Agent-Based Logics in Dependent Type Theory

Colm Baston

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy, September 2024

University of Nottingham
UK | CHINA | MALAYSIA

Abstract

This thesis is on the formalisation of mathematics in Martin-Löf type theory. This is a class of dependently-typed functional programming languages whose rules form a language suitable for the statement and proof of mathematical theorems. Programs in type theory can be mechanically checked to be well-typed, which corresponds to proofs being checked to be valid. In particular, we explore the semantics of certain agent-based logics, and seek to give them appropriate representations using the concepts of type theory.

We embed the syntax of epistemic modal logic in type theory as predicates over a type of possible worlds. Knowledge operators are defined in this setting by stating a set of properties they must satisfy. We prove this knowledge operator semantics equivalent to the traditional relational semantics of epistemic logic. Common knowledge is then defined in the embedding as a coinductive predicate, whose proofs are infinite data structures. We prove this representation is equivalent to the intuitive definition as iterated universal knowledge, and prove it is equivalent to the relational interpretation. In coalition logic, we represent game forms and playable effectivity functions in type theory, and outline a proof of their equivalence.

Acknowledgements

Foremost, I would like to thank Venanzio Capretta, who agreed to supervise this project and whose knowledge and research expertise have been indispensible ever since. I am also very appreciative of Graham Hutton, who has been very encouraging from the start, but took on the role of my second supervisor midway through, and was of great assistance towards the end of my studies.

More broadly, I am grateful to the past and present members of the Functional Programming Lab, and others from the School of Computer Science, for many insightful discussions over the years. In addition, I greatly enjoyed the undergraduate teaching opportunities offered by the school, which gave a welcome break from research.

Thanks to Nicolai Kraus and Roy Crole for their constructive comments and suggestions in finalising this thesis. Finally, without the support of my parents, Peter and Noëlle, completion would not have been possible.

Contents

Chapter I

Introduction

Martin-Löf type theories are a class of functional programming languages which give strict rules for the construction and use of different *types* of data. For example, a function defined to take a Boolean value as argument cannot be applied to a natural number, and vice versa. While languages with strong type systems are common throughout computer science and software engineering, these most often treat data types and their values as completely separate entities that cannot be mixed. In type theories, however, types are themselves first-class values, able to be stored in data structures, returned by functions, and so on, giving rise to *dependent types* — types which depend on values.

Dependent types have applications in common programming tasks. A type for matrices may depend on natural number values which state their dimensions. A matrix multiplication function can then be defined which only multiplies $a \times b$ and $b \times c$ matrices, with a matching dimension b, and the type of the output matrix is also labelled with its size, $a \times c$. The type system ensures statically, at the time the program is compiled, that all matrix multiplications are applied to appropriately-sized matrices, with no runtime checks necessary.

Simultaneously, through the phenomenon of the Curry-Howard correspondence, Martin-Löf type theory serves as a foundation for mathematics. Types can be seen as mathematical statements, their values as proofs for the statement, with the specific type giving the valid ways of constructing a valid proof of the statement. Dependent types are predicates whose truth depends on the values to which they are applied, and type theory has connectives analogous to the universal and existential quantifiers of a predicate calculus.

There are many implementations of dependently-typed languages, some of which are specifically designed as *proof assistants*, with features that assist in the construction of mathematical proofs. The type-checking apparatus of their compilers double as proof checkers — if a program compiles without error, the corresponding mathematical proofs are validated to be correct.

Even within computer science, mathematics has traditionally been done on pen and paper, new theorems communicated in the literature through prose. While care is taken to ensure constructions are compatible with a mathematical foundation, most often a system of set theory, rarely are proofs rigorously checked by a mechanical process, and mistakes sometimes slip through. In recent years, even pure mathematicians with no previous connection to computer science have turned to type theory as a potential solution to this problem [8, 11].

This thesis is on formalising mathematical results in the language of type theory, with the objective that they can be rigorously verified by a proof assistant. In particular, it explores the semantics of certain agent-based logics, epistemic modal logic and coalition logic, and their representation within type theory.

In epistemic modal logic, an embedding of the logic is defined in the language of type theory, with proof that the embedding corresponds to the traditional semantics. This allows features of the type theory to be used in epistemic logic, for example, the notion of common knowledge is be defined using the feature of coinductive types. In coalition logic, we focus on the formalisation of an equivalence proof between two semantic interpretations of the logic.

1.1 Thesis Structure and Contributions

The thesis aims to be accessible to those with a mathematical background, but with no prior exposure to Martin-Löf type theory, so Chapter II begins with a brief exposition. This is established material which can be pieced together from other sources, but the chapter provides a consistent syntax and an introduction to concepts that will be essential in later chapters. This chapter has a corresponding formalisation in the Agda proof assistant, provided in Appendix A.1, whose definitions are then used as a basis for later formalisations.

The content of Chapters III and IV is based on the following publication [5].

- The Coinductive Formulation of Common Knowledge
  Joint work with Venanzio Capretta
  Published in the Proceedings of the 9th International Conference on Interactive Theorem Proving, Oxford 2018

In Chapter III, we consider *epistemic modal logic*, which adds operators to the syntax of a propositional logic which model the knowledge of agents. The chapter begins with an introduction to the syntax of the logic and its traditional semantics, based on equivalence relations, and goes on to develop a representation for them in type theory.

This contributions of this chapter are as follows.

- An embedding for the syntax of the logic is defined in type theory, based on predicates over a type of possible worlds, Definitions 3.3.2 and 3.3.3.
- A semantic specification for knowledge operators within this embedding, Definition 3.4.2. In particular, this knowledge operator semantics introduces a new, infinitary deduction rule, *preservation of semantic entailment*, Definition 3.4.1, which is proved to subsume two axioms of epistemic logic, Theorems 3.4.3 and 3.4.4.
- Transformation functions between the knowledge operator semantics and the relational semantics, Definitions 3.5.2 and 3.5.4, with proof that the transformations yield the opposite semantics, Theorems 3.5.3 and 3.5.9.
- Proof that these transformations are inverse, forming an isomorphism up to propositional equivalence, Theorems 3.5.10 and 3.5.11.
- A formalisation of these results in the Agda proof assistant, Appendix A.2.

Chapter IV continues the focus on epistemic modal logic, extending it with multiple agents, each with their own knowledge operator. The concept of *common knowledge* is introduced, which exists among groups of agents when a fact is known universally, the fact of universal knowledge is itself known universally, and so on ad infinitum. The semantics of the common knowledge operator is explored in the context of a famous logic puzzle, which involves common knowledge in a subtle way, before going on to define common knowledge in type theory.

The contributions of this chapter are as follows.

- In the setting of the embedding of Chapter III, an operator for common knowledge is defined as a coinductive type, Definition 4.2.5, and is shown to satisfy the knowledge operator properties, Theorem 4.4.7.
- The coinductive common knowledge operator is shown to correspond to the intuitive idea of iterated universal knowledge, Theorems 4.2.10 and 4.2.11.
- The relational interpretation of common knowledge is transformed, using the isomorphism of the previous chapter, into a common knowledge operator, Definition 4.3.2.
- The coinductive and relational versions of the common knowledge operator are proven to coincide, Theorem 4.4.2.
- A formalisation of these results in the Agda proof assistant, Appendix A.3.

The content of Chapter V is based on the following publication [6].

- GAME FORMS FOR COALITION EFFECTIVITY FUNCTIONS
  Joint work with Venanzio Capretta
  Published in the Proceedings of the 25th International Conference on Types for Proofs and Programs, Oslo 2019

In Chapter V, we turn our attention to *coalition logic*, in which agents can act together towards a common set of goals. Coalition logic has two equivalent semantics based on game forms and playable effectivity functions, and an outline of the equivalence proof is given, with the objective of a formalisation of the proof in type theory.

The proof is involved and heavily set-theoretic, posing significant challenges for formalisation in type theory. This was an ambitious aim late into the research period, and the formalisation is not complete enough to be fully-verified by proof assistant. However the chapter proposes solutions to some of these challenges. These include representation of subsets with decidable membership as Boolean functions, Definition 5.3.1, and the use of partial functions to define type families over only a subset of a type at a time, Definition 5.4.2.

Chapter II

Dependent Type Theory

This chapter introduces a formal system of dependent type theory which will be used for the developments in later chapters. The coverage aims to be sufficient to gain an intuition, but not completely comprehensive — it won't be possible to implement a type-theoretic proof assistant based on this chapter alone.

The basic rules of type formation, introduction, and elimination are discussed in the context of a simply-typed λ-calculus. We then go on to say how this system forms a mathematical foundation via the Curry-Howard correspondence and how it differs from systems based on set theory. Dependent types and propositional equality are introduced, giving rise to a system of intuitionistic higher-order logic. Eventually, we will cover inductive types, which allow for the definition of well-founded tree-like structures in the type theory, and coinductive types, which are non-wellfounded trees with potentially infinite depth.

2.1 Background

In the late 19th and into the early 20th century, significant effort was being made to base the previous discoveries of mathematics in a unified foundation of mathematical logic. Foremost among the proposed systems was a theory of sets, where all mathematical structures could be built up from sets, with formal rules for set formation and to determine set membership. However, it was discovered that any such system which includes an unrestricted comprehension principle is inconsistent, a result today known as Russell's paradox — does the set of all sets that are not members of themselves contain itself?

Bertrand Russell proposed his theory of types to avoid the paradox [53]. There is a hierarchy of *types*, with each set being assigned to unique type. Instead of adopting an unrestricted comprehension principle, sets may only be composed of elements of a strictly lower type in the hierarchy. This stratification of types renders the problematic Russell set inexpressable.

The primary system of concern to this thesis is the type theory due to Per Martin-Löf [43, 44]. While it may bear little resemblance to Russell's theory, the idea of types remains an essential concept. Each object is assigned a unique type, and operations may only be performed on objects of appropriate types. While this may seem restrictive when compared to more free-form set theories, a well-designed type system ultimately provides a great amount of utility, as well as desirable computational properties.

Based on Alonzo Church's typed λ-calculus [16] and its later extensions, such as System F [29, 50], Martin-Löf type theory serves as both a dependently-typed functional programming language and a foundation for mathematics. It may be used to implement algorithms, but also to state and prove mathematical statements, all within the same language. Throughout this thesis, "dependent type theory", or simply "type theory", will refer specifically to a system of Martin-Löf type theory.

2.2 Simple Types

The constructions of type theory, called *terms*, are assigned to a unique type from the moment they are introduced and only exist in the context of their type. The language of type theory is defined by *judgements*, the most fundamental of which is the typing relation.

$$\Gamma \vdash t : T \qquad \text{“in context } \Gamma\text{, term } t \text{ has type } T\text{”}$$

The context Γ is included here so that typing judgements may follow from prior assumptions. For the purposes of this introduction, a context may be considered a mapping of variable names to their types: $v_1 : T_1$, …, $v_n : T_n$. When a judgement is asserted unconditionally, with an empty context of assumptions, it is common to omit the context in the notation. For example, when we introduce the type of natural numbers, we may simply write $\mathsf{zero} : \mathbb{N}$.

Before defining other types, we fix a *type universe* Type — a type whose terms are themselves types. The judgement $T : \mathsf{Type}$ is understood to mean that T is a type as well as a term, and it may in turn have its own terms. If we derive $\Gamma \vdash t : T$ without having first established $\Gamma \vdash T : \mathsf{Type}$, this is considered to be an implicit premise.

We define the valid judgements of a type theory using inference rules in the style of natural deduction. Our first types, the empty type $\mathbb{0}$, and the unit type $\mathbb{1}$ containing exactly one term, are defined axiomatically.

$$\mathbb{0}\text{-}\textsc{Form}\;\frac{}{\mathbb{0} : \mathsf{Type}} \qquad \mathbb{1}\text{-}\textsc{Form}\;\frac{}{\mathbb{1} : \mathsf{Type}} \qquad \mathbb{1}\text{-}\textsc{Intro}\;\frac{}{\star : \mathbb{1}}$$

Rules for creating new types are called *formation rules*, while rules for creating new terms of a type are called *introduction rules*. We give a formation rule for each of the new types, but only $\mathbb{1}$ has an introduction rule to define its term $\star$, as $\mathbb{0}$ is empty, uninhabited by terms.

At this point, we could go on to define the Boolean type $\mathbb{2}$, containing two terms in a similar way, with one formation rule and two introduction rules, but it is more convenient to define it in terms of a compound type. We define a binary sum operator on types that forms a new type combining both types' terms, tagged with which of the summed types the term originated.

$$+\text{-}\textsc{Form}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A + B : \mathsf{Type}}$$

$$+\text{-}\textsc{Intro}_{\mathsf{inl}}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}\ a : A + B} \qquad +\text{-}\textsc{Intro}_{\mathsf{inr}}\ \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}\ b : A + B}$$

With this type formation rule, without considering nesting, we can form four new types with each combination of $\mathbb{0}$ and $\mathbb{1}$. Since neither side of $\mathbb{0} + \mathbb{0}$ has any way of constructing a term, this type is uninhabited, but the other types have the following terms.

$$\mathsf{inl}\ \star : \mathbb{1} + \mathbb{0} \qquad \mathsf{inr}\ \star : \mathbb{0} + \mathbb{1} \qquad \mathsf{inl}\ \star : \mathbb{1} + \mathbb{1} \qquad \mathsf{inr}\ \star : \mathbb{1} + \mathbb{1}$$

We will use the type $\mathbb{1} + \mathbb{1}$ as our Boolean type, with the two terms listed above representing true and false. We introduce a new judgement to state when two terms are definitionally equal.

$$\Gamma \vdash a = b : T \qquad \text{“in context } \Gamma\text{, } a \text{ and } b \text{ are equal terms of type } T\text{”}$$

To be useful as a notion of equality, we need basic rules to endow some standard properties on this relation: reflexivity, symmetry, transitivity, and congruence rules, omitted here. These rules ensure that the type theory makes no distinction between the two terms — one can always be used in place of the other.

For convenience, we can add rules to assert $\mathbb{2} = \mathbb{1} + \mathbb{1}$, $\mathsf{true} = \mathsf{inl}\ \star$, and $\mathsf{false} = \mathsf{inr}\ \star$. It is important to note that there is no intrinsic value in these names, they are just syntactic identifiers — we could just as easily define true and false the other way around.

The choice of $\mathbb{1}+\mathbb{1}$ to represent the Boolean type is appropriate because it has exactly two terms. That is, it can be shown that any term of the type constructed in an empty context will be judgementally equal to either $\mathsf{inl}\ \star$ or $\mathsf{inr}\ \star$, and these terms are distinct from one another. This is unsurprising at the moment, as the only way of constructing terms for $\mathbb{1}+\mathbb{1}$ is by using the introduction rules directly, but will remain to be the case as new rules are added to the system. We will later be able to observe this fact from within the type theory, proving it as a theorem using an internal notion of propositional equality introduced in Section 2.5.

A second compound type is the product type. It is defined as another binary operator on types which results in the type of pairs formed by terms of the corresponding types.

$$\times\text{-}\textsc{Form}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \times B : \mathsf{Type}}$$

$$\times\text{-}\textsc{Intro}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a,\, b\rangle : A \times B}$$

As an example, the type $\mathbb{2} \times \mathbb{1}$ consists of pairs where the first component is a Boolean and the second is $\star$ — it has only two terms $\langle \mathsf{true},\, \star\rangle$ and $\langle \mathsf{false},\, \star\rangle$ that can be distinguished by judgemental equality. Similarly, $\mathbb{2} \times \mathbb{2}$ has four terms $\langle \mathsf{true},\, \mathsf{true}\rangle$, $\langle \mathsf{false},\, \mathsf{true}\rangle$, $\langle \mathsf{true},\, \mathsf{false}\rangle$, and $\langle \mathsf{false},\, \mathsf{false}\rangle$, while any product of $\mathbb{0}$ is uninhabited since no terms may be constructed to occupy the corresponding position in the pair.

As the notation suggests, the sum and product types parallel sums and products of arithmetic, and we can form types with any desired, finite number of terms by nesting them. For example, $\mathbb{1} + (\mathbb{1} + \mathbb{1})$ is a type of exactly three terms, up to judgemental equality. This analogy holds when considering the properties of associativity, commutativity, and distributivity — the type $(\mathbb{1} + \mathbb{1}) + \mathbb{1}$ also has three terms. When including sums of $\mathbb{0}$, there are infinitely many ways to form *equivalent* versions of each finite type.

These different constructions, though equivalent, are considered distinct types with distinct terms — the judgement $\mathbb{1} + (\mathbb{1} + \mathbb{1}) = (\mathbb{1} + \mathbb{1}) + \mathbb{1}$ cannot be derived, the term $\mathsf{inr}\ (\mathsf{inl}\ \star)$ has the left type but not the right, and so on. For this reason, the type theory presented here is called *intensional*, as opposed to *extensional*. This distinction will be revisited, and a precise definition of type equivalence given, alongside the introduction of propositional equality in Section 2.5.

We've seen how to construct some basic types, but we can do little else with them at present. The function type is at the core of how we introduce computation into the system. Like sums and products, it is a binary operator on types.

$$\to\text{-}\textsc{Form}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \to B : \mathsf{Type}}$$

$$\to\text{-}\textsc{Intro}\ \frac{\Gamma, v : A \vdash b : B}{\Gamma \vdash \lambda(v : A).\ b : A \to B}$$

The premise of the introduction rule shows the first non-trivial usage of contexts — $\Gamma, v : A$ denotes the context formed by extending Γ with an additional assumption. Term b is allowed to have occurrences of variable v within its substructure, so the judgement $b : B$ is conditional on the assumption of $v : A$. In the conclusion of the rule, the assumption is discharged and term $\lambda(v : A).\ b$ is now a function of variable v, resulting in term b. For notational convenience, we may sometimes drop the type of the bound variable where it can be inferred and would provide no additional clarity, simply writing $\lambda v.\ b$.

Strictly speaking, all functions in type theory are unary, taking exactly one input and producing exactly one output. Functions of higher arity may be simulated using products or, as is more common in functional programming, by *Currying* [21] — returning another function which is responsible for handling the next argument. For example, the types $A \times B \to C$ and $A \to (B \to C)$ both represent binary functions, the latter being a Curried version of the former.

To apply a function to an argument, we add an *elimination rule* to the type theory. Elimination rules specify how terms of a particular type may be used in computation, complementing the introduction rules for that type.

$$\to\text{-}\textsc{Elim}\ \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

Application of the function f to argument a is denoted here by juxtaposition in the tradition of the λ-calculus and functional programming. The actual result of the function application is specified as a judgemental equality, the *β-reduction* rule.

$$\beta\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash (\lambda(v : A).\ b) : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(v : A).\ b)\ a = b[v/a] : B}$$

In the rule's conclusion, $b[v/a]$ denotes syntactic substitution of all free occurrences of the variable v by term a inside the substructure of b. We shall omit the full intricacies of α-conversion and variable capture here. It is sufficient to say that the choice of variable name is arbitrary, and we identify λ-terms which only differ by the names of their bound variables. If free variables in a would be captured by naïve substitution of $b[v/a]$, we first perform α-conversion — all free occurrences of v in b are renamed such that they are not free in a.

An η-reduction rule is also defined. This states that a λ-term which only passes its argument through to another function may be identified with that function.

$$\eta\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash (\lambda(v : A).\ f\ v) = f : A \to B}$$

Furthering the arithmetic analogy, function types are exponentials. Interpreting types A and B as cardinal numbers, the function type $A \to B$ is inhabited by B^A functions that can be distinguished pointwise. For example, there are no functions $\mathbb{1} \to \mathbb{0}$, one function $\mathbb{0} \to \mathbb{1}$, four functions $\mathbb{2} \to \mathbb{2}$ — the identity, a constant function for each of true and false, and Boolean negation — and so on. The rules given so far allow us to define some of these Boolean functions.

$$
\begin{aligned}
\mathsf{id}_{\mathbb{2}} &= \lambda b.\ b &&: \mathbb{2} \to \mathbb{2} \\
\mathsf{const}_{\mathsf{true}} &= \lambda b.\ \mathsf{true} &&: \mathbb{2} \to \mathbb{2} \\
\mathsf{const}_{\mathsf{false}} &= \lambda b.\ \mathsf{false} &&: \mathbb{2} \to \mathbb{2}
\end{aligned}
$$

We cannot yet define Boolean negation as we can only interact with the function argument generically — we cannot do one thing when the argument is true and a different thing when it is false. To remedy this, we add elimination and reduction rules to our basic types. First, we give elimination rules for $\mathbb{0}$ and $\mathbb{1}$.

$$
\mathbb{0}\text{-}\textsc{Elim}\ \frac{\Gamma \vdash e : \mathbb{0}}{\Gamma \vdash \mathsf{elim}_{\mathbb{0}}\ e : T}
\qquad\qquad
\mathbb{1}\text{-}\textsc{Elim}\ \frac{\Gamma \vdash t : T \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash \mathsf{elim}_{\mathbb{1}}\ t\ u : T}
$$

In each case, these say how to transform an element of the type into a term of an arbitrary type T. For $\mathbb{1}$, we must first be given a $t : T$, but for $\mathbb{0}$ we allow this without restriction as it should not be possible to construct an $e : \mathbb{0}$ in an empty context. If we are able to derive $e : \mathbb{0}$, it is only because it is implied by Γ, such as in the case of a function whose domain is $\mathbb{0}$. To illustrate this, we give a derivation for the function $\lambda e.\ \mathsf{elim}_{\mathbb{0}}\ e : \mathbb{0} \to \mathbb{1}$ which uses this rule.

$$
\to\text{-}\textsc{Intro}\ \dfrac{\mathbb{0}\text{-}\textsc{Elim}\ \dfrac{\dfrac{\mathbb{1}\text{-}\textsc{Form}\ \dfrac{}{\mathbb{1} : \mathsf{Type}}}{e : \mathbb{0} \vdash \mathbb{1} : \mathsf{Type}} \qquad \dfrac{}{e : \mathbb{0} \vdash e : \mathbb{0}}}{e : \mathbb{0} \vdash \mathsf{elim}_{\mathbb{0}}\ e : \mathbb{1}}}{\lambda e.\ \mathsf{elim}_{\mathbb{0}}\ e : \mathbb{0} \to \mathbb{1}}
$$

This derivation uses two unlabelled rules which haven't been discussed. One is context weakening, the other an identity axiom for when a concluded typing judgement is already assumed in the context.

For the reasons outlined above, there is no reduction rule for $\mathbb{0}$. We need to give one rule for for $\mathbb{1}$ as it has only one term, simply reducing to the provided value $t : T$ when applied $\star$. The rule therefore does no meaningful work, simply acting as a constant function returning t.

$$
\mathbb{1}\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{elim}_{\mathbb{1}}\ t\ \star = t : T}
$$

The sum type $A + B$ has two cases, one when the term is of the form $\mathsf{inl}\ a$, carrying term $a : A$, and the other for $\mathsf{inr}\ b$, where $b : B$. The elimination and reduction rules describe how we can compute a value of type T in either case.

$$\textsc{+-Elim}\ \frac{\Gamma \vdash f : A \to T \qquad \Gamma \vdash g : B \to T \qquad \Gamma \vdash s : A + B}{\Gamma \vdash \mathsf{elim}_{+}\ f\ g\ s : T}$$

$$\textsc{+-Reduce}_{\mathsf{inl}}\ \frac{\Gamma \vdash f : A \to T \qquad \Gamma \vdash g : B \to T \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{elim}_{+}\ f\ g\ (\mathsf{inl}\ a) = f\ a : T}$$

$$\textsc{+-Reduce}_{\mathsf{inr}}\ \frac{\Gamma \vdash f : A \to T \qquad \Gamma \vdash g : B \to T \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{elim}_{+}\ f\ g\ (\mathsf{inr}\ b) = g\ b : T}$$

To compute a value of type T from $A + B$, we provide a function f to handle the inl case and a function g to handle inr. One reduction rule is specified for each case, applying the appropriate function to the carried term.

Recall that we defined $\mathbb{2} = \mathbb{1} + \mathbb{1}$, $\mathsf{true} = \mathsf{inl}\ \star$, and $\mathsf{false} = \mathsf{inr}\ \star$. We can therefore use elim_{+} and $\mathsf{elim}_{\mathbb{1}}$ to define functions which can discriminate the Boolean value passed to them, such as Boolean negation.

$$\mathsf{not}_{\mathbb{2}} = \mathsf{elim}_{+}\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{inr}\ \star))\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{inl}\ \star)) : \mathbb{2} \to \mathbb{2}$$

The first function will handle the true case, converting inl into inr, and vice versa for the second function which handles false. Note that using the η-reduction rule, elim_{+} is treated here as a Curried function, being partially-applied to only two of its arguments, leaving the final Boolean argument implicit.

The terms of the product type $A \times B$ may only be of the form $\langle a,\ b\rangle$, where $a : A$ and $b : B$. We therefore provide the eliminator $\mathsf{elim}_{\times}$ with a single Curried function to handle both a and b at once.

$$\textsc{$\times$-Elim}\ \frac{\Gamma \vdash f : A \to B \to T \qquad \Gamma \vdash p : A \times B}{\Gamma \vdash \mathsf{elim}_{\times}\ f\ p : T}$$

$$\textsc{$\times$-Reduce}\ \frac{\Gamma \vdash f : A \to B \to T \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{elim}_{\times}\ f\ \langle a,\ b\rangle = f\ a\ b : T}$$

For example, we can now define projection functions to extract just one part of the pair, or a function to swap the components of a pair. These examples assume a context Γ including $A : \mathsf{Type}$ and $B : \mathsf{Type}$.

$$\begin{aligned}
\Gamma \vdash \quad \pi_1 &= \mathsf{elim}_{\times}\ (\lambda a.\ \lambda b.\ a) &&: A \times B \to A \\
\Gamma \vdash \quad \pi_2 &= \mathsf{elim}_{\times}\ (\lambda a.\ \lambda b.\ b) &&: A \times B \to B \\
\Gamma \vdash \mathsf{swap} &= \mathsf{elim}_{\times}\ (\lambda a.\ \lambda b.\ \langle b,\ a\rangle) &&: A \times B \to B \times A
\end{aligned}$$

The rules introduced up to now constitute a simply-typed λ-calculus extended with sums and products. We will now observe some important computational properties exhibited by the system.

The judgemental equality relation presented here is symmetric — it can be applied in either direction. However, it is useful to consider a one-way reduction relation which simplifies the term at each step in order to compute a final result. We can form such a relation by taking the reduction rules of the system and substituting $=$ by $\rightsquigarrow$. In each rule, the equation has been arranged so the left-to-right direction represents a simplification step.

We don't imbue $\rightsquigarrow$ with the equivalence relation properties of equality but do allow congruence rules, so subterms may be reduced without affecting outer parts of an expression. We also define $\rightsquigarrow^*$ as the reflexive, transitive closure, representing a sequence containing any number of individual reduction steps. A term with no possible reductions is called a *normal form*, and the process of applying reductions to reach a normal form is called *normalisation*.

For example, take the example expression $\mathsf{swap}\ \langle \mathsf{not}_{\mathbb{2}}\ \mathsf{true}, \star\rangle : \mathbb{1} \times \mathbb{2}$ which uses some of our previous definitions. Here are two reductions for this term, one first reducing swap, the other first reducing $\mathsf{not}_{\mathbb{2}}$.

$$\begin{array}{llcll}
 & \mathsf{swap}\ \langle \mathsf{not}_{\mathbb{2}}\ \mathsf{true}, \star\rangle & \qquad & & \mathsf{swap}\ \langle \mathsf{not}_{\mathbb{2}}\ \mathsf{true}, \star\rangle \\
\rightsquigarrow^* & \langle \star, \mathsf{not}_{\mathbb{2}}\ \mathsf{true}\rangle & & \rightsquigarrow^* & \mathsf{swap}\ \langle \mathsf{false}, \star\rangle \\
\rightsquigarrow^* & \langle \star, \mathsf{false}\rangle & & \rightsquigarrow^* & \langle \star, \mathsf{false}\rangle \\
= & \langle \star, \mathsf{inr}\ \star\rangle & & = & \langle \star, \mathsf{inr}\ \star\rangle
\end{array}$$

This illustrates the property of *confluence* — when two different reduction steps are possible from the same expression, there will always be subsequent reductions which lead both forks back together. The choice of which reduction to use gives rise to reduction strategies such as normal-order and applicative-order reduction, and the concept of lazy evaluation as is implemented in the functional programming language Haskell [42].

Furthermore, the system is *strongly normalising* — the reduction relation is wellfounded, so a normal form will always be reached in a finite number of steps. This is a consequence of only allowing well-typed terms, which prohibits divergent terms that are possible in the untyped λ-calculus, such as the term $(\lambda\omega.\ \omega\ \omega)\ (\lambda\omega.\ \omega\ \omega)$, which reduces to itself in a single step. In a typed setting, however, the self-application $\omega\ \omega$ is forbidden as it cannot be assigned a type.

Additionally, normal forms exhibit *canonicity* — for each type former excluding functions, closed normal forms are of canonical form. That is, when a term can be constructed in an empty context, its normal form will have one of the syntactic patterns defined by the introduction rules for its type, called its constructors. Since $\mathbb{0}$ has no introduction rules, and therefore no constructors, it also has no canonical forms, while $\mathbb{1}$ has a single constructor and canonical form $\star$. A sum's normal forms will always be constructed by one of inl or inr, and a product's by the pairing constructor $\langle \_, \_ \rangle$ — the subterms carried by the constructors will themselves be of canonical form with respect to their own types.

Finally, the typing and equality judgements are *decidable* — whether a judgement is derivable can be definitively determined by a mechanical process. This is essential for practical implementations of type theories as programming languages, where a type-checking algorithm needs to be able to decide whether a given program is valid.

2.3 Type Theory as a Logic

The Curry-Howard correspondence is the observation of the intersection between formal systems of computation and logic. Indeed, the computational calculus of the previous section was defined using inference rules in the style of natural deduction, a proof calculus. More than just an analogy, the correspondence implies that certain computational systems may themselves serve as proof systems.

Our simply-typed λ-calculus coincides with a system of intuitionistic propositional logic. Logical formulae are interpreted as types, with their proofs being the terms of the corresponding types. When a type is inhabited by terms, the corresponding formula may be seen as true, or provable, while uninhabited types correspond to false, or unprovable, formulae. The following table summarises the type-theoretic interpretations for each of the logical constants and connectives of propositional logic.

	Logical Syntax	Type Theory Syntax
False Constant	$\bot$	$\mathbb{0}$
True Constant	$\top$	$\mathbb{1}$
Disjunction	$P \vee Q$	$P + Q$
Conjunction	$P \wedge Q$	$P \times Q$
Implication	$P \Rightarrow Q$	$P \to Q$
Negation	$\neg P$	$P \to \mathbb{0}$
Equivalence	$P \Leftrightarrow Q$	$(P \to Q) \times (Q \to P)$

The type $\mathbb{0}$ is uninhabited, by definition, corresponding to the fact that no proofs of $\bot$ exist. The $\mathbb{0}$-eliminator doubles as the logical principle of explosion, "*ex falso quodlibet*". Conversely, $\mathbb{1}$ has the constructor $\star$, valid in any context, indicating that a proof of $\top$ may be trivially given at any time. The $\mathbb{1}$-eliminator corresponds to the fact that assuming $\top$ as a premise gives a logical argument no additional strength.

Disjunctions are interpreted as sum types — to prove $P \vee Q$, it suffices to prove either P, using the inl constructor, or Q, using inr. For a conjunction $P \wedge Q$, proofs for both P and Q must be given, collected into a pair. Implications $P \Rightarrow Q$ are functions, transforming a proof of P into a proof of Q — the function elimination rule is the logical *modus ponens* rule.

Let us demonstrate some of the laws of propositional logic and their proofs in type theory. We assume a context with arbitrary types $P : \mathsf{Type}$, $Q : \mathsf{Type}$, and $R : \mathsf{Type}$ for the rest of this section.

First, some basic properties of implication, the *law of identity*, $P \Rightarrow P$, states that implication is a reflexive relation. In addition, implication transitivity is the fact that implications $P \Rightarrow Q$ and $Q \Rightarrow R$ may be chained to conclude $P \Rightarrow R$.

Theorem 2.3.1. *Reflexivity, $P \to P$.*

Proof. $\lambda(p : P).\ p$

We are proving an implication by defining a function — we use a λ-term to bind the premise of the implication to a variable, $p : P$, assuming it as a hypothetical. As the conclusion of the implication is also P, we can simply return the assumption p as the result of the function. The term is both a proof that implication is reflexive, but also the identity function specialised to type P. $\square$

Theorem 2.3.2. *Transitivity, $(P \to Q) \to (Q \to R) \to P \to R$.*

Proof. $\lambda(f : P \to Q).\ \lambda(g : Q \to R).\ \lambda(p : P).\ g\ (f\ p)$

Assume hypotheses $f : P \to Q$, $g : Q \to R$, and $p : P$, applying the two functions in turn to conclude $g\ (f\ p) : R$. This proof term corresponds to a higher-order function which composes two argument functions, $g \circ f$. $\square$

Next, the *law of non-contradiction* asserts a formula cannot be both true and false simultaneously, $\neg(P \wedge \neg P)$, a principle essential for any consistent proof system.

