JFP **33**, e7, 51 pages, 2023. © The Author(s), 2023. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S0956796823000060

Normalization by evaluation for modal dependent type theory

JASON Z. S. HU

School of Computer Science, McGill University, Montréal, Canada
(*e-mail:* zhong.s.hu@mail.mcgill.ca)

JUNYOUNG JANG

School of Computer Science, McGill University, Montréal, Canada
(*e-mail:* junyoung.jang@mail.mcgill.ca)

BRIGITTE PIENTKA

School of Computer Science, McGill University, Montréal, Canada
(*e-mail:* bpientka@cs.mcgill.ca)

Abstract

We present the Kripke-style modal type theory, MINT, which combines dependent types and the necessity modality. It extends the Kripke-style modal lambda-calculus by Pfenning and Davies to the full Martin-Löf type theory. As such it encompasses dependently typed variants of system K, T, $K4$, and $S4$. Further, MINT seamlessly supports a full universe hierarchy, usual inductive types, and large eliminations. In this paper, we give a modular sound and complete normalization-by-evaluation (NbE) proof for MINT based on an untyped domain model, which applies to all four aforementioned modal systems without modification. This NbE proof yields a normalization algorithm for MINT, which can be directly implemented. To further strengthen our results, our models and the NbE proof are fully mechanized in Agda and we extract a Haskell implementation of our NbE algorithm from it.

1 Introduction

Over the past two decades, modal logic's notion of necessity and possibility has provided precise characterizations for a wide range of computational phenomena: from reasoning about different stages of computation (Davies & Pfenning, 2001; Jang *et al.*, 2022) and meta-programming (Schürmann *et al.*, 2001; Pientka *et al.*, 2019) to homotopy type theory (Licata *et al.*, 2018; Shulman, 2018) and guarded recursions (Nakano, 2000; Clouston *et al.*, 2015). One might say that these applications bear witness to the unusual effectiveness of modalities in programming languages and logic.

To support these applications in various areas, the foundational study of the necessity modality (□) has started in the early 1990s. The study was primarily driven by the computational interpretation of modal logics from a proof-theoretic point of view. In his PhD thesis, Borghuis (1994) gives a formulation of pure type system with a □ modality.

Similarly, Pfenning & Wong (1995) and Davies & Pfenning (2001) propose an intuitionistic formulation of the all sublogics of the modal logic $S4$. Martini & Masini (1996) explore reading modal proofs from various modal logics as programs. All the prior work mentioned uses a *context stack* structure to organize modal assumptions in order to incorporate the $\square$ modality.

In this work, we contribute to the foundational landscape of modal type theories using context stacks by investigating MINT, a **M**odal **IN**tuitionistic **T**ype theory, which directly extends the simply typed Kripke-style modal lambda-calculus by Davies & Pfenning (2001) to the dependently typed setting. In particular, MINT adds to the usual Martin-Löf type theory (MLTT) the necessity modality ($\square$) and supports a full hierarchy of cumulative universes, inductive types, and large eliminations. MINT captures dependently typed variants of modal Systems K, T,[1] $K4$, and $S4$, complementing previous work by Gratzer *et al*. (2019), which only supports idempotent $S4$.

The Kripke style and context stacks. Following Davies & Pfenning (2001), MINT uses a context stack where each context corresponds to a Kripke world to model the Kripke semantics (Kripke, 1963). A term t is then typed in a context stack $\vec{\Gamma}$, which at the beginning is a singleton with an empty context (i.e. $\varepsilon; \cdot$).

$$\varepsilon; \Gamma_1; \ldots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \vec{\Gamma} \vdash t : T$$

where the topmost context denotes the current world. Subsequently, we use "context" and "world" interchangeably.

In the simply typed setting, the rules for $\square$ introduction and elimination are as follows:

$$\frac{\vec{\Gamma}; \cdot \vdash t : T}{\vec{\Gamma} \vdash \texttt{box}\ t : \square T} \qquad \frac{\vec{\Gamma} \vdash t : \square T}{\vec{\Gamma}; \Delta_1; \ldots; \Delta_n \vdash \texttt{unbox}_n\ t : T}$$

In the $\square$ introduction rule, we enter a new world by appending an empty context to the context stack. The $\square$ elimination rule allows us to use the fact that $\square T$ is true. In particular, if $\square T$ holds in a context stack $\vec{\Gamma}$, then we can use T in any world that is accessible from it. Which previous world can be reached is controlled by the unbox level n, which we call *modal offset*. Modal offsets eliminate the need of explicit structural rules to manage the context stack structure as done by Borghuis (1994) and Pfenning & Wong (1995). Following Davies & Pfenning (2001), we refer to the systems with context stacks and unbox for elimination as *the Kripke-style systems*.

There are two reasons why we are investigating a modal dependent type theory in the Kripke style:

- Firstly, the Kripke style, as indicated by its name and our previous introduction, corresponds to the Kripke semantics directly. In other words, the Kripke-style systems provide syntactic theories for the corresponding Kripke semantics. Moreover, in the Kripke style, we can elegantly and uniformly capture various modal systems such as K, T, $K4$, and $S4$ simply by restricting modal offsets (Davies & Pfenning, 2001; Hu & Pientka, 2022*a*). Modal offsets enable us to study properties such as

[1] One should not confuse the modal system T with the unrelated Gödel's system T.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

normalization uniformly for all subsystems of $S4$ and give rise to a fresh perspective of corresponding semantic concepts that internalize this syntactic context stack structure.

- Secondly, from a practical point of view, the Kripke style provides a foundation for the usual meta-programming style of quasi-quoting, as observed by Davies & Pfenning (2001). We are interested in reasoning directly about meta-programs for languages such as MetaML (Taha & Sheard, 1997; Taha, 2000) and similar staged- or meta-programming systems (see also Brady & Hammond, 2006) that use quasi-quotation. In this work, we hence develop the equational theory of the Kripke style formulation of the $\square$ modality. This allows MINT's $S4$ variant to be used as a program logic for reasoning directly about meta-programs. We give an example in Section 3.3.

Overview. In this paper, we establish normalization of MINT using normalization by evaluation (NbE) (Martin-Löf, 1975; Berger & Schwichtenberg, 1991). This yields an algorithm that can be directly implemented. An NbE algorithm typically consists of two steps: (1) we evaluate terms from the type theory in a chosen mathematical model and (2) we extract normal forms from that model. Here, we follow work by Abel (2013) and choose an untyped domain as the mathematical model. In the literature, there are other possible choices like presheaf categories (Altenkirch *et al.*, 1995; Kaposi & Altenkirch, 2017), but untyped domains are simpler to work with and mechanize. Further, they allow us to derive an actual normalization algorithm that can be implemented. In fact, as we will show, such an implementation of NbE can be extracted from our mechanization in Agda.

One problem in NbE is how we should model the Kripke structure introduced by $\square$ in the semantics. This is intimately related to how we characterize $\square$ elimination. In our constructions, the Kripke structure is captured by a novel algebra *truncoid* and is *internalized* in the semantics. Due to the internalization, our proof structure is just a moderate extension of Abel (2013). Since truncoids precisely capture various meta-theoretical structures in MINT including substitutions and evaluation environments, as a bonus, this allows us to elegantly and uniformly capture the differences across all subsystems. Further, the soundness and completeness proofs of NbE can easily be re-used for all the subsystems of $S4$ mentioned above *without change*.

Contributions. Our main contributions in this paper are:

- We present a core modal dependent type theory, MINT, as an explicit substitution calculus (Section 4) together with an equational theory. We successfully scale the concept of *Kripke-style substitutions* from Hu & Pientka (2022*a*) to dependent types, so that we obtain a unified representation of the modal and local structural properties of Kripke-style context stacks.
- Following Abel (2013), we develop an NbE algorithm (Section 5) for MINT which we prove complete (Section 6) and sound (Section 7). It builds on an untyped domain model. Central to the models and the NbE algorithm is the algebra of *truncoid* (Section 4.2). Truncoid provides an algebraic description of the Kripke structure of MINT, so that the Kripke structure in the semantics is *internalized* in the model

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

as *untyped modal transformations (UMoTs)*. As a result, our constructions are very adaptive, accommodating all four subsystems of $S4$ *without change*.

- The NbE algorithm of MINT with a full cumulative universe hierarchy and its soundness and completeness proofs are fully mechanized in Agda. Our mechanization of the NbE algorithm only relies on two standard extensions: induction–recursion (Dybjer, 2000) and function extensionality, which are more familiar than the advanced combination of induction–induction (Nordvall Forsberg & Setzer, 2010) and quotient inductive types exploited by Altenkirch & Kaposi (2016*b*,a) and Kaposi & Altenkirch (2017).
- We adjust our models so that they explicitly maintain universe levels and no longer fundamentally rely on cumulativity as in Abel (2013), Abel *et al.* (2017), and Gratzer *et al.* (2019). On the one hand, this adjustment directly enables a type-theoretic mechanization, as former on-paper approaches require taking limits of universe levels to infinity due to cumulativity. On the other hand, this adjustment also seems to suggest a more robust model construction that applies for both cumulative and non-cumulative universe hierarchies.
- Last, we extract from our Agda mechanization a Haskell implementation of NbE for MINT (Section 8), which may serve as a verified kernel of an implementation of MINT. We also provide an executable example for normalizing programs in MINT in our mechanization.

Our Agda mechanization consists of ∼11k LoC. This is close to or fewer than existing mechanizations (Abel *et al.*, 2017; Kaposi & Altenkirch, 2017; Pujet & Tabareau, 2022, 2023). Please refer to our technical report (Hu & Pientka, 2022*b*) and Agda mechanization for full details. The paper contains hyperlinks to an online artifact for an easy correspondence between code and discussions.

2 The Kripke style in the landscape of modal type theories

Over the past three decades, modal type systems have been studied from different perspectives. For example, Bierman & de Paiva (1996, 2000) use a regular context by requiring a substitution for modal assumptions during the introduction of $\square$. In 2001, Davies & Pfenning (2001) and Pfenning & Davies (2001) propose an alternative to the Kripke-style formulation where they separate modal assumptions and regular (or local) assumptions into two contexts. Hence, this formulation is often referred to as dual-context style. They also provide a translation for simply typed terms between the Kripke-style and the dual-context formulation. This translation can be viewed as compiling Kripke-style representations which corresponds to meta- or staged programming idioms under Curry–Howard correspondence to the dual-context representation which makes evaluation order more clear.

