Foundations and Applications of Modal Type Theories

JASON Z. S. HU, McGill University, Canada

Modalities are truth quantifiers and alter the meanings of truths. Traditionally, modalities are studied in the field of modal logic from a philosophical perspective. Under Curry-Howard correspondence, we realize that modalities also add various capabilities to programming languages. These capabilities have found many useful applications in areas like meta- and multi-staged programming, guarded recursions, and theoretical foundations of programming languages. Yet, the foundational study of combining modalities and types (especially dependent types) is notoriously challenging. In this thesis proposal, we investigate a number of theories that combines modalities and (dependent) types, complementing some existing work in the community. As a first step, we restudy the system $\lambda^{\to\Box}$ proposed by Pfenning, Wong and Davies and prove its normalization, a corollary of which is its consistency when viewed as a logical system. Then we scale $\lambda^{\to\Box}$ to Martin-Löf-style dependent types and again prove its normalization. At last, as the remaining goal of this thesis, we focus on developing a modal type theory for dependently typed meta-programming and pattern matching on code.

Additional Key Words and Phrases: modal logic, dependent types, normalization, meta-programming

1 TYPE THEORIES WITH MODALITY

Modalities are ubiquitous in computer science. On a practical side, the modal logic $S4$ corresponds to meta- and multi-staged programming [Davies and Pfenning 2001; Jang et al. 2022] where the necessity modality ($\Box$) is used to represent the code of a program and the evaluation of code is divided into different stages of computation. The modal logic $S5$ is used in distributed computing [Murphy VII et al. 2004] and modalities are used to represent mobile code to be executed and addresses of remote values. Jeffrey [2012] discovers that linear temporal logic corresponds to functional reactive programming, which is widely used in graphical interfaces and web services. Guarded recursion [Clouston et al. 2015; Nakano 2000] is a technique that ensures productivity of a recursion while relaxes the usual syntactic criteria. Modalities can also be used to track algebraic effects [Zyuzin and Nanevski 2021]. Other applications of modalities include quantitative analysis, differential privacy and information-flow security [Abel and Bernardy 2020; Reed and Pierce 2010]. On a theoretical side, Pientka et al. [2019] propose to encapsulate terms of an object language in a modality and hence ease the mechanization of the meta-theory of the object language. Modalities also find applications in the recent area of homotopy type theory [Licata et al. 2018; Shulman 2018], where modalities provide a convenient machinery to distinguish topological structures. In contrary to the wide range of applications of modalities, not all modal type theories are well-understood to their foundations. In fact, the foundational study of modalities are generally challenging.

In this thesis, we contribute to the recent advancements of modal type theory and conduct a general study of a specific $\Box$ modality in both simply typed and dependently typed cases. Then we focus on applying modal type theory to meta-programming. Our attention is primarily on the modal logic $S4$ with the $\Box$ modality and its sublogics. The modal logic $S4$ is characterized by three axioms: K, T and the Axiom 4. Among Axioms K, T and 4, we must admit K but T and 4 are optional, which yield in total four different modal logics (i.e. Systems K, T, $K4$ and $S4$). The following table summarizes the relation between axioms and modal logics:

Author's address: Jason Z. S. Hu, zhong.s.hu@mail.mcgill.ca, School of Computer Science, McGill University, McConnell Engineering Bldg. , 3480 University St., Montréal, Québec, Canada, H3A 0E9.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Axiom \ System	K	T	$K4$	$S4$
K: $\square(S \longrightarrow T) \to \square S \to \square T$	✓	✓	✓	✓
T: $\square T \to T$		✓		✓
4: $\square T \to \square\square T$			✓	✓

On the semantic side, Kripke [1963] show that many standard modal axioms correspond to algebraic structures of the accessibility relation in a multi-world semantic model (or a Kripke semantics nowadays). A Kripke semantics provides not only a unified semantic framework to study modal logic, but also a theoretical device for understanding a wide range of properties of programming languages.

There is a long history of combining modalities and programming languages. Prawitz [1965] first proposes a formulation in natural deduction for the $S4$ modal logic. However, as pointed out by Bierman and de Paiva [2000]; Pfenning and Davies [2001], the formulation given by Prawitz [1965] is not closed under substitution, and thus the system is unsound. Indeed, making substitutions work with modalities is the main challenge of all studies of modalities, including this thesis. Many have looked into how to develop a modal system so that substitutions behave coherently [Bierman and de Paiva 1996, 2000; Borghuis 1994; Davies and Pfenning 2001; Martini and Masini 1996; Pfenning and Davies 2001; Pfenning and Wong 1995, etc.]. One fix proposed by Bierman and de Paiva [2000] is to "bolt" substitutions onto the introduction rule:

$$\frac{\forall i \in [0,n).\Gamma \vdash s_i : \square S_i \qquad x_0 : \square S_0, \cdots, x_{n-1} : \square S_{n-1} \vdash t : \square T}{\Gamma \vdash \mathsf{box}\ t\ \mathtt{with}\ s_0/x_0, \cdots, s_{n-1}/x_{n-1} : \square T}$$

Here the introduction rule for $\square$ maintains a list of substitutions of variables which replaces all modal assumptions that t depends on (replacing s_i for x_i for all i). Notice that t must depend on assumptions of type $\square S_i$ only and not in any other form, because $\square$ in $S4$ represents validity or necessity, which is stronger than a truth assertion. The elimination rule is simple:

$$\frac{\Gamma \vdash t : \square T}{\Gamma \vdash \mathsf{unbox}\ t : T}$$

The reduction rule simply applies the pre-stored substitutions:

$$\mathsf{unbox}\ (\mathsf{box}\ t\ \mathtt{with}\ s_0/x_0, \cdots, s_{n-1}/x_{n-1}) \approx t[s_0/x_0, \cdots, s_{n-1}/x_{n-1}]$$

Though this formulation makes the type theory closed under substitutions, it does not seem very practical. For one, having to specify a complete list of substitutions when constructing a $\square$ seems too restrictive. We would hope to only provide substitutions when applying the elimination rule. For comparison, function arguments are provided during function application; it would make function abstraction pointless if one must provide the arguments during the definition of a function. For this reason, we turn to Davies and Pfenning [2001]; Pfenning and Davies [2001]; Pfenning and Wong [1995] for more practical formulations.

Davies and Pfenning [2001]; Pfenning and Davies [2001]; Pfenning and Wong [1995] propose two different formulations: the dual-context style (or the explicit style) and the Kripke style (or the implicit style). On a high level, two formulations start from different philosophical perspectives: the dual-context style starts from the distinction between validity and truth, and the Kripke style, as indicated by its name, starts from the Kripke semantics of modal logic. In the dual-context style, there are two categories of facts: valid ones and true ones. Truths are interpreted as usual; validities, on the other hand, are stronger, representing "necessary truths". In particular, the proof of a valid fact can only depend on other valid facts, but no mere truth. The $\square$ modality then bridges validity and truth. More concretely, the introduction and elimination rules are:

$$\frac{\Psi; \cdot \vdash t : T}{\Psi; \Gamma \vdash \mathsf{box}\ t : \square T} \qquad \frac{\Psi; \Gamma \vdash t : \square T \qquad \Psi, u : T; \Gamma \vdash t' : T'}{\Psi; \Gamma \vdash \mathtt{letbox}\ u = t\ \mathtt{in}\ t' : T'}$$

In the dual-context style, two contexts are used to keep track of assumptions, where Ψ maintains the valid assumptions, while Γ maintains the true ones. On the left is the introduction rule, where $\cdot$ denotes the empty context. This rule is significantly simpler than the rule by Bierman and de Paiva [2000] above because it does not require a substitution. Instead, the true assumptions are cleared out, only keeping the valid context. This is equivalent to Bierman and de Paiva [2000]'s rule only depending on other $\square$ assumptions. The elimination rule on the right then unpacks $\square T$ by pattern matching. We consider this formulation more practical than Bierman and de Paiva [2000] because the introduction rule no longer requires a pre-determined substitution. A substitution is deferred to the reduction time of the elimination rule:

$$\texttt{letbox}\ u = \mathsf{box}\ t\ \texttt{in}\ t' \approx t'[t/u]$$

The use of box t is determined after by t', instead of pre-determined. This adjustment makes the dual-context style very easy to apply. There are several recent extensions of the dual-context style. Licata et al. [2018]; Shulman [2018] extend the dual-context style to dependent types. Jang et al. [2022] give a System F-style extension for meta-programming and add support for pattern matching on the representation of programs. Gratzer [2022]; Gratzer et al. [2020, 2021] extend the dual-context style to a multi-context style with a 2-category parameter. Their system admits more than one modalities and the interactions of these modalities are described by the 2-category parameter.

