} = []
indices {suc n} = zero ∷ Vec.map suc indices
```

3.2 Constraints

In constraint programming, a constraint is a restriction, or relation, on set of variables. They are formulated as properties, or conditions, set upon these variables[2]. Since propositions are themselves types, a constraint on some collection of variables is a type family on the values of these variables, such that a constraint is considered satisfied if its type is inhabited.

For instance, constraining a natural number to be less than 5, would translate to the type

```
≤5 : ℕ → Type₀
≤5 n = n ≤ 5
```

In general, it is not required that a constraint is a mere proposition, though as we are mostly interested in inhabitance rather that specific values, the constraints in the rest of the thesis will all be mere propositions.

3.3 Models

Every MiniZinc model will be represented by a product type consisting of arbitrarily many fields and parameters

```
ExampleModel : (param : ℕ) → Type₀
ExampleModel n = Σ[ v ∈ ℕ ] v ≤ n
```

Here, we define a model by the name of Model, with parameter param of type ℕ. The variables and constraints are represented as fields in the record, which when constructing elements of the record type need to be supplied. Thus, a solution to the model represented is an element of the Model type, and its satisfiability corresponds to inhabitance.

We can now formalise the model in Listing 3.1.

Note that in order to restrict the integer data type in MiniZinc to the natural numbers, we simply constrain each element of the array to be greater than or equal to zero. This is automatically satisfied when modeling using natural numbers, so the constraint is not represented in the Agda model.

```
Plus>Times : ℕ → Type₀
Plus>Times n = Σ[ v ∈ Vec ℕ n ] (foldr (_*_) 1 v ≤ foldr (_+_) 0 v)
```

And provide a solution to it.

```
pf : Plus>Times 2
pf = (1 :: 0 :: []) , zero-≤
```

Note that in order for a model to be considered satisfied, it requires a proof for every constraint. In this instance, it is a rather simple task, since the calculations are done by the compiler, and we merely need to supply the proof that $1 > 0$.

However, for more intricate constraints, using the property of decidability to allow for the compiler to provide automatic proofs, we can simplify this process slightly.

3.4 Decidable Constraints in Models

Given a decidable type, we can let the Agda compiler use the decidability to automatically infer proofs of statements. This enables us to easily check that constraints hold on solutions.

Given a decidable type, we can convert the proof to a boolean.

```
isYes : Dec A → Bool
isYes (yes p) = true
isYes (no ¬p) = false
```

A boolean can in turn be represented as either the unit type, or the empty type.

```
T : Bool → Type₀
T true = Unit
T false = ⊥.⊥
```

isYes converts the truth value of Q into a boolean, and back to a type through T.

```
Trueₓ : Dec A → Type₀
Trueₓ Q = T (isYes Q)
```

Thus, if True is inhabited, Q is true, and there is exactly one element tt : True Q. This enables the Agda compiler to, after evaluating Q, infer the type of True Q. If it is indeed true, the compiler will instantiate it with tt. If it is false, it will fail to compile.

```
AutoProof : (Q : Dec A) → {True Q} → A
AutoProof (yes p) = p
```

As a simple example, we can let agda prove that $20 \leq 42$, through

```
20≤42 : 20 ≤ 42
20≤42 = AutoProof (decidable≤ 20 42)
```

while this is easy to prove manually through

```
20≤42' : 20 ≤ 42
20≤42' = ≤-+k {n = 22} zero-≤
```

for more complex constraints, allowing automatic proof can significantly lessen the need for manual labour.

3.5 Global Constraints

A global constraint is a constraint that represents a relation between a nonfixed number of variables. We will be using the MiniZinc global constraint **alldifferent**, which takes an array as input, and presents a set of $\binom{n}{2}$ inequalities. For instance, **alldifferent** $[a, b, c]$, would produce $(a \neq b) \wedge (a \neq c) \wedge (b \neq c)$. Note that this is equivalent, by associativity of logical conjunction, to $(a \neq b) \wedge (a \neq c) \wedge$ **alldifferent** $[b, c]$. The first two clauses together state that a is not an element of $[b, c]$. This can be represented as **¬elem** a $[b, c]$, so that **alldifferent** $[a, b, c] =$ (**¬elem** a $[b, c] \times$ **alldifferent** $[b, c]$). In Agda, this is built as two recursive definitions.