In recent years, the dual-context representation of modal logic has been well studied. For example, Kavvos (2017) looks into the dual-context style and gives formulations for systems K, T, $K4$, and GL. Shulman (2018) uses spatial type theory, a dependently typed variant of the dual-context style, to separate discrete and continuous assumptions to establish Brouwer's fixed-point theorem in HoTT. Licata *et al.* (2018) restrict spatial

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

type theory and obtain crisp type theory to internally represent universes in HoTT. The implementation of crisp type theory, Agda-flat, is also in the dual-context style. A recent general framework for modal type theories, multi-mode type theory (MTT) (Gratzer *et al.*, 2020; Gratzer, 2022), is a generalization of the dual-context style due to its elimination principle of the modalities. We therefore review here the dual-context style formulation and discuss differences between both styles.

The Kripke and dual-context styles differ in the organization of assumptions and the elimination principle of □. As opposed to context stacks in the Kripke style, a dual-context-style system has precisely two contexts. One context stores modal (or global) assumptions, while the other one stores regular (or local) assumptions. The defining difference between both styles is the elimination principle of the □ modality. In simply typed $S4$, the introduction and elimination rules for □ are defined as follows:

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

In the judgments, Ψ is the global context while Γ is the local one. The introduction rule requires that t is well typed only with the global assumptions. In the elimination rule, $s : \square T$ is eliminated by `letbox`; essentially `letbox` is a form of pattern matching.[2] The body t is type-checked with a global context extended with an extra global assumption $u : T$. This elimination principle implies that □ is viewed as sums. One shortcoming of sum types is the lack of extensionality. Extensionality of sum types including □ in the dual-context style requires a set of special equivalences like commuting conversions to be included (Lindley, 2007). Such commuting conversions are usually *absent* with dependent types. On the other hand, in the Kripke style, we use `unbox` for elimination. `unbox` is a projection and treats □ as products. □ in the Kripke style is therefore extensional due to its η equivalence rule. This difference in elimination principles finds a very close resemblance in the two styles of defining Σ types. When we define Σ inductively, its elimination is pattern matching.[3] If Σ is defined as a product, then we use projections for elimination. These two styles are not always equivalent and we may even clearly distinguish them in some settings, e.g., a substructural system. Moreover, even in regular MLTT, Σ as a product is extensional, because products admit η equivalence, while inductive types usually do not. Furthermore, the situation of □ is more complex. Davies & Pfenning (2001) only show a translation between the Kripke and the dual-context styles in the simply typed case. This translation in fact is not preserved by equivalence. Due to type-level computation, the translation given by Davies & Pfenning (2001) seems very challenging to extend to dependent types. In fact, due to the lack of extensionality in the dual-context style, we do not currently think that such a translation is possible between the Kripke-style and the dual-context style system.

Though the Kripke style presents appealing advantages of extensionality and uniformity in formulation, it is technically more challenging to study than the dual-context style. For example, direct normalization proofs of the simply typed Kripke-style systems were established (Hu & Pientka, 2022*a*; Valliappan *et al.*, 2022) only recently. The ultimate reason why the Kripke style is more challenging than the dual-context style is the lack of a proper notion of simultaneous substitutions. In the dual-context style, its notion of

[2] It is an *irrefutable pattern*.

[3] Also an irrefutable pattern.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

simultaneous substitutions is clear: there are two list of terms, one substituting the global context and the other for the local one. The obvious definition of simultaneous substitutions in the dual-context style is a significant advantage, both when implementing and when reasoning about these systems. In the Kripke style, it is less obvious how to define simultaneous substitutions. Gratzer *et al.* (2019) have given a substitution calculus for the dependently typed idempotent $S4$ and Birkedal *et al.* (2020*a*) have given a version for dependently typed K, but the general cases for other modal systems like T and non-idempotent $S4$ remain unknown. In our previous work (Hu & Pientka, 2022*a*), we propose the Kripke-style substitutions (or K-substitutions), which is shown to be a proper notion of simultaneous substitutions for the Kripke style with simple types. By introducing MINT and its foundational study, we have shown that K-substitutions and our normalization proof scale to the dependently typed settings. Therefore, our work is a significant step forward to a deeper understanding of the Kripke-style systems.

3 Introducing MINT by examples

Before introducing MINT and its NbE proof, we illustrate how to write programs that exploit the □ modality. In particular, we will use these programs to highlight different design decisions. We use an Agda-like syntax as our front-end language.

3.1 Axioms in S4

MINT is a system that captures dependently typed variants of four different modal systems: K, T, $K4$, and $S4$. These systems are distinguished by the logical axioms that they admit. In MINT, we can implement the following modal axioms generically:

$$\begin{aligned} K&: \Box(A \to B) \to \Box A \to \Box B \\ T&: \Box A \to A \\ 4&: \Box A \to \Box\Box A \end{aligned}$$

In these axioms, A and B refer to any propositions and hence are generic. Recall that in all four subsystems of $S4$, Axiom K is mandatory. The system that only admits K is System K. If Axiom T is added to System K, then we obtain System T. If we further add Axiom 4 to System T, then we have $S4$. System $K4$ only admits Axioms K and 4, but not T. To actually write down these axioms in MINT, intuitively we would like to give T, for example, the following type in MINT:

```
T : {A : Ty} → □ A → A
```

Here we use `Ty` to denote universes to avoid clashing with Agda's terminology. Unfortunately, this type does not type-check, because `A` is in the current world, but `□` requires the type `A` to be meaningful in the next world. The correct implementation of `T` states that `A` is a type that is universally accessible by giving it the kind `□ Ty`. This ensures that `A` remains accessible. When we want to use `A` in the definition of `T`, we now need to first `unbox` it with a proper level.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

```
  T : {A : □ Ty} → □ (unbox₁ A) → unbox₀ A
  T x = unbox₀ x
```

It might appear counter-intuitive at first glance, why we distinguish between a type of kind □ Ty and a type of kind Ty. This distinction is necessary, as MINT does not support cross-stage persistence (Taha & Sheard, 1997), i.e., the axiom R: $A \to \Box A$ or more specifically Ty → □Ty. In particular, there is no way to implement a function that would lift any type of kind Ty such that it would have kind □Ty. As a consequence, to ensure that all our types, in particular types such as $\Box A$, are well kinded, we need to ensure that A is globally meaningful. A similar design decision has been taken by Jang *et al*. (2022) in their work on developing a polymorphic modal type system that supports the generation of polymorphic code.

We are now in a position to also implement the other two axioms similarly:

```
K : {A B : □ Ty} → □ (unbox₁ A → unbox₁ B) → □ (unbox₁ A) → □ (unbox₁
  B)
K f x = box ((unbox₁ f) (unbox₁ x))

A4 : {A : □ Ty} → □ (unbox₁ A) → □ □ (unbox₂ A)
A4 x = box (box (unbox₂ x))
```

3.2 Lifting of natural numbers

As previously discussed, MINT does not support cross-stage persistence and the axiom $A \to \Box A$ is not admissible for all A. Nevertheless, there are types where we can explicitly lift an element of type A to $\Box A$. The type for natural numbers is one such example. In MINT, we need to implement such lifting functions when required to ensure cross-stage persistence. Since MINT supports inductive types just as MLTT does, we define natural numbers, Nat, in the usual way, with zero and succ as the constructors. Then, we define its lift function, which shows that natural numbers do admit Axiom $A \to \Box A$:

```
  lift : Nat → □ Nat
  lift zero     = box zero
  lift (succ n) = box (succ (unbox₁ (lift n)))
```

Note that this function is implemented by recursion on the input number. If the input is just zero, then the solution is easy: it is just box zero. We can refer to zero inside of a box, which requires a term in the next world, because Nat is a closed definition, which can be automatically lifted to any other world. In the succ case, we first provide a box as required by the output type obligation. Then, we must perform a recursion somehow. Luckily, unbox$_1$ brings us back to the current world, which allows us access to n, which is precisely needed for a structural recursion.

Just as pure MLTT, MINT can be used to prove properties about a definition. For example, we can show that unbox$_0$ is an inverse of lift:

```
  unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
  unbox-lift zero     = refl -- zero ≡ zero
  unbox-lift (succ n) = cong succ (unbox-lift n)
```

In the base case, the left-hand side evaluates to unbox$_0$ (box zero), which is just equivalent to zero. Therefore, reflexivity (refl) suffices to prove this goal. In the step case, we need to prove

```
unbox0 (box (succ (unbox1 (lift n)))) ≡ succ n
```

The left-hand side is reduced to succ (unbox$_0$ (lift n)) based on the equivalence rules that we describe in the next section. Note the recursive call unbox-lift n : unbox$_0$ (lift n) ≡ n. Therefore, we can conclude the goal by the recursive call modulo an extra congruence of succ.

3.3 Generating N-ary sum

According to Davies & Pfenning (2001), the modal logic $S4$ corresponds to staged computation under Curry–Howard correspondence, where $\Box A$ denotes the type of a computation of type A, the result of which is only available in some future stages of computation. Effectively, □ segments different computational stages, so that variables in past stages cannot be directly referred to in the current stage. The □ modality provides a logical foundation for multi-staged programming systems like MetaML (Taha & Sheard, 1997; Taha, 2000). By integrating □ into MLTT, we can use Mint as a program logic to model dependently typed staged computations and use Mint's *equational theory* to prove that the programs satisfy certain specifications. In this section, we show how the $S4$ variant of Mint can model staged programming and in the next, we prove that this program is correctly implemented. Proving the correctness of a staged or meta-program in MetaML or a similar system has not been previously considered, but with Mint, this capability comes very naturally. For more practicality, we postulate that certain extraction mechanisms can be employed here to extract the code to a mature staged programming system such as MetaML (Taha & Sheard, 1997) with proper type-level magic to erase the dependent types as commonly practiced in Coq and Agda.

Our task here is to model a meta-program that generates code for an n-ary sum function that sums up n numbers. If n is zero, then we return zero; if it is one, then we return the identity function; if it is two, then we return the function that sums up two arguments, i.e., box λ x y → x + y. Writing such an n-ary sum function in a type-safe manner can be achieved by exploiting large elimination in MLTT.