Nevertheless, for certain applications like meta-programming, the dual-context style is not completely satisfactory, as it does not formulate a typical, familiar programming paradigm. In Lisp and other meta-programming systems like MetaML, Isabelle/ML, etc., quasi-quoting is a more familiar style of meta-programming. With quasi-quoting, we can quote and convert a term into its representation, transform the term, splice the transformed term into some larger code fragment and eventually evaluate the meta-program to get an algorithmically constructed program. The dual-context style, on the other hand, does not directly model this (meta-)programming paradigm. Moreover, the dual-context style yields very different formulations for four subsystems of $S4$ [Kavvos 2017]. Luckily, Davies and Pfenning [2001]; Pfenning and Wong [1995] give another formulation $\lambda^{\to\square}$ in the Kripke style, which avoids all aforementioned problems.

The Kripke style models the Kripke semantics. In a Kripke semantics, there is a universe of worlds and worlds are connected by an accessibility relation. Different modal logics are obtained by giving algebraic structures to this accessibility relation. For example, if this relation is reflexive, then the semantics corresponds to the logic T; if the relation is reflexive and transitive, then the semantics corresponds to the logic $S4$. The Kripke style then uses a context stack to model a path of previously accessed worlds and considers $\square T$ as a fact T in the "next world". The introduction and elimination rules in the Kripke style are:

$$\frac{\vec{\Gamma}; \cdot \vdash t : T}{\vec{\Gamma} \vdash \mathsf{box}\ t : \square T} \qquad\qquad \frac{\vec{\Gamma} \vdash t : \square T \qquad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \mathsf{unbox}_n\ t : T}$$

Here we use the vector form ($\vec{\Gamma}$) to represent a context stack. A context stack in a judgment stores at least one context. Two adjacent contexts are separated by a semicolon (;) while assumptions in the same context are still separated by a comma (,). The introduction rule says that if T is true in the next world without any local assumption, then $\square T$ is true in the current world. The elimination rule allows us to travel to some previously accessed world by specifying an unbox level n of topmost contexts (or worlds), called the modal offset, as part of the eliminator. The elimination process then skips the topmost n contexts ($|\vec{\Delta}|$ counts the number of contexts in $\vec{\Delta}$) and requires t to have type $\square T$. This style is more akin to the familiar quasi-quoting style we mentioned above. If we consider meta-programs as just programs running in the next world, then box corresponds to quoting. unbox has two roles. unbox_n corresponds to splicing for $n > 0$ and unbox_0 corresponds to evaluation. The latter can be understood by specializing the elimination rule. For unbox_0, we have

$$\frac{\overrightarrow{\Gamma} \vdash t : \Box T}{\overrightarrow{\Gamma} \vdash \mathsf{unbox}_0\ t : T}$$

This rule extracts a T from a meta-program of T in the same context stack, so it is evaluation. The Kripke style also has the advantage of representing different variants of modal logic by tuning the range of modal offsets unbox levels [Davies and Pfenning 2001; Hu and Pientka 2022a; Pfenning and Wong 1995]. This is the primary reason why we consider the Kripke style in the first part of this thesis. However, we will see in Sec. 4 that the Kripke style has certain disadvantages, adding difficulties to its investigation.

The Kripke style was also investigated under different names. Borghuis [1994] formulates a version of modal pure type system (PTS) using the context stack structure (called generalized contexts), inspired by Fitch-style subordinate proof diagrams. Bellin et al. [2001]; Clouston [2018] later on refer to this style as the Fitch style. Instead of a context stack formulation, the latter flattens a context stack into one context, but places a special symbol (🔒) in between to segment a context into regions. That is, given a context stack $\epsilon; \Gamma_1; \cdots ; \Gamma_n$ in the Kripke style, its equivalent context is Γ_1, 🔒, $\cdots$, 🔒, Γ_n in the Fitch style. Though the change is merely syntactic, it does introduce some slight inconvenience. In the Kripke style, taking off a certain number of contexts is simply done by popping off the stack, while with 🔒, the same operation becomes inconvenient. It must scan the context and count the number of 🔒s countered so far. For this reason, many recent works on the Fitch style [Birkedal et al. 2020; Clouston 2018; Gratzer et al. 2019] only consider idempotency for simplicity. Valliappan et al. [2022] give a version of non-idempotent $S4$ where the unbox level in the Kripke style is replaced by a witness of the accessibility relation between the source and the target contexts. This treatment, however, is at the cost of losing the generality of models which allows us to talk about all four subsystems of $S4$ simultaneously as we will see very soon.

2 GOALS OF THIS THESIS

The high-level goal of this thesis is to develop firm foundations for various modal (dependent) type theories and to study their implementations and applications. We first begin with a general study of the $\Box$ modality. We study its behaviors in simply typed and dependently typed cases. After understanding the $\Box$ modality generally, we develop a modal dependent type theory with the capability of pattern matching on code, which finds an application to meta-programming. The thesis is broken down into the following concrete goals:

- We first reexamine $\lambda^{\to\Box}$ given by Davies and Pfenning [2001] and develop a better substitution calculus. Using this substitution calculus, we are able to prove the normalization property of $\lambda^{\to\Box}$ directly, not only justifying $\lambda^{\to\Box}$ as a logic, but also giving a normalization algorithm. In our work, we have succeeded in developing two different normalization proofs (hence two algorithms) based on two different mathematical models. As a bonus, our methods are general and modular enough so that they apply to all four subsystems of $S4$ without change.
- As a second step, based on our results from $\lambda^{\to\Box}$, we scale our development to Martin-Löf type theory, obtaining Mint. Mint can be specialized to any dependently typed variant of Systems K, T, $K4$ and $S4$. It appears that our method in our previous work in $\lambda^{\to\Box}$ also scales very naturally to dependent types. As a result, we prove the normalization of Mint.
- We mechanize the proofs in the previous two goals in Agda, a proof assistant implementing Martin-Löf type theory. We can extract normalization algorithms from these proofs in Agda to Haskell. These extracted algorithms can serve as trusted kernels for implementations of $\lambda^{\to\Box}$ and Mint.
- As the last piece of work to be done in the remainder of this PhD, we focus on the application to meta-programming and attempt to extend a modal dependent type theory with the ability of doing pattern

matching on code. Adding pattern matching on code allows us to write meta-programs which algorithmically generate programs or proofs of desired forms while still take advantages of the type safety of dependent types.

3 EXAMPLES FOR MODAL TYPE THEORIES

Before diving into the current progress of this thesis, let us consider a few examples for modal type theories. The examples for simple types and Kripke-style systems are genuine as they have been covered by previous work. There are some other examples that are more hypothetical; these are the examples which we would like to make them work in the remainder of this thesis.

3.1 Modal Axioms

Recall that modal logics are characterized by the axioms that they admit. In this thesis, we study the combinations of the following three axioms:

$$K : \square(A \longrightarrow B) \longrightarrow \square A \longrightarrow \square B$$
$$T : \square A \longrightarrow A$$
$$A4 : \square A \longrightarrow \square\square A$$

They can all be implemented by the systems we previously mentioned.

In the simply typed setting, their implementations are folklore. The following shows the implementations of these axioms in both Kripke style ($\lambda^{\rightarrow\square}$; left) and dual-context style (right):

```
K : □ (A ⟶ B) ⟶ □ A ⟶ □ B
K f x = box ((unbox1 f) (unbox1 x))


T : □ A ⟶ A
T x = unbox0 x

A4 : □ A ⟶ □ □ A
A4 x = box (box (unbox2 x))
```

```
K : □ (A ⟶ B) ⟶ □ A ⟶ □ B
K f x = let box f' = f in
        let box x' = x in box (f x)

T : □ A ⟶ A
T x = let box x' = x in x'

A4 : □ A ⟶ □ □ A
A4 x = let box x' = x in box (box x')
```

In the Kripke style, we can directly use a □ type immediately by doing unbox, while in the dual-context style, we must first use **let** box to extract the content inside of a □.