```
¬elem : {A : Type ℓ} → A → (Vec A n) → Type ℓ
```

There are no elements of the empty list

```
¬elem _ [] = Unit*
```

An element is not in a list if its not equal to the head of the list, and not an element of the tail.

```
¬elem a (x :: v) = (¬ a ≡ x) × (¬elem a v)
```

```
alldifferent : {A : Type ℓ} → Vec A n → Type ℓ
```

All elements of the empty list are different.

```
alldifferent [] = Unit*
```

All elements of a nonempty list are different if the head is not in the tail, and all elements of the tail are different.

```
alldifferent (x :: v) = ¬elem x v × alldifferent v
```

Interestingly, and of great use to us in section 4.1, the alldifferent constraint is equivalent to injectivity of the index function, also known as the lookup function.

We start by defining injectivity as a type.

```
injective : (A → B) → Type _
injective f = ∀ a b → f a ≡ f b → a ≡ b
```

Injectivity is a mere proposition

```
injective-isProp : ∀ {ϕ : isSet A} {f : A → B}
                 → isProp (injective f)
injective-isProp {ϕ = ϕ} = isPropΠ3 λ x y _ → ϕ x y
```

We define a type synonym for injectivity of the lookup function.

```
injective-lookup : {A : Type ℓ} → Vec A n → Type ℓ
injective-lookup v = injective (λ k → lookup k v)
```

And prove some theorems about it to assist us in the proof.

If the lookup function is injective on v, then the lookup function is injective on the tail of v.

```
injective-lookup-tail :
  ∀ {v : Vec A (suc (suc n))}
  → injective-lookup v
  → injective-lookup (tail v)

injective-lookup-tail {v = x :: v} ι a b p = injSucFin (ι (suc a) (suc b) p)
```

Injectivity of the lookup funciton is a mere proposition

```
injective-lookup-isProp : ∀{A : Type ℓ} {v : Vec A n}
                        → isProp (injective-lookup v)
injective-lookup-isProp {v = v}
  = injective-isProp {ϕ = isSetFin}
```

If the lookup function is injective on a vector, it is injective efter removing the second element. This lemma allows us to inductively operate on the tail of a list while keeping the head intact.

```
injective-lookup-skipelem :
  ∀ {v : Vec A (suc (suc n))}
  → injective-lookup v
  → injective-lookup ((head v) ∷ (tail (tail v)))

injective-lookup-skipelem ι zero zero p = refl
injective-lookup-skipelem {v = x ∷ x₁ ∷ v} ι zero (suc b) p =
  ⊥.rec (Fin.znots (ι zero (suc (suc b)) p))
injective-lookup-skipelem {v = x ∷ x₁ ∷ v} ι (suc a) zero p =
  ⊥.rec (Fin.snotz (ι (suc (suc a)) zero p))
injective-lookup-skipelem {v = x ∷ x₁ ∷ v} ι (suc a) (suc b) p =
  injective-lookup-tail ι (suc a) (suc b) p
```

If the lookup function is injective, the value of the head is not an element of the tail of the vector.

```
injective-lookup-¬elem : ∀ {v : Vec A (suc n)}
                        → injective-lookup v
                        → ¬elem (head v) (tail v)

injective-lookup-¬elem {v = x ∷ []} ι = tt*
injective-lookup-¬elem {v = x ∷ x₁ ∷ v} ι .fst π
  = Fin.znots (ι zero (suc zero) π)
injective-lookup-¬elem {v = x ∷ x₁ ∷ v} ι .snd
  = injective-lookup-¬elem (injective-lookup-skipelem ι)
```

We prove ¬elem is a mere proposition.

```
¬elem-isProp : ∀ {x : A} {v : Vec A n}
              → isProp (¬elem x v)

¬elem-isProp {v = []} a b = refl
¬elem-isProp {x = x} {v = v ∷ vs} a b
  = Σ≡Prop (λ _ → ¬elem-isProp) (isProp¬ (x ≡ v) (fst a) (fst b))
```

After which we prove that alldifferent is a mere proposition.