Theorem 2.3.3. *Non-contradiction, $(P \times (P \to \mathbb{0})) \to \mathbb{0}$.*

Proof. $\mathsf{elim}_\times\ (\lambda(p : P).\ \lambda(np : P \to \mathbb{0}).\ np\ p)$

We are defining a function whose domain is a product, $P \times (P \to \mathbb{0})$, so we apply $\mathsf{elim}_\times$ to decompose it, providing a function to handle the two components of the pair. We bind the those components using λ-terms, so now the context includes $p : P$ and $np : P \to \mathbb{0}$ with the goal of concluding $\mathbb{0}$. This is achieved by applying the function np to p. $\square$

DeMorgan's laws are good examples which use both product and sum types to represent conjunctions and disjunctions, respectively.

Theorem 2.3.4. *DeMorgan Law,* $(P + Q \to \mathbb{0}) \to (P \to \mathbb{0}) \times (Q \to \mathbb{0})$.

Proof. $\lambda(h : P + Q \to \mathbb{0}).\ \langle\lambda(p : P).\ h\ (\mathsf{inl}\ p),\ \lambda(q : Q).\ h\ (\mathsf{inr}\ q)\rangle$

Assume $h : P{+}Q \to \mathbb{0}$, and set out to prove both $P \to \mathbb{0}$ and $Q \to \mathbb{0}$, collecting the proofs into a pair. In each case, we bind the hypothesis to a variable with a λ-term and are then to prove $\mathbb{0}$. This is the conclusion of h, so we can apply it to a value of type $P + Q$ to obtain the desired result — we have $p : P$ in the first case and $q : Q$ in the second, so we use terms $\mathsf{inl}\ p$ and $\mathsf{inr}\ q$, respectively. □

Two of the remaining three DeMorgan laws are provable. We give their proof terms here for completeness, but without full explanation.

Theorem 2.3.5. *DeMorgan Law,* $(P \to \mathbb{0}) \times (Q \to \mathbb{0}) \to P + Q \to \mathbb{0}$.

Proof. $\mathsf{elim}_\times\ (\lambda(np : P \to \mathbb{0}).\ \lambda(nq : Q \to \mathbb{0}).\ \mathsf{elim}_+\ np\ nq)$ □

Theorem 2.3.6. *DeMorgan Law,* $(P \to \mathbb{0}) + (Q \to \mathbb{0}) \to P \times Q \to \mathbb{0}$.

Proof. $\mathsf{elim}_+\ (\lambda(np : P \to \mathbb{0}).\ \mathsf{elim}_\times\ (\lambda(p : P).\ \lambda(q : Q).\ np\ p))$
$(\lambda(nq : Q \to \mathbb{0}).\ \mathsf{elim}_\times\ (\lambda(p : P).\ \lambda(q : Q).\ nq\ q))$ □

However, the final DeMorgan law, $(P{\times}Q \to \mathbb{0}) \to (P \to \mathbb{0}){+}(Q \to \mathbb{0})$, cannot be proven in our system. The reason is that the system defined so far represents an intuitionistic propositional logic whose proofs are constructive. When we come to prove the conclusion $(P \to \mathbb{0}) + (Q \to \mathbb{0})$, we must choose to either prove the left side with inl or the right side with inr. However, the types P and Q are arbitrary variables which can represent any concrete type — we don't have sufficient information about them to be able to choose a side.

Similarly, the classical law of excluded middle, $P \vee \neg P$, and the equivalent double-negation elimination principle, $\neg\neg P \Rightarrow P$, are not intuitionistically valid. If excluded middle held in general, the properties of strong normalisation and canonicity guarantee that such a proof would ultimately reduce to inl carrying a proof of P, or inr carrying a proof of $\neg P$. In cases where $P + (P \to \mathbb{0})$ can be proven for some type P, we call P *decidable*, however it is well-known due to Gödel's incompleteness theorems that there will be undecidable propositions in any sufficiently-powerful proof system. If classical reasoning is desired, excluded middle may be postulated by including a term $\mathsf{exMid} : P + (P \to \mathbb{1})$ in the context without any defining equations, but the system will lose canonicity as exMid cannot be reduced into its constructors.

Strong normalisation is also important to ensure logical consistency. For example, if we allow general recursive definitions as exist in many practical functional programming languages, we lose strong normalisation. This would enable construction of a looping term to inhabit any type, $\Omega = \Omega : \mathbb{0}$, thereby proving any statement. In Section 2.6, we shall introduce a restricted form of recursion which preserves the wellfoundedness of the reduction relation.

The decidability of the typing and equality judgements allow type theories which correspond to logical systems to be implemented programmatically. This gives rise to type-theoretic *proof assistants* — software systems which provide a language for stating and proving theorems, as well as guiding the user while constructing proofs by showing the current proof goal, the assumptions available in the context, and so on. When the type-checking apparatus of the software decides the constructions of the program to be well-typed, this corresponds to the proofs being checked to be valid.

Some proof assistants, such as *Coq* [7] and *Lean* [22], provide a language of *tactics*, whose terms resemble intuitive proof steps, for example, “assume the premise as hypothesis h” and “split h into its possible cases”. Internally, these tactics are transformed into terms of the λ-calculus they are based on, and then type checking proceeds as normal. In other proof assistants, such as *Agda* [48], proofs are written directly in the primary language, but they implement language servers to provide interactive feedback during proof construction. *Idris* [9] is another implementation of type theory which may function as a proof assistant, but is more suited for leveraging the type system in practical programming tasks rather than proving mathematical theorems. Many of the results in this thesis have been formalised in Agda, with source code listings available in Appendix A.

2.4 Dependent Types

The simply-typed λ-calculus presented up to now isn’t truly a Martin-Löf type theory as it lacks *dependent types*, allowing types to depend on values. Types aren’t currently first-class entities, the rules defining the system enforcing a strict demarcation between types and their values — they cannot be stored in pairs, or returned from functions, and so on. This is because the type universe Type hasn’t itself been given a type.

It is tempting give a rule stating $\mathsf{Type} : \mathsf{Type}$, but this leads to a logical inconsistency known as Girard’s paradox [30, 36], analogous to Russell’s paradox in set theory. To solve this, the idea of stratification is borrowed from Russell’s theory of types [44] — there is an infinite, enumerable hierarchy of type universes, each assigned the type of the next.

$$\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \mathsf{Type}_3 : \mathsf{Type}_4 : \ldots$$

When multiple universe levels are used in the formation of a type, the result is at the universe level that is their least upper bound. We won't always be explicit about the levels of type universes in future constructions. That is, $T : \mathsf{Type}$, without specifying an index, is taken to be *universe polymorphic*, valid at any universe level. Most often in this thesis, the lowest universe Type_0 will suffice, but it will be highlighted when a higher level is required.

A function whose codomain is a type universe is called a *type family*. Given $A : \mathsf{Type}$ and $B : A \to \mathsf{Type}$, the result when applying type family B to a term $a : A$ is a type which may depend on the precise value of a. For example, we can create a type family $\mathsf{isTrue} : \mathbb{2} \to \mathsf{Type}$ which is inhabited when applied to true, but uninhabited for false.

$$\mathsf{isTrue} = \mathsf{elim}_{+}\ (\mathsf{elim}_{\mathbb{1}}\ \mathbb{1})\ (\mathsf{elim}_{\mathbb{1}}\ \mathbb{0}) : \mathbb{2} \to \mathsf{Type}$$

The logical interpretation of type families is that they correspond to predicates and relations. In the above case, $\mathsf{isTrue}\ \mathsf{true}$ reduces to $\mathbb{1}$ and so may be trivially proven by $\star$, while $\mathsf{isTrue}\ \mathsf{false}$ reduces to $\mathbb{0}$ and cannot be proven in an empty context.

We now introduce Π-types and Σ-types, utilising type families to define type-level connectives which bind terms. The Π-type can be seen as a dependent version of the function type, where the type of the output may depend on the value of the input. The following rules subsume those given for non-dependent functions.

$$\Pi\text{-}\textsc{Form}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, a : A \vdash B\ a : \mathsf{Type}}{\Gamma \vdash \Pi_{(a:A)}\ B\ a : \mathsf{Type}}$$

$$\Pi\text{-}\textsc{Intro}\ \frac{\Gamma, a : A \vdash b : B\ a}{\Gamma \vdash \lambda(a : A).\ b : \Pi_{(a:A)}\ B\ a}$$

$$\Pi\text{-}\textsc{Elim}\ \frac{\Gamma \vdash f : \Pi_{(a:A)}\ B\ a \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\ a}$$

$$\beta\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash (\lambda(v : A).\ b) : \Pi_{(a:A)}\ B\ a \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(v : A).\ b)\ a = b[v/a] : B\ a}$$

$$\eta\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash f : \Pi_{(a:A)}\ B\ a}{\Gamma \vdash (\lambda(v : A).\ f\ v) = f : \Pi_{(a:A)}\ B\ a}$$

The non-dependent function type can be recovered using a constant type family which does not depend on the input value. We may denote this by not using the variable bound by the Π-type former, $A \to B = \Pi_{(a:A)}\ B : \mathsf{Type}$.

Intuitively, the type $\Pi_{(a:A)}\ B\ a$ represents the product of B indexed by all terms $a : A$. For example, $\Pi_{(b:2)}\ B\ b$ is equivalent to $B\ \mathsf{true} \times B\ \mathsf{false}$. In the Curry-Howard correspondence, this is the interpretation of set-theoretic universal quantification, $\forall_{(a \in A)}\ B(a)$ — for any term of $a : A$, the function will provide a corresponding proof of $B\ a$ specialised to the input value.

One practical use of Π-types is defining *parametrically polymorphic* functions, that is, functions which operate uniformly on any type. For example, we've previously seen identity functions specialised to individual types, such as id_2. Similarly, we defined projection functions π_1 and π_2 for product types, however a context including types A and B was assumed. We can now define these functions universally, in an empty context — the argument types are parametrised by dependent functions.

$$\begin{aligned}
\mathsf{id} &= \lambda(A : \mathsf{Type}).\ \lambda a.\ a && : \Pi_{(A:\mathsf{Type})}\ A \to A \\
\pi_1 &= \lambda(A : \mathsf{Type}).\ \lambda(B : \mathsf{Type}).\ \mathsf{elim}_\times\ (\lambda a.\ \lambda b.\ a) && : \Pi_{(A:\mathsf{Type})}\ \Pi_{(B:\mathsf{Type})}\ A \times B \to A \\
\pi_2 &= \lambda(A : \mathsf{Type}).\ \lambda(B : \mathsf{Type}).\ \mathsf{elim}_\times\ (\lambda a.\ \lambda b.\ b) && : \Pi_{(A:\mathsf{Type})}\ \Pi_{(B:\mathsf{Type})}\ A \times B \to B
\end{aligned}$$

It is common to leave some of the parameters implicit in cases where they provide no extra clarity. In addition, we may place a function's parameters at the left side of a definition with the function being applied to them, writing $f = \lambda v.\ b$ as $f\ v = b$. For example, function composition can be defined generically without explicitly writing the Π-types or λ-terms that bind $A : \mathsf{Type}$, $B : \mathsf{Type}$, and $C : \mathsf{Type}$.

$$f \circ g = \lambda a.\ f\ (g\ a) : (B \to C) \to (A \to B) \to A \to C$$

The Σ-type is a dependent version of the product type, where the type of second element of the pair is allowed to depend on the value of the first. These rules subsume those previously given for products.

$$\Sigma\text{-}\textsc{Form}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, a : A \vdash B : A \to \mathsf{Type}}{\Gamma \vdash \Sigma_{(a:A)}\ B\ a : \mathsf{Type}}$$

$$\Sigma\text{-}\textsc{Intro}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B\ a}{\Gamma \vdash \langle a,\ b\rangle : \Sigma_{(a:A)}\ B\ a}$$

Instead of the Σ-eliminator describing how to compute a value of some arbitrary $T : \mathsf{Type}$, we allow the resulting type to depend on the value of the pair using a type family $T : (\Sigma_{(a:A)}\ B\ a) \to \mathsf{Type}$. The function f provided to operate on the pair's elements therefore needs to have a Π-type.

$$\Sigma\text{-}\textsc{Elim}\ \frac{\Gamma \vdash f : \Pi_{(a:A)}\ \Pi_{(b:B\ a)}\ T\ \langle a,\ b\rangle \qquad \Gamma \vdash p : \Sigma_{(a:A)}\ B\ a}{\Gamma \vdash \mathsf{elim}_\Sigma\ f\ p : T\ p}$$

$$\Sigma\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash f : \Pi_{(a:A)}\ \Pi_{(b:B\ a)}\ T\ \langle a,\, b\rangle \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B\ a}{\Gamma \vdash \mathsf{elim}_{\Sigma}\ f\ \langle a,\, b\rangle = f\ a\ b : T\ \langle a,\, b\rangle}$$

As with Π-types, a non-dependent product type may be recovered using a constant type family, $A \times B = \Sigma_{(a:A)}\ B : \mathsf{Type}$. As Π-types represent indexed products, Σ-types represent indexed sums. This is the interpretation of set-theoretic existential quantification, $\exists_{(a \in A)}\ B(a)$, though note that these existence proofs must be constructive.

Due to canonicity, a proof of $\Sigma_{(a:A)}\ B\ a$ reduces to a pair $\langle a,\, b\rangle$ where $a : A$ is a witness to the existential. For example, $\langle \mathsf{true},\, \star\rangle$ is a proof for $\Sigma_{(b:\mathbb{2})}\ \mathsf{isTrue}\ b$, with true witnessing the proof of $\star : \mathsf{isTrue}\ \mathsf{true}$. This rules out existence proofs by contradiction as they are an instance of the classical double-negation elimination principle, $\neg\neg(\exists_{a \in A}\ B(a)) \Rightarrow \exists_{a \in A}\ B(a)$, which doesn't hold constructively.

There can be some confusion caused by the name "dependent product". It is most often used to refer to the indexed products that are Π-types, but sometimes mistaken to mean the dependent generalisation of product types that are Σ-types. Throughout this thesis, this label isn't used, instead opting for "dependent function" for Π-types and "dependent pair" for Σ-types. The type of non-dependent pairs will remain the "product type".

We replace the non-dependent elimination and reduction rules of the other basic types with dependent versions which output in a type family indexed by their terms. As with the other rule upgrades, the originals can be reconstructed by instantiating T with constant type families.

$$\mathbb{0}\text{-}\textsc{Elim}\ \frac{\Gamma \vdash e : \mathbb{0}}{\Gamma \vdash \mathsf{elim}_{\mathbb{0}}\ e : T\ e}$$

$$\mathbb{1}\text{-}\textsc{Elim}\ \frac{\Gamma \vdash t : T\ \star \qquad \Gamma \vdash u : \mathbb{1}}{\Gamma \vdash \mathsf{elim}_{\mathbb{1}}\ t\ u : T\ u} \qquad \mathbb{1}\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash t : T\ \star}{\Gamma \vdash \mathsf{elim}_{\mathbb{1}}\ t\ \star = t : T\ \star}$$

$$+\text{-}\textsc{Elim}\ \frac{\begin{array}{c}\Gamma \vdash f : \Pi_{(a:A)}\ T\ (\mathsf{inl}\ a) \\ \Gamma \vdash g : \Pi_{(b:B)}\ T\ (\mathsf{inr}\ b) \qquad \Gamma \vdash s : A + B\end{array}}{\Gamma \vdash \mathsf{elim}_{+} f\ g\ s : T\ \sigma}$$

$$+\text{-}\textsc{Reduce}_{\mathsf{inl}}\ \frac{\begin{array}{c}\Gamma \vdash f : \Pi_{(c:A)}\ T\ (\mathsf{inl}\ c) \\ \Gamma \vdash g : \Pi_{(b:B)}\ T\ (\mathsf{inr}\ b) \qquad \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathsf{elim}_{+}\ f\ g\ (\mathsf{inl}\ a) = f\ a : T\ (\mathsf{inl}\ a)}$$

$$+\text{-}\textsc{Reduce}_{\mathsf{inr}}\ \frac{\begin{array}{c}\Gamma \vdash f : \Pi_{(a:A)}\ T\ (\mathsf{inl}\ a) \\ \Gamma \vdash g : \Pi_{(c:B)}\ T\ (\mathsf{inr}\ c) \qquad \Gamma \vdash b : B\end{array}}{\Gamma \vdash \mathsf{elim}_{+}\ f\ g\ (\mathsf{inr}\ b) = g\ b : T\ (\mathsf{inr}\ b)}$$

The dependent eliminators are more powerful reasoning principles than their non-dependent counterparts. Rather than just allowing for computation with the types, these eliminators facilitate proving predicates over them by case analysis. Like the non-dependent case, the $\mathbb{0}$-eliminator proves any proposition from an inconsistent assumption. To prove a property over $\mathbb{1}$, it suffices to consider the $\star$ case, while a sum has the cases of inl and inr to consider. The reduction rules are as before, with updated types.

With dependent types providing connectives for universal and existential quantification, our system now corresponds to a system of intuitionistic higher-order logic. Let us review how the concepts introduced align with those of a set theory with an associated predicate calculus, such as the widely-used Zermelo-Fraenkel system. While there are many points of analogy between the two foundations, there are also significant differences.

In set theory, objects are collected into sets, while type theory collects objects into types. An object in set theory may simultaneously be an element of many sets, with set membership $a \in A$ being a provable proposition within the logic. Each object in type theory, however, is only assigned to a unique type, known from the moment of the object's creation, $a : A$ being a judgement in the metatheory with no way to express this as a proposition within the type theory itself.

In set theory, the only fundamental construction is the set, with everything else encoded by combining sets. For example, functions are often encoded as sets of pairs which relate the input of the function to the output. In type theory, while encodings may be used, it is also possible to define new type primitives which more clearly express their intent. For example, the natural numbers may be encoded as Church numerals, $\mathbb{N} = \Pi_{(A:\mathsf{Type})}\ (A \to A) \to A \to A : \mathsf{Type}$, using the dependent function primitive, but we will see their direct definition using an inductive type in Section 2.6.

Typically, set-theoretic logics are classical, admitting the excluded middle and double-negation elimination reasoning principles, allowing proofs by contradiction. Type theory, however, is a computational calculus as well as a logic, with canonicity providing a strong guarantee about the result of reducing a term. This makes the logic inherently constructive, where disjunction proofs must reduce a proof for one of the two sides, and existence proofs must be accompanied by an algorithm for constructing a witness.

In set theory, there is a separation between the language of the logic and the language of the objects the logic describes. In type theory, propositions *are* types, with no distinction made between types inhabited by data structures and types inhabited by proofs — data and proofs are the same class of object, built from the same syntax. A term of type $\Pi_{(n:\mathbb{N})}\ \Sigma_{(p:\mathbb{N})}\ \mathsf{isPrime}\ p \times (p > n)$ is *both* a proof that there are infinitely many primes as well as as a function that computes a prime larger than any given input.

2.5 Equality and Equivalence

There is currently no way to state or prove the proposition that two terms are equal within the system of type theory we have defined. The only notion of equality seen so far is judgemental, part of the external metatheory used to define the rules of the type theory. We introduce *propositional equality*, also called the *identity type*, as a binary relation internal to the type theory.

$$\equiv\text{-}\textsc{Form}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \equiv b : \mathsf{Type}} \qquad \equiv\text{-}\textsc{Intro}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_{\equiv}\ a : a \equiv a}$$

Here, $a \equiv b$ is the type of proofs for the proposition that terms a and b are equal. The type has just one constructor, $\mathsf{refl}_{\equiv}$, which proves the reflexive case $a \equiv a$.

Judgemental equality implies propositional equality — judgementally-equal terms can always be reduced to their identical canonical forms, the equality then witnessed by $\mathsf{refl}_{\equiv}$. As an example illustrating this, we can show that function composition is associative.

Theorem 2.5.1. *Associativity,* $(f \circ (g \circ h))\ a \equiv ((f \circ g) \circ h)\ a$.

Proof. $\mathsf{assoc}_{\circ} = \mathsf{refl}_{\equiv}\ (f\ (g\ (h\ a)))$

By applying the definition of function composition, both sides of the equation reduce to $f\ (g\ (h\ a))$ by judgemental equalities, so it can be proved directly by the $\mathsf{refl}_{\equiv}$ constructor. □

Propositional equality is more powerful than judgemental equality, allowing us to prove terms equal which are not necessarily judgementally equal, however propositional equality is not decidable in general. We can see an application of the dependent $\mathbb{1}$-eliminator in combination with propositional equality by proving that $\star$ is the only canonical term for $\mathbb{1}$. Similarly, every Boolean value is either equal to true or false, whose proof also uses the dependent eliminator for sums.

Theorem 2.5.2. *Uniqueness,* $\Pi_{(u:\mathbb{1})}\ u \equiv \star$.

Proof. $\mathsf{elim}_{\mathbb{1}}\ (\mathsf{refl}_{\equiv}\ \star)$

Applying the eliminator, we are only required to prove the case where u is $\star$. The conclusion becomes $\star \equiv \star$, proved by $\mathsf{refl}_{\equiv}\ \star$. □

Lemma 2.5.3. $\Pi_{(b:\mathbb{2})}\ (b \equiv \mathsf{true}) + (b \equiv \mathsf{false})$

Proof. $\mathsf{elim}_{+}\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{inl}\ (\mathsf{refl}_{\equiv}\ \mathsf{true})))\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{inr}\ (\mathsf{refl}_{\equiv}\ \mathsf{false})))$

Since $\mathbb{2}$ is defined as $\mathbb{1} + \mathbb{1}$, the sum-eliminator splits the proof into the inl case and the inr case. In either case, the $\mathbb{1}$-eliminator has us consider $\star$ the only possible term being carried by each constructor. Considering the judgemental equality defining $\mathsf{true} = \mathsf{inl}\ \star$, in the first case the conclusion is $(\mathsf{true} \equiv \mathsf{true}) + (\mathsf{true} \equiv \mathsf{false})$, so we decide to prove the left side of the sum with $\mathsf{inl}\ (\mathsf{refl}_{\equiv}\ \mathsf{true})$. In the second case, we are to prove $(\mathsf{false} \equiv \mathsf{true}) + (\mathsf{false} \equiv \mathsf{false})$, so we elect for the right side this time. □

Lemma 2.5.3 validates there are no Boolean terms beside those equal to true or false, but it doesn't establish that these are actually distinct. For that, we need to prove $\mathsf{true} \equiv \mathsf{false} \to \mathbb{0}$, that true and false being equal leads to absurdity.

To prove statements with a propositional equality as a hypothesis, we need an elimination rule. Since the equality type former is a binary type family, the output predicate needs to be a dependent function which takes the parameters of the family, $T : \Pi_{(a,b:A)}\ a \equiv b \to \mathsf{Type}$.

$$\equiv\text{-}\textsc{Elim}\ (J)\ \frac{\Gamma \vdash f : \Pi_{(c:A)}\ T\ c\ c\ (\mathsf{refl}_{\equiv}\ c) \qquad \begin{array}{l}\Gamma \vdash a : A \\ \Gamma \vdash b : A \\ \Gamma \vdash p : a \equiv b\end{array}}{\Gamma \vdash \mathsf{elim}_{\equiv}\ f\ a\ b\ p : T\ a\ b\ p}$$

$$\equiv\text{-}\textsc{Reduce}\ \frac{\Gamma \vdash f : \Pi_{(c:A)}\ T\ c\ c\ (\mathsf{refl}_{\equiv}\ c) \qquad \begin{array}{l}\Gamma \vdash a : A \\ \Gamma \vdash \mathsf{refl}_{\equiv}\ a : a \equiv a\end{array}}{\Gamma \vdash \mathsf{elim}_{\equiv}\ f\ a\ a\ (\mathsf{refl}_{\equiv}\ a) = f\ a : T\ a\ a\ (\mathsf{refl}_{\equiv}\ a)}$$

The elimination rule, known as the *J-rule*, states that to prove a predicate on a equality proof $a \equiv b$, it suffices to prove the predicate in the reflexive case $\mathsf{refl}_{\equiv}\ c : c \equiv c$, for an arbitrary c of the type.

We need some basic properties of equality before we can prove that true and false are distinct, however. Let us start with symmetry, which instantiates the J-rule with predicate $T\ a\ b\ p = b \equiv a$.

Theorem 2.5.4. *Symmetry, $a \equiv b \to b \equiv a$.*

Proof. $\mathsf{sym}_{\equiv} = \mathsf{elim}_{\equiv}\ (\lambda(c : A).\ \mathsf{refl}_{\equiv}\ c)$

We assume $a \equiv b$ as a hypothesis, and are to prove $b \equiv a$. Applying the eliminator to the hypothesis, we need to provide a function of variable c which proves $T\ c\ c\ (\mathsf{refl}_{\equiv}\ c)$. Reducing this expression using the definition of T, this is simply the statement $c \equiv c$, which can be easily discharged using the $\mathsf{refl}_{\equiv}$ constructor. In essence, since we started from an assumption of $a \equiv b$, the J-rule allowed us to substitute all of the instances of a and b by a new variable c in the conclusion we wanted to reach. □

Similarly, transitivity can be proved by fixing $c : A$ and taking the predicate to be $T\ a\ b\ p = b \equiv c \to a \equiv c$.

Theorem 2.5.5. *Transitivity, $a \equiv b \to b \equiv c \to a \equiv c$.*

Proof. $\mathsf{trans}_{\equiv} = \mathsf{elim}_{\equiv}\ (\lambda(d : A).\ \mathsf{id}_{(d \equiv c)})$

Assume $a \equiv b$ as a hypothesis, but leave $b \equiv c$ unassumed — we must prove the implication $b \equiv c \to a \equiv c$. Apply the eliminator and instantiate the predicate with new variable d, leaving us to prove $d \equiv c \to d \equiv c$. This can be proven with an identity function id specialised to type $d \equiv c$ □

A common pattern in functional programming, and in mathematics more broadly, is theorem proving by *equational reasoning*, for example, sequencing equality proofs $a \equiv b \equiv \ldots \equiv y \equiv z$ to prove $a \equiv z$. This is justified in our type theory by chaining applications of $\mathsf{trans}_{\equiv}$.

As a final basic property we would expect an equality to exhibit, propositional equality forms a congruence relation. Sometimes called a substitution or transportation principle, this ensures that if we know $a \equiv b$, we may substitute b for a while preserving the proof of any predicate, $P\ a \to P\ b$. Taken in combination with symmetry, the direction of the implication is unimportant and the two terms become interchangeable.

Theorem 2.5.6. *Congruence, $P\ a \to a \equiv b \to P\ b$.*

Proof. $\mathsf{cong}\ (P : A \to \mathsf{Type}) = \lambda(p : P\ a).\ \lambda(q : a \equiv b).\ \mathsf{elim}_{\equiv}\ (\lambda(c : A).\ \mathsf{id}_{(P\ c)})\ q\ p$

Resembling the proof of transitivity, after applying the eliminator the conclusion becomes $P\ c \to P\ c$, proved by an identity function. □

A useful special case of congruence is that functions respect equality. That is, when proving an equality where both sides are applications of the same function, the function may be stripped away, simplifying the equation to be proved.

Theorem 2.5.7. *Functions respect equality, $a \equiv b \to f\ a \equiv f\ b$.*

Proof. $\mathsf{resp}\ (f : A \to B) = \mathsf{elim}_{\equiv}\ (\lambda(c : A).\ \mathsf{refl}_{\equiv}\ (f\ c))$ □

Using the congruence property with the previously-defined isTrue predicate which maps $\mathsf{true} \mapsto \mathbb{1}$ and $\mathsf{false} \mapsto \mathbb{0}$, we can prove that $\mathbb{2}$ is truly a Boolean type.

Lemma 2.5.8. $\mathsf{true} \equiv \mathsf{false} \to \mathbb{0}$

Proof. $\mathsf{cong}\ \mathsf{isTrue}\ \star$

Assume as a hypothesis that $\mathsf{true} \equiv \mathsf{false}$ and set out to prove $\mathbb{0}$, which is judgementally equal to $\mathsf{isTrue}\ \mathsf{false}$. Since we have $\mathsf{true} \equiv \mathsf{false}$ by assumption, congruence tells us that we may substitute within the conclusion, instead proving $\mathsf{isTrue}\ \mathsf{true}$, which is judgementally equal to $\mathbb{1}$. This is achieved with the trivial proof $\star$. □

Theorem 2.5.9. *Boolean,* $(\Pi_{(b:\mathbb{2})}\ (b \equiv \mathsf{true}) + (b \equiv \mathsf{false})) \times (\mathsf{true} \equiv \mathsf{false} \to \mathbb{0})$.

Proof. This is a straightforward pairing of the proof terms for Lemmas 2.5.3 and 2.5.8. □

As types are first-class values, it is also natural to ask when two types are equal. For types A and B, the type $A \equiv B$ contains proofs of their identification. Of course, judgementally-equal types can be proved propositionally equal by reflexivity, for example, $\mathsf{refl}_{\equiv}\ \mathbb{2} : \mathbb{2} \equiv \mathbb{2}$. However, in intensional type theories, such as the one defined in this chapter, we cannot go beyond judgemental equality of types and must instead talk of their equivalence.

The idea of equivalence has been used informally up until now, actually referring to three distinct notions. The first is propositional equivalence, corresponding to logical equivalence, $A \leftrightarrow B = (A \to B) \times (B \to A)$, indicating that the types are coinhabited — either they are both empty or they each have at least one term. However, this says nothing about the number of distinct terms inhabiting non-empty types, for example, $\mathbb{1} \leftrightarrow \mathbb{2}$ is itself inhabited.

Second, it has previously been noted that some types, such as $\mathbb{1} + (\mathbb{1} + \mathbb{1})$ and $(\mathbb{1} + \mathbb{1}) + \mathbb{1}$, are equivalent in the sense that they have the same number of canonical terms despite being structurally distinct. Let us now be more precise about what this form of type equivalence entails.

Definition 2.5.10. Two types $A : \mathsf{Type}$ and $B : \mathsf{Type}$ are equivalent when there is a bijective mapping between their terms — there are functions $f : A \to B$ and $g : B \to A$ which are inverse.

$$\begin{aligned} A \simeq B = \Sigma_{(f:A\to B)}\ \Sigma_{(g:B\to A)}\ &(\Pi_{(a:A)}\ a \equiv (g \circ f)\ a)) \\ &\times (\Pi_{(b:B)}\ b \equiv (f \circ g)\ b)) \end{aligned}$$

A proof of type equivalence is therefore a term whose canonical form consists of nested pairs, $\langle f, \langle g, \langle p, q\rangle\rangle\rangle$, containing two functions and two equality proofs. For notational convenience, we shall write this as a quadruple, $\langle f, g, p, q\rangle$.

While we cannot prove $\mathbb{1} + (\mathbb{1} + \mathbb{1})$ and $(\mathbb{1} + \mathbb{1}) + \mathbb{1}$ equal in an intensional type theory, we can prove them equivalent.

Theorem 2.5.11. $\mathbb{1} + (\mathbb{1} + \mathbb{1}) \simeq (\mathbb{1} + \mathbb{1}) + \mathbb{1}$

Proof. $\langle f, g, p, q\rangle$

Where f maps $\mathsf{inl}\ \star \mapsto \mathsf{inr}\ \star$, $\mathsf{inr}\ (\mathsf{inl}\ \star) \mapsto \mathsf{inl}\ (\mathsf{inr}\ \star)$, $\mathsf{inr}\ (\mathsf{inr}\ \star) \mapsto \mathsf{inl}\ (\mathsf{inl}\ \star)$, and g is its inverse. To be fully-formal, the terms for the function definitions, as well as proof terms p and q, would be be defined by case analysis, using the eliminator elim_{+}.

This is one possible equivalence proof out of a total of six for this type. The others may be produced by permuting the terms in the output of the map. □

Of course, like propositional equality, type equivalence forms an equivalence relation.