With dependent types, the situation is very similar. In Mint which is our dependently typed version of $\lambda^{\rightarrow\square}$ also admits these axioms and allows type parameters for A and B above:

```
K : {A B : □ Se} ⟶ □ ((unbox1 A) ⟶ (unbox1 B)) ⟶ □ (unbox1 A) ⟶ □ (unbox1 B)
K f x = box ((unbox1 f) (unbox1 x))

T : {A : □ Se} ⟶ □ (unbox1 A) ⟶ unbox0 A
T x = unbox0 x

A4 : {A : □ Se} ⟶ □ (unbox1 A) ⟶ □ □ (unbox2 A)
A4 x = box (box (unbox2 x))
```

where Se denotes the type for universes. We avoid Agda's **Set** because our mechanization is in Agda. The body of the definitions are identical to those in $\lambda^{\rightarrow\square}$. It is the types that are interesting. In T, for example, we cannot naively assign the following type

```
T : {A : Se} → □ A → A
```

because □ requires the type inside is well-formed in the next world while A is in the current world. To allow access to A inside of □, we must turn it into a □ Se so that it can be accessed via elimination. The same principle applies for K and A4.

On the other hand, in a dual-context-style dependent type theory, e.g. crisp type theory [Licata et al. 2018], the same axioms are implemented as:

```
K : {A B : □ Se} → let box A' = A in let box B' = B in □ (A' → B') → □ A' → □ B'
K {A} {B} = let box _ = A in let box _ = B in
            λ f x → let box f' = f in let box x' = x in box (f x)

T : {A : □ Se} → let box A' = A in □ A' → A'
T {A} = let box _ = A in λ x → let box x' = x in x'

A4 : {A : □ Se} → let box A' = A in □ A' → □ □ A'
A4 {A} = let box _ = A in λ x → let box x' = x in box (box x')
```

The types of these axioms follow the same principle as in Mint. The difference is due to the elimination of □. Instead of unbox, we use `let box`. As a consequence, we also need to eliminate the input types A and B in the body for typechecking purposes even if we do not actually use them. Consider for example the body of T. Before the first `let box`, we must provide a term of type `let box A' = A in □ A' → A'`. This type is not even known to be a function type, so we cannot even do a function introduction to construct a term of this type. To proceed, we must first `let box` so that our goal becomes `□ A' → A'`, which allows us to do a function introduction subsequently. Even in this simple example, it seems to indicate that the dual-context style is harder to use than the Kripke style.

3.2 Example for Mint: A Program Logic for Meta-programming

As part of the current progress, we have scale the Kripke style to full Martin-Löf type theory [Martin-Löf 1984] with Russell's cumulative universes [Palmgren 1998]. We call the resulting system Mint, a **M**odal **IN**tuitionistic **T**ype theory. According to Davies and Pfenning [2001], the modal logic *S*4 corresponds to meta-programming. This implies that we can use the *S*4 variant of Mint as a program logic for dependently typed meta-programming. In this section, we give one such example for a meta-program that generates *n*-ary sum function and prove that this function is implemented correctly.

The high-level idea is that this meta-program, when given 2, it generates a function `λ x y → x + y`; when given 3, it generates a function `λ x y z → x + y + z`; and so on. If we look at the types of these functions, they depend on the input n. We then need a function which large-eliminates a natural number to represent the type for these functions:

```
nary : Nat → Se
nary zero     = N
nary (succ n) = N → nary n
```

The nary function is our target function and generates an n-ary function that takes n numbers and return a number. Next we implement the meta-program which generates the desired function:

```
nary-sum : (n : Nat) → □ (nary (unbox₁ (lift n)))
nary-sum zero            = box zero
nary-sum (succ zero)     = box λ x → x
nary-sum (succ (succ n)) = box λ x y → (unbox₁ (nary-sum (succ n))) (x + y)
```

Let us first consider its type: it takes a number n which denotes the arity of the generated function. □ denotes that this is a meta-program. we want this meta-program to generate a nary n, so ideally we want the return type to be □ (nary n). However, since □ requires a type in the next world while nary n is only valid in the current one, □ (nary n) is not well-typed. To fix that, we need a lift function of type Nat ⟶ □ Nat which brings a number from the current world to the next one:

```
lift : N ⟶ □ N
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

Since lift n has type □ N, we then use unbox₁ to bring it to Nat but in the next world. The following are traces of invoking nary-sum:

```
nary-sum 1 = box λ x1 ⟶ x1
nary-sum 2 = box λ x2 x1 ⟶ (unbox₁ (nary-sum 1)) (x2 + x1)
           = box λ x2 x1 ⟶ (λ x1 ⟶ x1) (x2 + x1)
           = box λ x2 x1 ⟶ x2 + x1
nary-sum 3 = box λ x3 x2 ⟶ (unbox₁ (nary-sum 2)) (x3 + x2)
           = box λ x3 x2 ⟶ (λ x2 x1 ⟶ x2 + x1) (x3 + x2)
           = box λ x3 x2 x1 ⟶ x3 + x2 + x1
```

nary-sum seems to do what we expect for a finite number of test runs, but how do we prove that it is correct for all possible n? This is achieved by proving a lemma within Mint itself. Since Mint is a Martin-Löf type theory, it already supports proving properties of programs written in itself. The target theorem we are trying to establish here is, informally, applying this generated function with n numbers is indeed adding these n numbers together. To formally state this theorem, we first define the function sum which sums up all the numbers in a list xs of length n and the function ap-list which applies a function f : nary n to all the numbers in xs:

```
sum : (n : Nat) (xs : List Nat) ⟶ length xs ≡ n ⟶ Nat
sum zero              []            refl = zero
sum (succ zero)       (x ∷ [])      refl = x
sum (succ (succ n)) (x ∷ y ∷ xs) eq   = sum (succ n) ((x + y) ∷ xs) omitted-eq

ap-list : (n : Nat) (xs : List Nat) ⟶ length xs ≡ n ⟶ nary n ⟶ Nat
ap-list zero      []       refl x = x
ap-list (succ n) (x ∷ xs) eq   f = ap-list n xs omitted-eq (f x)
```

where omitted-eq has type length xs ≡ n when eq has type succ (length xs) ≡ succ n. We omit it to avoid being distracted by explicit reasoning of equality proofs. The slightly unorthodox definition of sum avoids auxiliary lemmas like the associativity of addition. Proving it equal to the standard definition is an easy exercise in pure MLTT, which we omit here. We then state the soundness theorem as follows:

```
nary-sum-sound : (n : Nat) (xs : List Nat)
    (eq : length xs ≡ n) (eq' : length xs ≡ unbox₀ (lift n)) ⟶
    ap-list (unbox₀ (lift n)) xs eq′ (unbox₀ (nary-sum n)) ≡ sum n xs eq
nary-sum-sound zero            []            refl refl = refl -- zero ≡ zero
nary-sum-sound (succ zero)     (x ∷ [])      refl refl = refl -- x ≡ x
nary-sum-sound (succ (succ n)) (x ∷ y ∷ xs) eq   eq′ =
  nary-sum-sound (succ n) ((x + y) ∷ xs) omitted-eq omitted-eq′
```

nary-sum-sound takes two equality proofs to simplify the formulation of this lemma. Only the recursive case is interesting. The proof obligation requires the following type:

```
ap-list (succ (succ (unbox₀ (lift n)))) (x :: y :: xs) eq′ (unbox₀ (nary-sum (succ (succ n))))
 ≡  sum (succ (succ n)) (x :: y :: xs) eq
```

By simplifying the left hand side, we obtain:

```
    ap-list (succ (succ (unbox₀ (lift n)))) (x :: y :: xs) eq′
            (unbox₀ (nary-sum (succ (succ n))))
  = ap-list (unbox₀ (lift n)) xs omitted-eq′
            ((λ x y → unbox₀ (nary-sum (succ n)) (x + y)) x y)
  = ap-list (unbox₀ (lift n)) xs omitted-eq′ ((unbox₀ (nary-sum (succ n))) (x + y))
```

On the other hand, the recursive call gives us:

```
ap-list (succ (unbox₀ (lift n))) ((x + y) :: xs) eq′ (unbox₀ (nary-sum (succ n)))
 ≡  sum (succ n) ((x + y) :: xs) eq
```

By again simplifying the left hand side, we conclude:

```
    ap-list (succ (unbox₀ (lift n))) ((x + y) :: xs) omitted-eq′
            (unbox₀ (nary-sum (succ n)))
  = ap-list (unbox₀ (lift n)) xs omitted-eq′ ((unbox₀ (nary-sum (succ n))) (x + y))
```

This concludes that the types check out and the soundness theorem is established.

3.3 Pattern Matching on Code

As the last goal in Sec. 2, we would like to add the capability of pattern matching to our modal type theory. This will extend the power of a type theory by allowing it to inspect the syntactic structures of programs written in itself. Since this work has not finished, the examples in this section are what we would like to make work and might require further adjustments in the future.

In this hypothetical type theory where we can do pattern matching on code, we must be able to internalize the representation of variables so that we can reason about open code. This is achieved by using contextual types [Nanevski et al. 2008] and a special type `Ctx` which represents the type of contexts. With pattern matching, we are able to inspect the syntactic structure of a piece of code. A ubiquitous example is the `is-app` function, which decides whether a piece of given code is a function application:

```
  is-app : {Ψ :: Ctx} {A :: ⌈ Ψ ⊢ Se ⌉} → ⌈ Ψ ⊢ A ⌉ → Bool
  is-app ⌈ _ ⊢ f x ⌉ = true
  is-app _           = false
```

In this example, a special binder `::` introduces a global variable so that it can be referred to in a contextual type e.g. `⌈ Ψ ⊢ Se ⌉`. The contextual type `⌈ Ψ ⊢ Se ⌉` denotes that `A` is a well-formed type in the context Ψ, so the type of the function argument `⌈ Ψ ⊢ A ⌉` is well-formed. The type `⌈ Ψ ⊢ A ⌉` denotes a well-typed code of type `A` in context Ψ. In the implementation, we inspect the syntactic structure of the argument. In the first case, we expect a function application where `f` and `x` capture the code of the function and the argument respectively, and return `true`. In any other cases, we return `false`.

A more complex example involves recursion on the syntactic structure of code. For example, we can count the number of arguments if a given code is a function application. The result is computed by recursion:

```
  num-args : {Ψ :: Ctx} {A :: ⌈ Ψ ⊢ Se ⌉} → ⌈ Ψ ⊢ A ⌉ → Nat
  num-args ⌈ _ ⊢ f x ⌉ = succ (num-args ⌈ _ ⊢ f ⌉)
  num-args _           = zero
```

If the input code is a function application, we recurse on the code of `f` recursively. The implement arguments can be automatically filled in because effectively we are recursing on a typed abstract syntax tree. Our goal is to develop a type theory in which these examples work while the consistency of the type theory is retained.

4 CURRENT PROGRESS

In this section, we discuss the current progress of the thesis. In particular, we will discuss a published work [Hu and Pientka 2022a] which proposes a way to represent simultaneous substitutions in the Kripke style in a way such that the type theory is much simpler to formulate and to prove normalization. We will also talk about another paper which is currently under preparation for a submission to the Journal of Functional Programming, which extends Hu and Pientka [2022a] to dependent types and also pushes the frontline of mechanzied meta-theory of dependent types in general by showing a full normalization proof with a universe hierarchy and the $\square$ modality.

4.1 Kripke-style Substitutions

In Sec. 1, we discuss the Kripke-style system $\lambda^{\to\square}$ and its advantages of familiar programming style and flexibility to model multiple modal logics. Nevertheless, these advantages are overshadowed by the modal transformation operation. Modal transformations are needed when we consider the dynamics of $\lambda^{\to\square}$. Intuitively, when we try to unbox_n a box, the modal offset n must somehow play a part. Consider the β equivalence rule:

$$\frac{\dfrac{\overrightarrow{\Gamma}; \cdot \vdash t : T}{\overrightarrow{\Gamma} \vdash \mathsf{box}\ t : \square T} \qquad |\overrightarrow{\Delta}| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash \mathsf{unbox}_n\ (\mathsf{box}\ t) \approx ??? : T}$$

As marked in red, the right hand side requires a term of type T in the context stack $\overrightarrow{\Gamma}; \overrightarrow{\Delta}$, but t is only meaningful in the context stack $\overrightarrow{\Gamma}; \cdot$. The modal transformation operation is the proposed solution by Davies and Pfenning [2001]. It shifts all modal offsets in t appropriately to transform a well-typed term in one context stack to another. Here, on the right hand side, we fill in $t\{n/0\}$:

$$\frac{\overrightarrow{\Gamma}; \cdot \vdash t : T \qquad |\overrightarrow{\Delta}| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash \mathsf{unbox}_n\ (\mathsf{box}\ t) \approx t\{n/0\} : T}$$

where $\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash t\{n/0\} : T$ holds. In general, the box and unbox cases of a modal transformation $t\{k/l\}$ is defined as follows:

$$\mathsf{box}\ t\{k/l\} := \mathsf{box}\ (t\{k/l+1\})$$

$$\mathsf{unbox}_n\ t\{k/l\} := \begin{cases} \mathsf{unbox}_n\ (t\{k/l-n\}) & \text{if } n \leq l \\ \mathsf{unbox}_{k+n-1}\ t & \text{if } n > l \end{cases}$$

The other cases simply push $\{k/l\}$ recursively inwards. The definition involves very detailed arithmetic in the unbox case and it is hard to motivate at the first glance. Overall, modal transformations must satisfy the following lemma:

Lemma 4.1. *If* $\overrightarrow{\Gamma}; \Gamma_0; \Delta_0; \cdots; \Delta_l \vdash t : T$, *then* $\overrightarrow{\Gamma}; \Gamma_0; \cdots; (\Gamma_n, \Delta_0); \cdots; \Delta_l \vdash t\{n/l\} : T$.

In particular, by letting $l = 0$, the lemma becomes

Lemma 4.2. *If* $\overrightarrow{\Gamma}; \Gamma_0; \Delta_0 \vdash t : T$, *then* $\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; \cdots; (\Gamma_n, \Delta_0) \vdash t\{n/0\} : T$.

If we further let $n = 0$, then we obtain modal fusion, because Γ_0 and Δ_0 are fused into one context:

Lemma 4.3. *If* $\overrightarrow{\Gamma}; \Gamma_0; \Delta_0 \vdash t : T$, *then* $\overrightarrow{\Gamma}; (\Gamma_0, \Delta_0) \vdash t\{n/0\} : T$.

Otherwise, the modal transformation $t\{n/0\}$ modal weakens the context stack by $n-1$ new contexts. A detailed analysis on the modal transformation operation can be found in Davies and Pfenning [2001]; Hu and Pientka [2022a], but the overall observation is that the modal transformation operation is challenging to implement and reason about due to the arithmetic and branching. This disadvantage makes Kripke-style systems much harder to work with than the dual-context style.

To avoid detailed analysis of modal transformations, Hu and Pientka [2022a] propose a solution which side-steps this problem and introduces the concept of Kripke-style substitutions or just K-substitutions. K-substitutions are a unifying concept of modal transformations and substitutions. Both modal transformations and substitutions are just special cases of K-substitutions. The deciding feature of K-substitutions is that they constitute a substitution calculus for Kripke-style systems. That is, K-substitutions have identity and composition:

$$\overrightarrow{\text{id}} : \vec{\Gamma} \Rightarrow \vec{\Gamma} \qquad \text{(identity)}$$

$$\_ \circ \_ : \vec{\Gamma}_1 \Rightarrow \vec{\Gamma}_2 \to \vec{\Gamma}_0 \Rightarrow \vec{\Gamma}_1 \to \vec{\Gamma}_0 \Rightarrow \vec{\Gamma}_2 \qquad \text{(composition)}$$

and they satisfies the usual laws of identity and associativity. This allows us to extend usual methods for studying simply typed λ-calculus (STLC) to understand $\lambda^{\to\square}$.

In our recent work and our technical report [Hu and Pientka 2022a,b], we minimally extend the typical presheaf model [Altenkirch et al. 1995] and an untyped domain model [Abel 2013] for STLC with Kripke structures. As a result, we obtain one normalization proof for $\lambda^{\to\square}$ for each model. For comparison, in Davies and Pfenning [2001], the consistency of $\lambda^{\to\square}$ is proved as a consequence of reduction to the dual-context style. There is a recent work [Valliappan et al. 2022] that proves a similar result but for the Fitch style. In Valliappan et al. [2022], one formulation is given for each subsystem of $S4$ and the normalization for each formulation is given separately. In contrast, in our methods, we take advantage of the commonalities of all Systems K, T, $K4$ and $S4$ and give a single normalization proof for all (and there are two such proofs).

These shared commonalities scale very well even to dependent types and we summarize them as a group of algebras called truncoid, which we will discuss very shortly.

4.2 Normalization and Explicit Substitutions

As a first step of this research, we first consider the simply typed system $\lambda^{\to\square}$ and develop two normalization proofs for it. These two proofs fall in the same category of normalization by evaluation (NbE) [Berger and Schwichtenberg 1991; Martin-Löf 1975]. There are two steps in an NbE proof: first, we evaluate a program in some mathematical domain; then we extract normal forms back from this domain. This approach has the advantage of being simple, because it does not require detailed reasoning of conversions of terms. The correctness of an NbE algorithm is composed of two theorems:

Theorem 4.4 (Completeness). *If $\vec{\Gamma} \vdash t \approx t' : T$, then $nbe^{T}_{\vec{\Gamma}}(t) = nbe^{T}_{\vec{\Gamma}}(t')$.*

Theorem 4.5 (Soundness). *If $\vec{\Gamma} \vdash t : T$, then $\vec{\Gamma} \vdash t \approx nbe^{T}_{\vec{\Gamma}}(t) : T$.*

where $\mathsf{nbe}^{T}_{\vec{\Gamma}}(t)$ denotes running the NbE algorithm on term t in context stack $\vec{\Gamma}$ with its type T. The completeness theorem states that two syntactically equivalent terms have equal normal forms. The soundness theorem states that a well-typed term is equivalent to its normal form. These theorems immediately imply that we can test whether two programs are equivalent by testing whether they have identical normal forms, which we obtain from the NbE algorithm. Normalization not only implies the consistency of a system in general but also is an essential building block for a typechecker.

One of our normalization proofs is based on the presheaf categories. Presheaf and category are concepts in the mathematical field of category theory, which is a generalization of algebraic theories that capture common characteristics of mathematical phenomena. As implied by its name, category theory studies categories. In computer science, category theory functions as a conceptual library and an "out-of-pocket" mathematical language for quick but organized inspirations. The presheaf categories are one particularly useful tool because they have many convenient structures to exploit. We take advantages of these structures and develop a simple normalization proof based on this model.

However, the use of presheaf categories is not continued in the dependently typed case because we find that constructing a presheaf category is way too complex in a proof assistant and prevents us from efficiently mechanizing this work. Fortunately, our second proof which uses an untyped domain model is not only easy to understand but also very suitable for mechanization. Later investigations also show that untyped domains scale very well to dependent types. To use an untyped domain model, we must first reformulate $\lambda^{\to\square}$ into a version with explicit substitutions. Having explicit substitutions benefits us on both hands: on the theoretical side, explicit substitutions give an axiomatization for K-substitutions. Mathematically, this axiomatization directly corresponds to a formulation of K-substitutions in category theory and opens doors for subsequent research from a categorical angle. On the practical side, explicit substitutions give a way to improve the implementation of $\lambda^{\to\square}$, by deferring the action of substitutions until it is actually needed, hence making the implementation more performant. The syntax of explicit substituions is:

$$\vec{\sigma}, \vec{\delta} := \vec{I} \mid \vec{\sigma}, t \mid \mathsf{wk} \mid \vec{\sigma}; \Uparrow^n \mid \vec{\sigma} \circ \vec{\delta} \qquad \text{(Unified substitutions)}$$

where $\vec{I}$ is the identity; $\vec{\sigma}, t$ extends a unified substituion $\vec{\sigma}$ with a term t; wk is weakening of the topmost variable; $\vec{\sigma} \circ \vec{\delta}$ is composition. At last, $\vec{\sigma}; \Uparrow^n$ is a modal extension and is unique in Kripke-style systems. It remembers a modal offset which is necessarily introduced by the β equivalence of $\square$. The following gives its typing rule:

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \qquad |\vec{\Gamma}'| = n}{\vec{\Gamma}; \vec{\Gamma}' \vdash \vec{\sigma}; \Uparrow^n : \vec{\Delta}; \cdot}$$

Instead of performing a modal transformation, eliminating a $\square$ now reduces to a substitution with a modal extension:

$$\frac{\vec{\Gamma}; \cdot \vdash t : T \qquad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \mathsf{unbox}_n\ (\mathsf{box}\ t) \approx t[\vec{I}; \Uparrow^n] : T}$$

The resulting system no longer requires detailed arithmetic as shown in Sec. 1 and becomes much easier to reason about. The formulation of explicit substitutions is very robust and scales to Mint, a Kripke-style Martin-Löf type theory. However, only an explicit substitution calculus is not enough to work with the Kripke structure in $\lambda^{\to\square}$. In this substitution calculus, we can modally extend a K-substitution but we have not seen what is the inverse operation, namely, can we truncate a K-substitution? The answer to this question is the algebra of truncoids.

4.3 Truncoid, An Algebra of Truncation

After modally extending a K-substitution, we sometimes want to perform the inverse operation. For example, let $\vec{\sigma}$ be $(\vec{\delta}; \Uparrow^1, t; \Uparrow^2, s)$ where $\vec{\Gamma} \vdash \vec{\delta} : \vec{\Delta}$ and $\vec{\Gamma}; \Gamma_0; \Gamma_1; \Gamma_2 \vdash \vec{\sigma} : \vec{\Delta}; \cdot.T; \cdot.S$. The latter typing judgment can be obtained by using the typing rule for modal extensions above. We want to recover $\vec{\delta}$ from $\vec{\sigma}$. The truncation $(\_ \mid \_)$ operation does precisely that. It truncates a substitution by a number, which denotes the number of modal extensions to drop. In this example, truncation $\overrightarrow{\sigma} \mid 2$ returns $\overrightarrow{\delta}$ after dropping everything up to two modal extensions $\Uparrow^1$ and $\Uparrow^2$. Since we are doing truncation by 2, it is intuitive to drop two contexts from the codomain context stack of $\overrightarrow{\sigma}$, which agrees with $\overrightarrow{\Delta}$, the codomain context stack of $\overrightarrow{\delta}$. But how do we get $\overrightarrow{\Gamma}$ from $\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; \Gamma_2$? We must drop three contexts. It turns out that we must drop as many contexts as the sum of the first two modal offsets, which is $2 + 1 = 3$. Therefore, we need another operation, the truncation offset operation ($O(\_, \_)$), to compute the sum of leading modal offsets in a K-substitution. In general, we want $\overrightarrow{\Gamma} \mid O(\overrightarrow{\sigma}, n) \vdash \overrightarrow{\sigma} \mid n : \overrightarrow{\Delta} \mid n$. Explicit substitutions allows us to give definitions of truncation and truncation offsets by doing recursion on the structure of inputs:

$$\begin{aligned}
\overrightarrow{\sigma} \mid 0 &:= \overrightarrow{\sigma} & O(\overrightarrow{\sigma}, 0) &:= 0\\
\overrightarrow{I} \mid 1 + n &:= \overrightarrow{I} & O(\overrightarrow{I}, 1 + n) &:= 1 + n\\
(\overrightarrow{\sigma}, t) \mid 1 + n &:= \overrightarrow{\sigma} \mid 1 + n & O((\overrightarrow{\sigma}, t), 1 + n) &:= O(\overrightarrow{\sigma}, 1 + n)\\
\mathsf{wk} \mid 1 + n &:= \overrightarrow{I} & O(\mathsf{wk}, 1 + n) &:= 1 + n\\
(\overrightarrow{\sigma}; \Uparrow^m) \mid 1 + n &:= \overrightarrow{\sigma} \mid n & O(\overrightarrow{\sigma}; \Uparrow^m, 1 + n) &:= m + O(\overrightarrow{\sigma}, n)\\
(\overrightarrow{\sigma} \circ \overrightarrow{\delta}) \mid 1 + n &:= (\overrightarrow{\sigma} \mid 1 + n) \circ (\overrightarrow{\delta} \mid O(\overrightarrow{\sigma}, 1 + n)) & O(\overrightarrow{\sigma} \circ \overrightarrow{\delta}, 1 + n) &:= O(\overrightarrow{\delta}, O(\overrightarrow{\sigma}, 1 + n))
\end{aligned}$$

In addition, truncation and truncation offset satisfy the following coherence conditions:

Lemma 4.6 (Coherence conditions).

- *Coherence of addition: for all $\overrightarrow{\sigma}$, m and n, $\overrightarrow{\sigma} \mid (n + m) = (\overrightarrow{\sigma} \mid n) \mid m$ and $O(\overrightarrow{\sigma}, n + m) = O(\overrightarrow{\sigma}, n) + O(\overrightarrow{\sigma} \mid n, m)$.*
- *Coherence of composition: for all $\overrightarrow{\sigma}$, $\overrightarrow{\delta}$ and m, $(\overrightarrow{\sigma} \circ \overrightarrow{\delta}) \mid n = (\overrightarrow{\sigma} \mid n) \circ (\overrightarrow{\delta} \mid O(\overrightarrow{\sigma}, n))$ and $O(\overrightarrow{\sigma} \circ \overrightarrow{\delta}, n) = O(\overrightarrow{\delta}, O(\overrightarrow{\sigma}, n))$.*

Truncation, truncation offset and their coherence is a recurring theme in both syntax and semantics, so it is worth it to give a more concrete algebraic characterization, which we call a truncoid:

Definition 4.7. A truncoid is a triple $(S, \_ \mid \_, O(\_, \_))$, where

- S is a set;
- the truncation operation $\_ \mid \_$ takes an S and a natural number and returns an S;
- the truncation offset operation $O(\_, \_)$ takes an S and a natural number and returns a natural number.

where the coherence of addition hold:

$$s \mid (n + m) = (s \mid n) \mid m \text{ and } O(s, n + m) = O(s, n) + O(s \mid n, m)$$

Following the common mathematical practice, we directly call S a truncoid if it has coherent truncation and truncation offset. We have already shown that K-substitutions are a truncoid. Truncoids also play important parts in the semantics. Essentially, the normalization proof is just a study of interactions among different truncoids. In fact, most truncoids we encounter in the normalization proof are more specific. One important kind is applicative truncoids, which allows a truncoid to be applied to another:

Definition 4.8. An applicative truncoid consists of a triple of truncoids (S_0, S_1, S_2) and an additional apply operation $\_[\_]$ which takes S_0 and S_1 and returns S_2. Moreover, the apply operation satisfies an extra coherence condition:

$$s[s'] \mid n = (s \mid n)[s' \mid O(s, n)] \text{ and } O(s[s'], n) = O(s', O(s, n))$$

Most semantic models that we depend on in the normalization proof are also applicative. For example, evaluation environments in the semantics is applicative. K-substitutions are also applicative, as we can just let the apply operation be composition. Following this intuition, we define a specialized applicative truncoid as a substitutional truncoid by asking the apply operation to behave like composition:

Definition 4.9. A substitutional truncoid S is an applicative truncoid (S, S, S) with an identity $\mathsf{id} \in S$. Note that the apply operation is essentially composition, which we just write as $\_ \circ \_$. The extra coherence conditions are for identity:

$$\mathsf{id} \mid n = \mathsf{id}, \mathcal{O}(\mathsf{id}, n) = n, \mathsf{id} \circ s = s \text{ and } s \circ \mathsf{id} = s$$

and associativity:

$$(s_0 \circ s_1) \circ s_2 = s_0 \circ (s_1 \circ s_2)$$

As expected, K-substitutions are a substitutional truncoid. The untyped modal transformations (UMoTs) which model the Kripke structure in the semantics are also substitutional. There is another way to specialize applicative truncoid. Starting from an applicative truncoid, if we allow S_1 to be substitutional, then obtain a closed truncoid:

Definition 4.10. A closed truncoid (S_0, S_1) is an applicative truncoid triple (S_0, S_1, S_0) and S_1 is a substitutional truncoid. We write id and $\_ \circ \_$ for identity and composition of S_1. The following additional conherence conditions are required:

- coherence of identity: $s[\mathsf{id}] = s$.
- coherence of composition: $s[s_1 \circ s_2] = s[s_1][s_2]$

We say that S_0 is closed under S_1. The apply operation of a closed truncoid must return S_0 as required by the definition. In the semantics, the evaluation environments are closed under UMoTs. Notice that K-substitutions are also closed under themselves. The laws of applicative and closed truncoids cover all the properties we need to reason about the normalization process. There are also non-closed applicative truncoids. The evaluation operation of K-substitutions to the semantics is one such example. In our investigation, these definitions not only help us to quickly identify the unique definitions of certain operations, but also provide a modular layer to our proof. Effectively, our proof is parameterized by a group of truncoids and each subsystem of $S4$ can be instantiated to its own group. This is the secret why our normalization proof is modular and applies to all four systems without change.

4.4 Mint: Scaling to Dependent Types

With the explicit substitution calculus given previously, we have succeeded in developing Mint, a **M**odal **IN**tuitionistic **T**ype theory. Mint is a Martin-Löf type theory with cumulative Russell's style universes and supports inductive types and large eliminations. Under Curry-Howard correspondence, Mint can be used to model dependently typed meta-programming, among other applications like providing an alternative type theory for studying universes in homotopy type theory. More formally, in Mint, $\square$ is characterized by the following rules:

$$\frac{\vec{\Gamma}; \cdot \vdash T : \mathsf{Se}_i}{\vec{\Gamma} \vdash \square T : \mathsf{Se}_i} \qquad \frac{\vec{\Gamma}; \cdot \vdash t : T}{\vec{\Gamma} \vdash \mathsf{box}\ t : \square T} \qquad \frac{\vec{\Gamma} \vdash t : \square T \qquad \vdash \vec{\Gamma}; \vec{\Delta} \qquad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \mathsf{unbox}_n\ t : T[\vec{I}; \Uparrow^n]}$$