```
alldifferent-isProp : ∀ {v : Vec A n}
                     → isProp (alldifferent v)
```

```
alldifferent-isProp {v = []} x y = refl
alldifferent-isProp {v = x₁ ∷ v}
  = isPropΣ ¬elem-isProp (λ _ → alldifferent-isProp)
```

We can now begin to prove that they imply each other. First, we prove that given that the lookup function is injective, alldifferent is satisfied.

```
injective-lookup→alldifferent : ∀ {v : Vec A n}
                               → injective-lookup v
                               → alldifferent v

injective-lookup→alldifferent {v = []} ι = tt*
injective-lookup→alldifferent {v = x ∷ []} ι
  = (injective-lookup-¬elem ι) , tt*
injective-lookup→alldifferent {v = x ∷ x₁ ∷ v} ι .fst
  = (injective-lookup-¬elem ι)
injective-lookup→alldifferent {v = x ∷ x₁ ∷ v} ι .snd
  = injective-lookup→alldifferent (injective-lookup-tail ι)
```

We provide a translation between ¬elem and lookup.

```
¬elem→¬lookup : ∀ {x : A} {v : Vec A n}
  → (b : Fin n)
  → ¬elem x v
  → ¬ (x ≡ lookup b v)
¬elem→¬lookup {v = x₁ ∷ v} zero ne p
  = fst ne p
¬elem→¬lookup {v = x ∷ v} (suc b) ne p
  = ¬elem→¬lookup b (snd ne) p
```

Which we use to prove that given alldifferent, the lookup function is injective.

```
alldifferent→injective-lookup : ∀ {v : Vec A n}
                               → alldifferent v
                               → injective-lookup v
alldifferent→injective-lookup {v = x ∷ []} ad zero zero _ = refl
alldifferent→injective-lookup {v = x ∷ y ∷ v} (cur , nxt) zero zero p = refl
alldifferent→injective-lookup {v = x ∷ y ∷ v} (cur , nxt) zero (suc b) p =
  ⊥.rec (¬elem→¬lookup b cur p)
```

```
alldifferent→injective-lookup {v = x ∷ y ∷ v} (cur , nxt) (suc a) zero p =
  ⊥.rec (¬elem→¬lookup a cur (sym p))
alldifferent→injective-lookup {v = x ∷ y ∷ v} (cur , nxt) (suc a) (suc b) p =
  cong suc (alldifferent→injective-lookup nxt a b p)
```

Finally, since we have functions in both directions between two mere propositions, we have an equality of types.

```
injective-lookup≡alldifferent : {v : Vec A n}
                              → injective-lookup v ≡ alldifferent v
injective-lookup≡alldifferent {v = v} =
  equivalence
    injective-lookup-isProp
    alldifferent-isProp
    injective-lookup→alldifferent
    alldifferent→injective-lookup
```

Further, we can prove that alldifferent is a decidable type.

```
dec¬elem : ∀ {n} {A : Type ℓ} {ϕ : Discrete A} {x : A} {v : Vec A n}
          → Dec (¬elem x v)
dec¬elem {n = .zero} {A} {ϕ} {x} {v = []} = yes tt*
dec¬elem {n = .(suc _)} {A} {ϕ} {x} {v = v ∷ vs} with ϕ x v
... | yes p = no (λ x₁ → fst x₁ p)
... | no ¬p with dec¬elem {ϕ = ϕ} {x = x} {v = vs}
... | yes q = yes (¬p , q)
... | no ¬q = no (λ x₁ → ¬q (snd x₁))

decAlldifferent : ∀ {n} {A : Type ℓ} {ϕ : Discrete A} (v : Vec A n)
                → Dec (alldifferent v)
decAlldifferent {n = zero} {A} {ϕ} [] = yes tt*
decAlldifferent {n = (suc n)} {A} {ϕ} (x ∷ v) with dec¬elem {ϕ = ϕ} {x = x} {v = v}
... | no ¬p = no (λ x₁ → ¬p (fst x₁))
... | yes p with decAlldifferent {n = n} {ϕ = ϕ} v
... | yes q = yes (p , q)
... | no ¬q = no (λ x₁ → ¬q (snd x₁))
```

And the same about lookup injectivity, using the implications defined above.