Theorem 2.5.12. *Reflexivity, $A \simeq A$.*

Proof. $\mathsf{refl}_{\simeq} = \langle \mathsf{id}_A,\ \mathsf{id}_A,\ \mathsf{refl}_{\equiv},\ \mathsf{refl}_{\equiv} \rangle$

The identity function id_A is used to instantiate both functions, so the final two elements must be of type $\Pi_{(a:A)}\ a \equiv \mathsf{id}_A\ (\mathsf{id}_A\ a)$. Since $\mathsf{id}_A\ (\mathsf{id}_A\ a)$ reduces to a, this is judgementally equal to the type of the $\mathsf{refl}_{\equiv}$ constructor. □

Theorem 2.5.13. *Symmetry, $A \simeq B \to B \simeq A$.*

Proof. $\mathsf{sym}_{\simeq} = \mathsf{elim}_{\Sigma}\ (\lambda f.\ \mathsf{elim}_{\Sigma}\ (\lambda g.\ \mathsf{elim}_{\Sigma}\ (\lambda p.\ \lambda q.\ \langle g,\ f,\ q,\ p \rangle)))$

Assume the equivalence $\langle f,\ g,\ p,\ q \rangle$ and simply swap the functions and swap the proofs, resulting in $\langle g,\ f,\ q,\ p \rangle$. □

Theorem 2.5.14. *Transitivity, $A \simeq B \to B \simeq C \to A \simeq C$.*

Proof. $\mathsf{trans}_{\simeq}\ (d : A \simeq B)\ (e : B \simeq C) = \ldots$

Assume the two equivalences $\langle f,\ g,\ p,\ q \rangle$ and $\langle f',\ g',\ p',\ q' \rangle$. To construct the two functions for the output equivalence, we compose the corresponding functions from the hypotheses, $f' \circ f$ and $g \circ g'$. For the proofs, we need to show, for arbitrary a and c, that $a \equiv ((g \circ g') \circ (f' \circ f))\ a$ and $c \equiv ((f' \circ f) \circ (g \circ g'))\ c$.

With the associative property, we can group the compositions such that our equality proofs p, q, p', and q' can eliminate them.

$$\begin{array}{lll} & & ((g \circ g') \circ (f' \circ f))\ a \\ (\mathsf{assoc}_{\circ}) & \equiv & (g \circ (g' \circ f') \circ f)\ a \\ (p') & \equiv & (g \circ f)\ a \\ (p) & \equiv & a \end{array} \qquad \begin{array}{lll} & & ((f' \circ f) \circ (g \circ g'))\ c \\ (\mathsf{assoc}_{\circ}) & \equiv & (f' \circ (f \circ g) \circ g')\ c \\ (q) & \equiv & (f' \circ g')\ c \\ (q') & \equiv & c \end{array}$$

The full proof term is omitted here to avoid the tedious decomposition of the two equivalences into their eight components using six applications of elim_{Σ}. Proof assistants provide *pattern matching* for cases like this, as seen in the version of this proof included in Appendix A. □

The third notion of equivalence is equivalence of functions. For example, bubble-sort and merge-sort can be considered equivalent functions in the sense they produce the same outputs when given the same inputs. However, since their implementations differ vastly, they cannot be proven propositionally equal.

Definition 2.5.15. Two functions $f : \Pi_{(a:A)}\ B\ a$ and $g : \Pi_{(a:A)}\ B\ a$ are equivalent when they are pointwise equal.

$$f \sim g = \Pi_{(a:A)}\ f\ a \equiv g\ a$$

Naturally, this also forms an equivalence relation. The proofs are immediate from the reflexivity, symmetry, and transitivity of the underlying propositional equality relation.

Theorem 2.5.16. *Reflexivity, $f \sim f$.*

Proof. $\mathsf{refl}_{\sim}\ (a : A) = \mathsf{refl}_{\equiv}\ (f\ a)$ □

Theorem 2.5.17. *Symmetry, $f \sim g \to g \sim f$.*

Proof. $\mathsf{sym}_{\sim}\ (e : f \sim g)\ (a : A) = \mathsf{sym}_{\equiv}\ (e\ a)$ □

Theorem 2.5.18. *Transitivity, $f \sim g \to g \sim h \to f \sim h$.*

Proof. $\mathsf{trans}_{\sim}\ (d : f \sim g)\ (e : g \sim h)\ (a : A) = \mathsf{trans}_{\equiv}\ (d\ a)\ (e\ a)$ □

As corollaries from their reflexivity, both type equivalence and function equivalence can be shown to be implied by propositional equality. Simply use congruence to substitute one of the sides of the conclusion so it becomes an instance of reflexivity.

Corollary 2.5.19. *Type equality implies equivalence, $A \equiv B \to A \simeq B$.*

Proof. $\mathsf{cong}\ (\lambda(C : \mathsf{Type}).\ A \simeq C)\ (\mathsf{refl}_{\simeq}\ A)$ □

Corollary 2.5.20. *Function equality implies equivalence, $f \equiv g \to f \sim g$.*

Proof. $\mathsf{cong}\ (\lambda(h : \Pi_{(a:A)}\ B\ a).\ f \sim h)\ (\mathsf{refl}_{\sim}\ f)$ □

Equality and equivalence are very active topics of research in the field of type theory. It is often desirable to eschew *intentional* type theories, where certain equality proofs may be difficult or impossible, and only focus on the *extensional* properties of the objects in question — if two distinct types or functions behave identically, why shouldn't they be considered equal?

For ease of reasoning about functions, the *axiom of function extensionality* may be assumed. This states the inverse of the above, that function equivalence implies equality, $\mathsf{funExt} : f \sim g \to f \equiv g$, allowing us to ignore their internal details and consider only their external behaviour. The axiom is independent of the intensional type theory we've defined — it cannot be proved directly, but its assumption is consistent with the existing rules [54]. However, as with adding classical reasoning principles as additional axioms, the funExt term has no reduction rules associated with it and so canonicity is lost.

The development of *homotopy type theory* [55] takes extensionality to its limit with the *univalence axiom*, due to Vladimir Voevodsky. Types are interpreted as topological spaces, with proofs of equality being paths between their points, and function equivalences regarded as homotopies deforming one function into the other. Notably, univalence implies, in addition to function extensionality, that type equivalence implies equality, $A \simeq B \to A \equiv B$ — equivalent types are truly equal, their implementation details irrelevant. There has been a recent effort to give computational content to univalence, one result being the system of *cubical type theory* [17], where univalence is a theorem rather than an axiom, and canonicity is preserved [35].

Since extensionality is not the focus of this thesis, only a very basic overview has been given. For the most part, we will operate within an intentional type theory, although function extensionality is assumed for Chapter V.

Finally in this section, we summarise the various equality and equivalence relations of type theory, the syntax used for them throughout this thesis, and compare their relative logical strength.

- Judgemental Equality, $a = b$: a statement in the defining metatheory, used to add new definitions to the type theory and to specify how terms are evaluated to their canonical forms. It is the strongest notion of equality, implying every relation on this list. However, it cannot be expressed as a type in the syntax of the type theory itself, so we cannot reason about judgemental equalities from within the system.

- Propositional Equality, $a \equiv b$: the notion of equality internal to the system of type theory, able to be stated as a type and reasoned about from within the system. Propositional equality is weaker than judgemental equality, but still implies all of the equivalences lower down on this list.

- Type Equivalence, $A \simeq B$: a bijective mapping between the terms of two types, establishing that the types have the same size. Type equivalence is weaker than propositional equality, but stronger than propositional equivalence.

- Propositional Equivalence, $A \leftrightarrow B$: a mapping between the terms of two types, implied by type equivalence since there is no requirement for injectivity or surjectivity. This establishes that the types are coinhabited, both empty or each inhabited by at least one term. This is the Curry-Howard interpretation of two propositions being logically equivalent.

- Function Equivalence, $f \sim g$: a statement that two functions are pointwise equal, with equal inputs to the functions yielding equal outputs. Weaker than propositional equality in intensional type theories, but of the same strength as propositional equality in extensional type theories where function extensionality is adopted.

2.6 Inductive Types

Combining sum and product types with the basic types $\mathbb{0}$ and $\mathbb{1}$, we are able to form any finite type, but is also possible to encode more complex structures using functions, as Church did in his λ-calculus [14,15]. For example, the Church numerals are an encoding of the natural numbers as parametrically polymorphic functions, $\mathbb{N} = \Pi_{(A:\mathsf{Type})}\ (A \to A) \to A \to A$. The numeral representing natural number n takes a function as an argument and iterates it n times over a second argument.

$$\begin{aligned}
&0 = \lambda A.\ \lambda f.\ \lambda x.\ x \\
&1 = \lambda A.\ \lambda f.\ \lambda x.\ f\ x \\
&2 = \lambda A.\ \lambda f.\ \lambda x.\ f\ (f\ x) \\
&3 = \lambda A.\ \lambda f.\ \lambda x.\ f\ (f\ (f\ x)) \\
&\ldots
\end{aligned}$$

It can be shown by parametricity [52, 57] that the only closed terms of this type are extensionally equal to the Church numerals. However, in a setting without function extensionality, it is more convenient to define the natural numbers, and other tree-like structures, as dedicated types with new syntax. In addition, our strictly-enforced type system prevents the definition of recursive functions using fixed-point combinators, such as Haskell Curry's Y combinator, $\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$ — the self-application $x\ x$ is invalid as it cannot be assigned a type.

We define Peano-style natural numbers in the same way as our other types, with formation, introduction, elimination, and reduction rules.

$$\mathbb{N}\text{-}\textsc{Form}\ \frac{}{\mathbb{N} : \mathsf{Type}} \qquad \mathbb{N}\text{-}\textsc{Intro}_{\mathsf{zero}}\ \frac{}{\mathsf{zero} : \mathbb{N}} \qquad \mathbb{N}\text{-}\textsc{Intro}_{\mathsf{succ}}\ \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\ n : \mathbb{N}}$$

$$\mathbb{N}\text{-}\textsc{Elim}\ \frac{\Gamma \vdash z : T\ \mathsf{zero} \qquad \Gamma \vdash s : \Pi_{(n:\mathbb{N})}\ T\ n \to T\ (\mathsf{succ}\ n) \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{elim}_{\mathbb{N}}\ z\ s\ n : T\ n}$$

$$\mathbb{N}\text{-}\textsc{Reduce}_{\mathsf{zero}}\ \frac{\Gamma \vdash z : T\ \mathsf{zero} \qquad \Gamma \vdash s : \Pi_{(n:\mathbb{N})}\ T\ n \to T\ (\mathsf{succ}\ n)}{\Gamma \vdash \mathsf{elim}_{\mathbb{N}}\ z\ s\ \mathsf{zero} = z : T\ \mathsf{zero}}$$

$$\mathbb{N}\text{-}\textsc{Reduce}_{\mathsf{succ}}\ \frac{\Gamma \vdash z : T\ \mathsf{zero} \qquad \Gamma \vdash s : \Pi_{(n:\mathbb{N})}\ T\ n \to T\ (\mathsf{succ}\ n) \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{elim}_{\mathbb{N}}\ z\ s\ (\mathsf{succ}\ n) = s\ n\ (\mathsf{elim}_{\mathbb{N}}\ z\ s\ n) : T\ (\mathsf{succ}\ n)}$$

The constructors for the natural numbers, defined by the two introduction rules, are zero and succ. Unlike previous introduction rules, the succ rule is recursive, requiring a natural number n to be constructed before the succ constructor may be applied to yield a new natural number. The canonicity principle specialised to natural numbers states that any closed term of type $\mathbb{N}$ is either the numeral zero, or the succ constructor applied to a numeral.

$$\begin{aligned} 0 &= \mathsf{zero} \\ 1 &= \mathsf{succ\ zero} \\ 2 &= \mathsf{succ\ (succ\ zero)} \\ 3 &= \mathsf{succ\ (succ\ (succ\ zero))} \\ &\ldots \end{aligned}$$

Seen as a computation principle, the eliminator gives a mechanism for defining functions by primitive recursion, saying how to compute from zero and how to compute from succ n, with the reduction rules specifying the recursion scheme. Seen as a proof principle, the elimination rule allows proofs by induction over natural numbers — to prove a predicate T for all natural numbers, it suffices to prove the base case, T zero, and the inductive case, $\Pi_{(n:\mathbb{N})}\ T\ n \to T\ (\mathsf{succ}\ n)$.

To be convinced that this type faithfully represents the natural numbers, we would also expect the succ constructor to be injective, which is explicitly stated as one of Peano's axioms. We may prove this as a theorem rather than adding an axiom, requiring the definition of a type family indexed by the natural numbers.

Theorem 2.6.1. $\mathsf{succ}\ m \equiv \mathsf{succ}\ n \to m \equiv n$

Proof. $\mathsf{cong}\ (\mathsf{elim}_{\mathbb{N}}\ \mathbb{0}\ (\lambda o.\ \lambda r.\ m \equiv o))\ (\mathsf{refl}_{\equiv}\ m)$

Using the natural number eliminator, define the type family $P : \mathbb{N} \to \mathsf{Type}$ which maps $\mathsf{succ}\ o \mapsto m \equiv o$. The type zero maps to is of no consequence and is chosen arbitrarily.

Specialising cong P to the case where both numbers are successors, its type is $P\ (\mathsf{succ}\ m) \to \mathsf{succ}\ m \equiv \mathsf{succ}\ n \to P\ (\mathsf{succ}\ n)$. Reducing the applications of P yields $m \equiv m \to \mathsf{succ}\ m \equiv \mathsf{succ}\ n \to m \equiv n$. After giving a proof of $m \equiv m$, this is exactly the statement that succ is injective. □

A basic operator that doesn't require recursion is the predecessor function, which removes an application of succ from positive natural numbers.

$$\mathsf{pred} = \mathsf{elim}_{\mathbb{N}}\ \mathsf{zero}\ (\lambda n.\ \lambda r.\ n) : \mathbb{N} \to \mathbb{N}$$

More complex arithmetic operators may be defined by primitive recursion. For example, an intuitive definition of addition of two natural numbers can be defined by pattern matching on the second argument. In the base case, the first argument is returned, while the recursive case moves the application of succ away from the second argument.

$$\begin{aligned}
&\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathsf{add}\ m\ \mathsf{zero} \qquad = m \\
&\mathsf{add}\ m\ (\mathsf{succ}\ n) = \mathsf{succ}\ (\mathsf{add}\ m\ n)
\end{aligned}$$

This definition is realised by applying the eliminator non-dependently, instantiated at constant type family $T\ n = \mathbb{N}$.

$$\mathsf{add} = \lambda m.\ \mathsf{elim}_{\mathbb{N}}\ m\ (\lambda n.\ \lambda r.\ \mathsf{succ}\ r) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

The first argument to to $\mathsf{elim}_{\mathbb{N}}$ represents the base case, while the second argument is the recursive case, with the result of the recursive call being bound to variable r. Applying the reduction rule for succ, this can be seen to reduce to the expected expression.

$$\begin{aligned}
\mathsf{add}\ m\ (\mathsf{succ}\ n) &= \quad \mathsf{elim}_{\mathbb{N}}\ m\ (\lambda n.\ \lambda r.\ \mathsf{succ}\ r)\ (\mathsf{succ}\ n) \\
&\rightsquigarrow \quad (\lambda n.\ \lambda r.\ \mathsf{succ}\ r)\ n\ (\mathsf{elim}_{\mathbb{N}}\ m\ (\lambda n.\ \lambda r.\ \mathsf{succ}\ r)\ n) \\
&\rightsquigarrow^* \mathsf{succ}\ (\mathsf{elim}_{\mathbb{N}}\ m\ (\lambda n.\ \lambda r.\ \mathsf{succ}\ r)\ n) \\
&= \quad \mathsf{succ}\ (\mathsf{add}\ m\ n)
\end{aligned}$$

As addition was defined by recursion on the second argument, the fact that zero is the right identity for addition is a judgemental equality — the proof of $\mathsf{add}\ n\ \mathsf{zero} \equiv n$ is simply $\mathsf{refl}_{\equiv}\ n$. However, for the left identity, $\mathsf{add}\ \mathsf{zero}\ n$ cannot be reduced without knowing whether n is constructed by zero or succ. To show this, we use the eliminator dependently, instantiating the predicate with $T\ n = \mathsf{add}\ \mathsf{zero}\ n \equiv n$, as a proof by induction.

Theorem 2.6.2. $\Pi_{(n:\mathbb{N})}\ \mathsf{add}\ \mathsf{zero}\ n \equiv n$

Proof. $\mathsf{elim}_{\mathbb{N}}\ (\mathsf{refl}_{\equiv}\ \mathsf{zero})\ (\lambda n.\ \lambda r.\ \mathsf{resp}\ \mathsf{succ}\ r)$

Proceed by induction on n. The base case is $\mathsf{add}\ \mathsf{zero}\ \mathsf{zero} \equiv \mathsf{zero}$. By definition of add, this reduces to $\mathsf{zero} \equiv \mathsf{zero}$, which is proved by $\mathsf{refl}_{\equiv}\ \mathsf{zero}$. This therefore becomes the first argument to $\mathsf{elim}_{\mathbb{N}}$.

In the inductive case, we have inductive hypothesis $\mathsf{add}\ \mathsf{zero}\ n \equiv n$ and must show $\mathsf{add}\ \mathsf{zero}\ (\mathsf{succ}\ n) \equiv \mathsf{succ}\ n$, which reduces to $\mathsf{succ}\ (\mathsf{add}\ \mathsf{zero}\ n) \equiv \mathsf{succ}\ n$. Since succ respects equality, by Theorem 2.5.7, it can be stripped from both sides of the conclusion, with the remaining term being identical to the inductive hypothesis. In the proof term, it can be seen that the inductive case is a dependent function where variable r binds the inductive hypothesis. □

Generalising from the definition of natural numbers, we can define an entire class of types called *inductive types* whose terms are wellfounded trees. However, instead of explicitly specifying the rules for all these types, we use a more concise notation which specifies the type of the former, and the types of all constructors.

$$\begin{aligned}&\mathsf{inductive}\ \mathbb{N} : \mathsf{Type}\\ &\quad \mathsf{zero} : \mathbb{N}\\ &\quad \mathsf{succ} : \mathbb{N} \to \mathbb{N}\end{aligned}$$

This definition of natural numbers is understood to be shorthand for all of its rules. That is, a type formation rule, an introduction rule for each constructor, an elimination rule, and a reduction rule for each constructor.

We can define inductive type families of arbitrary arity by setting the type former to be a function type. Each constructor may also be of arbitrary arity, and may depend recursively on other elements of the type, forming new branches in the tree. These types may be given as instances of W-types [45], but we will opt to allow the direct definition of recursive functions by pattern matching [23].

Natural numbers are unary trees whose nodes are applications of the succ constructor and whose leaves are zero. The tree carries no additional data at the nodes aside from subtrees, the only information represented being the tree's depth. If we instead define the inductive type of unary trees where the nodes additionally carry a term of type A, we obtain the List data structure.

$$\begin{aligned}&\mathsf{inductive}\ \mathsf{List} : \mathsf{Type} \to \mathsf{Type}\\ &\quad [\,] : \Pi_{(A:\mathsf{Type})}\ \mathsf{List}\ A\\ &\quad :: \ : \Pi_{(A:\mathsf{Type})}\ A \to \mathsf{List}\ A\ \to \mathsf{List}\ A\end{aligned}$$

List is defined as a type family, giving dependent function types for its constructors to allow for parametric polymorphism, but dependent types may also be used to impose constraints on when the use of a constructor is valid. We can define the Fin inductive family of finite types, indexed by natural numbers, where $\mathsf{Fin}\ n$ has exactly n canonical terms, representing those natural numbers strictly less than n.

$$\begin{aligned}&\mathsf{inductive}\ \mathsf{Fin} : \mathbb{N} \to \mathsf{Type}\\ &\quad \mathsf{zero}_{\mathsf{Fin}} : \Pi_{(n:\mathbb{N})}\ \mathsf{Fin}\ (\mathsf{succ}\ n)\\ &\quad \mathsf{succ}_{\mathsf{Fin}} : \Pi_{(n:\mathbb{N})}\ \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{succ}\ n)\end{aligned}$$

Since both constructors result in $\mathsf{Fin}\ (\mathsf{succ}\ n)$, there is no way of constructing a term of type $\mathsf{Fin}\ 0$. The only way of constructing a term of type $\mathsf{Fin}\ 1$ is using

$\mathsf{zero}_{\mathsf{Fin}}$, as $\mathsf{succ}_{\mathsf{Fin}}$ would have be applied to a term of type $\mathsf{Fin}\ 0$. The $\mathsf{succ}_{\mathsf{Fin}}$ constructor may be used for $\mathsf{Fin}\ 2$ onwards, but only a limited number of times before $\mathsf{zero}_{\mathsf{Fin}}$ is forced.

Inductive families representing predicates and relations may also be defined. For example, the ordering of natural numbers can be established with an inductive binary relation.

$$\begin{aligned}
&\mathsf{inductive}\ \leq\ : \mathbb{N} \to \mathbb{N} \to \mathsf{Type}\\
&\quad \mathsf{zero}_{\leq}\ : \Pi_{(n:\mathbb{N})}\ \mathsf{zero} \leq n\\
&\quad \mathsf{succ}_{\leq}\ : \Pi_{(m,n:\mathbb{N})}\ m \leq n \to \mathsf{succ}\ m \leq \mathsf{succ}\ n
\end{aligned}$$

The strength of inductive types is that, in lieu of an elimination rule, we allow direct definitions by *structural recursion*, pattern matching on the possible ways of constructing a type. To be certain of normalisation, recursive calls must be on a structurally smaller term than the original argument, that is a subexpression with strictly fewer constructors.

This is more general than the primitive recursion allowed by $\mathsf{elim}_{\mathbb{N}}$, allowing multiple constructors to be stripped at once. For example, we can define the function which computes $n \mapsto \lfloor n/2 \rfloor$ by pattern matching two constructors deep.

$$\begin{aligned}
&\mathsf{half} : \mathbb{N} \to \mathbb{N}\\
&\mathsf{half}\ \mathsf{zero} &&= \mathsf{zero}\\
&\mathsf{half}\ (\mathsf{succ}\ \mathsf{zero}) &&= \mathsf{zero}\\
&\mathsf{half}\ (\mathsf{succ}\ (\mathsf{succ}\ n)) &&= \mathsf{succ}\ (\mathsf{half}\ n)
\end{aligned}$$

Seen as universally-quantified proofs, dependent functions defined by structural recursion correspond to proofs by structural induction. For example, the reflexivity of $\leq$ can be established by induction using the eliminator $\mathsf{elim}_{\mathbb{N}}$, but we demonstrate the proof as a structurally-recursive function.

Theorem 2.6.3. *Reflexivity, $n \leq n$.*

Proof.

$$\begin{aligned}
&\mathsf{refl}_{\leq} : \Pi_{(n:\mathbb{N})}\ n \leq n\\
&\mathsf{refl}_{\leq}\ \mathsf{zero} &&= \mathsf{zero}_{\leq}\\
&\mathsf{refl}_{\leq}\ (\mathsf{succ}\ n) &&= \mathsf{succ}_{\leq}\ (\mathsf{refl}_{\leq}\ n)
\end{aligned}$$

Proceed by induction on n. In the base case, we have to show $\mathsf{zero} \leq \mathsf{zero}$, established by the $\mathsf{zero}_{\leq}$ constructor which states that zero is less than or equal any other natural number.

In the inductive case, we have inductive hypothesis $n \leq n$ and need to prove $\mathsf{succ}\ n \leq \mathsf{succ}\ n$. The $\mathsf{succ}_{\leq}$ constructor states that the relation respects succ, so we are just left to prove a statement equal to the inductive hypothesis. To access the inductive hypothesis in the proof term we call $\mathsf{refl}_{\leq}$ recursively, at a structurally-smaller index. □

In the presence of inductive families with dependently-typed constructors, structural recursion becomes a powerful reasoning tool, allows patterns to be refined according to the type of the constructor [18, 31]. We demonstrate this by proving the transitivity of $\leq$ by performing structural recursion on a proof of the relation rather than a natural number.

Theorem 2.6.4. *Transitivity,* $m \leq n \to n \leq o \to m \leq o$*.*

Proof.

$$\begin{aligned}
&\mathsf{trans}_{\leq} : \Pi_{(m,n,o:\mathbb{N})}\ m \leq n \to n \leq o \to m \leq o\\
&\mathsf{trans}_{\leq}\ \mathsf{zero}_{\leq}\ \quad q \quad\quad\ = \mathsf{zero}_{\leq}\\
&\mathsf{trans}_{\leq}\ (\mathsf{succ}_{\leq}\ p)\ (\mathsf{succ}_{\leq}\ q) = \mathsf{succ}_{\leq}\ (\mathsf{trans}_{\leq}\ p\ q)
\end{aligned}$$

Assume $m \leq n$ and $n \leq o$, towards a proof of $m \leq o$. Proceed by structural induction on the proof of $m \leq n$.

In the base case, it is constructed by $\mathsf{zero}_{\leq} : \mathsf{zero} \leq n$, so the only possibility is when m is zero. The conclusion is therefore refined to $\mathsf{zero} \leq o$, which is established by $\mathsf{zero}_{\leq}$.

In the inductive case, the proof is constructed by $\mathsf{succ}_{\leq}\ p : \mathsf{succ}\ m \leq \mathsf{succ}\ n$, the only possible pattern is when both sides of the relation are successors. The type of our second assumption is refined to $\mathsf{succ}\ n \leq o$, which forces it to be constructed by $\mathsf{succ}_{\leq}\ q : \mathsf{succ}\ n \leq \mathsf{succ}\ o$, the third number must also be a successor. In light of these refinements, the conclusion is $\mathsf{succ}\ m \leq \mathsf{succ}\ o$, so we can apply the $\mathsf{succ}_{\leq}$ constructor and then prove $m \leq o$ by recursion on the structurally smaller terms p and q. □

2.7 Coinductive Types

Inductive types represent wellfounded trees, structural recursion restricting them to finite depth, however infinite structures are often objects of study. In functional programming languages featuring general recursion and lazy evaluation, such as Haskell [42], no distinction is made between inductive types with finite objects and those with infinite objects [34]. For example, a definition like the following would be valid, defining the infinite list $\mathsf{nats} = 0 :: 1 :: 2 :: \ldots$ of all natural numbers — in fact, this is so common in Haskell that the shorthand $[0\,..]$ is provided

$$\begin{aligned}
&\mathsf{nats} : \mathsf{List}\ \mathbb{N} &&\mathsf{from} : \mathbb{N} \to \mathsf{List}\ \mathbb{N}\\
&\mathsf{nats} = \mathsf{from}\ \mathsf{zero} &&\mathsf{from}\ n = n :: \mathsf{from}\ (\mathsf{succ}\ n)
\end{aligned}$$

However, in our setting, strong normalisation must be preserved to ensure logical consistency. The definition of from cannot be accepted as it performs a recursive call on a term which is not structurally smaller than the argument.

We would still like to represent such objects, but care must be taken to construct them in a coherent way. The class of *coinductive types* [13,19] is introduced to this end. For example, the Stream coinductive family collects the types of infinite sequences.

$$\begin{aligned}
&\mathsf{coinductive\ Stream} : \mathsf{Type} \to \mathsf{Type}\\
&\quad \triangleright : \Pi_{(A:\mathsf{Type})}\ A \to \mathsf{Stream}\ A \to \mathsf{Stream}\ A
\end{aligned}$$

The definition is very similar to that of the List inductive family. We omit an empty constructor here, so these streams are necessarily infinite, though an alternative definition of lists which are potentially infinite is also possible, which more closely aligns with lazy lists as they exist in Haskell.

Crucially, declaring the type $\mathsf{coinductive}$ lifts the requirement that terms of this type must be wellfounded. If we had instead declared the type $\mathsf{inductive}$, it could be shown that $\mathsf{Stream}\ A \simeq \mathbb{0}$, as we could never construct a base value for the constructor's second argument. In particular, we don't require functions constructing streams to be structurally recursive, so the above definitions can be ported to result in $\mathsf{Stream}\ \mathbb{N}$.

$$\begin{aligned}
&\mathsf{nats} : \mathsf{Stream}\ \mathbb{N} &\qquad& \mathsf{from} : \mathbb{N} \to \mathsf{Stream}\ \mathbb{N}\\
&\mathsf{nats} = \mathsf{from\ zero} && \mathsf{from}\ n = n \triangleright \mathsf{from}\ (\mathsf{succ}\ n)
\end{aligned}$$

However, not all recursive definitions are allowed. We must ensure canonicity, so that a term of type $\mathsf{Stream}\ A$ is actually constructed by $\triangleright$ applied to the appropriate arguments. This will justify being able define functions by pattern matching on the type, such as head and tail projection functions.

$$\begin{aligned}
&\mathsf{head} : \Pi_{(A:\mathsf{Type})}\ \mathsf{Stream}\ A \to A &\qquad& \mathsf{tail} : \Pi_{(A:\mathsf{Type})}\ \mathsf{Stream}\ A \to \mathsf{Stream}\ A\\
&\mathsf{head}\ (h \triangleright t) = h && \mathsf{tail}\ (h \triangleright t) = t
\end{aligned}$$

Strong normalisation is preserved if we only reduce recursive calls lazily [25, 32]. That is, only when the result of reducing the term is needed in order to reduce a pattern-matching function.

In general, recursive calls must be *guarded* by a constructor of the coinductive type [28, 47], that is, occurring within a subexpression of a constructor. This guarantees the function must make some progress at each recursive step. In the definition of from, we can see that guardedness is satisfied, the recursive call appearing as the second argument of the $\triangleright$ constructor.

An example that is ruled out by the guardedness condition is the function which filters a stream according to a decidable predicate, keeping only those elements which satisfy the predicate. For lists, this has a valid definition by structural recursion.

$$\begin{aligned}
&\mathsf{filter} : \Pi_{(A:\mathsf{Type})}\ (A \to \mathbb{2}) \to \mathsf{List}\ A \to \mathsf{List}\ A\\
&\mathsf{filter}\ f\ [\,] \qquad\quad = [\,]\\
&\mathsf{filter}\ f\ (h :: t) = \mathsf{elim}_{+}\ (\mathsf{elim}_{\mathbb{1}}\ (h :: \mathsf{filter}\ f\ t))\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{filter}\ f\ t))\ (f\ h)
\end{aligned}$$

This definition has two recursive calls on the tail of the list. The first is when $f\ h$ is true, so h will be kept in the resulting list, $h :: \mathsf{filter}\ f\ t$, while $f\ h$ is false in the second, dropping h from the result, $\mathsf{filter}\ f\ t$.

When attempting to define a similar function for type $\mathsf{Stream}\ A$, the second recursive call would be problematic as it makes no progress in constructing a new $h \triangleright t$ node. In general, there is no guarantee that the predicate will ever return true — given a stream $s : \mathsf{Stream}\ A$, the term $\mathsf{head}\ (\mathsf{filter}\ (\lambda(a : A).\ \mathsf{false})\ s)$ would loop forever, never producing a head for the new stream.

Other familiar list functions may have corresponding definitions for streams as long as they are productive. For example, a function may be mapped pointwise over an stream to yield a new stream, and two streams may be zipped to form a stream of pairs.

$$\begin{aligned}
&\mathsf{map} : \Pi_{(A,B:\mathsf{Type})}\ (A \to B) \to \mathsf{Stream}\ A \to \mathsf{Stream}\ B\\
&\mathsf{map}\ f\ (h \triangleright t) = f\ h \triangleright \mathsf{map}\ f\ t
\end{aligned}$$

$$\begin{aligned}
&\mathsf{zip} : \Pi_{(A,B:\mathsf{Type})}\ \mathsf{Stream}\ A \to \mathsf{Stream}\ B \to \mathsf{Stream}\ (A \times B)\\
&\mathsf{zip}\ (h_1 \triangleright t_1)\ (h_2 \triangleright t_2) = \langle h_1,\, h_2\rangle \triangleright \mathsf{zip}\ t_1\ t_2
\end{aligned}$$

Coinductive types may also represent propositions whose proofs are infinite structures. For example, the $\mathsf{Always}\ P$ predicate on streams asserts that when predicate P holds at the head of a stream and, recursively, $\mathsf{Always}\ P$ holds on the tail, then it holds at every element.

$$\begin{aligned}
&\mathsf{coinductive\ Always} : \Pi_{(A:\mathsf{Type})}\ (A \to \mathsf{Type}) \to \mathsf{Stream}\ A \to \mathsf{Type}\\
&\quad \mathsf{always} : \Pi_{(P:A\to\mathsf{Type})}\ \Pi_{(s:\mathsf{Stream}\ A)}\\
&\qquad\qquad P\ (\mathsf{head}\ s) \to \mathsf{Always}\ P\ (\mathsf{tail}\ s) \to \mathsf{Always}\ P\ s
\end{aligned}$$

A proof of $\mathsf{Always}\ P\ s$ can itself be seen as an infinite stream of proofs, albeit each element having a different type, the first being $P\ (\mathsf{head}\ s)$, the second $P\ (\mathsf{head}\ (\mathsf{tail}\ s))$, then $P\ (\mathsf{head}\ (\mathsf{tail}\ (\mathsf{tail}\ s)))$, and so on.