Since Mint is dependently typed, we need a type formation rule to specify the well-formedness of $\square T$. The introduction rule is unchanged. The elimination rule is slightly more complex. $\vdash \vec{\Gamma}; \vec{\Delta}$ denotes the well-formedness of the context stack $\vec{\Gamma}; \vec{\Delta}$ and it is required for the syntactic validity. Due to dependent types, the result type becomes $T[\overrightarrow{I}\,;\Uparrow^n]$ to move T from $\overrightarrow{\Gamma};\cdot$ to $\overrightarrow{\Gamma};\overrightarrow{\Delta}$. The β and η equivalence rules are:

$$\frac{\overrightarrow{\Gamma};\cdot \vdash t : T \qquad \vdash \overrightarrow{\Gamma};\overrightarrow{\Delta} \qquad |\overrightarrow{\Delta}| = n}{\overrightarrow{\Gamma};\overrightarrow{\Delta} \vdash \mathsf{unbox}_n\ (\mathsf{box}\ t) \approx t[\overrightarrow{I}\,;\Uparrow^n] : T[\overrightarrow{I}\,;\Uparrow^n]} \qquad \frac{\overrightarrow{\Gamma} \vdash t : \square T}{\overrightarrow{\Gamma} \vdash t \approx \mathsf{box}\ (\mathsf{unbox}_1\ t) : \square T}$$

We also scale our normalization proof based on an untyped domain to dependent types and the proof is mechanized in Agda. The normalization proof is just a moderate extension of Abel [2013]. The proof indicates that the concept of K-substitutions as well as various formulations of truncoids also scale to dependent types. Moreover, the mechanization in itself is useful and novel because it leads to a formulation of semantics that is more suitable for mechanization in type theory and reveals certain inaccuracies in on-paper proofs in some previous work. In particular, previous work [Abel 2013; Abel et al. 2017; Gratzer et al. 2019] seems to oversimplify the proof of cumulativity of the semantic models. Cumulativity is a property of universes, with which a type in a lower universe also lives in all higher universes. In the semantic models, we must show the models are also cumulative, i.e. predicates hold on a lower level also hold on all higher ones. It turns out that in order to establish cumulativity of semantic models, a lowering property must be mutually proved, which state that under some condition, we can lower a predicate hold on a higher level back down to some lower level. Lowering is technically required to establish cumulativity but seems glossed over in previous work. In our mechanization, we also adjust our models so that they no longer fundamentally depend on cumulativity as in Abel [2013]; Abel et al. [2017]; Gratzer et al. [2019]. This potentially allows us to give an NbE proof for non-cumulative universes, which is not seen in the literature to date.

4.5 Contextual Types with Context Stacks

Another development becoming available after developing the notion of K-substitutions is contextual types. Since the introduction rule of $\square$ (no matter in which style) wipes off the current context (c.f. Sec. 1), $\square$ inherently only models closed code in meta-programming under Curry-Howard correspondence. Contextual types [Nanevski et al. 2008] is originally developed for the dual-context style to model meta-programming with open code. However, Nanevski et al. [2008] does not give a formulation in the Kripke style. One important reason is that the substitution calculus for the Kripke style was not clear at that time. With K-substitutions, contextual types for the Kripke style become possible.

$$S, T := \cdots \mid \lceil \overrightarrow{\Delta} \vdash T \rceil \qquad\qquad s, t, u := \cdots \mid \lceil \overrightarrow{\Delta} \vdash t \rceil \mid \lfloor t \rfloor_{\overrightarrow{\sigma}}$$

$\lceil \overrightarrow{\Delta} \vdash T \rceil$ is a contextual type, denoting a code of type T open w.r.t. a context stack $\overrightarrow{\Delta}$ (which can be empty). The introduction rule for contextual types is a straightforward extension of $\square$:

$$\frac{\overrightarrow{\Gamma};\overrightarrow{\Delta} \vdash t : T}{\overrightarrow{\Gamma} \vdash \lceil \overrightarrow{\Delta} \vdash t \rceil : \lceil \overrightarrow{\Delta} \vdash T \rceil}$$

If we let $\overrightarrow{\Delta} = \epsilon;\cdot$, then we recover $\square$. If we let $\overrightarrow{\Delta} = \epsilon;\Delta$ for some Δ, then we have an open term t which uses only assumptions in the same stage. If $\overrightarrow{\Delta}$ has more contexts, then t is an open term which uses assumptions from previous stages. We can also let $\overrightarrow{\Delta} = \epsilon$. In this case, $\lceil \epsilon \vdash T \rceil$ is isomorphic to T and is not too meaningful but allowing so makes our formulation mathematically cleaner.

The elimination rule, on the other hand, becomes significantly more complex:

$$\frac{\overrightarrow{\Gamma} \mid O(\overrightarrow{\sigma}) \vdash t : \lceil \overrightarrow{\Delta} \vdash T \rceil \qquad \overrightarrow{\sigma} : \overrightarrow{\Gamma} \Rightarrow_s \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \lfloor t \rfloor_{\overrightarrow{\sigma}} : T}$$

It is no longer enough to eliminate with just an unbox level because the eliminator must specify how to replace all variables in $\overrightarrow{\Delta}$ and how contexts in $\overrightarrow{\Gamma}$ and $\overrightarrow{\Delta}$ relate. This information is collectively stored in a semi-K-substitution $\overrightarrow{\sigma}$ (notice the semi-arrow). The modal offset sum ($\mathcal{O}(\_)$) computes the sum of all modal offsets in a semi-K-substitution. The full discussion on the definitions and the machinery behind is too verbose and technical, so we omit the rest of the details here. Readers can refer to Hu and Pientka [2022a] for the complete development.

5 FUTURE DIRECTIONS AND TIMELINE

In the previous section, we discussed some finished work on Kripke-style modal type theories. However, for an application to meta-programming, all these type theories have a critical but missing feature: pattern matching on code. Under Curry-Howard correspondence, box is a constructor for code. With a code representation, we should be able to analyze the code by its syntactic structure, i.e. pattern matching on code. For example, we should be able to pattern match on the code of the current goal and depending on the goal, we algorithmically compute a proof that satisfies this goal. Despite a significant amount of time invested in the Kripke style, we realize that it is challenging to achieve this goal in the Kripke style, for two reasons:

(1) Philosophically, the Kripke style employs a "projection" style for elimination, while pattern matching is a "destruction" style of elimination which is not always obviously compatible with projections.
(2) The Kripke style has been proven difficult to extend historically, primarily because its context stack structure is too complex to make compatible with new features. Though K-substitutions have simplified the problem, it is still not clear how to easily extend the Kripke style in general.

Hence, in the remainder of this thesis, we would look into possibilities of adding pattern matching to the dual-context style. The dual-context style is much easier to work with than the Kripke style from a historical point of view. Moreover, extending the dual-context style seems more viable than continuing with the Kripke style, as Jang et al. [2022] has shown that pattern matching on code is possible to achieve in System F. Nevertheless, it is still not immediate how to extend their method to Martin-Löf style dependent types, because impredicativity of System F seems to play a crucial part, while Martin-Löf type theories are predicative. Moreover, we want to not only have pattern matching, but also do recursion in arms of pattern matching. See a desirable example in Sec. 3.3.

The investigation of eventually supporting dependently typed meta-programming is planned to carry out in a similar way to the investigation of the Kripke-style systems:

(1) [3 months] First, we consider the simply typed case. We work based on the contextual modal type theory by Nanevski et al. [2008]. We can try to backport Jang et al. [2022] to simple types so that we have a base system to extend. In this step, our goal is to develop a substitution calculus and obtain a semantic model for pattern matching on code in its simplest form, so that the difficulties of introducing these features are exposed, understood and conquered. The final result of this step is a normalization algorithm[1] and its correctness theorems. These results will directly help our extension to dependent types.
(2) [4 months] With the results from the previous step, we work on the extension to Martin-Löf type theory. If our substitution calculus is developed properly in the previous step, it should be carried over to dependent types easily as the case in the Kripke style. The final result of this phase is a type theory which does dependently typed meta-programming with support of pattern matching on code, its normalization algorithm and the correctness theorems. This step achieves the last goal set in Sec. 2 for this thesis.
(3) [1 month] After developing the type theory in the previous step, we would like to know how practical this type theory is. In this step, we will perform some case analysis. Primarily, we write meta-programs like

[1]Note that though we focus on NbE for the Kripke-style systems, we are still open to conversion-based methods, which in general have more control over terms than NbE.

those in Sec. 3.3 and analyze their behaviors during execution. We compare meta-programs in this type theory and meta-programs in various systems [Anand et al. 2018; Delahaye 2000; Kaiser et al. 2018; Pédrot 2019; Ziliani et al. 2013, 2015] and understand their pros and cons.