We first define a type-level function nary n, which computes the type of an n-ary function:

```
nary : Nat → Ty
nary zero     = Nat
nary (succ n) = Nat → nary n
```

We then define the type of nary-sum as taking in a natural number n : Nat and intuitively returning code of type nary n. This, however, does not quite work, as □ (nary n) is ill-typed. Note that n is defined in the current world, but we need to use it inside □ (i.e. in the next world). We hence need to first lift the n : Nat to □ Nat using the previously defined lift function to then be able to splice it in. The need to lift values such as natural numbers to have access to both their values and their code representations is a common theme when writing staged programs. In a dependently typed setting, we need to use such lifting functions also on the type level to support a form of cross-stage persistence of values. We hence arrive at the type (n : Nat) → □ (nary (unbox$_1$ (lift n))) for the nary-sum function. Its implementation follows, in fact, directly from our intention.

```
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)
```

Note that in the base case of `zero`, the return type is □ `Nat`; in the case of `succ zero`, we return the boxed identity function; in the case of `succ (succ n)`, `nary-sum (succ (succ n))` returns a term of type □ `(nary (unbox₁ (lift (succ (succ n)))))`. The recursive call `nary-sum (succ n)` has type □ `(nary (unbox₁ (lift (succ n))))`. Further, we have

```
  nary (unbox₁ (lift (succ n)))
= Nat → nary (unbox₁ (lift n))
  nary (unbox₁ (lift (succ (succ n))))
= nary (succ (succ (unbox₁ (lift n))))
= Nat → Nat → nary (unbox₁ (lift n))
```

To compute the final result of `nary-sum (succ (succ n))`, we first unbox the code generated by `nary-sum (succ n)`, which has type `Nat → nary (unbox₁ (lift n))`, and apply it to the sum of the first two arguments. For convenience, we use numeric literals `0`, `1`, etc. for natural numbers `zero`, `succ zero`, etc. interchangeably. To illustrate, let us normalize `nary-sum 3`:

```
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
```

The last equation shows in MINT that `nary-sum 3` and the code of `λ x y z → (x + y) + z` are definitionally equal due to the congruence of `box`:

```
nary-sum-3 : nary-sum 3 ≡ box λ x y z → (x + y) + z
nary-sum-3 = refl
```

MINT admits the congruence of `box` and as a result, reductions occur freely even inside of a `box` as in MetaML (Taha, 2000). The congruence of `box` is essential to model MetaML and particularly helpful when using MINT as a program logic, for the same reason as having the congruence of `λ`. Moreover, the congruence of `box` allows a significantly simpler semantic model leading to a straightforward normalization proof.

3.4 Soundness of N-ary sum

Previously, we have shown a specific proof for the ternary sum. MINT can take one step further: we can prove general properties about `nary-sum`. In particular, we can prove that given a list `xs` of natural numbers, which has length `n`, adding up all numbers in `xs` returns the same result as using the code generated by `nary-sum n` to add them up. To make this theorem precise, 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 equational reasoning. The slightly unorthodox definition of `sum` is defined by recursion on `n` just as `ap-list`, so that auxiliary lemmas such as the associativity of addition are avoided in our subsequent soundness theorem. Proving it equal to the standard definition is an easy exercise in pure MLTT, which we omit here. We now can state and prove our target theorem:

```
nary-sum-sound : (n : Nat) (xs : List N)
    (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. When using `nary-sum-sound`, `eq'` can be derived from `eq` and `unbox-lift` defined above. The first two base cases are easy. In the last case, a recursive call suffices. We reason as follows. The expected return type is