Two streams are extensionally equal when they are pointwise equal. Even if their definitions may radically differ, they are considered equivalent if they produce the same sequence of values. This form of equivalence is called *bisimulation*, and can be defined in terms of the Always type.

$$\begin{aligned}&\mathsf{Bisim} : \Pi_{(A:\mathsf{Type})}\ \mathsf{Stream}\ A \to \mathsf{Stream}\ A \to \mathsf{Type}\\&\mathsf{Bisim}\ r\ s = \mathsf{Always}\ (\lambda(p : A \times A).\ \pi_1\ p \equiv \pi_2\ p)\ (\mathsf{zip}\ r\ s)\end{aligned}$$

Or, equivalently, as a dedicated coinductive type, a binary type family indexed by streams.

$$\begin{aligned}&\mathsf{coinductive\ Bisim} : \Pi_{(A:\mathsf{Type})}\ \mathsf{Stream}\ A \to \mathsf{Stream}\ A \to \mathsf{Type}\\&\quad\mathsf{bisim} : \Pi_{(a,b:\mathsf{Stream}\ A)}\ \mathsf{head}\ a \equiv \mathsf{head}\ b \to \mathsf{Bisim}\ (\mathsf{tail}\ a)\ (\mathsf{tail}\ b) \to \mathsf{Bisim}\ a\ b\end{aligned}$$

For example, we can show that mapping succ onto stream from n is the same as applying tail. These are two streams that can be proved bisimilar by coinduction.

Theorem 2.7.1. $\Pi_{(n:\mathbb{N})}\ \mathsf{Bisim}\ (\mathsf{map\ succ}\ (\mathsf{from}\ n))\ (\mathsf{tail}\ (\mathsf{from}\ n))$

Proof. First, use the bisim constructor to form a term of the coinductive Bisim type. We now need to provide two proofs, one showing the equality of the streams' heads, the other showing that their tails are bisimilar.

For the first proof, observe that the heads of both streams reduce to succ n, so they may be proven equal by $\mathsf{refl}_{\equiv}$ (succ n).

$$\begin{aligned}\mathsf{head}\ (\mathsf{map\ succ}\ (\mathsf{from}\ n)) &= \mathsf{succ}\ (\mathsf{head}\ (\mathsf{from}\ n)) && \text{(definition of map)}\\&= \mathsf{succ}\ n && \text{(definition of from)}\end{aligned}$$

$$\begin{aligned}\mathsf{head}\ (\mathsf{tail}\ (\mathsf{from}\ n)) &= \mathsf{head}\ (\mathsf{from}\ (\mathsf{succ}\ n)) && \text{(definition of from)}\\&= \mathsf{succ}\ n && \text{(definition of from)}\end{aligned}$$

For the bisimilarity of the tails, the relaxed limits of recursion allow us to assume the statement we are proving as a coinductive hypothesis. Since we are guarded by an application of constructor bisim, we may now use the hypothesis without further restriction.

Instantiating the coinductive hypothesis at index succ n gives the statement Bisim (map succ (from (succ n))) (tail (from (succ n))). It can be seen that that this is judgementally equal to the statement we want to prove.

$$\begin{aligned}\mathsf{map\ succ}\ (\mathsf{from}\ (\mathsf{succ}\ n)) &= \mathsf{map\ succ}\ (\mathsf{tail}\ (\mathsf{from}\ n)) && \text{(definition of from)}\\&= \mathsf{tail}\ (\mathsf{map\ succ}\ (\mathsf{from}\ n)) && \text{(definition of map)}\end{aligned}$$

$$\mathsf{tail}\ (\mathsf{from}\ (\mathsf{succ}\ n)) = \mathsf{tail}\ (\mathsf{tail}\ (\mathsf{from}\ n)) \qquad \text{(definition of } \mathsf{from}\text{)}$$

So we have $\mathsf{Bisim}\ (\mathsf{tail}\ (\mathsf{map}\ \mathsf{succ}\ (\mathsf{from}\ n)))\ (\mathsf{tail}\ (\mathsf{tail}\ (\mathsf{from}\ n)))$ by coinductive hypothesis, which is precisely the statement that the tails of the two streams are bisimilar. □

Formally, the proof can be written as a function defined by the following recursive equation, where the recursive call is guarded by the bisim constructor.

$$\begin{aligned} &f : \Pi_{(n:\mathbb{N})}\ \mathsf{Bisim}\ (\mathsf{map}\ \mathsf{succ}\ (\mathsf{from}\ n))\ (\mathsf{tail}\ (\mathsf{from}\ n)) \\ &f\ n = \mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ (\mathsf{succ}\ n))\ (f\ (\mathsf{succ}\ n)) \end{aligned}$$

Applying the function to 0 gives a proof that $\mathsf{map}\ \mathsf{succ}\ \mathsf{nats}$ and $\mathsf{tail}\ \mathsf{nats}$ are bisimilar. Unravelling the recursive calls in the proof term, we see that it actually an infinite sequence of $\mathsf{refl}_{\equiv}$ proofs, a new one generated at each recursive step.

$$\begin{aligned} f\ 0 &= \mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 1)\ (f\ 1) \\ &= \mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 1)\ (\mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 2)\ (f\ 2)) \\ &= \mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 1)\ (\mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 2)\ (\mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 3)\ (f\ 3))) \\ &= \mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 1)\ (\mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 2)\ (\mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ 3)\ (\ldots))) \end{aligned}$$

Bisimilarity is an equivalence relation. The properties of reflexivity, symmetry, and transitivity may each be shown by coinductive proof. We demonstrate these by giving a recursive function for each.

Theorem 2.7.2. *Reflexivity,* $\mathsf{Bisim}\ a\ a$.

Proof.

$$\begin{aligned} &\mathsf{refl}_{\mathsf{Bisim}} : \Pi_{(A:\mathsf{Type})}\ \Pi_{(a:\mathsf{Stream}\ A)}\ \mathsf{Bisim}\ a\ a \\ &\mathsf{refl}_{\mathsf{Bisim}}\ (h \triangleright t) = \mathsf{bisim}\ (\mathsf{refl}_{\equiv}\ h)\ (\mathsf{refl}_{\mathsf{Bisim}}\ t) \end{aligned}$$

We define the function by pattern matching on the stream, then use the bisim constructor to form the resulting proof. We must therefore prove $h \equiv h$ and $\mathsf{Bisim}\ t\ t$.

Using the reflexivity constructor of propositional equality, the head is equal to itself, $\mathsf{refl}_{\equiv}\ h$. To prove the tail bisimilar to itself, since we are guarded by an application of bisim, we may recursively call the function at the tail, proving the statement by coinductive hypothesis. □

Theorem 2.7.3. *Symmetry,* $\mathsf{Bisim}\ a\ b \to \mathsf{Bisim}\ b\ a$.

Proof.

$$\begin{aligned}&\mathsf{sym}_{\mathsf{Bisim}} : \Pi_{(A:\mathsf{Type})}\ \Pi_{(a,b:\mathsf{Stream}\ A)}\ \mathsf{Bisim}\ a\ b \to \mathsf{Bisim}\ b\ a\\ &\mathsf{sym}_{\mathsf{Bisim}}\ (\mathsf{bisim}\ h\ t) = \mathsf{bisim}\ (\mathsf{sym}_{\equiv}\ h)\ (\mathsf{sym}_{\mathsf{Bisim}}\ t)\end{aligned}$$

Here, we pattern match on the proof of $\mathsf{Bisim}\ a\ b$, which is formed by proofs $h : \mathsf{head}\ a \equiv \mathsf{head}\ b$ and $t : \mathsf{Bisim}\ (\mathsf{tail}\ a)\ (\mathsf{tail}\ b)$. After using the bisim constructor, we must prove the symmetric forms of these assumptions.

The first, $\mathsf{head}\ b \equiv \mathsf{head}\ a$, is clear from the symmetry of equality, $\mathsf{sym}_{\equiv}\ h$. The second, $\mathsf{Bisim}\ (\mathsf{tail}\ b)\ (\mathsf{tail}\ a)$, can be shown by coinductive hypothesis, recursively calling the function on t. □

Theorem 2.7.4. *Transitivity,* $\mathsf{Bisim}\ a\ b \to \mathsf{Bisim}\ b\ c \to \mathsf{Bisim}\ a\ c$.

Proof.

$$\begin{aligned}&\mathsf{trans}_{\mathsf{Bisim}} : \Pi_{(A:\mathsf{Type})}\ \Pi_{(a,b,c:\mathsf{Stream}\ A)}\ \mathsf{Bisim}\ a\ b \to \mathsf{Bisim}\ b\ c \to \mathsf{Bisim}\ a\ c\\ &\mathsf{trans}_{\mathsf{Bisim}}\ (\mathsf{bisim}\ h_1\ t_1)\ (\mathsf{bisim}\ h_2\ t_2) = \mathsf{bisim}\ (\mathsf{trans}_{\equiv}\ h_1\ h_2)\ (\mathsf{trans}_{\mathsf{Bisim}}\ t_1\ t_2)\end{aligned}$$

The structure of this proof is the same as that of symmetry, pattern matching on the proofs of bisimilarity. We use the transitive property of equality for the head, and the coinductive hypothesis for the tail. □

As a final note for this introduction, we have focused on infinite linear sequences as these are most relevant for Chapter IV, but coinductive types for other kinds of non-wellfounded tree may also be defined. A type of infinitely-deep binary trees is produced with a constructor that takes two recursive arguments instead of one.

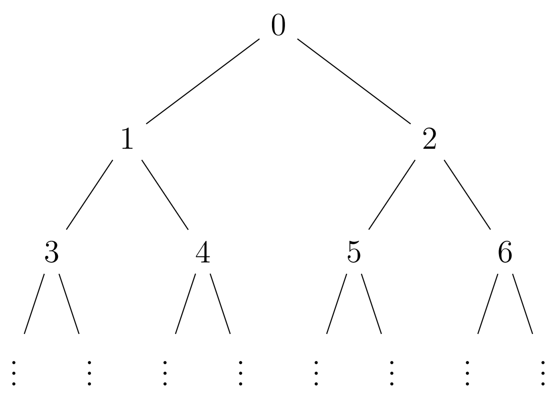
$$\begin{aligned}&\mathsf{coinductive\ Tree} : \mathsf{Type} \to \mathsf{Type}\\ &\qquad \mathsf{node} : \Pi_{(A:\mathsf{Type})}\ A \to \mathsf{Tree}\ A \to \mathsf{Tree}\ A \to \mathsf{Tree}\ A\end{aligned}$$

The requirement for guarded recursion remains the same. For example, the binary tree with whose nodes state their index in a breadth-first traversal can be defined by the following equation whose recursive calls are guarded by the node constructor.

$$\begin{aligned}&\mathsf{bft} : \mathbb{N} \to \mathsf{Tree}\ \mathbb{N}\\ &\mathsf{bft}\ n = \mathsf{node}\ n\ (\mathsf{bft}\ (\mathsf{add}\ 1\ (\mathsf{add}\ n\ n)))\ (\mathsf{bft}\ (\mathsf{add}\ 2\ (\mathsf{add}\ n\ n)))\end{aligned}$$

Applying the bft function to argument 0 gives the actual tree.

$$\begin{aligned}\mathsf{bft}\ 0 = \mathsf{node}\ 0\ &(\mathsf{node}\ 1\ (\mathsf{node}\ 3\ (\dots)\ (\dots))\ (\mathsf{node}\ 4\ (\dots)\ (\dots)))\\ &(\mathsf{node}\ 2\ (\mathsf{node}\ 5\ (\dots)\ (\dots))\ (\mathsf{node}\ 6\ (\dots)\ (\dots)))\end{aligned}$$



If val, left, and right are the projection functions for the node constructor, tree bisimulation may then be defined as a coinductive family indexed by trees.

$$\begin{aligned}
&\mathsf{coinductive}\ \mathsf{Bisim}_{\mathsf{Tree}} : \Pi_{(A:\mathsf{Type})}\ \mathsf{Tree}\ A \to \mathsf{Tree}\ A \to \mathsf{Type}\\
&\qquad \mathsf{bisim}_{\mathsf{Tree}} : \Pi_{(a,b:\mathsf{Tree}\ A)}\ \mathsf{val}\ a \equiv \mathsf{val}\ b \to \mathsf{Bisim}_{\mathsf{Tree}}\ (\mathsf{left}\ a)\ (\mathsf{left}\ b) \to\\
&\qquad\qquad\qquad \mathsf{Bisim}_{\mathsf{Tree}}\ (\mathsf{right}\ a)\ (\mathsf{right}\ b) \to \mathsf{Bisim}_{\mathsf{Tree}}\ a\ b
\end{aligned}$$

Its status as an equivalence relation can be proved analogously to bisimilarity of streams, by coinduction. For each property, the defining recursive function will have two recursive calls, instantiated separately for the left and right subtrees.

Chapter III

Epistemic Modal Logic

In this chapter, we will introduce the axiomatic system of epistemic modal logic and its standard relational semantics. Initially, this will be presented in isolation from the system of type theory defined Chapter II, but we will later go on to define a shallow embedding of the logic into our type theory.

The embedding constitutes a novel semantics for epistemic logic based on *knowledge operators*, which will be shown to correspond to the relational semantics. An Agda formalisation of the embedding and the semantics correspondence results are included in Appendix A.2.

3.1 Background

Modal logics are a rich field of study throughout mathematics and philosophy, as well as computer science. They were created in their modern form to study the linguistic modality "necessarily", which qualifies the truth of a statement [20, 27, 40]. For example, the statements "it is raining" and "it is necessarily raining" have different denotations,

Formally, the syntax of a propositional logic is extended with a new unary operator $\Box$ on formulae, with $\Box\,\phi$ interpreted as the statement "ϕ is necessary". There are a number of axiomatisations for the new operator, the weakest of which is a logic called *System K*, which introduces two new rules. The rules use the judgement $\phi_1, \ldots, \phi_n \vdash \psi$ which asserts that the propositional formula ψ is derivable under hypotheses ϕ_1 through ϕ_n. When ψ can be derived axiomatically, we may write $\vdash \psi$. Note that while the notation is similar, these rules are defining a distinct logical system from the type theory defined in Chapter II.

$$\textsc{Necessitation}\,\frac{\vdash \phi}{\vdash \Box\,\phi} \qquad \textsc{Axiom K}\,\frac{}{\vdash \Box\,(\phi \Rightarrow \psi) \Rightarrow \Box\,\phi \Rightarrow \Box\,\psi}$$

The necessitation rule states that when ϕ is a tautology of the underlying propositional logic, then so is $\Box\,\phi$. That is, tautologies of the logic are necessary.

Axiom K asserts necessity distributes over implication. It is a version of the modus ponens rule that can be applied to necessary formulae — if both a formula and an implication with that formula as the premise are necessary, then the conclusion must also be necessary.

To strengthen the logic, several more axioms may be added depending on the intended use. We will skip straight to the modal logic *System 5 (S5)* including the necessitation rule, Axiom K, and three additional axioms.

$$\textsc{Axiom T}\,\frac{}{\vdash \Box\,\phi \Rightarrow \phi} \qquad \textsc{Axiom 4}\,\frac{}{\vdash \Box\,\phi \Rightarrow \Box\,\Box\,\phi}$$

$$\textsc{Axiom 5}\,\frac{}{\vdash \neg\Box\,\phi \Rightarrow \Box\,\neg\Box\,\phi}$$

These axioms dictate what can be derived from the assumption that ϕ is necessary or unnecessary. From Axioms T and 4, if a formula is necessary, then it must actually be true, and its necessity is necessary.

In some presentations of modal logic, Axiom 5 is written $\Diamond\,\phi \Rightarrow \Box\,\Diamond\,\phi$. This uses a new modal operator $\Diamond$, with $\Diamond\,\phi$ meaning "ϕ is possible", with the axiom reading that if a formula is possible, it is necessarily possible. Possibility may be defined in terms of necessity, $\Diamond\,\phi = \neg\Box\,\neg\phi$, that is, a formula is possible when it's not necessarily false. The above version of the axiom, with a single modal operator, is classically equivalent and more suited for use in epistemic modal logic.

If tautological formulae are necessary, and necessary formulae are true, it might not be clear what we gain by adding the $\Box$ modality — what is the difference between the propositions $\Box\,\phi$ and ϕ? The answer is that modal logics are used to model scenarios containing unknowns, where we may be in any one of a number of *possible worlds*, where some formulae might be true, but cannot be derived tautologically.

For example, our earlier statement "it is raining" is true on some days, but false on others. In a mathematical game with hidden information, such as card games or a prisoner dilemma, the state of the play may not be known by a particular player — the truth of the statement "my opponent holds the $Q\heartsuit$ card" is contingent on the state. We add primitive formulae, called *events*, to the logic depending on the scenario, whose truth value is a function of the precise state of the world.

The traditional reading of $\Box\,\phi$ as "ϕ is necessary" can be modified, with different axioms selected, according to the scenario. In philosophy, moral obligation may be studied with a *deontic modal logic*, where $\Box\,\phi$ is read "ϕ is obligatory", with $\Diamond\,\phi$ therefore meaning "ϕ is permitted", and which lacks Axiom T — even if an action is obligatory, it may not actually occur. In a *temporal modal logic*, the modal operator refers to tense, one possible reading of $\Box\,\phi$ being "ϕ is true at all times", and $\Diamond\,\phi$ meaning "ϕ is true some of the time". In this chapter, we are concerned with *epistemic modal logic* [24, 33], where the modal operator is written $\mathsf{K}\ \phi$ with the interpretation "ϕ is known".

3.2 Epistemic Logic and Relational Semantics

In this section, we will see how the S5 modal logic forms an epistemic logic, appropriate for the modelling of knowledge. We will also discuss the traditional semantics of modal logic, based on binary relations, and how these relations are interpreted in epistemic logic.

We have a modal operator which can be applied to logical formulae, $\mathsf{K}\ \phi$ interpreted as "ϕ is known", but known by whom? It could be a player in a mathematical game, a computer on a network, or a Byzantine general planning a coordinated attack on an enemy city [38]. We will not presume any particular scenario, leaving the subject as an abstract *agent*. A natural extension is postulating many such agents, each agent a having their own modal operator K_a, but for now we will only consider a logic with one K operator.

The S5 modal logic introduced in Section 3.1 can be utilised to give an account of an agent's knowledge. Let us consider each rule, giving a brief justification for how it is interpreted in epistemic logic.

- Knowledge Generalisation, if $\vdash \phi$, then $\vdash \mathsf{K}\ \phi$

  This is the necessitation rule of System K, called *knowledge generalisation* in the context of epistemic logic. It states that if ϕ is derivable, then so is $\mathsf{K}\ \phi$. In other words, the agent is capable of using logic to derive theorems.

  This may seem an overly-strong assumption, tantamount to the agent being a perfect logician. However, consider that in order for us to derive $\mathsf{K}\ \phi$ by knowledge generalisation, we must actually have a proof of ϕ. If formula ψ is the statement of an unsolved problem in mathematics, such as the Collatz conjecture, at present we can show neither $\mathsf{K}\ \psi$ nor $\mathsf{K}\ \neg\psi$.

- Axiom K, $\mathsf{K}\ (\phi \Rightarrow \psi) \Rightarrow \mathsf{K}\ \phi \Rightarrow \mathsf{K}\ \psi$

  If both an implication and its premise are known, then so is its conclusion. That is, the agent is capable of using modus ponens with their own knowledge.

  In cases where both ϕ and $\phi \Rightarrow \psi$ are theorems, then so is ψ, and knowledge generalisation is sufficient to conclude $\mathsf{K}\ \psi$. However, this rule allows for the agent to apply logic to known events which are not derivable tautologically.

- Axiom T, $\mathsf{K}\ \phi \Rightarrow \phi$

  If a formula is known, it must actually be true in the present state of the world. This is another strong assumption, but this is what distinguishes knowledge from mere belief or opinion.

  Belief can itself be expressed in *doxastic modal logic*, which might adopt a weaker axiom of belief non-contradiction, that an agent cannot simultaneously believe a statement and its negation, $\neg(\mathsf{B}\ \phi \land \mathsf{B}\ \neg\phi)$.

- Axiom 4, $\mathsf{K}\ \phi \Rightarrow \mathsf{K}\ (\mathsf{K}\ \phi)$

  If a formula is known, then this knowledge is itself known. This is a principle of self-reflection, that our agent is aware of their own knowledge.

- Axiom 5, $\neg\mathsf{K}\ \phi \Rightarrow \mathsf{K}\ (\neg\mathsf{K}\ \phi)$

  If a formula isn't known, then this lack of knowledge is itself known. This is a negative version of Axiom 4, when the agent doesn't know something, they at least know they do not know it.

  When these two axioms are taken together, the agent is perfectly introspective, aware of both what they know and what they don't. This could be called a Socratic principle — *"I know that I know nothing"*.

The relational semantics of modal logic, and therefore epistemic logic, was originally devised by Saul Kripke, who proved it sound and complete with respect to the axiomatic system [37]. The semantics operates on a set of possible worlds, relating them according to the knowledge of the agent.

Formally, a *Kripke frame* is a pair $\langle S,\ R\rangle$ where S is a set of world states and R is a binary relation on states. A Kripke frame with an arbitrary relation R is sufficient to model the modal logic System K, but additional properties can be imposed on the relation which correspond to additional axioms. The following the properties are required of R to model the modal logic S5.

- Knowledge Generalisation, if $\vdash \phi$, then $\vdash \mathsf{K}\ \phi$

  Modelled by any relation R.

- Axiom K, $\mathsf{K}\ (\phi \Rightarrow \psi) \Rightarrow \mathsf{K}\ \phi \Rightarrow \mathsf{K}\ \psi$

  Modelled by any relation R.

- Axiom T, $\mathsf{K}\ \phi \Rightarrow \phi$

  Modelled when R is reflexive, $w\ R\ w$.

- Axiom 4, $\mathsf{K}\ \phi \Rightarrow \mathsf{K}\ (\mathsf{K}\ \phi)$

  Modelled when R is transitive, $w\ R\ v \Rightarrow v\ R\ u \Rightarrow w\ R\ u$.

- Axiom 5, $\neg\mathsf{K}\ \phi \Rightarrow \mathsf{K}\ (\neg\mathsf{K}\ \phi)$
  Modelled when R is Euclidean, $w\ R\ v \Rightarrow w\ R\ u \Rightarrow u\ R\ v$.

A reflexive, Euclidean relation must also be symmetric, and a symmetric, transitive relation is Euclidean, so S5 is modelled precisely when R is an equivalence relation. We omit the full details of the correspondence here as a version of it, embedded in type theory, is formally proven in Section 3.5.

Intuitively, the relation, called an *accessibility relation* or *knowledge relation* in the context of epistemic logic, relates the states of the world that agent's knowledge cannot distinguish. That is, two states w and v are related, $w\ R\ v$, when they differ only by the truth of those formulae ϕ for which $\neg\mathsf{K}\ \phi$.

If $\forall_{(w \in S)}\exists_{(v \in S)}\ w\ R\ v \Rightarrow w = v$, that is, each state is disconnected from all others, beside itself reflexively, the agent modelled by R is omniscient, able to precisely determine the state of the world. If the agent were then to forget an event ϕ, the relation would expand to form equivalence classes relating those states which only differ by the truth of ϕ. In the extreme case, when $\forall_{(w \in S)}\forall_{(v \in S)}\ w\ R\ v$, when R is strongly connected, the agent is solipsistic, completely ignorant of the world around them, only knowing tautologies through knowledge generalisation and the state of their own knowledge though Axioms 4 and 5.

3.3 Type-Theoretic Embedding

We now return to a type-theoretic setting to define an embedding of epistemic modal logic into the logic of type theory. Broadly, there are two methods to embed one logic into another [51], serving different purposes depending on the desired application.

In a *deep embedding*, the syntax of the object logic is defined directly in the metalogic. For example, a deep embedding of epistemic modal logic in type theory could utilise an inductive type to define its syntax trees.

```
inductive Syntax : Type
    ⊥ : Syntax
    ¬ : Syntax → Syntax
    ∨ : Syntax → Syntax → Syntax
    K : Syntax → Syntax
```

To keep the eliminator for the inductive type as simple as possible, only the essential connectives are defined as its constructors. The others may be defined in terms of these using standard classical equivalences.

$$\begin{array}{ll} \top : \mathsf{Syntax} & \wedge : \mathsf{Syntax} \to \mathsf{Syntax} \to \mathsf{Syntax} \\ \top = \neg\bot & \phi \wedge \psi = \neg(\neg\phi \vee \neg\psi) \end{array}$$

$$\begin{array}{ll} \Rightarrow : \mathsf{Syntax} \to \mathsf{Syntax} \to \mathsf{Syntax} & \Leftrightarrow : \mathsf{Syntax} \to \mathsf{Syntax} \to \mathsf{Syntax} \\ \phi \Rightarrow \psi = \neg\phi \vee \psi & \phi \Leftrightarrow \psi = (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \end{array}$$

An axiomatic system is specified over the syntax by defining a relation to establish the valid derivations of the logic. Here, an inductive type family indexed by both Syntax and List Syntax is used, the list representing prior assumptions.

$$\begin{array}{l} \mathsf{inductive\ Derives} : \mathsf{List\ Syntax} \to \mathsf{Syntax} \to \mathsf{Type} \\ \quad \begin{array}{lll} \mathsf{K\text{-}gen} & : \Pi_{(\phi:\mathsf{Syntax})} & \mathsf{Derives}\ [\,]\ \phi \to \mathsf{Derives}\ [\,]\ (\mathsf{K}\ \phi) \\ \mathsf{axiom}_T & : \Pi_{(\phi:\mathsf{Syntax})} & \mathsf{Derives}\ [\,]\ (\mathsf{K}\ \phi \Rightarrow \phi) \\ \mathsf{axiom}_K & : \Pi_{(\phi,\psi:\mathsf{Syntax})} & \mathsf{Derives}\ [\,]\ (\mathsf{K}\ (\phi \Rightarrow \psi) \Rightarrow \mathsf{K}\ \phi \Rightarrow \mathsf{K}\ \psi) \\ \mathsf{axiom}_4 & : \Pi_{(\phi:\mathsf{Syntax})} & \mathsf{Derives}\ [\,]\ (\mathsf{K}\ \phi \Rightarrow \mathsf{K}\ (\mathsf{K}\ \phi)) \\ \mathsf{axiom}_5 & : \Pi_{(\phi:\mathsf{Syntax})} & \mathsf{Derives}\ [\,]\ (\neg\mathsf{K}\ \phi \Rightarrow \mathsf{K}\ (\neg\mathsf{K}\ \phi)) \\ \ldots & & \end{array} \end{array}$$

A closed term of type Derives $\Delta\ \phi$ is a proof tree for the derivation of $\Delta \vdash \phi$ in epistemic logic. Only the additional rules of epistemic logic are shown above, but all of the rules for the base propositional logic will also need be included in the relation.

This style of embedding is appropriate for studying the metatheoretic properties of the logic [39]. An interpretation function may be defined, mapping from Syntax to a semantic model, with soundness and completeness results able to be stated and proved in the metalogic of type theory by structural induction. However, proving theorems of the object logic using a deep embedding can be cumbersome — the rules for both the object logic and the metalogic must be observed.

Alternatively, a *shallow embedding* may be used if the objective is theorem proving in the object logic. The metalogic is used as a semantic model of the object logic — rather than confining syntax trees to a single type, the connectives of the metalogic are used to encode them. This allows for the full feature set of the metalogic to be brought to bear in embedded proofs [12].

The rest of this chapter is concerned with a shallow embedding of epistemic logic in type theory. We will see how type-theoretic features such as type families and, in Chapter IV, coinduction may be used to define the rules of epistemic logic at a higher level than is possible with a deep embedding.

The embedding is based on the concept of possible worlds, so we fix a type to represent these worlds, $\mathsf{State} : \mathsf{Type}$. A formula in the syntax of epistemic logic may be interpreted as true in some worlds but false in others, so embedded formulae are events, predicates on states, encoded as type families indexed by State.

Definition 3.3.1. An event is a type family indexed by the type of possible worlds.

$$\begin{aligned} &\mathsf{Event} : \mathsf{Type} \\ &\mathsf{Event} = \mathsf{State} \to \mathsf{Type} \end{aligned}$$

As a reminder, to ensure consistency, since both State and Type are used in its definition, there are constraints on which type universe Event itself inhabits. Its level must be at least as high as the universe of State and strictly higher than the universe that is the codomain of the type family, with the actual level being the least value which satisfies both constraints. With this noted, we shall continue being implicit about levels in future definitions — the Agda formalisation in Appendix A details them in full.

The syntax of a propositional logic is built atop the constructs of type theory. The constants are constant type families, while the connectives are functions combining type families using the type formers introduced in Chapter II.

Definition 3.3.2. Event constants and connectives.

$$\begin{array}{lll} \bot : \mathsf{Event} & \top : \mathsf{Event} & \neg : \mathsf{Event} \to \mathsf{Event} \\ \bot = \lambda(w : \mathsf{State}).\ \mathbb{0} & \top = \lambda(w : \mathsf{State}).\ \mathbb{1} & \neg\phi = \lambda(w : \mathsf{State}).\ \phi\ w \to \mathbb{0} \end{array}$$

$$\begin{array}{ll} \vee : \mathsf{Event} \to \mathsf{Event} \to \mathsf{Event} & \wedge : \mathsf{Event} \to \mathsf{Event} \to \mathsf{Event} \\ \phi \vee \psi = \lambda(w : \mathsf{State}).\ \phi\ w + \psi\ w & \phi \wedge \psi = \lambda(w : \mathsf{State}).\ \phi\ w \times \psi\ w \end{array}$$

$$\begin{array}{ll} \Rightarrow : \mathsf{Event} \to \mathsf{Event} \to \mathsf{Event} & \Leftrightarrow : \mathsf{Event} \to \mathsf{Event} \to \mathsf{Event} \\ \phi \Rightarrow \psi = \lambda(w : \mathsf{State}).\ \phi\ w \to \psi\ w & \phi \Leftrightarrow \psi = \lambda(w : \mathsf{State}).\ \phi\ w \leftrightarrow \psi\ w \end{array}$$

An event may be seen set-theoretically as the subset of states where the event occurs. With this view, applying an event to a state, $\phi\ w$, is obtaining the truth assignment of $w \in \phi$. The constants $\bot$ and $\top$ are the empty and universal sets, $\neg$ is the complement, with $\vee$, $\wedge$, and $\Rightarrow$ being the union, intersection, and exponent, respectively.

We can also define predicates and relations over events. Rather than representing connectives in the syntax of the object logic, these correspond to metatheoretic statements — a shallow embedding allows these to coexist in the same language.

Definition 3.3.3. Event predicates and relations.

$$\begin{aligned} &\forall\!\!\!\forall : \mathsf{Event} \to \mathsf{Type} \\ &\forall\!\!\!\forall\, \phi = \Pi_{(w:\mathsf{State})}\ \phi\ w \end{aligned}$$

$$\begin{aligned} &\subseteq\, : \mathsf{Event} \to \mathsf{Event} \to \mathsf{Type} && \approx\, : \mathsf{Event} \to \mathsf{Event} \to \mathsf{Type} \\ &\phi \subseteq \psi = \forall\!\!\!\forall(\phi \Rightarrow \psi) && \phi \approx \psi = \forall\!\!\!\forall(\phi \Leftrightarrow \psi) \end{aligned}$$

The statement $\forall\!\!\!\forall\, \phi$ states that ϕ is a tautology in the object logic, true in every possible world. Continuing the set theory analogy, it means that ϕ is the universal set, with $w \in \phi$ for any state w. The tautological implication, $\phi \subseteq \psi$, is the subset relation, whenever $w \in \phi$, then $w \in \psi$. Two events are propositionally equivalent, $\phi \approx \psi$, when this holds bidirectionally, being equal sets.

One advantage of the shallow embedding style is immediately apparent. As we have defined the connectives of the embedded propositional logic using those of type theory, their properties follow straightforwardly with little extra effort. For example, unwrapping the definitions in $\phi \wedge \psi \subseteq \psi \wedge \phi$, it can be seen to be nothing more than the type-theoretic theorem that products are commutative, specialised to type families ϕ and ψ.

$$\Pi_{(w:\mathsf{State})}\ \phi\ w \times \psi\ w \to \psi\ w \times \phi\ w$$

This embedding is enough to begin modelling some basic scenarios in epistemic logic. For example, imagine that a coin has been tossed and a six-sided die rolled. Our chosen type of states should at least distinguish these primitive events.

C_H	"the coin landed heads-side up"
C_T	"the coin landed tails-side up"

D_1	"the die rolled a 1"
	$\vdots$
D_6	"the die rolled a 6"

Then, for example, $D_2 \vee D_4 \vee D_6$ is the event that occurred in those states where the die rolled an even number. We would also expect certain statements like $\forall\!\!\!\forall\, \neg(C_H \wedge C_T)$ to hold, so there cannot be inconsistent states where the coin or die give multiple outcomes. One possible choice of states for this scenario is $\mathsf{State} = \mathsf{Fin}\ 2 \times \mathsf{Fin}\ 6$, using finite types to represent the outcomes for the coin and die independently.

Notably absent, however, are any knowledge modalities. Being unary operators in the syntax of the embedded epistemic logic, it is clear that they should have type $\mathsf{Event} \to \mathsf{Event}$. Their semantics will be defined formally in Section 3.4, but for now assume we do have a function K_a representing the knowledge of some agent a. In our example, we now have additional events to model, $\mathsf{K}_a\ D_1$, $\mathsf{K}_a\ D_2$, and so on, but also more complex events like $\neg(\mathsf{K}_a\ C_H \vee \mathsf{K}_a\ C_T)$, "agent a doesn't know on which side the coin landed". The state type $\mathsf{Fin}\ 2 \times \mathsf{Fin}\ 6$ is no longer rich enough to encode this detail.