```
decInjective-lookup : ∀ {n} {A : Type ℓ} {ϕ : Discrete A} (v : Vec A n)
                    → Dec (injective-lookup v)
```

```
decInjective-lookup {ϕ = ϕ} v with decAlldifferent {ϕ = ϕ} v
... | yes p = yes (alldifferent→injective-lookup p)
... | no ¬p = no (λ x → ¬p (injective-lookup→alldifferent x))
```

Thus we have provided a method of representing constraints that allow for proofs of their equivalence. In short, we define constraints as propositions, in this case mere propositions. We can then prove their equivalence and optionally provide a proof of their decidability for use in confirming solutions we might provide to our models later.

Using this, we can move on to reasoning about models.

Chapter 4

Model Reasoning

In this chapter we will examine equivalencies of models, and then reason about constraint modeling techniques, and their effect on these equivalencies.

4.1 N-Queens

One of the classical constraint programming problems is the n-queens problem. The goal is to place n queens on a $n-by-n$ chessboard, without any one queen threatening another. We will prove the equivalence of two different models of this problem.

In order to properly define the types we need, we define some helper types. diagup and diagdown represent the diagonals in a positional matrix of queens defined by the type Vec (Fin n) n.

```
diagup : ∀{n} → Vec (Fin n) n → Vec ℕ n
diagup {n} positions = (Vec.map
            (λ x → ((toℕ (fst x)) + toℕ (snd x) + 1))
            (zip positions indices))
```

```
diagdown : ∀{n} → Vec (Fin n) n → Vec ℕ n
diagdown {n} positions = (Vec.map
   (λ x → ((n - 1) + toℕ (fst x) - toℕ (snd x)))
     (zip positions indices))
```

And the nqueens models themselves are as follows

```
include "globals.mzn";
int: w;
set of int: dim = 0..(w-1);

array[dim] of var dim: positions;
% No two queens on any one row
constraint alldifferent(positions);
% Diagonals
constraint alldifferent(
                [positions[x] + x | x in dim]
);
constraint alldifferent(
                [positions[x] - x | x in dim]
);
```

Listing 4.1: nQueens1.mzn

which in Agda is

```
q1 : ℕ → Type₀
q1 n = Σ[ v ∈ Vec (Fin n) n ]
         alldifferent v ×
         alldifferent (diagup v) ×
         alldifferent (diagdown v)
```

and

```
int: w;

set of int: dim = 1..w;

array[dim] of var dim: positions;

predicate injective_index(array[dim] of var int: x) =
forall (i, j in dim) (x[i] = x[j] -> i = j);

constraint injective_index(positions);

constraint injective_index(
           [positions[x] + x | x in dim]
);

constraint injective_index(
           [positions[x] - x | x in dim]
);
```

Listing 4.2: nQueens2.mzn

which in Agda is

```
q2 : ℕ → Type₀
q2 n = Σ[ v ∈ Vec (Fin n) n ]
         injective-lookup v ×
         injective-lookup (diagup v) ×
         injective-lookup (diagdown v)
```

Note that the models have the same variable type Vec (Fin n) n). In other words, they only differ in the constraints. This is not required, but makes the equality proof much simpler.

In order to prove the equivalence between these models, we need to prove that alldifferent is equivalent to injective-lookup.

The proof of equality between these models is constructed through transporting through the injective-lookup≡alldifferent equality.

```
q1≡q2 : ∀{n} → q1 n ≡ q2 n
q1≡q2 {n} = λ i → Σ[ v ∈ Vec (Fin n) n ]
                injective-lookup≡alldifferent {v = v} (~ i) ×
                injective-lookup≡alldifferent {v = diagup v} (~ i) ×
                injective-lookup≡alldifferent {v = diagdown v} (~ i)
```

Thus we have proven that the two models of the n-queens problem are equivalent, however the important part is found in the methodology used. We manage to use our framework to define two models of the same problem, and prove an equality between the models through the usage of univalence.

4.2 Symmetry Breaking

When creating models, it is often the case that the set of possible solutions contain several symmetries. As a simple example, one might consider the model