```
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))
```

Therefore by definitional equality, `nary-sum-sound` is a valid proof.

4 Definition of Mint

In this section, we formally introduce Mint. We introduce its syntax, typing rules, and equivalence rules. To manipulate the modal structure, we introduce two operations on Kripke-style substitutions, truncation and truncation offset, to the system. Both operations turn out to be components of an algebraic structure, called *truncoid*, which captures the Kripke structure of Mint generically. We use this algebraic structure throughout our technical development and have corresponding definitions in the semantics.

4.1 Syntax and judgments of Mint

Mint has the following syntax:

$$
\begin{array}{rcll}
m, n & \in & \mathbb{N} & (\mathtt{unbox}\text{ levels or modal offsets}) \\
x, y & \in & \mathbb{N} & (\text{de Bruijn indices}) \\
s, t, M, S, T & := & v_x \mid \mathtt{Nat} \mid \square T \mid \Pi S.T \mid \mathtt{Ty}_i \mid \mathtt{zero} \mid \mathtt{succ}\ \ t \mid \mathtt{elim}\, M\ s\ s'\ t & \\
 & \mid & \mathtt{box}\ t \mid \mathtt{unbox}_n\ t \mid \lambda t \mid s\ t \mid t[\overrightarrow{\sigma}] & (\text{Terms}, \mathsf{Trm}) \\
\overrightarrow{\sigma}, \overrightarrow{\delta} & := & \overrightarrow{I} \mid \overrightarrow{\sigma}, t \mid \mathsf{wk} \mid \overrightarrow{\sigma}; \Uparrow^n \mid \overrightarrow{\sigma} \circ \overrightarrow{\delta} & \\
 & & & (\text{Kripke-style substitutions or K-substitutions}, \mathsf{Substs}) \\
\Gamma, \Delta, \Phi & := & \cdot \mid \Gamma.T & (\text{Contexts}, \mathsf{Ctx}) \\
\overrightarrow{\Gamma}, \overrightarrow{\Delta} & := & \varepsilon \mid \overrightarrow{\Gamma}; \Gamma & (\text{Context stacks}, \overrightarrow{\mathsf{Ctx}}) \\
w, W & := & u \mid \mathtt{Nat} \mid \mathtt{Ty}_i \mid \square W \mid \Pi W.W' \mid \mathtt{zero} \mid \mathtt{succ}\ \ w \mid \mathtt{box}\ w \mid \lambda w & \\
 & & & (\text{Normal forms}, \mathsf{Nf}) \\
u, V & := & v_x \mid \mathtt{elim}\ W\ w\ w'\ u \mid \mathtt{unbox}_n\ u \mid u\ w & (\text{Neutral forms}, \mathsf{Ne})
\end{array}
$$

Mint extends MLTT with context stacks and the □ modality. As discussed in Section 1, Mint models and reasons about the Kripke semantics (Kripke, 1963). In particular, we use context stacks to keep track of assumptions in all accessed worlds. Mint has natural numbers ($\mathtt{Nat}$, $\mathtt{zero}$, $\mathtt{succ}\ \ t$), Π types, cumulative universes, written as $\mathtt{Ty}_i$, and explicit Kripke-style substitutions, which model the mapping between context stacks. Here, the cumulativity of universes means if a type is in the universe of level i, then it is also in the universe of level $1 + i$, not a stronger notion of cumulativity based on universe subtyping working even for function types and others. We also include a recursor on natural numbers ($\mathtt{elim}\, M\ s\ s'\ t$). In this expression, t is the scrutinee describing a natural number, and s and s' are referring to the two possible cases where t is $\mathtt{zero}$ and the successor, respectively; M is the motive describing essentially the overall type skeleton of the recursor. As the overall type of the recursor depends on t, we model the motive M as a type with one free variable. Subsequently, we omit most discussions of natural numbers for brevity and refer the interested readers to our technical report and our mechanization. To construct and use a term of Π type, we have λ abstraction and function application as usual. We have discussed $\mathtt{box}$ and $\mathtt{unbox}$, the constructor and eliminator of □ types, in Section 3. When $\mathtt{unbox}$'ing a term t, we must specify a number n, i.e., $\mathtt{unbox}_n\ t$, to describe the Kripke world we travel back to when type-checking t. By restricting n, Mint is specialized to one of the dependently typed generalizations of four different modal systems: K, T, $K4$, and $S4$. For example, if the modal offset must be 1, i.e., only $\mathtt{unbox}_1\ t$ is possible, then we obtain the System K. By allowing the modal offset to be either 0 or 1, we obtain the System T. If the

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

modal offset must be positive, we obtain the System $K4$. If there is no restriction at all, then we obtain the most general System $S4$. The Kripke style has the advantage of being specialized to different modal systems by controlling the modal offsets. This observation has been made by Pfenning & Wong (1995), Davies & Pfenning (2001), and Hu & Pientka (2022*a*). At last, since MINT is formulated with explicit Kripke-style substitutions, the substitution closure $t[\vec{\sigma}]$ is defined as a form of syntax. We emphasize that the $t[\vec{\sigma}]$ has a very low binding precedence in our representation. For example, when we write $t\ s[\vec{\sigma}]$, we mean $(t\ s)[\vec{\sigma}]$. If we want $\vec{\sigma}$ to be applied to s only, we add explicit parentheses $t\ (s[\vec{\sigma}])$. Similarly, $\texttt{unbox}_n\ t[\vec{\sigma}]$ denotes $(\texttt{unbox}_n\ t)[\vec{\sigma}]$.

Moreover, in this paper, we explicitly work with de Bruijn indices to stay close to our mechanization. The de Bruijn index of a variable is to be understood relative to the topmost context in a context stack, keeping in mind that box and unbox modify the context stack and change the topmost context. For example, given a context stack $\square\texttt{Nat}; \Pi\texttt{Nat}.\texttt{Nat}$ the term $v_0\ (\texttt{unbox}_1\ v_0)$ is well formed, but the first occurrence of v_0 refers to the function $\Pi\texttt{Nat}.\texttt{Nat}$, while the second occurrence in $\texttt{unbox}_1\ v_0$ refers to $\square\texttt{Nat}$, as $\texttt{unbox}_1\ v_0$ refers to a term in the prior world. Notationally, we consistently use upper cases for types and lower cases otherwise.

We turn the formulation of Kripke-style substitutions (or just *K-substitutions*) from our previous work at Mathematical Foundations of Programming Semantics (MFPS) (Hu & Pientka, 2022*a*) between context stacks into explicit K-substitutions. K-substitutions are a generalization of the usual simultaneous substitutions. Instead of representing a mapping between two contexts in the typical case, a K-substitution is a mapping between two *context stacks*. In our explicit K-substitutions, the following syntax directly replicates the usual formulation of simultaneous substitutions: $\vec{I}$ is the identity K-substitutions between two context stacks; $\vec{\sigma}, t$ extends a K-substitution with a term; wk is local weakening supporting weakening of the topmost context of a context stack; and $\vec{\sigma} \circ \vec{\delta}$ composes $\vec{\sigma}$ with $\vec{\delta}$. Furthermore, a K-substitution is no longer simply a list of terms. Modal extension $\vec{\sigma}; \Uparrow^n$ is added and unique in Kripke-style modal type theories, and models modal weakening of context stacks. The modal offset n originates from the modal transformation operation (MoT) (Davies & Pfenning, 2001) and is used to keep track of the number of contexts, which are added to the codomain context stack as in the following rule. A modal extension only introduces *one empty context* to the domain context stack:

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

Please refer to our previous work at MFPS (Hu & Pientka, 2022*a*) for a complete motivation for modal extensions. $\vec{\sigma}$ provides a mapping from $\vec{\Gamma}$ to $\vec{\Delta}$, while $\vec{\sigma}; \Uparrow^n$ maps from $\vec{\Gamma}; \vec{\Gamma}'$ to $\vec{\Delta}; \cdot$ where the number of contexts in $\vec{\Gamma}'$ is n. This modal weakening is crucial to characterize the β rule of $\square$ in a type-safe manner (see β RULE OF $\square$). Hence, we incorporate modal weakening into our definition of K-substitution.

We give some selected typing rules in Figure 1. The full set of rules can be found in Appendix A. The definition of MINT consists of six judgments: $\vdash \vec{\Gamma}$ denotes that the context stack $\vec{\Gamma}$ is well formed; $\vdash \vec{\Gamma} \approx \vec{\Delta}$ denotes that $\vec{\Gamma}$ and $\vec{\Delta}$ are equivalent context stacks; $\vec{\Gamma} \vdash t : T$ denotes that term t has type T in context stack $\vec{\Gamma}$; $\vec{\Gamma} \vdash t \approx s : T$ denotes

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

$\boxed{\vdash \overrightarrow{\Gamma}}$ and $\boxed{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}$ well formedness of $\overrightarrow{\Gamma}$ and equivalence between $\overrightarrow{\Gamma}$ and $\overrightarrow{\Delta}$

$$\frac{}{\vdash \varepsilon ; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma}}{\vdash \overrightarrow{\Gamma} ; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma} ; \Gamma \qquad \overrightarrow{\Gamma} ; \Gamma \vdash T : \mathtt{Ty}_i}{\vdash \overrightarrow{\Gamma} ; \Gamma . T} \qquad \frac{}{\vdash \varepsilon ; \cdot \approx \varepsilon ; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}{\vdash \overrightarrow{\Gamma} ; \cdot \approx \overrightarrow{\Delta} ; \cdot}$$

$$\frac{\begin{array}{c} \vdash \overrightarrow{\Gamma} ; \Gamma \approx \overrightarrow{\Delta} ; \Delta \qquad \overrightarrow{\Gamma} ; \Gamma \vdash T \approx T' : \mathtt{Ty}_i \\ \overrightarrow{\Delta} ; \Delta \vdash T \approx T' : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} ; \Gamma \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Delta} ; \Delta \vdash T' : \mathtt{Ty}_i \end{array}}{\vdash \overrightarrow{\Gamma} ; \Gamma . T \approx \overrightarrow{\Delta} ; \Delta . T'}$$

$\boxed{\overrightarrow{\Gamma} \vdash t : T}$ t has type T in $\overrightarrow{\Gamma}$

$$\frac{\overrightarrow{\Gamma} \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash T : \mathtt{Ty}_{1+i}} \qquad \frac{\overrightarrow{\Gamma} ; \Gamma \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} ; \Gamma . S \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma} ; \Gamma \vdash \Pi S . T : \mathtt{Ty}_i}$$

$$\frac{\overrightarrow{\Gamma} ; \Gamma \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} ; \Gamma . S \vdash t : T}{\overrightarrow{\Gamma} ; \Gamma \vdash \lambda t : \Pi S . T} \qquad \frac{\begin{array}{c} \overrightarrow{\Gamma} ; \Gamma \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} ; \Gamma . S \vdash T : \mathtt{Ty}_i \\ \overrightarrow{\Gamma} ; \Gamma \vdash t : \Pi S . T \qquad \overrightarrow{\Gamma} ; \Gamma \vdash s : S \end{array}}{\overrightarrow{\Gamma} ; \Gamma \vdash t\ s : T[\overrightarrow{I}, s]}$$

$$\frac{\overrightarrow{\Gamma} ; \cdot \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash \Box T : \mathtt{Ty}_i} \qquad \frac{\overrightarrow{\Gamma} ; \cdot \vdash t : T}{\overrightarrow{\Gamma} \vdash \mathtt{box}\ t : \Box T} \qquad \frac{\begin{array}{c} \overrightarrow{\Gamma} ; \cdot \vdash T : \mathtt{Ty}_i \\ \overrightarrow{\Gamma} \vdash t : \Box T \qquad \vdash \overrightarrow{\Gamma} ; \overrightarrow{\Delta} \qquad |\overrightarrow{\Delta}| = n \end{array}}{\overrightarrow{\Gamma} ; \overrightarrow{\Delta} \vdash \mathtt{unbox}_n\ t : T[\overrightarrow{I} ; \Uparrow^n]}$$