If we then include another agent b with associated knowledge operator K_b, we also have events about one agents knowledge of the other's knowledge to contend with. For example, the event $\mathsf{K}_a\ (\mathsf{K}_b\ D_3) \wedge \neg\mathsf{K}_b\ (\mathsf{K}_a\ (\mathsf{K}_b\ D_3))$ states "agent a knows that agent b knows the die rolled a 3, but b doesn't know that a knows this". Due to this added complexity, it is simpler to only talk abstractly about states, postulating properties such as the coin consistency principle stated above, rather than giving State a concrete definition which must distinguish the knowledge of agents.

3.4 Knowledge Operator Semantics

We now come to defining what it means to be a knowledge operator in our embedding. Instead of defining a function $\mathsf{K} : \mathsf{Event} \to \mathsf{Event}$ directly as we have with the other syntactic constructs of epistemic logic, we shall define them semantically, stating a set of properties a function must satisfy in order to be appropriate to represent the knowledge of an agent. The semantics will be based on the rules of the S5 system of modal logic introduced in Sections 3.1 and 3.2, defined in the syntax of our embedded epistemic logic.

$$
\begin{aligned}
&\Cup\phi \to \Cup\mathsf{K}\ \phi && \text{(Knowledge Generalisation)}\\
&\mathsf{K}\ (\phi \Rightarrow \psi) \subseteq \mathsf{K}\ \phi \Rightarrow \mathsf{K}\ \psi && \text{(Axiom K)}\\
&\mathsf{K}\ \phi \subseteq \phi && \text{(Axiom T)}\\
&\mathsf{K}\ \phi \subseteq \mathsf{K}\ (\mathsf{K}\ \phi) && \text{(Axiom 4)}\\
&\neg\mathsf{K}\ \phi \subseteq \mathsf{K}\ (\neg\mathsf{K}\ \phi) && \text{(Axiom 5)}
\end{aligned}
$$

We are not limited to the finitary syntax of the object logic, however. We are able to make stronger semantic statements which may involve entailments to and from potentially infinite sets of formulae.

Recall that a type family indexed by type T is a function $T \to \mathsf{Type}$ which may be thought of as a set of types. In the same way, we define event families as functions $T \to \mathsf{Event}$, which are sets of formulae in epistemic logic. In full, an event family is a function $T \to \mathsf{State} \to \mathsf{Type}$, a binary type family indexed by T and State.

Definition 3.4.1. An event family Φ semantically entails an event family Ψ when, in every state, if Φ holds universally, at all of its indices, then Ψ must also hold universally.

$$\begin{aligned}&\vDash : (A \to \mathsf{Event}) \to (B \to \mathsf{Event}) \to \mathsf{Type}\\ &\Phi \vDash \Psi = \Pi_{(w:\mathsf{State})}\ (\Pi_{(a:A)}\ \Phi\ a\ w) \to (\Pi_{(b:B)}\ \Psi\ b\ w)\end{aligned}$$

By composing an event family Φ with a function $\mathsf{K} : \mathsf{Event} \to \mathsf{Event}$, we form a new event family, indexed by the same type, where K is mapped onto the family. When $\Phi \vDash \Psi$ implies $\mathsf{K} \circ \Phi \vDash \mathsf{K} \circ \Psi$, we say that K *preserves semantic entailment*. We require that knowledge operators preserve semantic entailment in place of the knowledge generalisation and the Axiom K property.

In what follows, we are most often concerned about entailments from an event family into a single event, instantiating Ψ with a constant event family, $\lambda b.\ \psi$. We shall write this without the extraneous λ-term, $\Phi \vDash \psi$.

Definition 3.4.2. A knowledge operator is any function $\mathsf{K} : \mathsf{Event} \to \mathsf{Event}$ satisfying preservation of semantic entailment in addition to the S5 properties Axiom T, Axiom 4, and Axiom 5. That is, the type of knowledge operators is defined as the following dependent tuple type.

$$\begin{aligned}&\mathsf{KOp} : \mathsf{Type}\\ &\mathsf{KOp} = \Sigma_{(\mathsf{K}:\mathsf{Event}\to\mathsf{Event})} && \Pi_{(\Phi:\mathsf{Event}\to\mathsf{Type})}\ \Pi_{(\psi:\mathsf{Event})} && \Phi \vDash \psi \to \mathsf{K} \circ \Phi \vDash \mathsf{K}\ \psi\\ & && \times\ \Pi_{(\phi:\mathsf{Event})} && \mathsf{K}\ \phi \subseteq \phi\\ & && \times\ \Pi_{(\phi:\mathsf{Event})} && \mathsf{K}\ \phi \subseteq \mathsf{K}\ (\mathsf{K}\ \phi)\\ & && \times\ \Pi_{(\phi:\mathsf{Event})} && \neg\mathsf{K}\ \phi \subseteq \mathsf{K}\ (\neg\mathsf{K}\ \phi)\end{aligned}$$

Though not explicitly included in their definition, it can be shown that knowledge operators retain knowledge generalisation and the Axiom K property. In each case, this follows from the preservation of semantic entailment by choosing an appropriate event family.

Theorem 3.4.3. *Knowledge operators satisfy knowledge generalisation,* $\forall\!\!\forall\, \phi \to \forall\!\!\forall\, \mathsf{K}\ \phi$.

Proof. Unfolding $\forall\!\!\forall\, \phi \to \forall\!\!\forall\, \mathsf{K}\ \phi$ according to Definition 3.3.3, we are to prove the statement $(\Pi_{(w:\mathsf{State})}\ \phi\ w) \to (\Pi_{(v:\mathsf{State})}\ \mathsf{K}\ \phi\ v)$. We therefore proceed by assuming that ϕ holds in any state, and continue to show $\mathsf{K}\ \phi\ v$ for an arbitrary state v.

This is equivalent to the nullary entailment $\mathsf{K} \circ \mathsf{elim}_{\mathbb{0}} \vDash \mathsf{K}\ \phi$, from an event family indexed by the empty type, $\mathbb{0} \to \mathsf{Event}$. Applying preservation of semantic entailment, we have to show that $\mathsf{elim}_{\mathbb{0}} \vDash \phi$, whose conclusion is proved by our original assumption that ϕ holds universally. □

Theorem 3.4.4. *Knowledge operators satisfy Axiom K,* $\mathsf{K}\ (\phi \Rightarrow \psi) \Rightarrow \mathsf{K}\ \phi \Rightarrow \mathsf{K}\ \psi$.

Proof. The theorem statement $\mathsf{K}\ (\phi \Rightarrow \psi) \subseteq \mathsf{K}\ \phi \Rightarrow \mathsf{K}\ \psi$ is judgementally equal to $\Pi_{(w:\mathsf{State})}\ \mathsf{K}\ (\phi \Rightarrow \psi)\ w \to \mathsf{K}\ \phi\ w \to \mathsf{K}\ \psi\ w$. We assume hypotheses $\mathsf{K}\ (\phi \Rightarrow \psi)\ w$ and $\mathsf{K}\ \phi\ w$ towards the goal $\mathsf{K}\ \psi\ w$.

Define the event family $B = \mathsf{elim}_{+}\ (\mathsf{elim}_{\mathbb{1}}\ (\phi \Rightarrow \psi))\ (\mathsf{elim}_{\mathbb{1}}\ \phi) : \mathbb{2} \to \mathsf{Event}$, indexed by the Booleans. Applying preservation of semantic entailment with B, it suffices to show $B \vDash \psi$ and $\Pi_{(b:\mathbb{2})}\ \mathsf{K}\ (B\ b)\ w$, the latter of which is simply a pairing of our hypotheses.

To prove $B \vDash \psi$, we may assume a dependent function of type $\Pi_{(b:\mathbb{2})}\ B\ b\ w$ and use it to obtain a value of type $\psi\ w$. Applying the function to true and false, we have $\phi\ w \to \psi\ w$ and $\phi\ w$, respectively, yielding $\psi\ w$ by modus ponens. □

3.5 Correspondence with Relational Semantics

To prove the knowledge operator semantics correspond to the relational Kripke semantics of epistemic logic, we define transformation functions between their types. First, the type of Kripke frames is defined in a similar way to the knowledge operator semantics, as a dependent tuple type collecting an equivalence relation and its properties.

Definition 3.5.1. An equivalence relation over some type A is a binary type family $R : A \to A \to \mathsf{Type}$ satisfying the properties of reflexivity, symmetry, and transitivity.

$$\begin{aligned}
&\mathsf{EquivRel} : \mathsf{Type} \to \mathsf{Type}\\
&\mathsf{EquivRel}\ A = \Sigma_{(R:A\to A\to\mathsf{Type})}\ && \Pi_{(a:A)} && a\ R\ a\\
& && \times\ \Pi_{(a,b:A)} && a\ R\ b \to b\ R\ a\\
& && \times\ \Pi_{(a,b,c:A)} && a\ R\ b \to b\ R\ c \to a\ R\ c
\end{aligned}$$

A Kripke frame is an equivalence relation over the type of possible worlds, $\mathsf{Kripke} = \mathsf{EquivRel}\ \mathsf{State}$.

We will define the transformation from a function $\mathsf{K} : \mathsf{Event} \to \mathsf{Event}$ to a relation $\mathsf{State} \to \mathsf{State} \to \mathsf{Type}$ without considering the knowledge operator and equivalence relation properties. Separately, we will demonstrate that if K is a knowledge operator, then the transformation must endow the equivalence relation properties on its output.

Definition 3.5.2. Transforming an event operator K, define the binary relation which holds between states whose events are not distinguished by K. That is, $w\ R\ v$ when, $\mathsf{K}\ \phi\ w$ and $\mathsf{K}\ \phi\ v$ are propositionally equivalent for all ϕ.

If classical reasoning is admitted, only one direction of the equivalence is sufficient, the other being derivable. For simplicity, this is the version of the transformation which will be used going forward.

$$\begin{aligned}
&R_{[\_]} : (\mathsf{Event} \to \mathsf{Event}) \to \mathsf{State} \to \mathsf{State} \to \mathsf{Type} \\
&w\ R_{[\mathsf{K}]}\ v = \Pi_{(\phi:\mathsf{Event})}\ \mathsf{K}\ \phi\ w \to \mathsf{K}\ \phi\ v
\end{aligned}$$

Note that, since this transformation is quantified over events, the resulting type universe must be at a higher level than the one Event itself resides in.

Theorem 3.5.3. *If* K *is a knowledge operator, then* $R_{[\mathsf{K}]}$ *is an equivalence relation.*

Proof. Reflexivity and transitivity are immediate from fact that the underlying implication relation is a preorder — see Theorems 2.3.1 and 2.3.2.

To show symmetry, assume $w\ R_{[\mathsf{K}]}\ v$, which is $\Pi_{(\phi:\mathsf{Event})}\ \mathsf{K}\ \phi\ w \to \mathsf{K}\ \phi\ v$ according to Definition 3.5.2. We are to prove the opposite direction for this implication, so also assume $\mathsf{K}\ \phi\ v$ and aim to show $\mathsf{K}\ \phi\ w$. Using the classical principle of double-negation elimination, assume $\mathsf{K}\ \phi\ w \to \mathbb{0}$, towards a contradiction.

By Definition 3.3.2, this assumption is equal to $(\neg\mathsf{K}\ \phi)\ w$. Since K is a knowledge operator, using the Axiom 5 property we can derive $\mathsf{K}\ (\neg\mathsf{K}\ \phi)\ w$, from which we can conclude $\mathsf{K}\ (\neg\mathsf{K}\ \phi)\ v$ by instantiating our first hypothesis at index $\neg\mathsf{K}\ \phi$. By Axiom T, this implies $(\neg\mathsf{K}\ \phi)\ v$, which can be written $\mathsf{K}\ \phi\ v \to \mathbb{0}$, but this contradicts our assumption of $\mathsf{K}\ \phi\ v$. Therefore, $\mathsf{K}\ \phi\ w$ is proved by contradiction. $\square$

We now come to defining the opposite transformation, from a binary relation $R : \mathsf{State} \to \mathsf{State} \to \mathsf{Type}$ into an operator $\mathsf{Event} \to \mathsf{Event}$.

Definition 3.5.4. Transforming a binary relation R, define the event operator whose knowledge is preserved by R. That is, $\mathsf{K}\ \phi$ holds in state w when ϕ holds in all states that are related to w.

$$\begin{aligned}
&\mathsf{K}_{[\_]} : (\mathsf{State} \to \mathsf{State} \to \mathsf{Type}) \to \mathsf{Event} \to \mathsf{Event} \\
&\mathsf{K}_{[R]}\ \phi\ w = \Pi_{(v:\mathsf{State})}\ w\ R\ v \to \phi\ v
\end{aligned}$$

We demonstrate each knowledge operator property of $\mathsf{K}_{[R]}$ only assuming the relevant equivalence relation properties in each case.

Lemma 3.5.5. *Without any constraints on the relation,* $\mathsf{K}_{[R]}$ *preserves semantic entailment.*

Proof. Assume for some event family $\Phi : A \to \mathsf{Event}$ and event ψ that $\Phi \vDash \psi$. We want to reach the conclusion $\mathsf{K}_{[R]} \circ \Phi \vDash \mathsf{K}_{[R]}\ \psi$, which, applying Definitions 3.4.1

and 3.5.4, is the following statement.

$$\Pi_{(w:\mathsf{State})}\ (\Pi_{(a:A)}\ \Pi_{(v:\mathsf{State})}\ w\ R\ v \to \Phi\ a\ v) \to \Pi_{(v:\mathsf{State})}\ w\ R\ v \to \psi\ v$$

With additional hypotheses $(\Pi_{(a:A)}\ \Pi_{(v:\mathsf{State})}\ w\ R\ v \to \Phi\ a\ v)$ and $w\ R\ v$, we need to show $\psi\ v$. This follows from the original assumption of $\Phi \vDash \psi$ if we can show $\Phi\ a\ v$. But this is proved by our first additional hypothesis, instantiated at indices a and v, as we have already assumed $w\ R\ v$. □

Lemma 3.5.6. *When R is reflexive, $\mathsf{K}_{[R]}$ satisfies Axiom T.*

Proof. Assume $\mathsf{K}_{[R]}\ \phi\ w$ with the goal of proving $\phi\ w$. By Definition 3.5.4, the assumption expands to $\Pi_{(v:\mathsf{State})}\ w\ R\ v \to \phi\ v$, so instantiate it at w to obtain $w\ R\ w \to \phi\ w$. Our desired $\phi\ w$ therefore follows from the reflexivity of R. □

Lemma 3.5.7. *When R is transitive, $\mathsf{K}_{[R]}$ satisfies Axiom 4.*

Proof. Assume $\mathsf{K}_{[R]}\ \phi\ w$, which, as above, is $\Pi_{(v:\mathsf{State})}\ w\ R\ v \to \phi\ v$. We want to reach the conclusion $\mathsf{K}_{[R]}\ (\mathsf{K}_{[R]}\ \phi)\ w$, which is judgementally equal to the following.

$$\Pi_{(v:\mathsf{State})}\ w\ R\ v \to \Pi_{(u:\mathsf{State})}\ v\ R\ u \to \phi\ u$$

Assume $w\ R\ v$ and $v\ R\ u$ and use transitivity to obtain $w\ R\ u$. The result $\phi\ u$ is then proved by applying the original assumption at index u. □

Lemma 3.5.8. *When R is symmetric and transitive, $\mathsf{K}_{[R]}$ satisfies Axiom 5.*

Proof. Assume $\neg\mathsf{K}_{[R]}\ \phi\ w$. After unfolding $\mathsf{K}_{[R]}\ (\neg\mathsf{K}_{[R]}\ \phi)\ w$, we are to derive a contradiction.

$$\Pi_{(v:\mathsf{State})}\ w\ R\ v \to (\Pi_{(u:\mathsf{State})}\ v\ R\ u \to \phi\ u) \to \mathbb{0}$$

If we can prove $\mathsf{K}_{[R]}\ \phi\ w$, which is judgementally equal to $\Pi_{(t:\mathsf{State})}\ w\ R\ t \to \phi\ t$, this will contradict our assumption. To this end, assume additional hypotheses $w\ R\ v$, $w\ R\ t$, and $\Pi_{(u:\mathsf{State})}\ v\ R\ u \to \phi\ u$, setting out to prove $\phi\ t$.

Using symmetry and transitivity with the first two of these, we derive $v\ R\ t$. Then, applying the third hypothesis at index t reaches the desired result. □

Theorem 3.5.9. *When R is an equivalence relation, $\mathsf{K}_{[R]}$ is a knowledge operator.*

Proof. Aggregating Lemmas 3.5.5 to 3.5.8 gives the full theorem. □

Theorems 3.5.3 and 3.5.9 show that transforming knowledge operators and equivalence relations always yields equivalence relations and knowledge operators, respectively. For a complete correspondence between the two semantics, we furthermore want to show that the transformations are inverse, that knowledge operators and equivalence relations are isomorphic.

The isomorphism will be established up to propositional equivalence. For example, we won't show $\mathsf{K} \equiv \mathsf{K}_{[R_{[\mathsf{K}]}]}$, using propositional equality to state they are actually identical objects. Rather, we will show that the types $\mathsf{K}\ \phi\ w$ and $\mathsf{K}_{[R_{[\mathsf{K}]}]}\ \phi\ w$ are coinhabited — if either can be proved, then so can the other.

Theorem 3.5.10. *When $R_{[\_]}$ is applied to a knowledge operator, $\mathsf{K}_{[\_]}$ is its inverse, up to propositional equivalence. That is, $\mathsf{K}\ \phi \approx \mathsf{K}_{[R_{[\mathsf{K}]}]}\ \phi$.*

Proof. The left-to-right direction is $\mathsf{K}\ \phi \subseteq \mathsf{K}_{[R_{[\mathsf{K}]}]}\ \psi$, so assume $\mathsf{K}\ \phi\ w$. The conclusion is judgementally equal to the following statement.

$$\Pi_{(v:\mathsf{State})}\ (\Pi_{(\psi:\mathsf{Event})}\ \mathsf{K}\ \psi\ w \to \mathsf{K}\ \psi\ v) \to \phi\ v$$

With the additional assumption of $\Pi_{(\psi:\mathsf{Event})}\ \mathsf{K}\ \psi\ w \to \mathsf{K}\ \psi\ v$ instantiated at ϕ, we can derive $\mathsf{K}\ \phi\ v$, and then $\phi\ v$ follows from the Axiom T property.

For the right-to-left direction, assume $\mathsf{K}_{[R_{[\mathsf{K}]}]}\ \phi\ w$ — this is the same as the conclusion for the left-to-right direction, so its full definition is above. To show $\mathsf{K}\ \phi\ w$, we will use preservation of semantic entailment using as event family all those events which are known in w. Formally, we define the event family $\Phi = \lambda(p : \Sigma_{(\psi:\mathsf{Event})}\ \mathsf{K}\ \psi\ w).\ \mathsf{K}\ (\mathsf{elim}_\Sigma\ (\lambda\psi.\ \lambda k.\ \psi))$. It is indexed by dependent pairs, $\langle\psi,\ k\rangle$, where k is a proof of $\mathsf{K}\ \psi\ w$, with the output being the event $\mathsf{K}\ \psi$.

To see that our premise $\mathsf{K}_{[R_{[\mathsf{K}]}]}\ \phi\ w$ implies the entailment $\Phi \vDash \phi$, assume $\Pi_{(p:\Sigma_{(\psi:\mathsf{Event})}\ \mathsf{K}\ \psi\, w)}\ \Phi\ p\ v$. Then, given a proof $k : \mathsf{K}\ \psi\ w$, we can instantiate the assumption at index $\langle\psi,\ k\rangle$, yielding $\Phi\ k\ v$, which is $\mathsf{K}\ \psi\ v$. Applying the premise at event ϕ therefore proves $\phi\ v$.

Finally, we must show $\Pi_{(p:\Sigma_{(\psi:\mathsf{Event})}\ \mathsf{K}\ \psi\, w)}\ (\mathsf{K} \circ \Phi)\ p\ w$. Given an arbitrary index $\langle\psi,\ k\rangle$ where $k : \mathsf{K}\ \psi\ w$, this is $\mathsf{K}\ (\mathsf{K}\ \psi)\ w$, so we apply the Axiom 4 property to the second element of the pair. □

Theorem 3.5.11. *When $\mathsf{K}_{[\_]}$ is applied to an equivalence relation, $R_{[\_]}$ is its inverse, up to propositional equivalence. That is, $w\ R\ v \leftrightarrow w\ R_{[\mathsf{K}_{[R]}]}\ v$.*

Proof. In the left-to-right direction, expanding only the outer transformation, we want to demonstrate that the knowledge information of $\mathsf{K}_{[R]}$ can be transported between states related by R.

$$w\ R\ v \to \Pi_{(\phi:\mathsf{Event})}\ \mathsf{K}_{[R]}\ \phi\ w \to \mathsf{K}_{[R]}\ \phi\ v$$

But, in fact, this is an instance the Axiom 4 property specialised to $\mathsf{K}_{[R]}$, whose use is justified by Theorem 3.5.9. This can be seen by selectively expanding definitions in the statement $\mathsf{K}_{[R]}\ \phi \subseteq \mathsf{K}_{[R]}\ (\mathsf{K}_{[R]}\ \phi)$.

$$\Pi_{(w:\mathsf{State})}\ \mathsf{K}_{[R]}\ \phi\ w \to \Pi_{(v:\mathsf{State})}\ w\ R\ v \to \mathsf{K}_{[R]}\ \phi\ v$$

For the right-to-left direction, assume $w\ R_{[\mathsf{K}_{[R]}]}\ v$. After fully-unfolding, this is judgementally equal to the following statement.

$$\Pi_{(\phi:\mathsf{Event})}\ (\Pi_{(u:\mathsf{State})}\ w\ R\ u \to \phi\ u) \to \Pi_{(u:\mathsf{State})}\ v\ R\ u \to \phi\ u$$

Instantiating with event $\phi = \lambda t.\ w\ R\ t$ and state v, its conclusion becomes our desired $w\ R\ v$.

$$(\Pi_{(u:\mathsf{State})}\ w\ R\ u \to w\ R\ u) \to v\ R\ v \to w\ R\ v$$

We need only provide the identity function $\mathsf{id}_{(wRu)}$ and a proof of the reflexive property, $v\ R\ v$. □

This isomorphism justifies working with either representation of knowledge equivalently, whichever is more convenient at each point. Given a knowledge operator or an equivalence relation over states, we can leverage our transformations to obtain the other, complete with all of its expected properties, and preserving the truth value when transforming back.

In Chapter IV, we shall work with the shallow embedding defined in this chapter to define the concept of *common knowledge*. A type of agents will be assumed, with an equivalence relation postulated for each. The agents' knowledge operators are defined directly using the $\mathsf{K}_{[\_]}$ transformation, and the correspondence between the two semantics will be essential.

Chapter IV

Coinductive Common Knowledge

In this chapter, we introduce the concept of *common knowledge* in epistemic modal logic with multiple agents, and show how the relational semantics is extended to support the new operator. We will then use the embedding defined in Chapter III to define a common knowledge operator directly, using the feature of coinductive types in type theory.

This coinductive common knowledge operator will be shown to correspond to the intuitive notion of iterated universal knowledge. Furthermore, the common knowledge operator induced by the relation semantics will also be defined in the embedding, and proved equivalent to the coinductive definition.

An Agda formalisation of both definitions for common knowledge and their equivalence is included in Appendix A.3

4.1 Background

Common knowledge is a concept in epistemic logic when it is extended to more than one agent. If a fact ϕ is common knowledge among the agents, it not only implies that it is known universally, but also that the agents can reason about each others' knowledge of ϕ to an arbitrary degree. While common knowledge was first given a mathematical definition by Robert Aumann [2], the notion was discussed much earlier in the philosophical literature [41].

Assumptions of common knowledge permeate society. For example, it can be assumed that it is common knowledge among UK drivers that they are to drive on the left side of the road. Not only do drivers assume that all other drivers know this, they also assume that all drivers assume this of all other drivers, and so on. If it were announced that the nation would suddenly switch to driving on the right, common knowledge of this fact would have to be established before drivers could feel reasonably safe on the road again.

Formally, in the syntax of epistemic logic, given a set of agents A, there is a modal operator K_a for each agent $a \in A$, and we have new modal operators for universal knowledge, EK read "everyone knows", and for common knowledge, CK. Universal knowledge of ϕ is defined as the conjunction of the knowledge for all agents, $\mathsf{EK}\ \phi = \bigwedge_{a \in A} \mathsf{K}_a\ \phi$, while common knowledge of ϕ is understood as an infinite conjunction, $\mathsf{CK}\ \phi = \bigwedge_{n \in \mathbb{N}} \mathsf{EK}^n\ \phi$, combining all finite iterations of EK [4]. That is, from an assumption of $\mathsf{CK}\ \phi$, we can derive $\mathsf{EK}\ \phi$, and therefore $\mathsf{K}_a\ \phi$ for any agent a, as well as $\mathsf{EK}\ (\mathsf{EK}\ \phi)$, $\mathsf{EK}\ (\mathsf{EK}\ (\mathsf{EK}\ \phi))$, and so on ad infinitum.

Let us explore the semantics of common knowledge in the context of a famous logic puzzle, known as the *Blue Eyes Puzzle* or the *Muddy Children Puzzle*, among other variants. The following is a rephrasing of the first version I heard, from Randall Munroe's XKCD website [46], but the puzzle was widely shared as at least as early as the 1950s [10, 26].

> A group of logicians are performing an experiment on a remote island. Every night, a ferry arrives which will allow them to leave, but only those who can accurately determine the colour of their eyes are be allowed to board. While every logician does want to leave, they are committed to the experiment and will never guess their eye colour, only leaving when they can be certain by logical deduction. Due to the experimental parameters, the logicians cannot communicate in any way, but each is able to see the others, and each trusts in the deduction ability of the others.
>
> One day, impatient after no logician has made an attempt to leave, the experiment organiser makes an announcement, *"at least one of you has blue eyes"*. In fact, there are three total blue-eyed logicians, and everyone already knows the fact that was announced — those without blue eyes can see the three with blue eyes, and each of those three can see two others. Nevertheless, all three blue-eyed logicians leave on the third ferry after the announcement. Why?

The puzzle involves common knowledge in two ways. First, we may assume that the setup of the puzzle is common knowledge among all the logicians. For example, each will leave the island when deducing their own eye colour, but they also assume this of the others, and assume that this is common knowledge. This is similar to Aumann's assumption of common knowledge of *rationality* in game theory [3] — each player will maximise their own payoff, assumes the other players will do the same, and that this is common knowledge.

Second, when the announcement is made, it might initially seem that no additional information is given since the statement was already known universally.

However, assuming the announcement is trusted, and the trust is itself common knowledge, what was universal knowledge before the announcement becomes common knowledge after the announcement.

Let ϕ be the statement of the announcement, that there is at least one blue-eyed logician. Since each logician can see at least two others with blue eyes, and the rationality of the logicians is common knowledge, they are all able to deduce $\mathsf{EK}\ \phi$ before the announcement. However, after the announcement, $\mathsf{CK}\ \phi$ holds and only then are they able to make the crucial deduction, $\mathsf{EK}\ (\mathsf{EK}\ \phi)$. This can be generalised by the following inductive argument.

Theorem 4.1.1. *When there are $n \geq 1$ blue-eyed logicians, they will all leave on the n^{th} ferry after the announcement.*

Proof. If there is only one blue-eyed logician, before the announcement he doesn't know ϕ. After the announcement, $\mathsf{EK}\ \phi$ holds, so the blue-eyed logician can deduce ϕ. Seeing no one else with blue eyes, the statement must refer to him, so he leaves on the first ferry.

If there are two blue-eyed logicians, they each know ϕ before the announcement, but not $\mathsf{EK}\ \phi$ — not knowing they have blue eyes themselves, they each reason that they could be in a possible world corresponding to one blue-eyed case. After the announcement, $\mathsf{EK}\ (\mathsf{EK}\ \phi)$ holds, so each can deduce $\mathsf{EK}\ \phi$. In particular, they each now know that ϕ is known by the other. The next day, upon seeing the other still present, they rule out that possible world, leaving only the one where they both have blue eyes, so they both leave on the second ferry.

In general, when there are $n \geq 2$ blue-eyed logicians, each of them knows $\mathsf{EK}^{n-2}\ \phi$ from their own observations, but $\mathsf{EK}^{n-1}\ \phi$ isn't universally known. After the announcement, $\mathsf{EK}^{n}\ \phi$ holds, so they can now deduce $\mathsf{EK}^{n-1}\ \phi$. Given this, all blue-eyed logicians reason by inductive hypothesis that if there are only $n-1$ of them, they will all leave on night $n-1$. However, $n-1$ nights pass without anyone leaving, falsifying the premise of the hypothesis — there must actually be at least n blue-eyed logicians. Only seeing $n-1$, they each conclude there must be exactly n, which must include them, therefore all n leave the next night, on the n^{th} ferry. $\square$

Recall that in the semantics of epistemic logic, the knowledge of an agent may be modelled by an equivalence relation over possible worlds, called a Kripke frame. In multi-agent scenarios, each agent has their own relation independent of the others. We can describe the knowledge content of the puzzle scenario by giving a knowledge relation for each logician.

The relevant information about the state of the world is whether each logician has blue eyes or not. We can represent this information as a tuple of Booleans, for example $\langle \top, \top, \top \rangle$ for three total logicians, each of whom have blue eyes. In this state, since the first logician doesn't know his own eye colour, he could be in state $\langle \bot, \top, \top \rangle$ as far as he is aware, so these states are related in the model, but he can distinguish states which differ by the other components, so these are not related.

In general, each logician's knowledge relation corresponds to an axis of an n-dimensional cube [24]. For the three-logician case, this is can be shown by the following three graphs, leaving the reflexive connections implicit.

$$\begin{tikzcd}
 & \langle \bot, \top, \top \rangle \arrow[rr, dash] & & \langle \top, \top, \top \rangle \\
\langle \bot, \top, \bot \rangle \arrow[rr, dash] & & \langle \top, \top, \bot \rangle & \\
 & \langle \bot, \bot, \top \rangle \arrow[rr, dash] & & \langle \top, \bot, \top \rangle \\
\langle \bot, \bot, \bot \rangle \arrow[rr, dash] & & \langle \top, \bot, \bot \rangle &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \langle \bot, \top, \top \rangle \arrow[dd, dash] & & \langle \top, \top, \top \rangle \arrow[dd, dash] \\
\langle \bot, \top, \bot \rangle \arrow[dd, dash] & & \langle \top, \top, \bot \rangle \arrow[dd, dash] & \\
 & \langle \bot, \bot, \top \rangle & & \langle \top, \bot, \top \rangle \\
\langle \bot, \bot, \bot \rangle & & \langle \top, \bot, \bot \rangle &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \langle \bot, \top, \top \rangle \arrow[dl, dash] & & \langle \top, \top, \top \rangle \arrow[dl, dash] \\
\langle \bot, \top, \bot \rangle & & \langle \top, \top, \bot \rangle & \\
 & \langle \bot, \bot, \top \rangle \arrow[dl, dash] & & \langle \top, \bot, \top \rangle \arrow[dl, dash] \\
\langle \bot, \bot, \bot \rangle & & \langle \top, \bot, \bot \rangle &
\end{tikzcd}$$

Common knowledge is itself interpreted with its own knowledge relation. An event which is common knowledge must be known universally, so states must be related by the common knowledge relation when they are related by any one of the individual agents' knowledge relations. Set-theoretically, this is the union of all agents' relations, intuitively taking the union of their ignorance about the world. For the above three-logician example, this operation completes the cube.

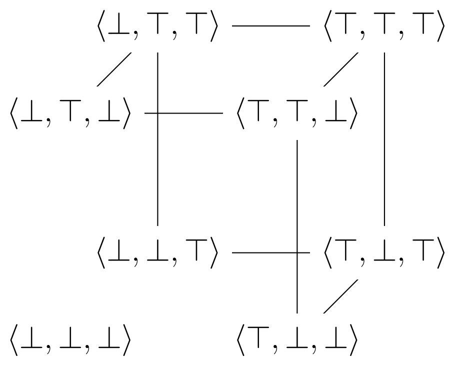
$$\begin{tikzcd}
 & \langle \bot, \top, \top \rangle \arrow[rr, dash] \arrow[dl, dash] \arrow[dd, dash] & & \langle \top, \top, \top \rangle \arrow[dl, dash] \arrow[dd, dash] \\
\langle \bot, \top, \bot \rangle \arrow[rr, dash] \arrow[dd, dash] & & \langle \top, \top, \bot \rangle \arrow[dd, dash] & \\
 & \langle \bot, \bot, \top \rangle \arrow[rr, dash] \arrow[dl, dash] & & \langle \top, \bot, \top \rangle \arrow[dl, dash] \\
\langle \bot, \bot, \bot \rangle \arrow[rr, dash] & & \langle \top, \bot, \bot \rangle &
\end{tikzcd}$$

This is the relational interpretation of the universal knowledge operator, EK. Observe that states with two or more blue-eyed logicians are not directly connected to $\langle \bot, \bot, \bot \rangle$, which corresponds to the fact that everyone already knows there is at least one blue-eyed logician in these states, as is stated in the puzzle text. The relation is reflexive and symmetric, but not transitive, which implies that the corresponding operator lacks the introspective properties corresponding to Axioms 4 and 5 of the S5 modal logic. From only an assumption of $\mathsf{EK}\ \phi$, we cannot in general conclude $\mathsf{EK}\ (\mathsf{EK}\ \phi)$.