```
pairSum : Type₀
pairSum = Σ[ x ∈ ℕ × ℕ ] (fst x) + (snd x) ≡ 5
```

which models the existence of two natural numbers, with a parameterised sum. However, the two solutions

```
s₁ s₂ : pairSum

s₁ = (1 , 4) , refl

s₂ = (4 , 1) , refl
```

present a symmetry, due to the commutativity of addition. In this case, any solution (a , b) also presents a solution (b , a). By taking advantage of known symmetries, we can lessen the search space by adding additional constraints. For instance, we can ensure that a ≤ b by adding it as an additional constraint.

```
pairSum₂ : Type₀
pairSum₂ = Σ[ x ∈ ℕ × ℕ ] ((fst x) + (snd x) ≡ 5) × ((fst x) ≤ (snd x))
```

This model does not express the previously mentioned symmetry. Generally, symmetries can be eliminated through some constraint. We can represent a symmetry through a projection, by sending all symmetric values onto a chosen candidate. In essence, this relates to the idea of a quotient map.

```
symmetry : (A : Type ℓ) (B : A → Type ℓ') → Type _
symmetry A B = Σ[ f ∈ (A → A) ] ((x : Σ A B) → B (f (fst x)))
```

The symmetry breaking can then be encoded as the constraint that the element is projected onto itself by the symmetry projection.

```
elimSym : ∀{A : Type ℓ} {B : A → Type ℓ'} → symmetry A B → Type (ℓ-max ℓ ℓ')
elimSym {A = A} {B} (f , pf) = Σ[ x ∈ A ] (B x) × (f x ≡ x)
```

In order to show this methodology, we can describe the symmetry in the pairSum model using the symmetry type. Before we can construct this element, we need to define our projection function. For the sake of simplicity of implementation, we define the function using the decidability of the ≤ operator.

```
proj : {a b : ℕ} → Dec (a ≤ b) → ℕ × ℕ
proj {a} {b} (yes _) = a , b
proj {a} {b} (no _) = b , a
```

We can then proceed to define the symmetry

```
pairSumSymmetry : symmetry (ℕ × ℕ) (λ x → (fst x) + (snd x) ≡ 5)

pairSumSymmetry .fst (a , b) = proj (decidable≤ a b)
pairSumSymmetry .snd ((a , b) , p) with decidable≤ a b
... | yes _ = p
... | no _ = +-comm b a • p
```

which we can prove is equivalent to our example symmetry break as follows.

First, we show that a ≤ b is equivalent to the projection function projecting a value onto itself.

```
≤-proj≡ : {a b : ℕ} {p : Dec (a ≤ b)} → a ≤ b → proj p ≡ (a , b)
≤-proj≡ {p = p} pf with p
... | yes q = refl
... | no ¬q = ⊥.rec (¬q pf)

proj≡-≤ : {a b : ℕ} {p : Dec (a ≤ b)} → proj p ≡ (a , b) → a ≤ b
proj≡-≤ {p = yes p} const = p
proj≡-≤ {p = no ¬p} const = 0 , (λ i → snd (const i))

≤≡const : ∀{a b} {p : Dec (a ≤ b)} → (proj p ≡ (a , b)) ≡ (a ≤ b)
≤≡const {a} {b} {p} = isoToPath
  (iso
    proj≡-≤
    ≤-proj≡
    (λ b₁ → m≤n-isProp (proj≡-≤ {p = p} (≤-proj≡ b₁)) b₁)
    λ a₁ → isSetΣ isSetℕ (λ _ → isSetℕ) (proj p) (a , b) (≤-proj≡ (proj≡-≤ a₁)) a₁
  )
```

Which we can use to finally prove the equivalence of the two models.