$\boxed{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}$ $\overrightarrow{\sigma}$ is a well formed K-substitution from $\overrightarrow{\Delta}$ to $\overrightarrow{\Gamma}$

$$\frac{\vdash \overrightarrow{\Gamma}}{\overrightarrow{\Gamma} \vdash \overrightarrow{I} : \overrightarrow{\Gamma}} \qquad \frac{\vdash \overrightarrow{\Gamma} ; \Gamma . T}{\overrightarrow{\Gamma} ; \Gamma . T \vdash \mathsf{wk} : \overrightarrow{\Gamma} ; \Gamma} \qquad \frac{\begin{array}{c} \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}' ; \Gamma \\ \overrightarrow{\Gamma}' ; \Gamma \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : T[\overrightarrow{\sigma}] \end{array}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}, t : \overrightarrow{\Gamma}' ; \Gamma . T}$$

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

$\boxed{\overrightarrow{\Gamma} \vdash t \approx t' : T}$ t and t' of type T are equivalent in $\overrightarrow{\Gamma}$

$$\frac{\begin{array}{c} \overrightarrow{\Gamma} ; \Gamma \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} ; \Gamma . S \vdash T : \mathtt{Ty}_i \\ \overrightarrow{\Gamma} ; \Gamma . S \vdash t : T \qquad \overrightarrow{\Gamma} ; \Gamma \vdash s : S \end{array}}{\overrightarrow{\Gamma} ; \Gamma \vdash (\lambda t)\ s \approx t[\overrightarrow{I}, s] : T[\overrightarrow{I}, s]} \qquad \frac{\begin{array}{c} \overrightarrow{\Gamma} ; \Gamma \vdash S : \mathtt{Ty}_i \\ \overrightarrow{\Gamma} ; \Gamma . S \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} ; \Gamma \vdash t : \Pi S . T \end{array}}{\overrightarrow{\Gamma} ; \Gamma \vdash t \approx \lambda (t[\mathsf{wk}]\ v_0) : \Pi S . T}$$

$$\frac{\begin{array}{c} \overrightarrow{\Gamma} ; \cdot \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} ; \cdot \vdash t : T \\ \vdash \overrightarrow{\Gamma} ; \overrightarrow{\Delta} \qquad |\overrightarrow{\Delta}| = n \end{array}}{\overrightarrow{\Gamma} ; \overrightarrow{\Delta} \vdash \mathtt{unbox}_n\ (\mathtt{box}\ t) \approx t[\overrightarrow{I} ; \Uparrow^n] : T[\overrightarrow{I} ; \Uparrow^n]} \qquad \frac{\overrightarrow{\Gamma} ; \cdot \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : \Box T}{\overrightarrow{\Gamma} \vdash t \approx \mathtt{box}\ (\mathtt{unbox}_1\ t) : \Box T}$$

Fig. 1. Selected rules for MINT.

that terms t and s of type T are equivalent in context stack $\overrightarrow{\Gamma}$; $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$ denotes that $\overrightarrow{\sigma}$ is a K-substitution susbtituting terms in $\overrightarrow{\Delta}$ into ones in $\overrightarrow{\Gamma}$; and $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\delta} : \overrightarrow{\Delta}$ denotes that $\overrightarrow{\sigma}$ and $\overrightarrow{\delta}$ are equivalent in K-substituting terms in $\overrightarrow{\Delta}$ into ones in $\overrightarrow{\Gamma}$. Due to explicit K-substitutions, there needs to be more judgments than usual. The equivalence between

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

context stacks $\vdash \vec{\Gamma} \approx \vec{\Delta}$ is necessary, because we must specify a conversion rule for the equivalence between K-substitutions:

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \qquad \vdash \vec{\Delta} \approx \vec{\Delta}'}{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}'}$$

In turn, we need the equivalence between K-substitutions $\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\delta} : \vec{\Delta}$ due to the following congruence rule for the K-substitution closure:

$$\frac{\vec{\Delta} \vdash t \approx t' : T \qquad \vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Delta}}{\vec{\Gamma} \vdash t[\vec{\sigma}] \approx t'[\vec{\sigma}'] : T[\vec{\sigma}]}$$

Without explicit (K-)substitutions, both $\vdash \vec{\Gamma} \approx \vec{\Delta}$ and $\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\delta} : \vec{\Delta}$ are defined after the typing judgments, instead of mutually defined. This style of definitions follows Abel (2013) closely.

Following Harper & Pfenning (2005), we introduce extra premises in the rules in order to establish syntactic properties like *context stack conversion* and *presupposition*. Context stack conversion states that all syntactic judgments respect context stack equivalence $\vdash \vec{\Gamma} \approx \vec{\Delta}$. Presupposition (or syntactic validity) includes for example facts such as if $\vec{\Gamma} \vdash t : T$, then $\vdash \vec{\Gamma}$ and $\vec{\Gamma} \vdash T : \mathtt{Ty}_i$ for some i.

Theorem 4.1 (Context stack conversion). *Given* $\vdash \vec{\Gamma} \approx \vec{\Delta}$,

- *if* $\vec{\Gamma} \vdash t : T$, *then* $\vec{\Delta} \vdash t : T$;
- *if* $\vec{\Gamma} \vdash t \approx s : T$, *then* $\vec{\Delta} \vdash t \approx s : T$;
- *if* $\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Gamma}'$, *then* $\vec{\Delta} \vdash \vec{\sigma} : \vec{\Gamma}'$;
- *if* $\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Gamma}'$, *then* $\vec{\Delta} \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Gamma}'$.

Theorem 4.2 (Presupposition).

- *If* $\vdash \vec{\Gamma} \approx \vec{\Delta}$, *then* $\vdash \vec{\Gamma}$ *and* $\vdash \vec{\Delta}$.
- *If* $\vec{\Gamma} \vdash t : T$, *then* $\vdash \vec{\Gamma}$ *and* $\vec{\Gamma} \vdash T : \mathtt{Ty}_i$ *for some* i.
- *If* $\vec{\Gamma} \vdash t \approx t' : T$, *then* $\vdash \vec{\Gamma}$, $\vec{\Gamma} \vdash t : T$, $\vec{\Gamma} \vdash t' : T$ *and* $\vec{\Gamma} \vdash T : \mathtt{Ty}_i$ *for some* i.
- *If* $\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}$, *then* $\vdash \vec{\Gamma}$ *and* $\vdash \vec{\Delta}$.
- *If* $\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Delta}$, *then* $\vdash \vec{\Gamma}$, $\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}$, $\vec{\Gamma} \vdash \vec{\sigma}' : \vec{\Delta}$ *and* $\vdash \vec{\Delta}$.

After proving these two properties, we remove the highlighted premises and call the resulting judgments the *true and golden* definition of MINT. These two definitions are equivalent:

Theorem 4.3. *The two sets of rules of* MINT *given above are equivalent.*

4.2 Truncoid: Algebra of truncation and truncation offset

MINT fundamentally relies on two operations, truncation $(\_ \mid \_)$ and truncation offset $(\mathscr{O}(\_, \_))$ to handle its Kripke structure. Given $\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}$ and some $n < |\vec{\Delta}|$, truncation

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

of a K-substitution, written as $\vec{\sigma} \mid n$, returns a prefix of $\vec{\sigma}$ with domain context stack $\vec{\Delta} \mid n$ where n modal extensions and accordingly n contexts from $\vec{\Delta}$ are dropped. The main question then is, what is the codomain context stack of this K-substitution? – Since we drop modal extensions including their modal offsets from $\vec{\sigma}$, the codomain context stack $\vec{\Gamma}$ must also be adjusted. Recall that each modal offset in a modal extension accounts for a number of contexts in $\vec{\Gamma}$. Hence, intuitively, we should drop k contexts from $\vec{\Gamma}$ where k is the sum of all dropped modal offsets.

For example, let $\vec{\sigma}$ be $(\vec{\delta}; \Uparrow^1, t; \Uparrow^2, s)$ where $\vec{\Gamma}; \Gamma_0; \Gamma_1; \Gamma_2 \vdash \vec{\sigma} : \vec{\Delta}; \cdot.T; \cdot.S$. Truncation $\vec{\sigma} \mid 2$ then returns the prefix $\vec{\delta}$ after dropping everything up to two modal extensions $\Uparrow^1$ and $\Uparrow^2$. This resulting K-substitution intuitively must have domain context stack $\vec{\Delta}$ and codomain context stack $\vec{\Gamma}$, i.e., $\vec{\Gamma} \vdash \vec{\delta} : \vec{\Delta}$. How do we obtain $\vec{\Gamma}$ from $\vec{\Gamma}; \Gamma_0; \Gamma_1; \Gamma_2$? – The answer is by simply dropping the top 3 contexts where 3 is exactly the sum of the modal offsets in the dropped part of $\vec{\sigma}$.

To compute the sum of modal offsets, we define the truncation offset $\mathcal{O}(\vec{\sigma}, n)$ below. In general, we have $\vec{\Gamma} \mid \mathcal{O}(\vec{\sigma}, n) \vdash \vec{\sigma} \mid n : \vec{\Delta} \mid n$.

Our syntax of explicit K-substitutions is carefully designed such that truncation and truncation offset can be defined by recursion on the structure of the inputs:

Truncation (_ | _)

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

Truncation Offset ($\mathcal{O}$(_, _))

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

These two operations satisfy the following two coherence conditions:

Lemma 4.4 (Coherence conditions).

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

These two operations describe how K-substitutions interact with the Kripke structure of MINT so that the whole system remains coherent, e.g., in the following rule:

$$\frac{\vec{\Delta}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Delta} \vdash t : \Box T \qquad |\vec{\Delta}'| = n \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}; \vec{\Delta}'}{\vec{\Gamma} \vdash \mathtt{unbox}_n\ t[\vec{\sigma}] \approx \mathtt{unbox}_{\mathcal{O}(\vec{\sigma}, n)}\ (t[\vec{\sigma} \mid n]) : T[\vec{\sigma} \mid n; \Uparrow^{\mathcal{O}(\vec{\sigma}, n)}]}$$

When applying a K-substitution $\vec{\sigma}$ to `unbox`, the K-substitution is pushed recursively inside with a truncated K-substitution and the `unbox` level adjusted by truncation offset.

Truncation and truncation offset turn out to be very general concepts and are central to modeling the Kripke structure in both syntax and semantics. In particular, truncation and truncation offset form an algebra, which we call a *truncoid*. The following gives a concrete and the most basic algebraic characterization of a truncoid:

Definition 4.1. A truncoid is a triple $(S, \_ \mid \_, \mathcal{O}(\_, \_))$, where

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

where the coherence of addition holds:

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

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. In Section 5 and later sections, we will describe other instances of truncoids on the semantic side. 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, so it is worth organizing a few important and special truncoids ahead of time. The first kind is *applicative truncoids*, which allows a truncoid to be applied to another:

Definition 4.2. 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:

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

Most semantic models that we depend on in the normalization proof are also applicative. For example, evaluation environments in Section 5 are 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 4.3. 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:

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

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

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. We will see very soon that the 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 we obtain a *closed truncoid*:

Definition 4.4. A closed truncoid (S_0, S_1) is an applicative truncoid triple (S_0, S_1, S_0) where S_1 is a substitutional truncoid. We write id and $\_ \circ \_$ for identity and composition of S_1. The following additional coherence 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. Note 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 be shown in Section 5 is one such example.

5 Untyped domain

Starting from this section, we begin to prove the normalization of Mint. The plan is as follows:

1. In this section, we define the *untyped domain* to which we evaluate valid programs of Mint and the NbE algorithm.
2. In Section 6, we establish the completeness theorem, which states that equivalent terms evaluate to syntactically equal normal forms by our algorithm. Completeness is proved by constructing a partial equivalence relation (PER) model, which relates two values from the untyped domain.
3. In Section 7, we establish the soundness theorem, which states that a well-typed term is equivalent to the normal form returned by the NbE algorithm. The model to prove soundness is more sophisticated as it relies on the PER model defined for completeness. The model for soundness must *glue* both values in the untyped domain and the syntactic terms, so conventionally, we call this model a *gluing model*.

In our NbE algorithm, we first *evaluate* a well-typed syntactic term into an untyped semantic domain, D. This domain contains no β redex, so the evaluation process performs all β reductions. After obtaining a value in the untyped domain, we must convert this value back to the syntax as a normal form. This process is done by the *readback* functions. The readback functions also perform η expansion, so we eventually obtain a $\beta\eta$ normal form.

In this section, we give the definition of the untyped domain in which we operate and the NbE algorithm. For brevity, we omit most of discussion about natural numbers and recursion. We refer interested readers to our technical report and our Agda code. For mathematical accuracy, we use $:=$ for assignments and $=$ for equality. We use :: to denote a meta-level name binding, i.e., Agda level.

5.1 Definition of untyped domain

The untyped domain has the following syntax. :

$$\begin{array}{rcll}
z & \in & \mathbb{N} & \text{(Domain variables in de Bruijn levels)} \\
a, b, A & := & \mathsf{N} \mid \blacksquare A \mid \mathit{Pi}(A, T, \overrightarrow{\rho}) \mid \mathsf{U}_i \mid \mathtt{ze} \mid \mathtt{su}(a) \mid \Lambda(t, \overrightarrow{\rho}) \mid \mathtt{box}(a) \mid \uparrow^{A}(c) & \\
 & & & \text{(Domain values, } D\text{)} \\
c, C & := & l_z \mid \mathsf{rec}(M, a, t, c, \overrightarrow{\rho}) \mid c\ d \mid \mathtt{unbox}(n, c) & \text{(Neutral domain values, } D^{\mathsf{Ne}}\text{)} \\
d, D & := & \downarrow^{A}(a) & \text{(Normal domain values, } D^{\mathsf{Nf}}\text{)} \\
\rho & \in & \mathsf{Env} := \mathbb{N} \to D & \text{(Local evaluation environment)} \\
\overrightarrow{\rho} & \in & \mathsf{Envs} := \mathbb{N} \to \mathbb{N} \times \mathsf{Env} & \text{((Global) evaluation environment)}
\end{array}$$

In the untyped domain, variables are represented by de Bruijn *levels*. Consider a topmost context $\Gamma.T.\Delta$ in some stack. If a variable v_x is bound to type T, in the syntax, we use a de Bruijn *index* and $x = |\Delta|$. Its corresponding de Bruijn level z, on the other hand, equals to $|\Gamma|$. De Bruijn levels assign a *unique absolute name* to each variable in each context, so that we can avoid handling local weakening of variables in the semantics. Evidently, these two representations satisfy $z + x + 1 = |\Gamma.T.\Delta|$. This equation will be used in the readback functions to correspond syntactic and semantic variables.

In this untyped domain, values are effectively partially β-reduced values and classified into three categories: values (D), neutral values (D^{Ne}), and normal values (D^{Nf}). Same as before, we consistently use upper cases for semantic types and lower cases otherwise. In D, $\blacksquare$ models $\square$ semantically and U_i models a universe at level i. $\mathtt{ze}$ models $\mathtt{zero}$ and $\mathtt{su}(a)$ models successors. A neutral value c is embedded into D when annotated with a type A. Following Abel (2013), we use *defunctionalization* (Reynolds, 1998; Ager *et al.*, 2003) to capture open syntactic terms in the domain together with their surrounding evaluation environments, which enables formalization in Agda.

For example, a domain type $\mathit{Pi}(A, T, \overrightarrow{\rho})$ models a Π type and consists of a domain type A as the input type and a syntactic type T as the output type together with its ambient environment $\overrightarrow{\rho}$, which provides instantiations for all the free variables in T except the topmost one bound by Π in the syntax. Similarly, a domain value $\Lambda(t, \overrightarrow{\rho})$ models a λ term and describes a syntactic body t together with an environment $\overrightarrow{\rho}$, which provides values for all the free variables in t except the topmost one bound by λ in the syntax. For a neutral domain value for the recursor of natural numbers $\mathsf{rec}(M, a, t, c, \overrightarrow{\rho})$, the neutral domain value c describes the scrutinee, which is intended to be a natural number, while a describes the domain value for the base case. Next, t is an open syntactic term because it describes the term for the step-case. Last, M describes the motive in the syntax with one free variable expressing the dependency on the natural number c. Due to defunctionalization, we thus capture $\overrightarrow{\rho}$, the surrounding environment of M and t.

The NbE algorithm relies on local and global environments in the evaluation process. Local evaluation environments (Env) are functions mapping de Bruijn indices to domain values. Global evaluation environments (Envs), or just environments, are functions mapping modal offsets to tuples $\mathbb{N} \times \mathsf{Env}$ where the first projection is a modal offset, thereby allowing us to model different modal logics. An environment can be viewed as a stream of local environments paired with modal offsets. In the NbE algorithm, when evaluating a well-typed term, only a finite prefix of an environment is used, which is ensured by soundness.

We use $\texttt{emp} :: \mathsf{Env}$ for the empty local environment and $\texttt{empty} :: \mathsf{Envs}$ for the global environment.

$$\begin{array}{ll@{\qquad}ll} \texttt{emp} & :: \mathsf{Env} & \texttt{empty} & :: \mathsf{Envs} \\ \texttt{emp}(\_) & := \texttt{ze} & \texttt{empty}(\_) & := (1, \texttt{emp}) \end{array}$$

$\texttt{emp}$ and $\texttt{empty}$ are the empty local and global evaluation environments. Their definitions do not matter here because they only provide default values, which are guaranteed to be never used by soundness. Instead, we focus on how to extend local and global environments. First, we can modally extend an environment with a modal offset n:

$$\begin{array}{ll} \texttt{ext} & :: \mathsf{Envs} \to \mathbb{N} \to \mathsf{Envs} \\ \texttt{ext}(\vec{\rho}, n)(0) & := (n, \texttt{emp}) \\ \texttt{ext}(\vec{\rho}, n)(1+m) & := \vec{\rho}(m) \end{array}$$

We write $\texttt{ext}(\vec{\rho})$ for $\texttt{ext}(\vec{\rho}, 1)$. $\texttt{ext}$ models modal extension of a K-substitution $(\vec{\sigma}\,;\Uparrow^n)$, so n is not associated with any local environment, hence $\texttt{emp}$.

We can locally extend an environment with a value by inserting it into the topmost local environment. $\texttt{lext'}$ conses a value to a local environment, and $\texttt{lext}$ just extends a value to its topmost local environment by calling $\texttt{lext'}$:

$$\begin{array}{ll} \texttt{lext'} & :: \mathsf{Env} \to D \to \mathsf{Env} \\ \texttt{lext'}(\rho, a)(0) & := a \\ \texttt{lext'}(\rho, a)(1+m) & := \rho(m) \end{array}$$

$$\begin{array}{lll} \texttt{lext} & :: \mathsf{Envs} \to D \to \mathsf{Envs} & \\ \texttt{lext}(\vec{\rho}, a)(0) & := (n, \texttt{lext'}(\rho, a)) & \text{(where } (n, \rho) := \vec{\rho}(0)) \\ \texttt{lext}(\vec{\rho}, a)(1+m) & := \vec{\rho}(1+m) & \end{array}$$

The $\texttt{drop}$ operation drops the zeroth mapping from the topmost Env. It is needed for the interpretation of wk:

$$\begin{array}{lll} \texttt{drop} & :: \mathsf{Envs} \to \mathsf{Envs} & \\ \texttt{drop}(\vec{\rho})(0) & := (n, m \mapsto \rho(1+m)) & \text{(where } (n, \rho) := \vec{\rho}(0)) \\ \texttt{drop}(\vec{\rho})(1+m) & := \vec{\rho}(1+m) & \end{array}$$

Last, environments form a truncoid,[4] and we define truncation and truncation offset on $\vec{\rho}$:

$$\begin{array}{ll} \_\mid\_ & :: \mathsf{Envs} \to \mathbb{N} \to \mathsf{Envs} \\ (\vec{\rho} \mid n)(m) & := \vec{\rho}(n+m) \end{array}$$

$$\begin{array}{lll} \mathcal{O}(\_,\_) & :: \mathsf{Envs} \to \mathbb{N} \to \mathbb{N} & \\ \mathcal{O}(\vec{\rho}, 0) & := 0 & \\ \mathcal{O}(\vec{\rho}, 1+n) & := m + \mathcal{O}(\vec{\rho} \mid 1, n) & \text{(where } (m, \_) := \vec{\rho}(0)) \end{array}$$

[4] Coherence condition 1 and 2.

5.2 Untyped Modal Transformations

To model $\square$, we must consider how to model the Kripke structure of MINT. In our untyped domain model, we employ the *internal* approach, where the Kripke structure is *internalized* by UMoTs, ranged over by κ. Formally, a UMoT is just an (Agda) function of type $\mathbb{N} \to \mathbb{N}$, modeling a *stream* of modal offsets. Given a domain value $a \in D$ and a UMoT κ, the UMoT application operation $a[\kappa]$ applies κ to a and denotes sending a to another world according to κ. The internalization further requires subsequent models to satisfy *monotonicity* w.r.t. UMoTs (see Sections 6.2.3 and 7.3.1), which is the root of other properties involving UMoTs. The internalization seems to provide certain modularity, and we conjecture that our proof can be adapted to other modalities in Kripke style by only adjusting the definition of UMoTs and the proof of monotonicity. In fact, UMoTs have sufficiently captured the Kripke structures of all Systems K, T, $K4$, and $S4$, so as a bonus, their normalization is established simultaneously.

Our approach is in contrast to the *external* approach by Gratzer *et al.* (2019), where the model is parameterized by an extra layer of poset. Subsequent proofs thus must explicitly quantify over this poset, making their proof more difficult to adapt to other modalities (Gratzer *et al.*, 2020).

Before defining applications of UMoTs, we first define the following operations:

$$\begin{array}{lll}