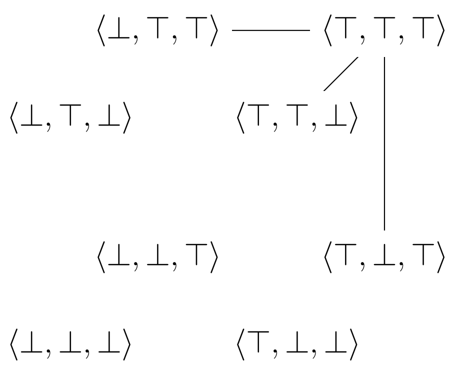
To get a full accounting of common knowledge, which includes the Axiom 4 and 5 properties, we further take the transitive closure. In the example, the relation will be strongly connected, all states in a single equivalence class. It is as if there is a hypothetical, maximally-ignorant agent whose knowledge relation is the common knowledge relation, unable to observe a difference between any pair of states.

However, following the announcement which makes state $\langle \bot, \bot, \bot \rangle$ impossible, the connections from this state in the logicians' individual relations are severed. In states where there is a single blue-eyed logician, that logician is now able to precisely determine the state, while the other two logicians still consider another state possible.

Constructing the updated common knowledge relation by union and transitive closure, state $\langle \bot, \bot, \bot \rangle$ now forms its own equivalence class. This indicates that it has become common knowledge that there is at least one blue-eyed logician in the other states. Even our hypothetical agent can now distinguish this state from the others. In the following graph for this relation, both the reflexive and transitive connections are left implicit.



The first night after the announcement, world states with only one blue-eyed logician become impossible from the perspective of logicians in states with at least two. In these states, it has therefore become common knowledge that there are at least two blue-eyed logicians. The common knowledge relation now has singleton equivalence classes for all states with fewer than two blue-eyed logicians.



This pattern continues after the second night, with the entire world state becoming common knowledge in the three-logician case, each state in its own singleton equivalence class. In the general case with arbitrarily many logicians, n of which have blue eyes, common knowledge of the value of n is established $n-1$ nights after the announcement.

4.2 Coinductive Definition

We now return to the embedding of epistemic logic into type theory defined in Chapter III. To define a common knowledge operator in the embedding, we need to add type of agents. We postulate this type abstractly, making only the assumption that it is non-empty. That is, we assume an element $\langle \mathsf{Agent}, \mathsf{agent} \rangle$ of type $\Sigma_{(\mathsf{Agent}:\mathsf{Type})}\ \mathsf{Agent}$. We therefore have a type, Agent, which has at least one concrete element, $\mathsf{agent} : \mathsf{Agent}$.

The isomorphism between the knowledge operator semantics and the relational semantics was established, so we may equivalently work with either representation. Here we assume equivalence relations as primitive, each agent a equipped with $\propto_a : \mathsf{EquivRel\ State}$.

Definition 4.2.1. The agents' knowledge operators are defined by the isomorphism, with Theorem 3.5.9 validating they have the appropriate properties.

$$\begin{aligned}&\mathsf{K} : \mathsf{Agent} \to \mathsf{Event} \to \mathsf{Event}\\&\mathsf{K}_a = \mathsf{K}_{[\propto_a]}\end{aligned}$$

With a type of agents now equipped with knowledge operators, universal knowledge among them has an obvious definition.

Definition 4.2.2. An event ϕ is universal knowledge when "everyone knows" ϕ. That is, each agent knows ϕ individually.

$$\begin{aligned}&\mathsf{EK} : \mathsf{Event} \to \mathsf{Event}\\&\mathsf{EK}\ \phi\ w = \Pi_{(a:\mathsf{Agent})}\ \mathsf{K}_a\ \phi\ w\end{aligned}$$

The EK operator is not a full knowledge operator. It preserves semantic entailments and satisfies the Axiom T property, but not the two introspective properties corresponding to Axioms 4 and 5. Intuitively, the relational representation of the operator is the union of all agents' individual relations — it is reflexive and symmetric, but not necessarily transitive.

Lemma 4.2.3. EK *preserves semantic entailment,* $\Phi \vDash \psi \to \mathsf{EK} \circ \Phi \vDash \mathsf{EK}\ \psi$.

Proof. Assume an event family $\Phi : T \to \mathsf{Event}$ such that $\Phi \vDash \psi$. Given hypothesis $\Pi_{(t:T)}\ \mathsf{EK}\ (\Phi\ t)\ w$, we need to show $\mathsf{EK}\ \psi\ w$. By Definitions 4.2.1 and 4.2.2, this is the following statement.

$$\Pi_{(a:\mathsf{Agent})}\ \Pi_{(v:\mathsf{State})}\ w \propto_a v \to \psi\ v$$

So we assume $w \propto_a v$ and apply the entailment $\Phi \vDash \psi$ at index v, the goal becoming $\Phi\ t\ v$. From our earlier hypothesis, we know $\mathsf{EK}\ (\Phi\ t)\ w$, so can derive $\mathsf{K}_a\ (\Phi\ t)\ w$ by instantiating universal knowledge specialised to agent a.

Using Axiom 4 as a transportation principle as in Theorem 3.5.11, we can transport this knowledge to state v as these states are bridged by $w \propto_a v$. Since $\mathsf{K}_a\ (\Phi\ t)\ v$, Axiom T validates $\Phi\ t\ v$ — if a knows $\Phi\ t$ at state v, the event must actually occur there. □

Lemma 4.2.4. EK *satisfies Axiom T,* $\mathsf{EK}\ \phi \subseteq \phi$*.*

Proof. Given $\mathsf{EK}\ \phi\ w$, instantiate this at the agent constant to derive $\mathsf{K}_{\mathsf{agent}}\ \phi\ w$. The conclusion $\phi\ w$ then follows from the Axiom T property of agent's knowledge operator. □

With universal knowledge defined, we can give common knowledge a definition as an operator on events. The intuitive idea of common knowledge as the infinite conjunction $\mathsf{EK}\ \phi \wedge \mathsf{EK}\ (\mathsf{EK}\ \phi) \wedge \ldots$ can be realised directly using a coinductive type.

Definition 4.2.5. Common knowledge of an event ϕ implies that it is universal knowledge and, corecursively, common knowledge of the universal knowledge.

$$\begin{aligned}&\mathsf{coinductive\ CK} : \mathsf{Event} \to \mathsf{Event}\\ &\qquad\mathsf{introCK} : \Pi_{(\phi:\mathsf{Event})}\ \mathsf{EK}\ \phi \wedge \mathsf{CK}\ (\mathsf{EK}\ \phi) \subseteq \mathsf{CK}\ \phi\end{aligned}$$

Expanding the type of the constructor using Definitions 3.3.2 and 3.3.3, it can be seen that the canonical way to prove $\mathsf{CK}\ \phi\ w$ is to provide proofs of $\mathsf{EK}\ \phi\ w$ and $\mathsf{CK}\ (\mathsf{EK}\ \phi)\ w$ collected as a pair.

$$\mathsf{introCK} : \Pi_{(\phi:\mathsf{Event})}\ \Pi_{(w:\mathsf{State})}\ \mathsf{EK}\ \phi\ w \times \mathsf{CK}\ (\mathsf{EK}\ \phi)\ w \to \mathsf{CK}\ \phi\ w$$

This coinductive definition resembles that of infinite streams. They each have only one constructor with a single recursive branch, carrying some additional data at each node. However, the type of the data carried by streams is the same at every node, where the common knowledge constructor is a dependent function which changes the type at each node, each step having one more application of EK than the previous.

Since a proof of $\mathsf{CK}\ \phi$ is constructed from a conjunction, it is immediate that either conjunct may be projected from the underlying product type.

Corollary 4.2.6. $\mathsf{CK}\ \phi \subseteq \mathsf{EK}\ \phi$

Corollary 4.2.7. $\mathsf{CK}\ \phi \subseteq \mathsf{CK}\ (\mathsf{EK}\ \phi)$

A similar property to Corollary 4.2.7, swapping the operators in the conclusion, states common knowledge of an event implies that it's universally known to be common knowledge. However, this is not immediate from the definition, requiring a coinductive proof.

Lemma 4.2.8. $\mathsf{CK}\ \phi \subseteq \mathsf{EK}\ (\mathsf{CK}\ \phi)$

Proof. In a coinductive proof of common knowledge, we may recursively use the statement we are proving as long as its use is guarded by an application of constructor $\mathsf{introCK}$ to prevent circularity. We therefore have the coinductive hypothesis $\Pi_{(\phi:\mathsf{Event})}\ \mathsf{CK}\ \phi \subseteq \mathsf{EK}\ (\mathsf{CK}\ \phi)$.

We assume $\mathsf{CK}\ \phi\ w$ and, from Definitions 4.2.1 and 4.2.2, $w \propto_a v$. From the first assumption and Corollary 4.2.7, we obtain $\mathsf{CK}\ (\mathsf{EK}\ \phi)\ w$. The conclusion is the statement $\mathsf{CK}\ \phi\ v$, so apply $\mathsf{introCK}$, thus requiring proofs for $\mathsf{EK}\ \phi\ v$ and $\mathsf{CK}\ (\mathsf{EK}\ \phi)\ v$.

To show $\mathsf{EK}\ \phi\ v$ from $\mathsf{CK}\ (\mathsf{EK}\ \phi)\ w$, use Corollary 4.2.6 to reach $\mathsf{EK}\ (\mathsf{EK}\ \phi)\ w$ and therefore its specialisation, $\mathsf{K}_a\ (\mathsf{EK}\ \phi)\ w$. By the assumption of $w \propto_a v$, we use Axiom 4 to transport the knowledge of $\mathsf{EK}\ \phi$ from state w to v.

To show $\mathsf{CK}\ (\mathsf{EK}\ \phi)\ v$ from $\mathsf{CK}\ (\mathsf{EK}\ \phi)\ w$, guardedness is satisfied, so we may use the coinductive hypothesis at index $\mathsf{EK}\ \phi$ to obtain $\mathsf{EK}\ (\mathsf{CK}\ (\mathsf{EK}\ \phi))\ w$ and its specialisation, $\mathsf{K}_a\ (\mathsf{CK}\ (\mathsf{EK}\ \phi))\ w$. By the same reasoning as above, the known event may then be transported to state v. □

By repeatedly applying Corollary 4.2.7 followed by an application of Corollary 4.2.6, we can show that EK can be iterated on ϕ any number of times from an assumption of $\mathsf{CK}\ \phi$, which matches the intuition of common knowledge as iterated universal knowledge. To demonstrate this formally, we define the event family $\mathsf{recCK}\ \phi$, indexed by the natural numbers, which contains all those events defined by iterating EK — $\mathsf{recCK}\ \phi\ 0 = \phi$, $\mathsf{recCK}\ \phi\ 1 = \mathsf{EK}\ \phi$, $\mathsf{recCK}\ \phi\ 2 = \mathsf{EK}\ (\mathsf{EK}\ \phi)$, and so on.

Definition 4.2.9. The event $\mathsf{recCK}\ \phi\ n$ is defined by recursively iterating the function $\mathsf{EK}\ n$ times over ϕ.

$$\begin{aligned}
&\mathsf{recCK} : \mathsf{Event} \to \mathbb{N} \to \mathsf{Event}\\
&\mathsf{recCK}\ \phi\ \mathsf{zero} &&= \phi\\
&\mathsf{recCK}\ \phi\ (\mathsf{succ}\ n) &&= \mathsf{recCK}\ (\mathsf{EK}\ \phi)\ n
\end{aligned}$$

Theorem 4.2.10. *The event* $\mathsf{CK}\ \phi$ *semantically entails event family* $\mathsf{recCK}\ \phi$.

Proof. Proceed by induction on the indices of event family $\mathsf{recCK}\ \phi$. The base case is $\Pi_{(\phi:\mathsf{Event})}\ \mathsf{CK}\ \phi \subseteq \mathsf{recCK}\ \phi\ 0$, whose conclusion reduces to ϕ. Assume $\mathsf{CK}\ \phi$, applying Corollary 4.2.6 to obtain $\mathsf{EK}\ \phi$, followed by the Axiom T property of EK, Lemma 4.2.4, to conclude ϕ.

We have inductive hypothesis $\Pi_{(\phi:\mathsf{Event})}\ \mathsf{CK}\ \phi \subseteq \mathsf{recCK}\ \phi\ n$ for the inductive case. Assume $\mathsf{CK}\ \phi$ and apply Corollary 4.2.7 to obtain $\mathsf{CK}\ (\mathsf{EK}\ \phi)$. Use the inductive hypothesis at event $\mathsf{EK}\ \phi$ to conclude $\mathsf{recCK}\ (\mathsf{EK}\ \phi)\ n$, which is equal to $\mathsf{recCK}\ \phi\ (\mathsf{succ}\ n)$. □

We can also show the reverse entailment as another example of a coinductive proof for common knowledge. Therefore the equivalence of event $\mathsf{CK}\ \phi$ and event family $\mathsf{recCK}\ \phi$ is established.

Theorem 4.2.11. *The event family* $\mathsf{recCK}\ \phi$ *semantically entails event* $\mathsf{CK}\ \phi$.

Proof. Assume, as the coinductive hypothesis, $\Pi_{(\phi:\mathsf{Event})}\ \mathsf{recCK}\ \phi \vDash \mathsf{CK}\ \phi$. Then assume $\Pi_{(n:\mathbb{N})}\ \mathsf{recCK}\ \phi\ n$ and use $\mathsf{introCK}$ to construct a term of coinductive type $\mathsf{CK}\ \phi$.

We need to provide a pair of proofs, the first of which, $\mathsf{EK}\ \phi$, is established by applying our assumption at index 1. For the second, $\mathsf{CK}\ (\mathsf{EK}\ \phi)$, guardedness is satisfied, so we apply the coinductive hypothesis at event $\mathsf{EK}\ \phi$. We then need to show $\Pi_{(n:\mathbb{N})}\ \mathsf{recCK}\ (\mathsf{EK}\ \phi)\ n$, which is the same as $\Pi_{(n:\mathbb{N})}\ \mathsf{recCK}\ \phi\ (\mathsf{succ}\ n)$, yielded by applying the assumption at index $\mathsf{succ}\ n$ □

Unlike the universal knowledge operator it's based on, the coinductive common knowledge operator is a true knowledge operator, satisfying all required properties. While each property may be proved directly, it is more convenient to leverage the equivalence with the relational definition of common knowledge. The knowledge operator proof will be shown with this equivalence in Section 4.4.

4.3 Relational Definition

The relational semantics of epistemic logic interprets common knowledge among a group of agents as the equivalence relation that is the transitive closure of the union of the individual agents' relations. This notion can be defined in our embedding directly as an inductive relation, denoted by $\propto_{\mathsf{Agent}}$. The knowledge operator semantics then follows from Theorem 3.5.9 provided we show $\propto_{\mathsf{Agent}}$ is an equivalence relation.

Definition 4.3.1. The statement $w \propto_{\mathsf{Agent}} v$ may be derived when there is an agent a such that $w \propto_a v$. Alternatively, it may be derived transitively.

$$\begin{aligned}
&\mathsf{inductive}\ \propto_{\mathsf{Agent}} : \mathsf{State} \to \mathsf{State} \to \mathsf{Type}\\
&\qquad \mathsf{union}_{\propto_{\mathsf{Agent}}} : \Pi_{(a:\mathsf{Agent})}\ \Pi_{(w,v:\mathsf{State})}\ w \propto_a v \to w \propto_{\mathsf{Agent}} v\\
&\qquad \mathsf{trans}_{\propto_{\mathsf{Agent}}} : \Pi_{(w,v,u:\mathsf{State})}\ w \propto_{\mathsf{Agent}} v \to v \propto_{\mathsf{Agent}} u \to w \propto_{\mathsf{Agent}} u
\end{aligned}$$

Definition 4.3.2. The relational common knowledge operator, rCK, is yielded when the $\mathsf{K}_{[\_]}$ transformation is applied to relation $\propto_{\mathsf{Agent}}$.

$$\begin{aligned}
&\mathsf{rCK} : \mathsf{Event} \to \mathsf{Event}\\
&\mathsf{rCK} = \mathsf{K}_{[\propto_{\mathsf{Agent}}]}
\end{aligned}$$

For rCK to be a knowledge operator, we must establish that $\propto_{\mathsf{Agent}}$ is indeed an equivalence relation. Transitivity is trivial, while reflexivity and symmetry remain to be proved.

Lemma 4.3.3. *Reflexivity,* $w \propto_{\mathsf{Agent}} w$.

Proof. Since there is at least one agent, witnessed by agent, whose knowledge relation is reflexive, we have $w \propto_{\mathsf{agent}} w$. Applying the $\mathsf{union}_{\propto_{\mathsf{Agent}}}$ constructor lifts this to $w \propto_{\mathsf{Agent}} w$, as desired. □

Lemma 4.3.4. *Symmetry,* $w \propto_{\mathsf{Agent}} v \to v \propto_{\mathsf{Agent}} w$.

Proof. Proceed by structural induction on the assumption of $w \propto_{\mathsf{Agent}} v$. We have two cases to consider, the base case when the assumption is constructed by $\mathsf{union}_{\propto_{\mathsf{Agent}}}$, and the inductive case when it is constructed by $\mathsf{trans}_{\propto_{\mathsf{Agent}}}$.

In the base case, there must be an agent a such that $w \propto_a v$, and whose knowledge relation is symmetric, so $v \propto_a w$. Applying $\mathsf{union}_{\propto_{\mathsf{Agent}}}$ shows $v \propto_{\mathsf{Agent}} w$.

In the inductive case, we have assumptions $w \propto_{\mathsf{Agent}} u$ and $u \propto_{\mathsf{Agent}} v$, each with an inductive hypothesis, $u \propto_{\mathsf{Agent}} w$ and $v \propto_{\mathsf{Agent}} u$, respectively. Recombining the inductive hypotheses in the opposite order with the $\mathsf{trans}_{\propto_{\mathsf{Agent}}}$ constructor therefore gives $v \propto_{\mathsf{Agent}} w$. □

Theorem 4.3.5. $\propto_{\mathsf{Agent}}$ *is an equivalence relation.*

Proof. Transitivity is by definition of $\mathsf{trans}_{\propto_{\mathsf{Agent}}}$, while reflexivity and symmetry are shown in Lemmas 4.3.3 and 4.3.4. □

Corollary 4.3.6. rCK *is a knowledge operator.*

In addition to the knowledge operator properties, rCK satisfies the additional properties expected of an interpretation for common knowledge. These are useful lemmas, corresponding to properties already shown for CK, which describe how assumptions of rCK interact with EK.

Lemma 4.3.7. $\mathsf{rCK}\ \phi \subseteq \mathsf{EK}\ \phi$

Proof. Assume $\mathsf{rCK}\ \phi\ w$ and $w \propto_a v$, setting out to prove $\phi\ v$. Using constructor $\mathsf{union}_{\propto_{\mathsf{Agent}}}$ with the second assumption, we obtain $w \propto_{\mathsf{Agent}} v$. By Definition 4.3.2, the first assumption is equal to $\Pi_{(v:\mathsf{State})}\ w \propto_{\mathsf{Agent}} v \rightarrow \phi\ v$, so applying this at index v reaches the desired conclusion. □

Lemma 4.3.8. $\mathsf{rCK}\ \phi \subseteq \mathsf{rCK}\ (\mathsf{EK}\ \phi)$

Proof. Assume $\mathsf{rCK}\ \phi\ w$, $w \propto_{\mathsf{Agent}} v$, and $v \propto_a u$. Using constructor $\mathsf{union}_{\propto_{\mathsf{Agent}}}$ with the last assumption, we obtain $v \propto_{\mathsf{Agent}} u$, and then, by $\mathsf{trans}_{\propto_{\mathsf{Agent}}}$ with this and the second assumption, $w \propto_{\mathsf{Agent}} u$. Applying the first assumption, gives the desired result, $\phi\ u$. □

Lemma 4.3.9. $\mathsf{rCK}\ \phi \subseteq \mathsf{EK}\ (\mathsf{rCK}\ \phi)$

Proof. Assume $\mathsf{rCK}\ \phi\ w$, $w \propto_a v$, and $v \propto_{\mathsf{Agent}} u$. As above, use the $\propto_{\mathsf{Agent}}$ constructors with the second and third assumptions, then apply the first assumption to conclude $\phi\ u$. □

4.4 Coinductive and Relational Equivalence

We now come to proving the equivalence of the two operators we have defined for common knowledge, $\mathsf{CK}\ \phi \approx \mathsf{rCK}\ \phi$. There is one final lemma we need which provides a bridge between these two interpretations.

From Corollary 4.3.6, we know that the relational definition is a knowledge operator, in particular satisfying Axiom 4, $\mathsf{rCK}\ \phi \subseteq \mathsf{rCK}\ (\mathsf{rCK}\ \phi)$. Recall that this property, for the K_a operator, implied a principle which allowed the agent's knowledge in one state to be transported to another, provided that these states were bridged by the agent's knowledge relation. Since the relational common knowledge operator is defined in the same way, using the $\mathsf{K}_{[\_]}$ transformation on an equivalence relation, we have a corresponding transportation principle for it.

$$\mathsf{rCK}\ \phi\ w \rightarrow w \propto_{\mathsf{Agent}} v \rightarrow \mathsf{rCK}\ \phi\ v$$

If we can show that coinductive common knowledge can also be transported across this relation, we have all we need to show the equivalence of the two operators.

Lemma 4.4.1. *Coinductive common knowledge can be transported across* $\propto_{\mathsf{Agent}}$*. That is,* $\mathsf{CK}\ \phi\ w \to w \propto_{\mathsf{Agent}} v \to \mathsf{CK}\ \phi\ v$*.*

Proof. Proceed by structural induction on $w \propto_{\mathsf{Agent}} v$. In the base case, $w \propto_{\mathsf{Agent}} v$ is constructed by $\mathsf{union}_{\propto_{\mathsf{Agent}}}$, so we need to show $\mathsf{CK}\ \phi\ w \to w \propto_a v \to \mathsf{CK}\ \phi\ v$. Assume $\mathsf{CK}\ \phi\ w$ and $w \propto_a v$, then we will utilise Lemma 4.2.8 rather than directly proving $\mathsf{CK}\ \phi\ v$ by coinduction. This tells us $\mathsf{EK}\ (\mathsf{CK}\ \phi)\ w$, which can be specialised $\mathsf{K}_a\ (\mathsf{CK}\ \phi)\ w$. Using Axiom 4 to transport this knowledge to state v, followed by Axiom T, we can conclude $\mathsf{CK}\ \phi\ v$.

In the inductive case, $w \propto_{\mathsf{Agent}} v$ is constructed by $\mathsf{trans}_{\propto_{\mathsf{Agent}}}$, so we have assumptions $\mathsf{CK}\ \phi\ w$, $w \propto_{\mathsf{Agent}} u$, and $u \propto_{\mathsf{Agent}} v$. The corresponding inductive hypotheses are $\mathsf{CK}\ \phi\ w \to \mathsf{CK}\ \phi\ u$ and $\mathsf{CK}\ \phi\ u \to \mathsf{CK}\ \phi\ v$. We chain the inductive hypotheses to conclude $\mathsf{CK}\ \phi\ v$. □

Theorem 4.4.2. *Coinductive and relational common knowledge are equivalent,* $\mathsf{CK}\ \phi \approx \mathsf{rCK}\ \phi$*.*

Proof. For the left-to-right direction, assume $\mathsf{CK}\ \phi\ w$ and $w \propto_{\mathsf{Agent}} v$, towards the conclusion $\phi\ v$. By Lemma 4.4.1, we derive $\mathsf{CK}\ \phi\ v$. Since we have not yet proved CK is a knowledge operator, we cannot apply Axiom T directly. Instead, we apply Corollary 4.2.6, to derive $\mathsf{EK}\ \phi\ v$, which has its own Axiom T principle, shown in Lemma 4.2.4.

For the right-to-left direction, assume $\mathsf{rCK}\ \phi\ w$ with the goal of showing $\mathsf{CK}\ \phi\ w$. This direction is by coinduction, so apply the constructor $\mathsf{introCK}$.

For the first field of the coinductive type, since we know $\mathsf{rCK}\ \phi\ w$, Lemma 4.3.7 tells us $\mathsf{EK}\ \phi\ w$. For the second field, use Lemma 4.3.8 to derive $\mathsf{rCK}\ (\mathsf{EK}\ \phi)\ w$. We can then use the coinductive hypothesis indexed with event $\mathsf{EK}\ \phi$ to reach $\mathsf{CK}\ (\mathsf{EK}\ \phi)\ w$, as desired. □

We will finally show that CK is itself a knowledge operator. This follows from Theorem 4.4.2 and the status of rCK as a knowledge operator, Corollary 4.3.6.

Unfolding $\mathsf{CK}\ \phi \approx \mathsf{rCK}\ \phi$ according to Definitions 3.3.2 and 3.3.3, we can see that it is a statement of propositional equivalence, $\Pi_{(w:\mathsf{State})}\ \mathsf{CK}\ \phi\ w \leftrightarrow \mathsf{rCK}\ \phi\ w$. If this were the stronger statement of propositional equality, we could appeal to Theorem 2.5.6 to substitute CK by rCK and the proof would be immediate, however propositional equivalence will only allow substitution between CK and rCK when one of these is the top-level connective of a hypothesis or the conclusion to be reached.

For example, to prove the Axiom 4 property, $\mathsf{CK}\ \phi \subseteq \mathsf{CK}\ (\mathsf{CK}\ \phi)$, after assuming the premise, we may substitute the conclusion $\mathsf{CK}\ (\mathsf{CK}\ \phi)\ w$ by $\mathsf{rCK}\ (\mathsf{CK}\ \phi)\ w$ using Theorem 4.4.2, but it will not allow direct substitution of the inner instance of CK. To make further progress with the proof, we must make use of the definition and properties of rCK. We proceed in this fashion for each of the knowledge operator properties, making liberal use of Theorem 4.4.2 to be able to utilise the properties of rCK.

Lemma 4.4.3. CK *preserves semantic entailment,* $\Phi \vDash \psi \to \mathsf{CK} \circ \Phi \vDash \mathsf{CK}\ \psi$*.*

Proof. For an arbitrary event family $\Phi : T \to \mathsf{Event}$, assume hypotheses $\Phi \vDash \psi$ and $\Pi_{(t:T)}\ \mathsf{CK}\ (\Phi\ t)\ w$. We are to prove $\mathsf{CK}\ \psi\ w$, but due to Theorem 4.4.2 we can equivalently prove $\mathsf{rCK}\ \psi\ w$. We may therefore assume $w \propto_{\mathsf{Agent}} v$ with the goal becoming $\psi\ v$.

Applying the first hypothesis, it is sufficient to prove $\Phi\ t\ v$ for some index $t : T$. We know $\mathsf{CK}\ (\Phi\ t)\ w$, which can be transported to state v using Lemma 4.4.1. Then, using the equivalence again, in the other direction, we have $\mathsf{rCK}\ (\Phi\ t)\ v$, and we may use Axiom T to reach the conclusion. □

Lemma 4.4.4. CK *satisfies Axiom T,* $\mathsf{CK}\ \phi \subseteq \phi$*.*

Proof. Assume $\mathsf{CK}\ \phi\ w$ and use Theorem 4.4.2 to derive $\mathsf{rCK}\ \phi\ w$. The conclusion $\phi\ w$ then follows from the Axiom T property of rCK. □

Lemma 4.4.5. CK *satisfies Axiom 4,* $\mathsf{CK}\ \phi \subseteq \mathsf{CK}\ (\mathsf{CK}\ \phi)$*.*

Proof. Assume $\mathsf{CK}\ \phi\ w$ and, using Theorem 4.4.2 with the conclusion as needed, $w \propto_{\mathsf{Agent}} v$ and $v \propto_{\mathsf{Agent}} u$. To conclude $\phi\ u$, we chain Lemma 4.4.1 twice with our assumptions to reach $\mathsf{CK}\ \phi\ u$, and use the Axiom T property from Lemma 4.4.4. □

Lemma 4.4.6. CK *satisfies Axiom 5,* $\neg\mathsf{CK}\ \phi \subseteq \mathsf{CK}\ (\neg\mathsf{CK}\ \phi)$*.*

Proof. Assume $(\neg\mathsf{CK}\ \phi)\ w$ and, with Theorem 4.4.2, $w \propto_{\mathsf{Agent}} v$. To prove the negation $(\neg\mathsf{CK}\ \phi)\ v$, which means $\mathsf{CK}\ \phi\ v \to \mathbb{0}$, we may assume $\mathsf{CK}\ \phi\ v$ to reach a contradiction. By symmetry, we have $v \propto_{\mathsf{Agent}} w$, then Lemma 4.4.1 allows us to derive $\mathsf{CK}\ \phi\ w$, which contradicts the first assumption. □

Theorem 4.4.7. CK *is a knowledge operator.*

Proof. The full theorem is formed by combining Lemmas 4.4.3 to 4.4.6. □

Chapter V

Coalition Logic

In this chapter, we introduce coalition logic and its semantics based on game forms and playable effectivity functions. As in previous chapters, this will at first be in isolation from type theory, using traditional set-theoretic notions to define the concepts of the logic. We will go on to describe a formulation of these concepts appropriate for use in type theory, and sketch an equivalence proof for game forms and playable effectivity functions.

The content of this chapter is based on work in progress, without a corresponding Agda formalisation in Appendix A. It would take significant future effort to develop the following outline so it can be fully formalised in a proof assistant.

5.1 Background

Coalition logic is a modal logic introduced by Marc Pauly [49]. Like epistemic logic, it is an agent-based logic, the fundamental concept being the *coalition*, a subset of agents. Rather than considering each agents in isolation, coalition logic explores what agents can achieve when they act in unison.

In the syntax of the logic, each coalition C has a modal operator, $[C]\,\phi$, with the reading that the agents of C, if they so are inclined to work together, could force ϕ to hold after their collective action, regardless of any other agents' actions. The base logic has no concept of knowledge, essentially modelling perfect-information strategic games, but it can be extended with modal operators for knowledge to model epistemic scenarios [1, 56].

The semantics of coalition logic is given by *game forms*. These are descriptions of a step in some strategic game where players may act simultaneously. A game form consists of a finite, non-empty set of agents, N, a set of actions, $\mathcal{A}_i$ for each $i \in N$, a set of outcomes, O, and a function which maps a choice of action for each agent to an outcome, $o : (\Pi_{(i \in N)}\ \mathcal{A}_i) \to O$.

We can derive an *effectivity function* for each game form G. An effectivity function $E_G : \mathcal{P}(N) \to \mathcal{P}(\mathcal{P}(O))$ describes, for each coalition of agents, those subsets of outcomes for which they are effective. That is, if $X \in E_G(C)$, coalition C has a joint strategy which guarantees the actual outcome must come from X, regardless of the strategies chosen by the other agents not in C. To define the function precisely, some additional notation is introduced.

Let C range over coalitions, subsets of agents, $C \in \mathcal{P}(N)$, and X range over subsets of outcomes, $X \in \mathcal{P}(O)$. By $\overline{C}$ and $\overline{X}$, we refer to the complements of C and X relative to their full sets, $\overline{C} = N \setminus C$, $\overline{X} = O \setminus X$. A strategy profile, σ_C, is a tuple containing the choice of action for each agent in a coalition, $\sigma_C : \Pi_{(i \in C)}\ \mathcal{A}_i$. A global strategy profile, $\sigma : \Pi_{(i \in N)}\ \mathcal{A}_i$, can be constructed by combining strategy profiles of complementary coalitions, $\sigma = \sigma_C \oplus \sigma_{\overline{C}}$. The effectivity function for game form G may then be defined.

$$E_G(C) = \{X \mid \exists_{\sigma_C} \forall_{\sigma_{\overline{C}}}\ o(\sigma_C \oplus \sigma_{\overline{C}}) \in X\}$$

Is is convenient to work abstractly with effectivity functions rather than game form definitions. Pauly shows that there is a set of properties for effectivity functions, *playability*, which characterises when an arbitrary effectivity function E is the effectivity function E_G for some game form G.