```
pairSum₂≡elimSym : pairSum₂ ≡ elimSym pairSumSymmetry
pairSum₂≡elimSym i =
 Σ (ℕ × ℕ)
   λ x →
     Σ (fst x + snd x ≡ 5)
       λ x₁ → ≤≡const {a = fst x} {b = snd x} {p = decidable≤ (fst x) (snd x)} (~ i)
```

The two main ways of breaking symmetries in models are through reformulation and through constraints. The latter of which was applied in the example above. The former mothod is more difficult to express using types, and is left as future work.

4.3 Implied Constraints

When modeling, it is often useful to supply the solvers with additional constraints, that are already implied by the other constraints. This is due to the fact that different solvers interact with constraints in different ways, such that some constraints allow better propagation of solution domains. Ideally, since no solutions are added nor removed by providing an implied constraint, the models are equivalent using the methodology presented here.

A good example of this is the magic series puzzle. A magic series of length n is an array of integers such that the element at index i is the number of occurrences of i. It is implied that the sum of the elements of the array is equal to the number n. However, as the proofs end up being fairly complicated for the general case, we will only prove that this condition is satisfied for one solution, to the model instance with $n = 4$.

In order to model the magic condition, we need to count the number of occurrences of an element in an array. This means that there must be an algorithm that decides whether two elements are equal, i.e. they need to be discrete.

```
count : {A : Type ℓ} (ϕ : Discrete A) → Vec A n → A → ℕ
count {A = A} ϕ [] x = 0
count {A = A} ϕ (x₁ ∷ v) x with ϕ x₁ x
... | yes p = suc (count ϕ v x)
... | no ¬p = count ϕ v x
```

Now we can state the magic condition

```
magicCondition : ∀ {n : ℕ} → Vec ℕ n → Type₀
magicCondition {n = n} v
  = ∀(i : Fin n)
    → (count (discreteℕ) v (toℕ i)) ≡ (lookup i v)
```

And a formulation for calculating the sum of a vector of natural numbers.

```
sum : Vec ℕ n → ℕ
sum = foldr _+_ 0
```

We also need to prove that the magic condition is decidable. First, we prove that given a family over Fin n that is decidable for all n, the type representing the quantified conjunction over Fin is decidable.

```
decForallFin : ∀{n} {P : Fin n → Type ℓ}
             → (∀ x → Dec (P x))
             → Dec ((x : Fin n) → P x)

decForallFin {n = zero} _ = yes (λ ())
decForallFin {n = suc n} f with f zero
... | no ¬p = no λ x → ¬p (x zero)
... | yes p with decForallFin (λ x → f (suc x))
... | no ¬q = no (λ x → ¬q (λ x₁ → x (suc x₁)))
... | yes q = yes (λ { zero → p
                     ; (suc x) → q x })
```

Which allows us to prove that the magic condition is decidable.

```
decidableMC : ∀ {n : ℕ} (v : Vec ℕ n)
             → Dec (magicCondition v)
decidableMC v
  = decForallFin (λ x →
       discreteℕ (count discreteℕ v (toℕ x)) (lookup x v)
    )
```

Finally, we can construct the models

```
MS₁ MS₂ : ℕ → Type₀

MS₁ n = Σ[ series ∈ Vec ℕ n ]
            magicCondition series

MS₂ n = Σ[ series ∈ Vec ℕ n ]
            magicCondition series ×
            (sum series ≡ n)
```

And present a solution to each

```
mS₁ : MS₁ 4
mS₁ = s , AutoProof (decidableMC s)
```

```
  where
    s = 1 :: 2 :: 1 :: 0 :: []

mS₂ : MS₂ 4
mS₂ = fst mS₁ , (snd mS₁ , refl)
```

Note that mSol_2 defines the variable value and the proof of the magic condition by directly referencing mSol.

The implied constraint is represented in the second constraint. Since we only provide an example where this holds in the case of our fixed series $[1, 2, 1, 0]$, the operations defined by the sum function simply evaluate, leaving the type of the implied constraint in mSol_2 as $4 \equiv 4$, which is proven with refl. However, it is much harder to prove that this constraint indeed is implied, that it holds for any valid assignment of the variable. We can formalise implied constraints by a dependent type

```
implied : ∀{A : Type ℓ} → (B : A → Type ℓ’) → (A → Type ℓ’) → Type (ℓ-max ℓ ℓ’)
implied {A = A} B C = ∀(x : Σ A B) → (C (fst x))
```

In general, any constraint that is automatically satisfied can be added to a model while still maintaining equality, as long as the constraint on any solution is a mere proposition.