\text{Truncation of UMoTs} & & \\
\_ \mid \_ & :: \text{UMoT} \to \mathbb{N} \to \text{UMoT} \\
(\kappa \mid n)\,(m) & := \kappa(n+m) \\
\text{Truncation Offset of UMoTs} & & \\
\mathcal{O}(\_,\_) & :: \text{UMoT} \to \mathbb{N} \to \mathbb{N} \\
\mathcal{O}(\kappa, 0) & := 0 \\
\mathcal{O}(\kappa, 1+n) & := \kappa(0) + \mathcal{O}(\kappa \mid 1, n)
\end{array}
\qquad
\begin{array}{lll}
\text{Identity UMoT} & & \\
\vec{1} & :: \text{UMoT} \\
\vec{1}\ (\_) & := 1 \\
\text{Cons of UMoTs} & & \\
\_;\Uparrow^{-} & :: \text{UMoT} \to \mathbb{N} \to \text{UMoT} \\
(\kappa;\Uparrow^{n})\,(0) & := n \\
(\kappa;\Uparrow^{n})\,(m) & := \kappa(1+m)
\end{array}$$

$$\begin{array}{ll}
\text{Composition of UMoTs} \quad \_\circ\_ & :: \text{UMoT} \to \text{UMoT} \to \text{UMoT} \\
(\kappa \circ \kappa')\,(0) & := \mathcal{O}(\kappa', \kappa(0)) \\
(\kappa \circ \kappa')\,(1+n) & := ((\kappa \mid 1) \circ (\kappa' \mid \kappa(0)))(n)
\end{array}$$

Cons of UMoTs is defined in a way similar to environments. UMoTs have composition, and we use $\vec{1}$ to represent the identity UMoT in our setting. As previously mentioned, UMoTs form a substitutional truncoid. A quick intuition is that UMoTs behave like substitutions in the semantics, except that it only brings values from one world to another without touching the variables. In the following, we give the definitions for applying a UMoT to domain values and environments:

$$\begin{array}{rcl}
\blacksquare A[\kappa] & := & \blacksquare(A[\kappa;\Uparrow^{1}]) \\
Pi(A, T, \overrightarrow{\rho})[\kappa] & := & Pi(A[\kappa], T, \overrightarrow{\rho}\,[\kappa]) \\
\mathsf{U}_i[\kappa] & := & \mathsf{U}_i \\
\mathtt{box}(a)[\kappa] & := & \mathtt{box}(a[\kappa;\Uparrow^{1}]) \\
\Lambda(t, \overrightarrow{\rho})[\kappa] & := & \Lambda(t, \overrightarrow{\rho}\,[\kappa]) \\
\uparrow^{A}(c)[\kappa] & := & \uparrow^{A[\kappa]}(c[\kappa]) \\
\downarrow^{A}(a)[\kappa] & := & \downarrow^{A[\kappa]}(a[\kappa])
\end{array}
\qquad
\begin{array}{rcl}
l_z[\kappa] & := & l_z \\
c\ d[\kappa] & := & (c[\kappa])\,(d[\kappa]) \\
\mathtt{unbox}(n, c)[\kappa] & := & \mathtt{unbox}(\mathcal{O}(\kappa, n), c[\kappa \mid n]) \\
 & & \\
\overrightarrow{\rho}\,[\kappa](0) & := & (\mathcal{O}(\kappa, n), \rho[\kappa]) \\
 & & \quad (\text{where } (n, \rho) := \overrightarrow{\rho}\,(0)) \\
\overrightarrow{\rho}\,[\kappa](1+n) & := & \overrightarrow{\rho} \mid 1[\kappa \mid \mathcal{O}(\overrightarrow{\rho}, 1)](n)
\end{array}$$

Following our previous convention, we let $a[\kappa]$ take a very low parsing precedence. Most cases just recursively push κ inwards, except the $\blacksquare$, `box` and `unbox` cases. The cases for

$\blacksquare$ and box are similar. $\kappa; \Uparrow^1$ instructs their recursions to enter a new world, indicated by cons'ing 1 to κ. The unbox case is similar to the rule in Section 4.2. The recursion is $c[\kappa \mid n]$ because c is in the n-th previous world. The unbox level is adjusted to $\mathcal{O}(\kappa, n)$ for coherence with the Kripke structure. The apply operation for a local environment ρ is just defined by applying κ to all values within pointwise. The apply operation for $\overrightarrow{\rho}$ can be motivated by making the triple (Envs, UMoT, Envs) an applicative truncoid. Indeed, this is the unique definition (up to isomorphism) to prove $\overrightarrow{\rho}[\kappa] \mid n = (\overrightarrow{\rho} \mid n)[\kappa \mid \mathcal{O}(\overrightarrow{\rho}, n)]$. From here, we can see the formulation of truncoids does guide us very quickly to the right definition of operations. Moreover, since UMoTs are substitutional, we would hope that Envs is closed under UMoTs. Indeed, this fact can be examined by checking the necessary laws imposed by a closed truncoid.

5.3 Evaluation

Next we consider the evaluation functions ($\llbracket \_ \rrbracket$), which, given $\overrightarrow{\rho}$, evaluates a syntactic term to a domain value or evaluates a K-substitution to another environment. In the following sections, we will define a number of partial functions (denoted by $\rightharpoonup$), which cannot be directly formalized in Agda. Instead, we define them as relations between inputs and outputs, and we prove that given the same inputs, the outputs are equal. When we refer to a result of a partial function, we implicitly existentially quantify this result for brevity.

$$
\begin{aligned}
\llbracket \_ \rrbracket &:: \mathsf{Trm} \rightharpoonup \mathsf{Envs} \to D \\
\llbracket \mathtt{Ty}_i \rrbracket(\overrightarrow{\rho}) &:= \mathsf{U}_i \\
\llbracket \square T \rrbracket(\overrightarrow{\rho}) &:= \blacksquare(\llbracket T \rrbracket(\mathtt{ext}(\overrightarrow{\rho}))) \\
\llbracket \Pi S.T \rrbracket(\overrightarrow{\rho}) &:= Pi(\llbracket S \rrbracket(\overrightarrow{\rho}), T, \overrightarrow{\rho}) \\
\llbracket v_x \rrbracket(\overrightarrow{\rho}) &:= \rho(x) \qquad (\text{where } (\_, \rho) := \overrightarrow{\rho}(0)) \\
\llbracket \mathtt{box}\ t \rrbracket(\overrightarrow{\rho}) &:= \mathtt{box}(\llbracket t \rrbracket(\mathtt{ext}(\overrightarrow{\rho}))) \\
\llbracket \mathtt{unbox}_n\ t \rrbracket(\overrightarrow{\rho}) &:= \mathtt{unbox} \cdot (\mathcal{O}(\overrightarrow{\rho}, n), \llbracket t \rrbracket(\overrightarrow{\rho} \mid n)) \\
\llbracket \lambda t \rrbracket(\overrightarrow{\rho}) &:= \Lambda(t, \overrightarrow{\rho}) \\
\llbracket t\ s \rrbracket(\overrightarrow{\rho}) &:= \llbracket t \rrbracket(\overrightarrow{\rho}) \cdot \llbracket s \rrbracket(\overrightarrow{\rho}) \\
\llbracket t[\overrightarrow{\sigma}] \rrbracket(\overrightarrow{\rho}) &:= \llbracket t \rrbracket(\llbracket \overrightarrow{\sigma} \rrbracket(\overrightarrow{\rho}))
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \_ \rrbracket &:: \mathsf{Substs} \rightharpoonup \mathsf{Envs} \to \mathsf{Envs} \\
\llbracket \overrightarrow{I} \rrbracket(\overrightarrow{\rho}) &:= \overrightarrow{\rho} \\
\llbracket \mathsf{wk} \rrbracket(\overrightarrow{\rho}) &:= \mathtt{drop}(\overrightarrow{\rho}) \\
\llbracket \overrightarrow{\sigma}, t \rrbracket(\overrightarrow{\rho}) &:= \mathtt{lext}(\llbracket \overrightarrow{\sigma} \rrbracket(\overrightarrow{\rho}), \llbracket t \rrbracket(\overrightarrow{\rho})) \\
\llbracket \overrightarrow{\sigma}; \Uparrow^n \rrbracket(\overrightarrow{\rho}) &:= \mathtt{ext}(\llbracket \overrightarrow{\sigma} \rrbracket(\overrightarrow{\rho} \mid n), \mathcal{O}(\overrightarrow{\rho}, n)) \\
\llbracket \overrightarrow{\sigma} \circ \overrightarrow{\delta} \rrbracket(\overrightarrow{\rho}) &:= \llbracket \overrightarrow{\sigma} \rrbracket(\llbracket \overrightarrow{\delta} \rrbracket(\overrightarrow{\rho}))
\end{aligned}
$$

Note that here (Substs, Envs, Envs) forms an applicative truncoid when the evaluation terminates. In the evaluation of Trm, we make use of partial functions, which evaluate box and Λ. These partial functions are defined below:

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

$$\begin{aligned}
\texttt{unbox}\cdot &:: \mathbb{N} \to D \rightharpoonup D \\
\texttt{unbox}\cdot(n, \texttt{box}(a)) &:= a[\overrightarrow{1}\,; \Uparrow^n] \\
\texttt{unbox}\cdot(n, \uparrow^{\blacksquare A}(c)) &:= \uparrow^{A[\overrightarrow{1};\Uparrow^n]}(\texttt{unbox}(n, c))
\end{aligned}$$

$$\begin{aligned}
\_\cdot\_ &:: D \rightharpoonup D \rightharpoonup D \\
(\Lambda(t, \overrightarrow{\rho}))\cdot a &:= [\![t]\!](\texttt{lext}(\overrightarrow{\rho}, a)) \\
(\uparrow^{Pi(A,T,\overrightarrow{\rho})}(c))\cdot a &:= \uparrow^{[\![T]\!](\texttt{lext}(\overrightarrow{\rho},a))}(c\ \downarrow^A(a))
\end{aligned}$$

Effectively, these partial functions remove all β redexes.

5.4 Readback functions

After evaluating a Trm to D, we have already got the corresponding β normal form in D. We need one last step, readback functions, to read from D back to normal form and do the η expansion at the same time to obtain a $\beta\eta$ normal form:

$$\begin{aligned}
\mathsf{R}^{\mathsf{Nf}} &:: \overrightarrow{\mathbb{N}} \rightharpoonup D^{\mathsf{Nf}} \rightharpoonup \mathsf{Nf} \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\mathsf{U}_i}(A)) &:= \mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(A) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\uparrow^A(c)}(\uparrow^{A'}(c'))) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c') \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\blacksquare A}(a)) &:= \texttt{box}\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};0}(\downarrow^A(\texttt{unbox}\cdot(1, a))) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};z}(\downarrow^{Pi(A,T,\overrightarrow{\rho})}(a)) &:= \lambda \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};1+z}(\downarrow^{[\![T]\!](\texttt{lext}(\overrightarrow{\rho},\uparrow^A(l_z)))}(a\cdot\uparrow^A(l_z)))
\end{aligned}$$

$$\begin{aligned}
\mathsf{R}^{\mathsf{Ty}} &:: \overrightarrow{\mathbb{N}} \rightharpoonup D \rightharpoonup \mathsf{Nf} \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(\mathsf{U}_i) &:= \texttt{Ty}_i \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(\blacksquare A) &:= \square \mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};0}(A) \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};z}(Pi(A, T, \overrightarrow{\rho})) &:= \Pi(\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};z}(A)).\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};1+z}([\![T]\!](\texttt{lext}(\overrightarrow{\rho}, \uparrow^A(l_z)))) \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(\uparrow^A(c)) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c)
\end{aligned}$$

$$\begin{aligned}
\mathsf{R}^{\mathsf{Ne}} &:: \overrightarrow{\mathbb{N}} \rightharpoonup D^{\mathsf{Ne}} \rightharpoonup \mathsf{Ne} \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z};z'}(l_z) &:= v_{\max(z'-z-1,0)} \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c\ d) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c)\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(d) \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(\texttt{unbox}(n, c)) &:= \texttt{unbox}_n\ \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}|n}(c)
\end{aligned}$$

A readback function takes as an argument $\overrightarrow{z} :: \overrightarrow{\mathbb{N}}$, which is a nonempty list of natural numbers. Each number in this list records the length of the context in that position of the context stack. This list supplies new de Bruijn levels (i.e. new absolute and fresh names) as the readback process continues. In the $\blacksquare$ case, since we enter a new world, 0 is pushed to the list because the new world has no assumption. In the Pi case, the topmost context is extended by one due to the argument of λ, so we also increment the topmost de Bruijn level by one. In the unbox case, we must truncate $\overrightarrow{z}$ in order to correctly keep track of the lengths of contexts in the stack because the context stack is also truncated. In the variable case in R^{Ne}, we use the aforementioned formula $z' - z - 1$ to compute the corresponding de Bruijn index in the syntax. If we begin with a well-typed term, then this formula is always non-negative, so we do not have to consider the case where we are cut off at 0.