- Safety, $\varnothing \notin E(C)$
- Liveness, $O \in E(C)$
- N-Maximality, $\overline{X} \notin E(\varnothing) \Rightarrow X \in E(N)$
- Outcome Monotonicity, $X_1 \subseteq X_2 \Rightarrow X_1 \in E(C) \Rightarrow X_2 \in E(C)$
- Superadditivity, $C_1 \cap C_2 = \varnothing \Rightarrow X_1 \in E(C_1) \Rightarrow X_2 \in E(C_2) \Rightarrow X_1 \cap X_2 \in E(C_1 \cup C_2)$

Two further properties can be shown to follow from those listed above.

- Regularity, $X \in E(C) \Rightarrow \overline{X} \notin E(\overline{C})$
- Coalition Monotonicity, $C_1 \subseteq C_2 \Rightarrow X \in E(C_1) \Rightarrow X \in E(C_2)$

Given a game form G, the playability of E_G is a routine question of checking each of the properties. The inverse direction, starting from a playable effectivity function E, requires construction of a game form G such that E and E_G are equivalent. For the rest of this chapter, we work towards giving a type-theoretic formulation for this direction of the correspondence.

5.2 Game Form Construction

Our construction of the game form is based on the idea of Pauly's original construction [49]. However, this is heavily set-theoretic and poses significant technical challenges to formulation in type theory. This section will first give an outline of the construction, and we will discuss some of the technical details in later sections.

Given a playable effectivity function $E : \mathcal{P}(N) \to \mathcal{P}(\mathcal{P}(O))$. The constructed game form will use the same agent and outcome sets, so we just need to define the actions available to each agent and the outcome function.

An action for agent i consists of a choice of coalition C such that $i \in C$, a set of outcome states $X \in E(C)$, a specific outcome $x \in X$, and a natural number. Formally, $\mathcal{A}_i$ is defined as the following set.

$$\mathcal{A}_i = \{\langle C,\, X,\, x,\, t\rangle \mid C \in \mathcal{P}(N),\, i \in C,\, X \in E(C),\, x \in X,\, t \in \mathbb{N}\}$$

Intuitively, if an agent chooses action $\langle C,\, X,\, x,\, t\rangle$, it means they want to be part of coalition C, cooperating to achieve an outcome in X, and they have a personal preference for specific outcome x if the choice were up to them. The natural number t will be used in the outcome function to determine which agent gets to make the final decision.

Given a strategy profile $\sigma : \Pi_{(i \in N)}\, \mathcal{A}_i$, we have a choice of action for each agent, $\langle C_i,\, X_i, x_i,\, t_i\rangle$. We call C a *σ-cooperative* coalition when, for every agent $i \in C$, $C = C_i$, and for every pair of agents $i, j \in C$, $X_i = X_j$. That is, a σ-cooperative coalition C agrees to work together towards a common set of outcomes X_C.

Let $[C_1,\, \ldots,\, C_m]$ be the list of all the non-empty σ-cooperative coalitions. There must be finitely many since N is itself finite. Let C_0 be the coalition of agents not in a σ-cooperative coalition, partitioning the full set of agents $[C_0,\, C_1,\, \ldots,\, C_m]$. Since C_0 has no agreed set of outcomes to aim for, define $X_{C_0} = O$, and define X_σ as the intersection of all chosen outcome sets X_{C_k}.

$$X_\sigma = \bigcap_{k=0}^{m} X_{C_k} = \bigcap_{k=1}^{m} X_{C_k}$$

It cannot be the case that $X_\sigma = \varnothing$. Since $[C_0,\, C_1,\, \ldots,\, C_m]$ is a partition, and all $X_{C_k} \in E(C_k)$ by definition of $\mathcal{A}$, the superadditivity property of E implies that $X_\sigma \in E(N)$, and safety tells us $\varnothing \notin E(N)$.

The outcome function will be defined to choose an outcome $x \in X_\sigma$. Recall that the number of agents is finite, and let n be their number, $n = |N|$. Labelling the set of agents by natural numbers $\{0,\, \ldots,\, n-1\}$, we sum modulo n the choice of t_i for each agent, $d = t_0 + \cdots + t_{n-1} \mod n$, to define the agent who decides the outcome.

It must be the case that $x_d \in X_{C_d}$, but it is not guaranteed that $x_d \in X_\sigma$. If $x_d \notin X_\sigma$, we revert to an arbitrary choice function $H : \Pi_{(X \in E(N))}\, X$, which exists constructively due to the safety property, $\varnothing \notin E(N)$. Since $X_\sigma \in E(N)$ from the above superadditivity argument, we can define the outcome function.

$$o(\sigma) = \begin{cases} x_d & \text{if } x_d \in X_\sigma \\ H(X_\sigma) & \text{otherwise} \end{cases}$$

Thus we have constructed the game form G. To show the correspondence between game forms and playable effectivity functions, we need to demonstrate that E is equivalent to the derived effectivity function E_G.

Lemma 5.2.1. *If $X \in E(C)$, then $X \in E_G(C)$.*

Proof. Assume $X \in E(C)$, then we have to show $\exists_{\sigma_C} \forall_{\sigma_{\overline{C}}}\ o(\sigma_C \oplus \sigma_{\overline{C}}) \in X$. We define σ_C by giving an action $\langle C_i, X_i, x_i, t_i \rangle$ for every $i \in C$. Set $C_i = C$ and $X_i = X$, choosing x_i and t_i arbitrarily — our assumption that $X \in E(C)$ together with the safety property, $\varnothing \notin E(C)$, ensure an $x \in X$ can be chosen.

Given any counter strategy $\sigma_{\overline{C}}$, we have strategy profile $\sigma = \sigma_C \oplus \sigma_{\overline{C}}$, where C is σ-cooperative by definition. C is therefore one of the partition classes whose chosen outcome set X is used to define X_σ by intersection, so $X_\sigma \subseteq X$. Since $o(\sigma) \in X_\sigma$, this subset relation implies $o(\sigma) \in X$, as desired. □

Lemma 5.2.2. *If $X \in E_G(C)$, then $X \in E(C)$.*

Proof. We assume that coalition membership is decidable, so we distinguish two cases, when $C = N$ and when there is at least one $i \in N$ where $i \in \overline{C}$.

In the first case, we assume $X \in E_G(N)$. By regularity of E_G, this implies $\overline{X} \notin E_G(\varnothing)$. Using the contrapositive of Lemma 5.2.1, we can derive $X \notin E(\varnothing)$. Then, N-maximality of E reaches the desired conclusion $X \in E(N)$.

Now consider the case with at least one $i \in \overline{C}$. Assume $X \in E_G(C)$, whose definition means that C has a strategy profile σ_C such that $\forall_{\sigma_{\overline{C}}}\ o(\sigma_C \oplus \sigma_{\overline{C}}) \in X$. Instantiating this with a strategy profile $\sigma_{\overline{C}}$ where $C_i = N$ and $X_i = O$ for every $i \in \overline{C}$ makes X_σ depend only on σ_C.

$$X_\sigma = \bigcap \{X_{C'} \mid C' \subseteq C,\ C' \text{ is } \sigma\text{-cooperative}\}$$

Since, by definition of $\mathcal{A}$, all $X_{C'} \in E(C')$, superadditivity and coalition monotonicity imply $X_\sigma \in E(C)$. If we can give appropriate values for x_i and t_i for each $i \in \overline{C}$ so that $X_\sigma \subseteq X$, this will show the desired $X \in E(C)$ by outcome monotonicity.

For each $x \in X_\sigma$, since we have at least one $i \in \overline{C}$, we can set $x_i = x$, and tweak the values of t_i so that $d = t_0 + \cdots + t_{n-1} \mod n$ ends up as one of i. This ensures $o(\sigma_C \oplus \sigma_{\overline{C}}) = x$, and so $x \in X$ by our earlier assumption, therefore $X_\sigma \subseteq X$. □

Theorem 5.2.3. *E and E_G are equivalent, $E(C) \subseteq E_G(C)$ and $E(C) \supseteq E_G(C)$.*

Proof. The left-to-right direction is Lemma 5.2.1, while the right-to-left direction is Lemma 5.2.2. □

5.3 Decidable Subsets

Back in the setting of type theory, we seek to represent the concepts of coalition logic. We begin by assuming appropriate types for agents and outcomes.

Unlike in Chapter IV, we impose the restriction that the agent type is finite, as well as non-empty. We fix a non-zero natural number n as the number of agents, $\Sigma_{(n:\mathbb{N})}\ n \not\equiv \mathsf{zero}$, and define $\mathsf{Agent} = \mathsf{Fin}\ n : \mathsf{Type}$. Our agents therefore correspond to the natural numbers $\{0, \ldots, n-1\}$, and since n must be a successor, we can always construct the agent $\mathsf{zero}_{\mathsf{Fin}} : \mathsf{Agent}$. We assume $\mathsf{Outcome} : \mathsf{Type}$ without any restrictions.

We must develop a library to work with subsets of these types. A type family $T \to \mathsf{Type}$, representing predicates over type T, can be seen as a subset of the type's elements — a term $t : T$ is part of the subset if and only if type $T\ t$ is inhabited. However, in the constructive setting of type theory, there is no guarantee that predicates are decidable in general, which will make working with this representation of subsets difficult to impossible. We instead encode the type of *decidable subsets* by functions into the Booleans, $T \to \mathbb{2}$, which correspond to decidable predicates whose truth values always reduce to true or false.

Definition 5.3.1. A decidable subset of type T is a function mapping from T into the Booleans.

$$\begin{aligned} &\mathcal{P} : \mathsf{Type} \to \mathsf{Type} \\ &\mathcal{P}\ T = T \to \mathbb{2} \end{aligned}$$

Given a decidable subset $S : \mathcal{P}\ T$ and a term $t : T$, we obtain the truth value of subset membership by function application, $S\ t : \mathbb{2}$. From here, the notation of set theory can be defined using Boolean algebra, with the true and false constants, and the basic operators $\mathsf{not}_{\mathbb{2}}$, $\mathsf{or}_{\mathbb{2}}$, and $\mathsf{and}_{\mathbb{2}}$.

Definition 5.3.2. Set-theoretic notation defined for decidable subsets.

$$\begin{aligned} &\mathsf{full} : \Pi_{(T:\mathsf{Type})}\ \mathcal{P}\ T \qquad && \varnothing : \Pi_{(T:\mathsf{Type})}\ \mathcal{P}\ T \\ &\mathsf{full} = \lambda t.\ \mathsf{true} && \varnothing = \lambda t.\ \mathsf{false} \end{aligned}$$

$$\overline{} : \Pi_{(T:\mathsf{Type})}\ \mathcal{P}\ T \to \mathcal{P}\ T \qquad \setminus : \Pi_{(T:\mathsf{Type})}\ \mathcal{P}\ T \to \mathcal{P}\ T \to \mathcal{P}\ T$$
$$\overline{S} = \lambda t.\ \mathsf{not}_{\mathbb{2}}\ (S\ t) \qquad A \setminus B = \lambda t.\ \mathsf{and}_{\mathbb{2}}\ (A\ t)\ (\mathsf{not}_{\mathbb{2}}\ (B\ t))$$

$$\cup : \Pi_{(T:\mathsf{Type})}\ \mathcal{P}\ T \to \mathcal{P}\ T \to \mathcal{P}\ T \qquad \cap : \Pi_{(T:\mathsf{Type})}\ \mathcal{P}\ T \to \mathcal{P}\ T \to \mathcal{P}\ T$$
$$A \cup B = \lambda t.\ \mathsf{or}_{\mathbb{2}}\ (A\ t)\ (B\ t) \qquad A \cap B = \lambda t.\ \mathsf{and}_{\mathbb{2}}\ (A\ t)\ (B\ t)$$

$$\in : \Pi_{(T:\mathsf{Type})}\ T \to \mathcal{P}\ T \to \mathsf{Type} \qquad \subseteq : \Pi_{(T:\mathsf{Type})}\ \mathcal{P}\ T \to \mathcal{P}\ T \to \mathsf{Type}$$
$$t \in S = S\ t \equiv \mathsf{true} \qquad A \subseteq B = \Pi_{(t:T)}\ t \in A \to t \in B$$

Most of these definitions are standard, needing little explanation. To state the set membership $t \in S$ as a proposition in type theory, that is as a type, we assert that the truth value resulting from $S\ t$ is propositionally equal to true. The statement $t \notin S$ can be defined simply as $t \in S \to \mathbb{0}$.

Theorem 5.3.3. *Set membership is decidable, $t \in S + t \notin S$.*

Proof. The full statement is $S\ t \equiv \mathsf{true} + (S\ t \equiv \mathsf{true} \to \mathbb{0})$. By case analysis on the Boolean value $S\ t$, using the eliminators elim_+ and $\mathsf{elim}_{\mathbb{1}}$, we have to demonstrate the statement when $S\ t$ is true and when it is false.

In the true case, we choose inl, proving the left equality, $\mathsf{refl}_\equiv$ true. In the false case, we choose inr, showing that false $\equiv$ true leads to absurdity, which is implied by Theorem 2.5.9. □

By pattern matching on the result of Theorem 5.3.3, it is possible to derive a function which decides the statement $t \in S$, giving the result as a Boolean value, inl $l \mapsto$ true, inr $r \mapsto$ false. This result of this function will coincide with the value of $S\ t$.

We do not define equality of decidable subsets with a dedicated relation. It could be stated $(A \subseteq B) \times (B \subseteq A)$, as is usual in set theory. However, as decidable subsets are functions, it is more convenient to postulate the principle of function extensionality, $\mathsf{funExt} : (\Pi_{(t:T)}\ A\ t \equiv B\ t) \to A \equiv B$. We consider two decidable subsets propositionally equal when they are pointwise equal, giving the same truth values for set membership at all indices.

For example, to show that $\overline{\mathsf{full}} \equiv \varnothing$, by appealing to function extensionality we can instead show $\Pi_{(t:T)}\ \overline{\mathsf{full}}\ t \equiv \varnothing\ t$. This reduces the question of subset equality into a Boolean equality, $\mathsf{not}_{\mathbb{2}}$ true $\equiv$ false, which can be shown with the usual proof methods of type theory, funExt ($\lambda t.\ \mathsf{refl}_\equiv$ false).

Lemma 5.3.4. *Complementary subsets are disjoint, $S \cap \overline{S} \equiv \varnothing$.*

Proof. $\mathsf{funExt}\ (\lambda t.\ \mathsf{elim}_+\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{refl}_\equiv\ \mathsf{false}))\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{refl}_\equiv\ \mathsf{false}))\ (S\ t))$

Using function extensionality, this can be proved as an instance of the Boolean theorem $\Pi_{(b:\mathbb{2})}\ \mathsf{and}_{\mathbb{2}}\ b\ (\mathsf{not}_{\mathbb{2}}\ b) \equiv \mathsf{false}$, where b is instantiated with the set membership truth value $S\ t$. The proof of the Boolean theorem is given by two cases, when b is true and when b is false, both of which reduce to proving false $\equiv$ false. □

Lemma 5.3.5. *Dual to the above,* $S \cup \overline{S} \equiv \mathsf{full}$.

Proof. $\mathsf{funExt}\ (\lambda t.\ \mathsf{elim}_{+}\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{refl}_{\equiv}\ \mathsf{true}))\ (\mathsf{elim}_{\mathbb{1}}\ (\mathsf{refl}_{\equiv}\ \mathsf{true}))\ (S\ t))$

The only difference from the proof of Lemma 5.3.4 is that the two cases reduce to $\mathsf{true} \equiv \mathsf{true}$ rather than $\mathsf{false} \equiv \mathsf{false}$. □

Using propositional equality for subsets gives us access to the full suite of tools for equality proofs, including the powerful congruence substitution principle. We will need many more properties of set equality, which can be proved first as Boolean equalities and then elevated to set equalities by function extensionality.

Given this representation of subsets, we are able to state the type of effectivity functions for our type of agents and outcomes, $\mathcal{P}\ \mathsf{Agent} \to \mathcal{P}\ (\mathcal{P}\ \mathsf{Outcome})$. We can further define the type of playable effectivity functions.

Definition 5.3.6. A playable effectivity function is an effectivity function satisfying the playability properties.

$$
\begin{aligned}
&\mathsf{Playable} : \mathsf{Type}\\
&\mathsf{Playable} = \Sigma_{(E:\mathcal{P}\,\mathsf{Agent}\to\mathcal{P}\,(\mathcal{P}\ \mathsf{Outcome}))}\\
&\qquad\qquad \Pi_{(C:\mathcal{P}\,\mathsf{Agent})}\ \varnothing \notin E\ C\\
&\qquad\quad \times\Pi_{(C:\mathcal{P}\,\mathsf{Agent})}\ \mathsf{full} \in E\ C\\
&\qquad\quad \times\Pi_{(X:\mathcal{P}\,\mathsf{Outcome})}\ \overline{X} \notin E\ \varnothing \to X \in E\ \mathsf{full}\\
&\qquad\quad \times\Pi_{(C:\mathcal{P}\,\mathsf{Agent})}\ \Pi_{(X_1,X_2:\mathcal{P}\,\mathsf{Outcome})}\ X_1 \subseteq X_2 \to X_1 \in E\ C \to X_2 \in E\ C\\
&\qquad\quad \times\Pi_{(C_1,C_2:\mathcal{P}\,\mathsf{Agent})}\ \Pi_{(X_1,X_2:\mathcal{P}\,\mathsf{Outcome})}\ C_1 \cap C_2 \equiv \varnothing \to\\
&\qquad\qquad X_1 \in E\ C_1 \to X_2 \in E\ C_2 \to X_1 \cap X_2 \in E\ (C_1 \cup C_2)
\end{aligned}
$$

The additional properties of regularity and coalition monotonicity can be shown to follow from playability.

Theorem 5.3.7. *Playable effectivity functions are regular,* $X \in E\ C \to \overline{X} \notin E\ \overline{C}$.

Proof. Assume $X \in E\ C$ and, towards a contradiction, $\overline{X} \in E\ \overline{C}$. Since C and $\overline{C}$ can be shown to be disjoint, superadditivity can be used to conclude $X \cap \overline{X} \in E\ (C \cup \overline{C})$. By using congruence to substitute with proofs of the equalities $X \cap \overline{X} \equiv \varnothing$ and $C \cup \overline{C} \equiv \mathsf{full}$, we have $\varnothing \in E\ \mathsf{full}$, contradicting the safety property of playability. □

Theorem 5.3.8. *Playable effectivity functions are coalition monotonic,* $C_1 \subseteq C_2 \to E\ C_1 \subseteq E\ C_2$.

Proof. Assume $C_1 \subseteq C_2$ and $X \in E\ C_1$, with the goal of proving $X \in E\ C_2$. Define $C_3 = C_2 \setminus C_1$, which gives equalities $C_1 \cup C_3 \equiv C_2$ and $C_1 \cap C_3 \equiv \varnothing$.

By the liveness property of playability, $\mathsf{full} \in E\ C_3$, so superadditivity implies $X \cap \mathsf{full} \in E\ (C_1 \cup C_3)$. Substituting, using congruence with proofs of $X \cap \mathsf{full} \equiv X$ and $C_1 \cup C_3 \equiv C_2$, gives the desired conclusion $X \in E\ C_2$. □

These proofs show just some of the standard set-theoretic properties that are needed. As described earlier, these follow from function extensionality and basic Boolean properties. Most of these properties will not be proved explicitly, but we will give one final example which was used in Theorem 5.3.8.

Lemma 5.3.9. $A \subseteq B \to A \cup (B \setminus A) \equiv B$

Proof. Assume $A \subseteq B$. By function extensionality, we can reduce the conclusion to the Boolean statement $\mathsf{or}_2\ (A\ t)\ (\mathsf{and}_2\ (B\ t)\ (\mathsf{not}_2\ (A\ t))) \equiv B\ t$. By case analysis, this can be checked to hold except when $A\ t \equiv \mathsf{true}$ and $B\ t \equiv \mathsf{false}$.

However, instantiating the assumption of $A \subseteq B$ at element t, it is the Boolean statement $A\ t \equiv \mathsf{true} \to B\ t \equiv \mathsf{true}$. Therefore, by congruence, we can construct $\mathsf{true} \equiv \mathsf{false}$ in the problematic case, ruling it absurd by Theorem 2.5.9, completing the proof using $\mathsf{elim}_{\mathbb{0}}$. □

5.4 Partial Functions

We can define a type for game forms using our existing Agent and $\mathsf{Outcome}$ types.

Definition 5.4.1. A game form is a dependent pair $\langle \mathcal{A}, o \rangle$ where $\mathcal{A}$ is a type family indexed by agents, and o is a function assigning an outcome to a choice of action for each agent.

$$\begin{aligned}&\mathsf{GameForm} : \mathsf{Type}\\&\mathsf{GameForm} = \Sigma_{(\mathcal{A}:\mathsf{Agent}\to\mathsf{Type})}\ (\Pi_{(i:\mathsf{Agent})}\ \mathcal{A}_i) \to \mathsf{Outcome}\end{aligned}$$

The outcome function o is total, only assigning outcomes to strategy profiles for the full set of agents, $\sigma : \Pi_{(i:\mathsf{Agent})}\ \mathcal{A}_i$. However, the definition of effectivity function E_G relies on having strategy profiles for coalitions of agents, σ_C, $\sigma_{\overline{C}}$. We use partial functions to bridge this gap.

Definition 5.4.2. A partial function mapping from $A : \mathsf{Type}$ to an element of type family $B : A \to \mathsf{Type}$ is a dependent function $\Pi_{(a:A)}\ B\ a + \mathbb{1}$.

$$\begin{aligned}&\rightharpoonup\ : \Pi_{(A:\mathsf{Type})}\ (A \to \mathsf{Type}) \to \mathsf{Type}\\&A \rightharpoonup B = \Pi_{(a:A)}\ B\ a + \mathbb{1}\end{aligned}$$

Of course, strictly speaking, $A \rightharpoonup B$ is not a partial function, but a total function into a codomain with exactly one additional element. The intuition is that if a partial function maps $a \mapsto b$, this is represented by mapping $a \mapsto \mathsf{inl}\ b$. When the partial function does not provide a mapping from a, we represent this by mapping to the dummy element, $a \mapsto \mathsf{inr}\ \star$.

It can also be the case by this definition that a so-called partial function is actually total, mapping to $a \mapsto \mathsf{inl}\ b$ for all $a : A$. We can characterise elements of A for which there are mappings, the domain of the partial function, using our decidable subset type.

Definition 5.4.3. The domain of a partial function $A \rightharpoonup B$ is the subset of A for which the function has mappings.

$$\begin{aligned}&\mathsf{Domain} : \Pi_{(A:\mathsf{Type})}\ \Pi_{(B:A\to\mathsf{Type})}\ (A \rightharpoonup B) \to \mathcal{P}\ A\\ &\mathsf{Domain}\ f = \lambda a.\ \mathsf{elim}_{+}\ (\lambda b.\ \mathsf{true})\ (\mathsf{elim}_{\mathbb{1}}\ \mathsf{false})\ (f\ a)\end{aligned}$$

For example, the partial function with no mappings has an empty domain, $\mathsf{Domain}\ (\lambda(a : A).\ \mathsf{inr}\ \star) \equiv \varnothing$. Given a total function $f : \Pi_{(a:A)}\ B\ a$, we can construct a corresponding partial function, $\mathsf{Domain}\ (\lambda(a : A).\ \mathsf{inl}\ (f\ a)) \equiv \mathsf{full}$.

We can join the domains of two partial functions $f, g : A \rightharpoonup B$ produce another partial function $f \oplus g : A \rightharpoonup B$. The resulting function only maps $a \mapsto \mathsf{inr}\ \star$ when neither joined function has a mapping from a.

$$(f \oplus g)\ a = \begin{cases}\mathsf{inl}\ b & \text{if } f\ a \equiv \mathsf{inl}\ b\\ \mathsf{inl}\ b & \text{if } g\ a \equiv \mathsf{inl}\ b\\ \mathsf{inr}\ \star & \text{otherwise}\end{cases}$$

While $\oplus$ is always associative, it is only commutative under the assumption that the domains of f and g are disjoint. This is because we default to the mapping provided by f in the case where both f and g provide mappings. The join produces a function whose domain is the union of the two original functions, $\mathsf{Domain}\ (f \oplus g) \equiv \mathsf{Domain}\ f \cup \mathsf{Domain}\ g$.

Given a game form $G = \langle \mathcal{A}, o \rangle$, a strategy profile for a coalition C is a partial function $\sigma_C : \mathsf{Agent} \rightharpoonup \mathcal{A}$, mapping to an action in $\mathcal{A}_i$ if and only if $i \in C$. That is, $\mathsf{Domain}\ \sigma_C \equiv C$. Since Lemma 5.3.5 tells us that $C \cup \overline{C} \equiv \mathsf{full}$, given strategy profiles σ_C and $\sigma_{\overline{C}}$ with these domains, we can show $\mathsf{Domain}\ (\sigma_C \oplus \sigma_{\overline{C}}) \equiv \mathsf{full}$. From this, we are able to construct a total strategy profile $\sigma : \Pi_{(i:\mathsf{Agent})}\ \mathcal{A}_i$.

Lemma 5.4.4. *Given a partial function $f : A \rightharpoonup B$ and an $a : A$ such that $a \in \mathsf{Domain}\ f$, then we can construct an element of type $B\ a$.*

Proof. Assume that $a \in \mathsf{Domain}\ f$, which means $\mathsf{Domain}\ f\ a \equiv \mathsf{true}$. Apply $f\ a$, which gives an element of type $B\ a + \mathbb{1}$. Pattern matching on the result, this must be of the form $\mathsf{inl}\ b$ where $b : B\ a$, or $\mathsf{inr}\ \star$.

In the first case, we can simply return b. In the second case, we cannot construct an element of type $B\ a$, but $\mathsf{Domain}\ f\ a$ must have been false, refining our assumption to $\mathsf{false} \equiv \mathsf{true}$. By Theorem 2.5.9 and $\mathsf{elim}_{\mathbb{0}}$, this case can be dismissed as absurd. □

Theorem 5.4.5. *When the domain of a partial function is the full set, a total function can be constructed,* $\Pi_{(f:A \rightharpoonup B)}\ \mathsf{Domain}\ f \equiv \mathsf{full} \to \Pi_{(a:A)}\ B\ a$.

Proof. Assume that $\mathsf{Domain}\ f \equiv \mathsf{full}$. Given an $a : A$, it must be the case that $a \in \mathsf{full}$ by definition. By congruence using our assumed equality, it must also be the case that $a \in \mathsf{Domain}\ f$. Our element of type $B\ a$ can then be constructed through Lemma 5.4.4. □

5.5 Future Work

We have given suitable constructions in type theory to represent game forms and playable effectivity functions. However, a full formalisation of the construction outlined in Section 5.2 is incomplete, and is left as future work. This section discusses some of the remaining challenges of the formalisation.

First, when defining the transformation from playable effectivity function E to game form, the definition $\mathcal{A}$ is clear, as a type family indexed by Agent.

Definition 5.5.1. An action for agent i is a choice of coalition that i is part of, a set of outcomes that coalition is effective for, a specific outcome from that set, and a natural number.

$$\begin{aligned}
&\mathcal{A} : \mathsf{Agent} \to \mathsf{Type} \\
&\mathcal{A}_i = \Sigma_{(C:\mathcal{P}\ \mathsf{Agent})}\quad i \in C \\
&\quad \times \Sigma_{(X:\mathcal{P}\ \mathsf{Outcome})}\quad X \in E\ C \\
&\quad \times \Sigma_{(x:\mathsf{Outcome})}\quad x \in X \\
&\quad \times \mathbb{N}
\end{aligned}$$

With projection functions $\mathcal{A}_C$ and $\mathcal{A}_X$, we can then define the σ-cooperative predicate for coalitions relative to a specific strategy profile σ.

Definition 5.5.2. Given a strategy profile σ, a coalition C is σ-cooperative when all of its members choose actions in σ which agree on C and X.

$$\begin{aligned}
&\mathsf{coop} : (\Pi_{(i:\mathsf{Agent})}\ \mathcal{A}_i) \to \mathcal{P}\ \mathsf{Agent} \to \mathsf{Type} \\
&\mathsf{coop}\ \sigma\ C = \Pi_{(i:\mathsf{Agent})}\ i \in C \to C \equiv \mathcal{A}_C\ (\sigma\ i) \\
&\qquad\qquad \times \Pi_{(j:\mathsf{Agent})}\ j \in C \to \mathcal{A}_X\ (\sigma\ i) \equiv \mathcal{A}_X\ (\sigma\ j)
\end{aligned}$$

The partitioning process will require a proof that this predicate is decidable, $\mathsf{coop}\ \sigma\ C + (\mathsf{coop}\ \sigma\ C \to \mathbb{0})$. One possibility for a data structure to describe the partition would be an inductive list.

Starting from a list of all agents, we can map a function pointwise which pairs the $\langle C_i,\, X_i\rangle$ choices for each agent, then filter out those elements whose C_i is not σ-cooperative. This gives a list of all non-empty, σ-cooperative coalitions and their choice of X_C, and we must prove each pair of coalitions disjoint so we may later use superadditivity. To define X_σ, we can use a recursive function which intersects the X_i choices of the partition from a base case of full.

$$\begin{aligned}
&\mathsf{intersect} : \mathsf{List}\ (\mathcal{P}\ \mathsf{Agent} \times \mathcal{P}\ \mathsf{Outcome}) \to \mathcal{P}\ \mathsf{Outcome}\\
&\mathsf{intersect}\ [\,] \qquad\qquad\quad\ = \mathsf{full}\\
&\mathsf{intersect}\ (\langle C_i,\, X_i\rangle :: t) = X_i \cap \mathsf{intersect}\ t
\end{aligned}$$

However, this must be done in such a way that the proof terms which verify that $X_i \in E\ C_i$ for each i, which are included the structure of action $\sigma\ i : \mathcal{A}_i$, are retained throughout the process. At each recursive step, we need to use superadditivity with this proof to produce a proof of $X_\sigma \in E\ \mathsf{full}$ along with X_σ. This will justify $X_\sigma \not\equiv \varnothing$ by safety, and allow for the definition of outcome function o.

With a game form $G = \langle \mathcal{A},\, o\rangle$, we have shown in Theorem 5.4.5 that two complementary strategy profiles can be joined to give a complete strategy profile $\sigma : \Pi_{(i:\mathsf{Agent})}\ \mathcal{A}_i$. Using this, we can formulate a version of the following set-theoretic statement in type theory, to be used in the definition of effectivity function E_G.

$$\exists \sigma_C \forall \sigma_{\overline{C}}\ o(\sigma_C \oplus \sigma_{\overline{C}}) \in X$$

To actually define the function, $\mathsf{GameForm} \to \mathcal{P}\ \mathsf{Agent} \to \mathcal{P}\ (\mathcal{P}\ \mathsf{Outcome})$, the result is a decidable subset. When applied to a coalition $C : \mathcal{P}\ \mathsf{Agent}$ and a set of outcomes $X : \mathcal{P}\ \mathsf{Outcome}$, this means the result must be Boolean, $E_G\ C\ X : \mathbb{2}$. However, the obvious translation into type theory, using a Σ-type and a Π-type to express the quantifiers, and omitting details establishing the domains of the partial functions, is a statement residing in a type universe.

$$\Sigma_{(\sigma_C:\mathsf{Agent}\rightharpoonup\mathcal{A})}\ \Pi_{(\sigma_{\overline{C}}:\mathsf{Agent}\rightharpoonup\mathcal{A})}\ o(\sigma_C \oplus \sigma_{\overline{C}}) \in X : \mathsf{Type}$$

To define the effectivity function with the type given above, it must be shown that this statement is decidable. Alternatively, the type of E_G could be reformulated as a non-decidable predicate, $\mathcal{P}\ \mathsf{Agent} \to \mathcal{P}\ \mathsf{Outcome} \to \mathsf{Type}$, but then the notion of playability in Definition 5.3.6 must also be reformulated.

Chapter VI

Concluding Remarks

In Chapter III of this thesis, we developed an embedding for epistemic modal logic within type theory. At first, this was without a type of agents, and we defined what it means to be a knowledge operator in this setting, giving a set of properties such an operator must satisfy. This included the preservation of semantic entailment property, which is an infinitary deduction rule that subsumes two axioms of the finitary axiomatic system of epistemic logic. We proved this knowledge operator semantics, including the additional rule, coincides with the traditional semantics based on equivalence relations, establishing an isomorphism between the two.

In Chapter IV, we extended the embedding with a type of agents, giving each a knowledge relation, and therefore a knowledge operator by using the isomorphism of Chapter III. We introduced a universal knowledge operator to the embedding, and used it to define a common knowledge operator as a coinductive predicate, whose proofs are infinite data structures. We proved that coinductive common knowledge is equivalent to a type family which iterates universal knowledge, and also to the relational interpretation of common knowledge, by taking the union and transitive closure of the individual agents' relations. Formalisations of all results from Chapters III and IV are available in Appendix A, verified by the Agda proof assistant.