```
implied≡ : ∀{A : Type ℓ} {B : A → Type ℓ’} {C : A → Type ℓ’}
           → implied B C → (∀ x → isProp (C x))
           → Σ A (λ x → B x × C x) ≡ Σ A B

implied≡ {A = A} {B = B} ι prop =
  Σ-cong-snd (λ x →
    ua (Σ-contractSnd
      (λ a →
        inhProp→isContr
          (ι (x , a))
          (prop x))))
```

However, we have not yet shown that sum series ≡ n is an implied constraint; to do that, we need to provide a proof of implied magicCondition (λ x → sum x ≡ n), which is left as an exercise to the reader.

Chapter 5

Conclusion

5.1 Results

We have explained the basics of MLTT, and how it differs from ZFC. Further, we have described univalence as well as the benefits of it. We have introduced Univalence in the form of an axiom for HoTT, and introduced the interval types for CTT, which allows univalence to be proven as a theorem. Using these concepts, we have constructed a new framework for representing a subset of MiniZinc models — containing arrays, natural numbers, ranges as well as the **alldifferent** global constraint. The natural numbers and the ranges are defined using the built-in types $\mathbb{N}$ and Fin n respectively. The variables and constraints are fields of a record type representing the model. The constraints are represented by propositions as types.

A solution to a model is an element of the model type. This element must have all its fields assigned, thus providing values for the variables, and proofs that each constraint is satisfied.

We have provided the basics required for automatic proof of constraints in a solution, through the use of decidable types, by providing algorithms to decide whether or not the constraint holds for given variables.

Through this framework we have defined two different models of the n-queens problem. Further, we have proven that these models are equivalent, and thus equal. Thus we can conclude that we can indeed use CTT to prove model equivalences, at least for simple models.

We have also examined two common ways to alter models in order to reduce search time; symmetry breaking and implied constraints, and concluded that each of these concepts are able to be represented and generalized using this framework.

However, the proofs for this was at times complicated, meaning more work is required in order to make this a practical approach, especially for more complex problems.

5.2 Related Work

The initial idea for the thesis stems from previous work dedicated to using HoTT for proving equivalences of SQL queries. In their work, they extend a previously existing formalisation of SQL's semiring semantics, by extending it with unbound summation and duplicate elimination[5].

They define the U-semiring as a tuple $(\mathcal{U}, 0, 1, +, \times, \| \bullet \|, \text{not}(\bullet), (\Sigma_D) D \in \mathcal{D})$ where

- $(\mathcal{U}, 0, 1, +, \times)$ forms a commutative semiring
- $\| \bullet \|$ — referred to as squash — and not$(\bullet)$ are unary operations satisfying a number of properties.
- $\mathcal{D}$ is a set of sets, where each $D \in \mathcal{D}$ is called a summation domain. For each D, the operation $\Sigma_D : (D \to \mathcal{U}) \to \mathcal{U}$ is called an unbounded summation taking a function $f : D \to \mathcal{U}$ and outputs a value in $\mathcal{U}$. Further, this summation is required to satisfy a number of axioms.

For the specific axioms and properties, refer to the paper[5]. They proceed to define a translation algorithm, for converting a SQL query into an U-expression — an expression using only the operations specified by the structure. These expressions are rewritten to a standard form, at which point they can construct a solver that can automatically prove whether or not queries are equivalent[5].

There is a very close connection between constraint satisfaction problems and databases. In fact, given a constraint satisfaction problem one can create a database encoding this problem, such that the database can act as a solver[10]. This similarity hints at the possibility of using a formalisation similar to this for MiniZinc.

Methodologically, the approaches taken by this thesis differ from the SQL paper. The SQL paper evaluated which rules and operations need to be available for the correct formalisation of SQL queries through the use of previous work. This enabled them to construct their abstractions and then convert from the SQL language to their formal representation. Such groundwork had not been laid for MiniZinc, which required the experimentation of which rules need to be satisfied, and which operations need to be available, leaving little time for the formalisation into an algebraic structure.

5.3 Future Work