(4) As an open-ended direction for more research in the future, we implement the type theory developed in the previous step. In this step, more engineering factors are likely to surface. Novel implementation techniques might be required to implement a dependently typed language that can analyze its own syntactic structures. This work is left as future work.

At last, the time for writing thesis is approximately 4 months.

6 CONCLUSIONS

In this proposal, we surveyed the broad research area of modal type theories and a few important modal systems that serve as base systems in our research. Then we discussed our current progress on the Kripke-style systems. In particular, we developed a substitution calculus for $\lambda^{\to\square}$ and proved its normalization with two different proofs. Our proofs are modular so that they proved the normalization of all four subsystems of $S4$ simultaneously. We extended our results in the simply typed case to dependent types. We developed Mint, combining the $\square$ modality and Martin-Löf type theory and proved its normalization. The normalization proof has been fully mechanized in Agda. At last, we discussed our plan for the remainder of this PhD. We planned to switch to the dual-context style to search for a combination of dependent types and pattern matching on code as a focused application of modal type theories.

REFERENCES

Andreas Abel. 2013. Normalization by evaluation: dependent types and impredicativity. Habilitation thesis. Ludwig-Maximilians-Universität München.

Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. Proceedings of the ACM on Programming Languages 4, ICFP (Aug. 2020), 90:1–90:28. https://doi.org/10.1145/3408972

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of conversion for type theory in type theory. Proceedings of the ACM on Programming Languages 2, POPL (Dec. 2017), 23:1–23:29. https://doi.org/10.1145/3158111

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In Category Theory and Computer Science, David Pitt, David E. Rydeheard, and Peter Johnstone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 182–199.

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In 9th International Conference Interactive Theorem Proving (ITP'18) (Lecture Notes in Computer Science (LNCS 10895)). Springer, 20–39. https://doi.org/10.1007/978-3-319-94821-8_2

Gianluigi Bellin, Valeria C. V. de Paiva, and Eike Ritter. 2001. Extended Curry-Howard Correspondence for a Basic Constructive Modal Logic. In In Proceedings of Methods for Modalities.

Ulrich Berger and Helmut Schwichtenberg. 1991. An inverse of the evaluation functional for typed lambda-calculus. In [1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science. 203–211. https://doi.org/10.1109/LICS.1991.151645

Gavin M. Bierman and Valeria C. V. de Paiva. 1996. Intuitionistic Necessity Revisited. Technical Report. University of Birmingham.

Gavin M. Bierman and Valeria C. V. de Paiva. 2000. On an Intuitionistic Modal Logic. Studia Logica 65, 3 (Aug. 2000), 383–416. https://doi.org/10.1023/A:1005291931660

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2020. Modal dependent type theory and dependent right adjoints. Mathematical Structures in Computer Science 30, 2 (Feb. 2020), 118–138. https://doi.org/10.1017/S0960129519000197 Publisher: Cambridge University Press.

V. A. J. Borghuis. 1994. Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus. PhD Thesis. Mathematics and Computer Science. https://doi.org/10.6100/IR427575

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science), Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 258–275. https://doi.org/10.1007/978-3-319-89366-2_14

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In Foundations of Software Science and Computation Structures, Andrew Pitts (Ed.). Vol. 9034. Springer Berlin

Heidelberg, Berlin, Heidelberg, 407–421. https://doi.org/10.1007/978-3-662-46678-0_26 Series Title: Lecture Notes in Computer Science.

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. J. ACM 48, 3 (May 2001), 555–604. https://doi.org/10.1145/382780.382785

David Delahaye. 2000. A Tactic Language for the System Coq. In Logic for Programming and Automated Reasoning (Lecture Notes in Artificial Intelligence), Michel Parigot and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 85–95. https://doi.org/10.1007/3-540-44404-1_7

Daniel Gratzer. 2022. Normalization for Multimodal Type Theory. In Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '22). Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3531130.3532398

Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20). Association for Computing Machinery, New York, NY, USA, 492–506. https://doi.org/10.1145/3373718.3394736

Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2021. Multimodal Dependent Type Theory. Log. Methods Comput. Sci. 17, 3 (2021). https://doi.org/10.46298/lmcs-17(3:11)2021

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a modal dependent type theory. Proceedings of the ACM on Programming Languages 3, ICFP (July 2019), 107:1–107:29. https://doi.org/10.1145/3341711

Jason Z. S. Hu and Brigitte Pientka. 2022a. A Categorical Normalization Proof for the Modal Lambda-Calculus. In Proceedings 38th Conference on Mathematical Foundations of Programming Semantics, MFPS 2022 (EPTCS).

Jason Z. S. Hu and Brigitte Pientka. 2022b. An Investigation of Kripke-style Modal Type Theories. CoRR abs/2206.07823 (2022). https://doi.org/10.48550/arXiv.2206.07823 arXiv:2206.07823

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Moebius: Metaprogramming using Contextual Types – The stage where System F can pattern match on itself. Proc. ACM Program. Lang. (PACMPL) POPL (2022).

Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012, Koen Claessen and Nikhil Swamy (Eds.). ACM, 49–60. https://doi.org/10.1145/2103776.2103783

Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: typed tactics for backward reasoning in Coq. Proceedings of the ACM on Programming Languages 2, ICFP (July 2018), 78:1–78:31. https://doi.org/10.1145/3236773

G. A. Kavvos. 2017. Dual-context calculi for modal logic. In 2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). 1–12. https://doi.org/10.1109/LICS.2017.8005089

Saul A. Kripke. 1963. Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi. Mathematical Logic Quarterly 9, 5-6 (1963), 67–96. https://doi.org/10.1002/malq.19630090502 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19630090502.

Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In 3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 108), Hélène Kirchner (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.22 ISSN: 1868-8969.

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In Studies in Logic and the Foundations of Mathematics, H. E. Rose and J. C. Shepherdson (Eds.). Logic Colloquium '73, Vol. 80. Elsevier, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

Per Martin-Löf. 1984. Intuitionistic type theory. Studies in proof theory, Vol. 1. Bibliopolis.

Simone Martini and Andrea Masini. 1996. A Computational Interpretation of Modal Proofs. In Proof Theory of Modal Logic, Heinrich Wansing (Ed.). Springer Netherlands, Dordrecht, 213–241. https://doi.org/10.1007/978-94-017-2798-3_12

Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. 2004. A Symmetric Modal Lambda Calculus for Distributed Computing. In 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings. IEEE Computer Society, 286–295. https://doi.org/10.1109/LICS.2004.1319623

Hiroshi Nakano. 2000. A modality for recursion. In Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332). 255–266. https://doi.org/10.1109/LICS.2000.855774 ISSN: 1043-6871.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. ACM Transactions on Computational Logic 9, 3 (June 2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

Erik Palmgren. 1998. On universes in type theory. In Twenty Five Years of Constructive Type Theory. Oxford University Press. https://doi.org/10.1093/oso/9780198501275.003.0012

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science 11, 04 (Aug. 2001). https://doi.org/10.1017/S0960129501003322

Frank Pfenning and Hao-Chi Wong. 1995. On a modal lambda calculus for S4. In Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995, Tulane University, New Orleans, LA, USA, March 29 - April 1, 1995 (Electronic Notes in Theoretical Computer Science, Vol. 1), Stephen D. Brookes, Michael G. Main, Austin Melton, and Michael W. Mislove (Eds.). Elsevier, 515–534. https://doi.org/10.1016/S1571-0661(04)00028-3

Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rebecca Zucchini. 2019. A Type Theory for Defining Logics and Proofs. In 34th IEEE/ ACM Symposium on Logic in Computer Science (LICS'19). IEEE Computer Society, 1–13.

Dag Prawitz. 1965. Natural Deduction: A Proof-theoretical Study. Stockholm.

Pierre-Marie Pédrot. 2019. Ltac2: tactical warfare. In The Fifth International Workshop on Coq for Programming Languages, CoqPL. 13–19.

Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010, Paul Hudak and Stephanie Weirich (Eds.). ACM, 157–168. https://doi.org/10.1145/1863543.1863568

Michael Shulman. 2018. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. Mathematical Structures in Computer Science 28, 6 (2018), 856–941. https://doi.org/10.1017/S0960129517000147 Publisher: Cambridge University Press.

Nachiappan Valliappan, Fabian Ruch, and Carlos Tomé Cortiñas. 2022. Normalization for fitch-style modal calculi. Proc. ACM Program. Lang. 6, ICFP (2022), 772–798. https://doi.org/10.1145/3547649

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: a monad for typed tactic programming in Coq. In ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 87–100. https://doi.org/10.1145/2500365.2500579

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A monad for typed tactic programming in Coq. J. Funct. Program. 25 (2015). https://doi.org/10.1017/S0956796815000118

Nikita Zyuzin and Aleksandar Nanevski. 2021. Contextual modal types for algebraic effects and handlers. Proc. ACM Program. Lang. 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473580