In Chapter V, we turned our attention to coalition logic, and its semantics based on game forms and playable effectivity functions. We outlined an equivalence proof for these with the intention of formalisation in type theory. Type-theoretic representations are given, using a type of decidable subsets to define playable effectivity functions, and partial functions used to solve a technical problem when defining an effectivity function from a given game form. However, the formalisation is not currently in a state that can be verified by proof assistant, and completion of the formalisation is left as future work. We sketched the remaining challenges which need to be overcome for the full formalisation.

Appendix A

Agda Formalisations

Formalisations based on the content of Chapter II, Chapter III, and Chapter IV, developed in the Agda proof assistant [48].

A.1 Dependent Type Theory

```
{-
  AGENT-BASED LOGICS IN DEPENDENT TYPE THEORY
  - by Colm Baston

  CHAPTER II: DEPENDENT TYPE THEORY
  - checked with Agda version 2.7.0.1
-}

module TypeTheory where


-- levels for universe polymorphism
open import Agda.Primitive
variable α β γ : Level


{- BASIC TYPE DEFINITIONS -}


data 𝟘 : Set where
  -- no constructors


elim-𝟘 : {T : 𝟘 → Set α} (e : 𝟘) → T e
elim-𝟘 ()
```

```
data 𝟙 : Set where
  ⋆ : 𝟙


elim-𝟙 : {T : 𝟙 → Set α} → T ⋆ → (u : 𝟙) → T u
elim-𝟙 t ⋆ = t


data _+_ (A : Set α) (B : Set β) : Set (α ⊔ β) where
  inl : A → A + B
  inr : B → A + B


elim-+ : {A : Set α} {B : Set β} {T : A + B → Set γ} →
         ((a : A) → T (inl a)) →
         ((b : B) → T (inr b)) →
         (s : A + B) → T s
elim-+ f _ (inl a) = f a
elim-+ _ g (inr b) = g b


𝟚 : Set
𝟚 = 𝟙 + 𝟙


true : 𝟚
true = inl ⋆


false : 𝟚
false = inr ⋆


data Σ (A : Set α) (B : A → Set β) : Set (α ⊔ β) where
  _,_ : (a : A) → B a → Σ A B


_×_ : Set α → Set β → Set (α ⊔ β)
A × B = Σ A (λ _ → B)


elim-Σ : {A : Set α} {B : A → Set β} {T : Σ A B → Set γ} →
         ((a : A) (b : B a) → T (a , b)) →
         (p : Σ A B) → T p
elim-Σ f (a , b) = f a b
```

```
record EquivRel (A : Set α) : Set (α ⊔ lsuc β) where
  field
    _R_ : A → A → Set β
    refl  : {a     : A} → a R a
    sym   : {a b   : A} → a R b → b R a
    trans : {a b c : A} → a R b → b R c → a R c

{- PROPOSITIONAL LOGIC EXAMPLES -}

module Propositional (P Q R : Set) where
  refl-→ : P → P
  refl-→ p = p


  trans-→ : (P → Q) → (Q → R) → P → R
  trans-→ f g p = g (f p)


  non-cont : (P × (P → 𝟘)) → 𝟘
  non-cont = elim-Σ (λ p np → np p)


  DeMorgan₁ : (P → 𝟘) + (Q → 𝟘) → P × Q → 𝟘
  DeMorgan₁ = elim-+ (λ np → elim-Σ (λ p _ → np p))
                     (λ nq → elim-Σ (λ _ q → nq q))


  DeMorgan₂ : (P → 𝟘) × (Q → 𝟘) → P + Q → 𝟘
  DeMorgan₂ = elim-Σ (λ np nq → elim-+ np nq)


  DeMorgan₃ : (P + Q → 𝟘) → (P → 𝟘) × (Q → 𝟘)
  DeMorgan₃ h =  (λ p → h (inl  p))
              , (λ q → h (inr  q))


  -- the final DeMorgan law cannot be proven constructively
  module Classical (excluded-middle : (S : Set) → S + (S → 𝟘)) where
    DeMorgan₄ : (P × Q → 𝟘) → (P → 𝟘) + (Q → 𝟘)
    DeMorgan₄ h = elim-+ (λ p  → elim-+  (λ q  → elim-𝟘 (h (p , q)))
                                         (λ nq → inr nq)
                                         (excluded-middle Q))
                         (λ np → inl np)
                         (excluded-middle P)
```

```
{- PARAMETRICALLY-POLYMORPHIC FUNCTIONS -}

id : {A : Set α} → A → A
id a = a

_$_ : {A : Set α} {B : A → Set β} → ((a : A) → B a) → (a : A) → B a
_$_ = id

const : {A : Set α} {B : Set β} → A → B → A
const a _ = a

if_then_else_ : {A : Set α} → 2 → A → A → A
if b then t else f = elim-+ (const t) (const f) b

flip : {A : Set α} {B : Set β} {C : A → B → Set γ} →
       ((a : A) → (b : B) → C a b) → (b : B) → (a : A) → C a b
flip f b a = f a b

_∘_ : {A : Set α} {B : A → Set β} {C : {a : A} → B a → Set γ} →
        ({a : A} → (b : B a) → C b) →
        (g : (a : A) → B a) →
        (a : A) → C (g a)
f ∘ g = λ a → f (g a)

π₁ : {A : Set α} {B : A → Set β} → Σ A B → A
π₁ = elim-Σ (λ a _ → a)

π₂ : {A : Set α} {B : A → Set β} → (p : Σ A B) → B (π₁ p)
π₂ = elim-Σ (λ _ b → b)

{- PROPOSITIONAL EQUALITY -}

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

```
elim-≡ : {A : Set α} {T : (a b : A) → a ≡ b → Set β} {a b : A} →
           ((c : A) → T c c (refl-≡ c)) → (p : a ≡ b) → T a b p
elim-≡ f (refl-≡ a) = f a


sym-≡ : {A : Set α} {a b : A} → a ≡ b → b ≡ a
sym-≡ = elim-≡ (λ c → refl-≡ c)


trans-≡ : {A : Set α} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
trans-≡ {c = c} = elim-≡ (λ d → id {A = d ≡ c})


cong : {A : Set α} (P : A → Set β) {a b : A} → P a → a ≡ b → P b
cong P = flip (elim-≡ (λ c → id {A = P c}))


resp : {A : Set α} {B : Set β} {a b : A} →
        (f : A → B) → a ≡ b → f a ≡ f b
resp f = elim-≡ (λ c → refl-≡ (f c))


uniqueness : (u : 𝟙) → u ≡ ⋆
uniqueness = elim-𝟙 (refl-≡ ⋆)


Boolean : ((b : 𝟚) → (b ≡ true) + (b ≡ false)) × (true ≡ false → 𝟘)
Boolean = true-or-false , absurd
  where
    true-or-false : (b : 𝟚) → (b ≡ true) + (b ≡ false)
    true-or-false = elim-+ (elim-𝟙 (inl (refl-≡ true)))
                           (elim-𝟙 (inr (refl-≡ false)))


    absurd : true ≡ false → 𝟘
    absurd = cong (if_then 𝟙 else 𝟘) ⋆


{- PROPOSITIONAL EQUIVALENCE -}


_↔_ : Set α → Set β → Set (α ⊔ β)
A ↔ B = (A → B) × (B → A)


refl-↔ : (A : Set α) → A ↔ A
refl-↔ A = id , id
```

```
sym-↔ : {A : Set α} {B : Set β} → A ↔ B → B ↔ A
sym-↔ = elim-Σ (λ f g → g , f)

trans-↔ : {A : Set α} {B : Set β} {C : Set γ} →
            A ↔ B → B ↔ C → A ↔ C
trans-↔ = elim-Σ (λ f g → elim-Σ (λ f' g' → f' ∘ f , g ∘ g'))

{- TYPE EQUIVALENCE -}

_≃_ : Set α → Set β → Set (α ⊔ β)
A ≃ B = Σ (A → B) (λ f →
          Σ (B → A) (λ g → ((a : A) → a ≡ (g ∘ f) a)
                         × ((b : B) → b ≡ (f ∘ g) b)))

equiv-3-3 : 𝟙 + (𝟙 + 𝟙) ≃ (𝟙 + 𝟙) + 𝟙
equiv-3-3 = f , g , p , q
  where
    f : 𝟙 + (𝟙 + 𝟙) → (𝟙 + 𝟙) + 𝟙
    f  = elim-+ (const (inr ⋆))
               (elim-+ (const (inl (inr ⋆)))
                       (const (inl (inl ⋆))))

    g : (𝟙 + 𝟙) + 𝟙 → 𝟙 + (𝟙 + 𝟙)
    g = elim-+ (elim-+ (const (inr (inr ⋆)))
                       (const (inr (inl ⋆))))
               (const (inl ⋆))

    p : (a : 𝟙 + (𝟙 + 𝟙)) → a ≡ (g ∘ f) a
    p = elim-+ (elim-𝟙 (refl-≡ (inl ⋆)))
               (elim-+ (elim-𝟙 (refl-≡ (inr (inl ⋆))))
                       (elim-𝟙 (refl-≡ (inr (inr ⋆)))))

    q : (b : (𝟙 + 𝟙) + 𝟙) → b ≡ (f ∘ g) b
    q = elim-+ (elim-+ (elim-𝟙 (refl-≡ (inl (inl ⋆))))
                       (elim-𝟙 (refl-≡ (inl (inr ⋆)))))
               (elim-𝟙 (refl-≡ (inr ⋆)))

refl-≃ : (A : Set α) → A ≃ A
refl-≃ A = id , id , refl-≡ , refl-≡
```

```
sym-≃ : {A : Set α} {B : Set β} → A ≃ B → B ≃ A
sym-≃ = elim-Σ (λ f → elim-Σ (λ g → elim-Σ (λ p q → g , f , q , p)))


trans-≃ : {A : Set α} {B : Set β} {C : Set γ} →
          A ≃ B → B ≃ C → A ≃ C
trans-≃ (f , g , p , q) (f' , g' , p' , q')
  = f' ∘ f , g ∘ g' , (λ a → cong (λ b → a ≡ g b) (p a) (p' (f a)))
                    , (λ c → cong (λ b → c ≡ f' b) (q' c) (q (g' c)))


{- FUNCTION EQUIVALENCE -}


_∼_ : {A : Set α} {B : A → Set β} →
      (f g : (a : A) → B a) →
      Set (α ⊔ β)
f ∼ g = (a : _) → f a ≡ g a


refl-∼ : {A : Set α} {B : A → Set β} →
         (f : (a : A) → B a) →
         f ∼ f
refl-∼ f a = refl-≡ (f a)


sym-∼ : {A : Set α} {B : A → Set β} →
        {f g : (a : A) → B a} →
        f ∼ g → g ∼ f
sym-∼ e a = sym-≡ (e a)


trans-∼ : {A : Set α} {B : A → Set β} →
          {f g h : (a : A) → B a} →
          f ∼ g → g ∼ h → f ∼ h
trans-∼ d e a = trans-≡ (d a) (e a)


FunExt : {A : Set α} {B : A → Set β} →
         (f g : (a : A) → B a) →
         Set (α ⊔ β)
FunExt f g = f ∼ g → f ≡ g
```

```
{- INDUCTIVE TYPES -}

data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ

elim-ℕ : {T : ℕ → Set α} → T 0 →
         ((n : ℕ) → T n → T (succ n)) →
         (n : ℕ) → T n
elim-ℕ z _ 0        = z
elim-ℕ z s (succ n) = s n (elim-ℕ z s n)

inj-succ : {m n : ℕ} → succ m ≡ succ n → m ≡ n
inj-succ {m} = cong (elim-ℕ 𝟘 (λ o _ → m ≡ o) ) (refl-≡ m)

pred : ℕ → ℕ
pred = elim-ℕ zero (λ n _ → n)

add : ℕ → ℕ → ℕ
add n = elim-ℕ n (λ _ → succ)

add-id-r : flip add 0 ∼ id
add-id-r = refl-≡

add-id-l : add 0 ∼ id
add-id-l = elim-ℕ (refl-≡ 0) (λ _ → resp succ)

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

data Fin : ℕ → Set where
  Fin-zero : {n : ℕ} → Fin (succ n)
  Fin-succ : {n : ℕ} → Fin n → Fin (succ n)

data _≤_ : ℕ → ℕ → Set where
  zero-≤ : {n : ℕ}     → zero ≤ n
  succ-≤ : {m n : ℕ} → m ≤ n → succ m ≤ succ n
```

```
half : ℕ → ℕ
half zero              = zero
half (succ zero)       = zero
half (succ (succ n)) = succ (half n)

refl-≤ : (n : ℕ) → n ≤ n
refl-≤ zero       = zero-≤
refl-≤ (succ n) = succ-≤ (refl-≤ n)

trans-≤ : {m n o : ℕ} → m ≤ n → n ≤ o → m ≤ o
trans-≤ zero-≤       q             = zero-≤
trans-≤ (succ-≤ p) (succ-≤ q) = succ-≤ (trans-≤ p q)

{- COINDUCTIVE TYPES -}

record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail  : Stream A

from : ℕ → Stream ℕ
head (from n) = n
tail  (from n) = from (succ n)

nats : Stream ℕ
nats = from 0

-- cannot be defined for streams
-- due to the unguarded recursive call
filter : {A : Set} → (A → 2) → List A → List A
filter f []         = []
filter f (h :: t) = if f h then h :: filter f t else filter f t

map : {A B : Set} → (A → B) → Stream A → Stream B
head (map f s) = f (head s)
tail  (map f s) = map f (tail s)
```

```
zip : {A B : Set} → Stream A → Stream B → Stream (A × B)
head (zip a b) = head a , head b
tail  (zip a b) = zip (tail a) (tail b)


record Always {A : Set} (P : A → Set) (s : Stream A) : Set where
  coinductive
  field
    always-head : P (head s)
    always-tail  : Always P (tail s)


record Bisim {A : Set} (a b : Stream A) : Set where
  coinductive
  field
    bisim-head : head a ≡ head b
    bisim-tail  : Bisim (tail a) (tail b)


map-succ-from : (n : ℕ) → Bisim (map succ (from n)) (tail (from n))
bisim-head (map-succ-from n) = refl-≡           (succ n)
bisim-tail  (map-succ-from n) = map-succ-from (succ n)


refl-Bisim : {A : Set} (a : Stream A) → Bisim a a
bisim-head (refl-Bisim a) = refl-≡      (head a)
bisim-tail  (refl-Bisim a) = refl-Bisim (tail   a)


sym-Bisim : {A : Set} {a b : Stream A} → Bisim a b → Bisim b a
bisim-head (sym-Bisim p) = sym-≡      (bisim-head p)
bisim-tail  (sym-Bisim p) = sym-Bisim (bisim-tail   p)


trans-Bisim : {A : Set} {a b c : Stream A} →
                Bisim a b → Bisim b c → Bisim a c
bisim-head (trans-Bisim p q) = trans-≡      (bisim-head p)
                                              (bisim-head q)
bisim-tail  (trans-Bisim p q) = trans-Bisim (bisim-tail   p)
                                              (bisim-tail   q)
```

A.2 Epistemic Modal Logic

```
{-
  CHAPTER III: EPISTEMIC MODAL LOGIC
  - checked with Agda version 2.7.0.1
-}

open import Agda.Primitive
open import TypeTheory

module EpistemicLogic (σ : Level) (State : Set σ) where

{- EVENTS AND EPISTEMIC LOGIC CONNECTIVES -}

Event : Set (σ ⊔ lsuc α)
Event {α} = State → Set α

⊥ : Event
⊥ = const 𝟘

⊤ : Event
⊤ = const 𝟙

¬_ : Event {α} → Event
¬ ϕ = λ w → ϕ w → 𝟘

_∨_ : Event {α} → Event {β} → Event
ϕ ∨ ψ = λ w → ϕ w + ψ w

_∧_ : Event {α} → Event {β} → Event
ϕ ∧ ψ = λ w → ϕ w × ψ w

_⇒_ : Event {α} → Event {β} → Event
ϕ ⇒ ψ = λ w → ϕ w → ψ w

_⇔_ : Event {α} → Event {β} → Event
ϕ ⇔ ψ = λ w → ϕ w ↔ ψ w
```

```
∀∀_ : Event {α} → Set (σ ⊔ α)
∀∀_ φ = {w : State} → φ w

_⊆_ : Event {α} → Event {β} → Set (σ ⊔ α ⊔ β)
φ ⊆ ψ = ∀∀ (φ ⇒ ψ)

_≈_ : Event {α} → Event {β} → Set (σ ⊔ α ⊔ β)
φ ≈ ψ = ∀∀ (φ ⇔ ψ)

{- KNOWLEDGE OPERATOR SEMANTICS -}

_⊨_ : {A : Set α} → (A → Event {β}) → Event {γ} → Set (σ ⊔ α ⊔ β ⊔ γ)
Φ ⊨ ψ = {w : State} → ((a : _) → Φ a w) → ψ w

record KOp : Set (σ ⊔ lsuc (α ⊔ β)) where
  field
    K           : Event {α} → Event {α}
    preserves-⊨ : {A : Set β} (Φ : A → Event) →
                  {ψ : Event} → Φ ⊨ ψ → K ∘ Φ ⊨ K ψ
    axiom-T     : {φ : Event} →     K φ ⊆ φ
    axiom-4     : {φ : Event} →     K φ ⊆ K (K φ)
    axiom-5     : {φ : Event} → ¬ K φ ⊆ K (¬ K φ)

K-gen : {{_ : KOp}} → {φ : Event {α}} → ∀∀ φ → ∀∀ K φ
K-gen h = preserves-⊨ elim-𝟘 (const h) elim-𝟘

axiom-K : {{_ : KOp}} → {φ ψ : Event {α}} → K (φ ⇒ ψ) ⊆ (K φ ⇒ K ψ)
axiom-K {_} {φ} {ψ} p q = preserves-⊨ (if_then φ ⇒ ψ else φ)
                                      (λ f → f true (f false))
                                      (elim-+ (const p) (const q))

{- CORRESPONDENCE WITH RELATIONAL SEMANTICS -}

Kripke : {α : Level} → Set (σ ⊔ lsuc α)
Kripke {α} = EquivRel {σ} {α} State

K[_] : (State → State → Set α) → Event {β} → Event {σ ⊔ α ⊔ β}
K[ _R_ ] φ w = {v : State} → w R v → φ v
```

```
_R[_]_ : State → (Event {α} → Event {β}) → State → Set (σ ⊔ lsuc α ⊔ β)
w R[ K ] v = {ϕ : Event} → K ϕ w → K ϕ v

Kripke-KOp : {{_ : Kripke {α}}} → KOp {σ ⊔ α} {β}
Kripke-KOp = record { K           = K[ _R_ ]                        ;
                      preserves-⊨ = λ Φ h p q → h (λ a → p a q)      ;
                      axiom-T     = λ k       → k refl              ;
                      axiom-4     = λ k p     → k ∘ trans p         ;
                      axiom-5     = λ h p k   → h (k ∘ trans (sym p)) }

module Classical (elim-¬¬ : {A : Set α} → ((A → 𝟘) → 𝟘) → A) where
  KOp-Kripke : {{_ : KOp {α} {β}}} → Kripke
  KOp-Kripke = record { _R_   = _R[ K ]_                   ;
                        refl  = id                         ;
                        sym   = λ p k → elim-¬¬ (flip axiom-T k
                                           ∘ p ∘ axiom-5) ;
                        trans = λ p q → q ∘ p                }

iso-R : {β : Level} {{_ : Kripke {σ ⊔ α}}} (w v : State) →
        w R v ↔ w R[ K[ _R_ ] ] v
iso-R {β} w v = ltr , rtl
  where
    ltr : w R v → w R[ K[ _R_ ] ] v
    ltr p k = axiom-4 {{Kripke-KOp {β = β}}} k p

    rtl : w R[ K[ _R_ ] ] v → w R v
    rtl p = p id refl

iso-K : {{_ : KOp}} (ϕ : Event {α}) → K ϕ ≈ K[ _R[ K ]_ ] ϕ
iso-K ϕ = ltr , rtl
  where
    ltr : K ϕ ⊆ K[ _R[ K ]_ ] ϕ
    ltr k p = axiom-T (p k)

    rtl : K[ _R[ K ]_ ] ϕ ⊆ K ϕ
    rtl k = preserves-⊨ (K ∘ π₁)
                        (λ h → k (λ {ψ} j → h (ψ , j)))
                        (axiom-4 ∘ π₂)
```

A.3 Coinductive Common Knowledge

```
{-
  CHAPTER IV: COINDUCTIVE COMMON KNOWLEDGE
  - checked with Agda version 2.7.0.1
-}

open import Agda.Primitive
open import TypeTheory

module CommonKnowledge
  (State : Set)
  (Agent : Set)
  (agent : Agent)
  (K-rel : Agent → EquivRel {β = lzero} State) where

open import EpistemicLogic lzero State

_∝[_]_ : State → Agent → State → Set
w ∝[ a ] v = (K-rel a)._R_ w v

K : Agent → Event {α} → Event
K a = K[ _∝[ a ]_ ]

K-op : Agent → KOp
K-op a = Kripke-KOp {lzero} {lzero} {{K-rel a}}

EK : Event {α} → Event
EK ϕ w = {a : Agent} → K a ϕ w

preserves-⊨-EK : {A : Set α} (Φ : A → Event) {ψ : Event {β}} →
                 Φ ⊨ ψ → EK ∘ Φ ⊨ EK ψ
preserves-⊨-EK Φ h i {a} p = h (λ t → axiom-T {{K-op a}}
                                  $ axiom-4 {{K-op a}} (i t) p)

axiom-T-EK : {ϕ : Event} → EK ϕ ⊆ ϕ
axiom-T-EK k = axiom-T {{K-op agent}} k
```

```
{- COINDUCTIVE COMMON KNOWLEDGE DEFINITION -}

record cCK (ϕ : Event) (w : State) : Set where
  coinductive
  field
    cCK-EK     : EK ϕ w
    cCK-cCKEK : cCK (EK ϕ) w

cCK-EKcCK : {ϕ : Event} → cCK ϕ ⊆ EK (cCK ϕ)
cCK-EK     (cCK-EKcCK k p) = cCK-EK     (cCK-cCKEK k) p
cCK-cCKEK (cCK-EKcCK k p) = cCK-EKcCK (cCK-cCKEK k) p

recCK : Event {α} → ℕ → Event
recCK ϕ zero     = ϕ
recCK ϕ (succ n) = recCK (EK ϕ) n

recCK-cCK : (ϕ : Event) → recCK ϕ ⊨ cCK ϕ
cCK-EK     (recCK-cCK ϕ h) = h 1
cCK-cCKEK (recCK-cCK ϕ h) = recCK-cCK (EK ϕ) (h ∘ succ)

cCK-recCK : (ϕ : Event) → (n : ℕ) → cCK ϕ ⊆ recCK ϕ n
cCK-recCK ϕ zero     k = axiom-T-EK (cCK-EK k)
cCK-recCK ϕ (succ n) k = cCK-recCK (EK ϕ) n (cCK-cCKEK k)

{- RELATIONAL COMMON KNOWLEDGE DEFINITION -}

data _∝_ : State → State → Set where
  union-∝ : {a : Agent} {w v : State} → w ∝[ a ] v → w ∝ v
  trans-∝ : {w v u : State} → w ∝ v → v ∝ u → w ∝ u

rCK : Event {α} → Event
rCK = K[ _∝_ ]

refl-∝ : {w : State} → w ∝ w
refl-∝ = union-∝ ((K-rel agent).refl)

sym-∝ : {w v : State} → w ∝ v → v ∝ w
sym-∝ (union-∝ {a} p) = union-∝ ((K-rel a).sym p)
sym-∝ (trans-∝ p q)     = trans-∝ (sym-∝ q) (sym-∝ p)
```

```
Kripke-∝ : Kripke
Kripke-∝ = record { _R_  = _∝_     ;
                    refl  = refl-∝  ;
                    sym   = sym-∝   ;
                    trans = trans-∝ }


rCK-op : KOp
rCK-op = Kripke-KOp {β = lzero} {{Kripke-∝}}


rCK-EK : {φ : Event {α}} → rCK φ ⊆ EK φ
rCK-EK r p = r (union-∝ p)


rCK-rCKEK : {φ : Event {α}} → rCK φ ⊆ rCK (EK φ)
rCK-rCKEK r p q = r (trans-∝ p (union-∝ q))


rCK-EKrCK : {φ : Event {α}} → rCK φ ⊆ EK (rCK φ)
rCK-EKrCK r p q = r (trans-∝ (union-∝ p) q)


{- EQUIVALENCE WITH RELATIONAL DEFINITION -}


cCK-transport : {φ : Event} {w v : State} → cCK φ w → w ∝ v → cCK φ v
cCK-transport k (union-∝ {a} p) = axiom-T {{K-op a}}
                                  $ axiom-4 {{K-op a}} (cCK-EKcCK k) p
cCK-transport k (trans-∝ p q)   = cCK-transport (cCK-transport k p) q


cCK-rCK : {φ : Event} → cCK φ ≈ rCK φ
cCK-rCK = ltr , rtl
  where
    ltr : {φ : Event} → cCK φ ⊆ rCK φ
    ltr k p = axiom-T-EK (cCK-EK (cCK-transport k p))


    rtl : {φ : Event} → rCK φ ⊆ cCK φ
    cCK-EK     (rtl k) = rCK-EK k
    cCK-cCKEK (rtl k) = rtl (rCK-rCKEK k)
```

```
{- COINDUCTIVE COMMON KNOWLEDGE IS A KNOWLEDGE OPERATOR -}


preserves-⊨-cCK : {A : Set α} (Φ : A → Event) {ψ : Event} →
                  Φ ⊨ ψ → cCK ∘ Φ ⊨ cCK ψ
preserves-⊨-cCK Φ h i = π₂ cCK-rCK (λ p → h (λ t → axiom-T {{rCK-op}}
                                                   ∘ π₁ cCK-rCK
                                                   $ cCK-transport (i t) p))


axiom-T-cCK : {ϕ : Event} → cCK ϕ ⊆ ϕ
axiom-T-cCK k = axiom-T {{rCK-op}} (π₁ cCK-rCK k)


axiom-4-cCK : {ϕ : Event} → cCK ϕ ⊆ cCK (cCK ϕ)
axiom-4-cCK k = π₂ cCK-rCK (λ p → π₂ cCK-rCK
                           (λ q → axiom-T-cCK
                              $ cCK-transport (cCK-transport k p) q))


axiom-5-cCK : {ϕ : Event} → ¬ cCK ϕ ⊆ cCK (¬ cCK ϕ)
axiom-5-cCK nk = π₂ cCK-rCK (λ p k → nk
                                ∘ cCK-transport k
                                $ sym-∝ p)


cCK-op : KOp {lzero} {β}
cCK-op = record { K            = cCK            ;
                  preserves-⊨ = preserves-⊨-cCK ;
                  axiom-T      = axiom-T-cCK    ;
                  axiom-4      = axiom-4-cCK    ;
                  axiom-5      = axiom-5-cCK    }
```

Bibliography

[1] Thomas Ågotnes and Natasha Alechina. Coalition logic with individual, distributed and common knowledge. *Journal of Logic and Computation*, 29:1041–1069, 2018.

[2] Robert J. Aumann. Agreeing to disagree. *Annals of Statistics*, 4:1236–1239, 1976.

[3] Robert J. Aumann. Backward induction and common knowledge of rationality. *Games and Economic Behavior*, 8(1):6–19, 1995.

[4] Jon Barwise. Three views of common knowledge. In *Proceedings of the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge, Pacific Grove, CA, March 1988*, pages 365–379, 1988.

[5] Colm Baston and Venanzio Capretta. The coinductive formulation of common knowledge. In *International Conference on Interactive Theorem Proving*, 2018.

[6] Colm Baston and Venanzio Capretta. Game forms for coalition effectivity functions. In *International Conference on Types for Proofs and Programs*, 2019.

[7] Yves Bertot and Pierre Castran. Interactive theorem proving and program development: Coq'art the calculus of inductive constructions. 2010.

[8] Mark Bickford, Fedor A. Bogomolov, and Yuri Tschinkel. Vladimir Voevodsky – work and destiny. 2017.

[9] Edwin C. Brady. Type-driven development with Idris. Manning, 2017.

[10] John Charles Burkill and John Edensor Littlewood. A mathematician's miscellany. *The Mathematical Gazette*, 38:47, 1954.

[11] Kevin Buzzard. *What is the Xena project?* `https://xenaproject.wordpress.com/what-is-the-xena-project/`.

[12] Venanzio Capretta. Common knowledge as a coinductive modality. In Erik Barendsen, Herman Geuvers, Venanzio Capretta, and Milad Niqui, editors, *Reflections on Type Theory, Lambda Calculus, and the Mind*, pages 51–61. ICIS, Faculty of Science, Radbout University Nijmegen, 2007. Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday.

[13] Venanzio Capretta. Coalgebras in functional programming and type theory. *Theoretical Computer Science*, 412(38):5006–5024, 2011. CMCS Tenth Anniversary Meeting.

[14] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346, 1932.

[15] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345, 1936.

[16] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56 – 68, 1940.

[17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. 2016.

[18] Thierry Coquand. Pattern matching with dependent types. 1992.

[19] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[20] Paolo Crivelli, Timothy Luke Williamson, Gareth Hughes, and Max J. Cresswell. A new introduction to modal logic. 1998.

[21] Haskell B. Curry. Some philosophical aspects of combinatory logic. *Studies in logic and the foundations of mathematics*, 101:85–101, 1980.

[22] Leonardo Mendonça de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *CADE*, 2021.

[23] Peter Dybjer. Inductive sets and families in Martin-Loϊf's type theory and their set-theoretic semantics. 1991.

[24] Ronald Fagin, Joseph Y. Halpern, Moshe Y. Vardi, and Yoram Moses. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 1995.

[25] Daniel P. Friedman and David S. Wise. Cons should not evaluate its arguments. In *International Colloquium on Automata, Languages and Programming*, 1976.

[26] George Gamow and Marvin Stern. *Puzzle Math.* Viking Press, New York, 1958.

[27] James Garson. Modal Logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Spring 2024 edition, 2024.

[28] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs. International Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[29] Jean-Yves Girard. Une extension de Ľinterpretation de Gödel a Ľanalyse, et son application a Ľelimination des coupures dans Ľanalyse et la theorie des types. *Studies in logic and the foundations of mathematics*, 63:63–92, 1971.

[30] Jean-Yves Girard. Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur. 1972.

[31] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Essays Dedicated to Joseph A. Goguen*, 2006.

[32] Peter B. Henderson and James H. Morris. A lazy evaluator. *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, 1976.

[33] Jaakko Hintikka. *Knowledge and Belief.* Ithaca: Cornell University Press, 1962.

[34] Ralf Hinze. The Bird tree. *Journal of Functional Programming*, 19:491 – 508, 2009.

[35] Simon Huber. Canonicity for cubical type theory. *Journal of Automated Reasining*, pages 173–210, 2019.

[36] Antonius J. C. Hurkens. A simplification of Girard's paradox. In *International Conference on Typed Lambda Calculus and Applications*, 1995.

[37] Saul A. Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24(1):1–14, 03 1959.

[38] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[39] Pierre Lescanne. Common knowledge logic in a higher order proof assistant. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 271–284. Springer, 2013.

[40] C.I. Lewis and C.H. Langford. *Symbolic Logic*. Century philosophy series. Century Company, 1932.

[41] David Lewis. Convention: A philosophical study. 1969.

[42] Simon Marlow. Haskell 2010 language report. 2010.

[43] Per Martin-Löf. An intuitionistic theory of types. 1972.

[44] Per Martin-Löf. An intuitionistic theory of types: Predicative part. *Studies in logic and the foundations of mathematics*, 80:73–118, 1975.

[45] Per Martin-Löf. Intuitionistic type theory. In *Studies in proof theory*, 1984.

[46] Randall Munroe. *XKCD: Blue Eyes*. `https://xkcd.com/blue_eyes.html`.

[47] Hiroshi Nakano. A modality for recursion. *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

[48] Ulf Norell. Dependently typed programming in Agda. In *Lecture Notes from the Summer School on Advanced Functional Programming, AFP 2008*, pages 230–266. Springer Berlin Heidelberg, 2009.

[49] Marc Pauly. A modal logic for coalitional power in games. *Journal of Logic and Computation*, 12, 02 2002.

[50] John C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, 1974.

[51] John C. Reynolds. *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*, pages 309–317. Springer-Verlag, New York, NY, 1978.

[52] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

[53] Bertrand Russell and Alfred North Whitehead. *Principia Mathematica*, volume 1. 1910.

[54] Thomas Streicher. Semantics of type theory - correctness, completeness and independence results. In *Progress in theoretical computer science*, 1991.

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

[56] Wiebe van der Hoek and Michael Wooldridge. Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. *Studia Logica*, 75:125–157, 2003.

[57] Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture*, 1989.