Being able to prove equivalences between models is a good start, however much remains to be done in order for this to be an effective method. The main problem with this framework is the amount of manual work required, and can be considered preparation for larger possible work. There are a couple of possible ways to continue this research;

One possibility is to represent the models as elements of some containing model type. This allows for the possibility of decidable equalities. However, as the space of models — even for a simple subset of the MiniZinc language — is undecidable[3], this is likely not possible.

Another possible source of improvement is the automatic reasoning and proof search in Agda, or alternatively using a theorem prover that is more focused on automatic reasoning, for instance Coq.

While the above suggestions both allow for simpler proofs and less manual labour, the main goal is still to produce a complete formalisation of the MiniZinc language. Using similar methodologies to the SQL paper[5], this should be the main focus of future work. A possible future research proposal could be outlined as

- Analyse which operations and rules need to be supported
- Construct an algebraic structure satisfying the above
- Formalise the conversion of MiniZinc code to elements of the structure
- Create a solver for the structure

This work will be a solid starting point, and can be used for analysing which operations and rules need to be supported.

5.4 Analysis

This work has been a big source of learning to me. Both HoTT and constraint programming were subjects I had only briefly looked at prior to writing this thesis. Taking a course in constraint programming taught me all I needed to know for the thesis, while the HoTT-book ([12]) served a valuable source of both methodology, theory and references.

While the initial project plan was to summarize and possibly repeat the previously mentioned SQL-paper ([5]) but for constraint programming, it eventually became clear that repeating it would be too much work, causing this thesis to instead end up serving as preparation for future work.

The hardest part of the thesis was coming to terms with proof assistants. Before eventually settling on Agda and the cubical library, I also became acquainted with Coq and the univalent foundations library. In retrospect, going with a proof assistant with better support for automated proof search – such as Coq – might have been a better choice.

Acronyms

CTT	Cubical Type Theory
HoTT	Homotopy Type Theory
MLTT	Martin-Löf Type Theory
ZFC	Zermelo-Frankel set theory

Bibliography

[1] Andrea Vezzosi Anders Mörtberg. Cubical Agda. Agda Library. URL: https://github.com/Agda/cubical.

[2] Krzysztof R. Apt. Principles of Constraint Programming. Cambridge University Press, 2003.

[3] Christian Bessière et al. “Reasoning about constraint models”. In: PRICAI 2014. Vol. 8862. LNCS. Springer, 2014, pp. 795–808.

[4] Marco Cadoli and Toni Mancini. “Using a theorem prover for reasoning on constraint problems”. In: Applied Artificial Intelligence 21.4–5 (May 2007), pp. 383–404.

[5] Shumo Chu et al. “Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries”. In: Proc. VLDB Endow. 11.11 (2018), pp. 1482–1495. DOI: 10.14778/3236187.3236200. URL: http://www.vldb.org/pvldb/vol11/p1482-chu.pdf.

[6] Cyril Cohen et al. “Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom”. In: FLAP 4.10 (2017), pp. 3127–3170. URL: http://collegepublications.co.uk/ifcolog/?00019.

[7] W. A. Howard. “Per Martin-Löf. Intuitionistic type theory. (Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.) Studies in proof theory. Bibliopolis, Naples1984, ix 91 pp.” In: Journal of Symbolic Logic 51.4 (1986), pp. 1075–1076. DOI: 10.2307/2273925.

[8] Nicholas Nethercote et al. “MiniZinc: Towards a standard CP modelling language”. In: CP 2007. Ed. by Christian Bessière. Vol. 4741. LNCS. The MiniZinc toolchain is available at https://www.minizinc.org. Springer, 2007, pp. 529–543.

[9] Nicholas Nethercote et al. “MiniZinc: Towards a standard CP modelling language”. In: In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming. Springer, 2007, pp. 529–543.

[10] Justin K. Pearson and Peter Jeavons. A survey of tractable constraint satisfaction problems. Tech. rep. CSD-TR-97-15. Computer Science Department, Royal Holloway, University of London, UK, July 1997.

[11] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597.

[12] The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[13] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types". In: Proc. ACM Program. Lang. 3.ICFP (July 2019). DOI: 10.1145/3341691. URL: https://doi.org/10.1145/3341691.