The readback process consists of three functions: R^{Nf} reads back a normal form; R^{Ne} reads back a neutral form; and R^{Ty} reads back a normal type. Note that R^{Nf} is type-directed, so η expansion is performed. With evaluation and readback, we are ready to give the definition of the NbE algorithm by first evaluating a term and its type to the domain and then read back as a normal form:

Definition 5.1. For $\vec{\Gamma} \vdash t : T$, the NbE algorithm is

$$\mathsf{nbe}^{T}_{\vec{\Gamma}}(t) := \mathsf{R}^{\mathsf{Nf}}_{\mathtt{map}(\Gamma \mapsto |\Gamma|, \vec{\Gamma})}(\downarrow^{[\![T]\!](\uparrow^{\vec{\Gamma}})}([\![t]\!](\uparrow^{\vec{\Gamma}})))$$

where the initial environment $\uparrow^{\vec{\Gamma}}$ is defined by the structure of $\vec{\Gamma}$:

$$\begin{aligned}
\uparrow &:: \overrightarrow{\mathsf{Ctx}} \rightharpoonup \mathsf{Envs} \\
\uparrow^{\varepsilon;\cdot} &:= \mathtt{empty} \\
\uparrow^{\vec{\Gamma};\cdot} &:= \mathtt{ext}(\uparrow^{\vec{\Gamma}}) \\
\uparrow^{\vec{\Gamma};(\Gamma.T)} &:= \mathtt{lext}(\vec{\rho}, \uparrow^{[\![T]\!](\vec{\rho})}(l_{|\Gamma|})) && \text{(where } \vec{\rho} := \uparrow^{\vec{\Gamma};\Gamma}\text{)}
\end{aligned}$$

We have an elaborated example for running the NbE algorithm in Appendix B.

6 PER model and completeness

In the previous section, we have given the full definition of the NbE algorithm. In this section, we follow Abel (2013) and define a PER model for the untyped domain and prove the completeness theorem, which states that equivalent terms evaluate to an equal normal form. There are two steps to establish completeness: (1) the fundamental theorems, which prove soundness of the PER model, and (2) the realizability theorem, which states that values related by the PER model have an equal normal form. Similar to other NbE proofs for dependent types (Abel, 2013; Abel *et al.*, 2017; Kaposi & Altenkirch, 2017; Wieczorek & Biernacki, 2018; Gratzer *et al.*, 2019), the soundness proof of NbE relies on the fundamental theorems of the PER model, so the PER model is a prerequisite for the next section.

Due to our intention of staying close to our mechanization, we assign names to judgments, ranging from $\mathscr{D}$, $\mathscr{E}$, and $\mathscr{J}$.

6.1 PER model

We follow Abel (2013) and first introduce the following relations.

$$\frac{\forall \vec{z}, \kappa. \mathsf{R}^{\mathsf{Nf}}_{\vec{z}}(d[\kappa]) = \mathsf{R}^{\mathsf{Nf}}_{\vec{z}}(d'[\kappa])}{d \approx d' \in \mathit{Nf}} \qquad \frac{\forall \vec{z}, \kappa. \mathsf{R}^{\mathsf{Ty}}_{\vec{z}}(A[\kappa]) = \mathsf{R}^{\mathsf{Ty}}_{\vec{z}}(A'[\kappa])}{A \approx A' \in \mathit{Ty}}$$

$$\frac{\forall \vec{z}, \kappa. \mathsf{R}^{\mathsf{Ne}}_{\vec{z}}(c[\kappa]) = \mathsf{R}^{\mathsf{Ne}}_{\vec{z}}(c'[\kappa])}{c \approx c' \in \mathit{Ne}}$$

where $\mathit{Nf} \subseteq D^{\mathsf{Nf}} \times D^{\mathsf{Nf}}$, $\mathit{Ty} \subseteq D \times D$ and $\mathit{Ne} \subseteq D^{\mathsf{Ne}} \times D^{\mathsf{Ne}}$. Nf relates two normal domain values iff their readbacks are equal given any context stack and UMoT. Ty and Ne are defined similarly. We will show by realizability (Section 6.2.5) that all forthcoming PERs are subsumed by Nf, so any two related values have equal readbacks as normal forms,

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

through which completeness is established. These relations are part of *dependently typed candidate space* introduced by Abel (2013). Since these relations are defined by equality, they are indeed PERs.

The premises of these PERs are all universally quantified over UMoTs. This universal quantification is and continues to be crucial in our PERs. Due to the substitutional truncoid structure of UMoTs, particularly composition, universally quantifying UMoTs allows us to *internalize* the Kripke structure and delegate the handling of the said structure to UMoTs, so that the proof structure is completely oblivious to the exact structure $\square$ possesses, making our constructions very adaptive as explained in Section 5.2.

Now we move on to define the actual PERs that relate domain values. The PER model consists of two PERs: $\mathscr{U}_i$ which denotes a universe and relates two domain types at level i, and $\mathbf{El}_i(\mathscr{D})$ which given $\mathscr{D} :: A \approx B \in \mathscr{U}_i$ relates two domain values of domain types A and B. The PER model resembles Tarski-style universes (Palmgren, 1998), in which universes contain "codes" and **El** converts these codes into actual types which contain values. Following Abel (2013) and Abel *et al.* (2017), $\mathscr{U}_i$ and $\mathbf{El}_i$ are defined inductive-recursively (Dybjer, 2000). Moreover, due to cumulative universes, they must in addition be defined with the well-foundedness of the universe levels. We refer interested readers to our Agda code for our actual formalization of these two relations.

Definition 6.1. The equivalence for domain types $\mathscr{D} :: A \approx B \in \mathscr{U}_i$ and the equivalence for domain values $a \approx b \in \mathbf{El}_i(\mathscr{D})$ are defined as follows:

- Neutral types and neutral values:

$$\mathscr{D} := \frac{C \approx C' \in Ne}{\uparrow^{A}(C) \approx \uparrow^{A'}(C') \in \mathscr{U}_i}$$

Then $a \approx b \in \mathbf{El}_i(\mathscr{D})$ iff $a \approx b \in Neu$, where Neu relates two values only when they are actually neutral:

$$\frac{c \approx c' \in Ne}{\uparrow^{A_1}(c) \approx \uparrow^{A_2}(c') \in Neu}$$

Note that the annotating domain types A_1 and A_2 do not matter as long as c and c' are related by Ne.

- Universes:

$$\mathscr{D} := \frac{j < i}{\mathsf{U}_j \approx \mathsf{U}_j \in \mathscr{U}_i}$$

Then $a \approx b \in \mathbf{El}_i(\mathscr{D})$ iff $a \approx b \in \mathscr{U}_j$. Note that here $\mathbf{El}_i(\mathscr{D})$ is defined in terms of $\mathscr{U}_j$. This is fine because of $j < i$ and the well-foundedness of universe levels.

- Semantic $\blacksquare$ types:

$$\mathscr{D} := \frac{\mathscr{J} :: \forall \kappa.\, A[\kappa] \approx A'[\kappa] \in \mathscr{U}_i}{\blacksquare A \approx \blacksquare A' \in \mathscr{U}_i}$$

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Then $a \approx b \in \mathbf{El}_i(\mathscr{D})$ iff for any UMoT κ and `unbox` level n, the `unboxing`'s of $a[\kappa]$ and $b[\kappa]$ remain related: $\texttt{unbox} \cdot (n, a[\kappa]) \approx \texttt{unbox} \cdot (n, b[\kappa]) \in \mathbf{El}_i(\mathscr{J}(\kappa; \Uparrow^n))$. In other words, if a and b are still related no matter how they travel in Kripke worlds and then are `unbox`'ed, then they are related by $\mathbf{El}_i(\mathscr{D})$.

- Semantic *Pi* types:

$$\mathscr{D} :=$$

$$\frac{\begin{array}{c}\mathscr{J}_1 :: \forall \kappa .\, A[\kappa] \approx A'[\kappa] \in \mathscr{U}_i \\ \mathscr{J}_2 :: \forall \kappa, a \approx a' \in \mathbf{El}_i(\mathscr{J}_1[\kappa]).\ [\![T]\!](\texttt{lext}(\overrightarrow{\rho}[\kappa], a)) \approx [\![T']\!](\texttt{lext}(\overrightarrow{\rho}'[\kappa], a')) \in \mathscr{U}_i\end{array}}{Pi(A, T, \overrightarrow{\rho}) \approx Pi(A', T', \overrightarrow{\rho}') \in \mathscr{U}_i}$$

Then $a \approx b \in \mathbf{El}_i(\mathscr{D})$ iff for any UMoT κ and related a' and b', i.e., $\mathscr{E} :: a' \approx b' \in \mathbf{El}_i(\mathscr{J}_1(\kappa))$, the results of applying $a[\kappa]$ and $b[\kappa]$ remain related: $a[\kappa] \cdot a' \approx b[\kappa] \cdot b' \in \mathbf{El}_i(\mathscr{J}_2(\kappa, \mathscr{E}))$. That is, a and b are related if all results of applying them in other worlds to related values are still related.

6.2 Properties for PERs

During mechanization, we are forced to make everything precise and type theory friendly and observe certain gaps between a set-theoretic proof and a type-theoretic one. In this section, we discuss how properties of our PERs are formulated and proved in a type-theoretic flavor. Our mechanization also exposes some oversimplifications about universes that are common in on-paper, set theoretic NbE proofs (Abel, 2013; Abel *et al.*, 2017; Gratzer *et al.*, 2019) in Section 6.2.4.

6.2.1 $\mathscr{U}$ irrelevance

While it comes for free in paper proofs, we must prove the intuition that $\mathbf{El}_i$ only relies on A and B in $A \approx B \in \mathscr{U}_i$, not how exactly they are related by $\mathscr{U}_i$. Effectively, we would like to show that $\mathscr{U}$ is proof-irrelevant:

Lemma 6.1 ($\mathscr{U}$ irrelevance)**.** *Given $\mathscr{D} :: A \approx B \in \mathscr{U}_i$ and $a \approx b \in \boldsymbol{El}_i(\mathscr{D})$,*

- *if $\mathscr{E}_1 :: A \approx B' \in \mathscr{U}_i$, then $a \approx b \in \boldsymbol{El}_i(\mathscr{E}_1)$;*
- *if $\mathscr{E}_2 :: A' \approx B \in \mathscr{U}_i$, then $a \approx b \in \boldsymbol{El}_i(\mathscr{E}_2)$.*

This lemma matches our set-theoretic intuition and states that a and b are related as long as we know they are related by one "representative" domain type.

*6.2.2 $\mathscr{U}$ and **El** are PERs*

When proving $\mathscr{U}$ and **El** being PERs, we were already facing strong scrutiny from Agda's termination checker, so we must adjust our statements to more type-theoretic ones. To even state symmetry, we have to introduce an extra premise $\mathscr{D}_2$. This extra assumption $\mathscr{D}_2$ exposes a clearer termination measure and allows Agda to admit our proof just by recognizing decreasing structures.

Lemma 6.2 (Symmetry). *Given $\mathscr{D}_1 :: A \approx B \in \mathscr{U}_i$,*

- *$B \approx A \in \mathscr{U}_i$;*
- *if $\mathscr{D}_2 :: B \approx A \in \mathscr{U}_i$, and $a \approx b \in \textbf{\textit{El}}_i(\mathscr{D}_1)$, then $b \approx a \in \textbf{\textit{El}}_i(\mathscr{D}_2)$.*

Proof Induction on i and $\mathscr{D}_1$ and inversion on $\mathscr{D}_2$ in the second statement. ■

The symmetry of **El** requires two $\mathscr{U}_i$ derivations: $\mathscr{D}_1$ relating A and B, and $\mathscr{D}_2$ relating B and A. They are used in the premise and the conclusion, respectively. $\mathscr{D}_2$ is important to handle the contravariance of the input and the output in the function case. $\mathscr{D}_2$ can be eventually discharged by combining the symmetry of $\mathscr{U}_i$ and irrelevance, but it is necessary to prove the lemma in a type-theoretic flavor.

Transitivity also requires a similar treatment but more complex:

Lemma 6.3 (Transitivity). *Given $\mathscr{D}_1 :: A_1 \approx A_2 \in \mathscr{U}_i$ and $\mathscr{D}_2 :: A_2 \approx A_3 \in \mathscr{U}_i$,*

- *$A_1 \approx A_3 \in \mathscr{U}_i$;*
- *if $\mathscr{D}_3 :: A_1 \approx A_3 \in \mathscr{U}_i$, $A_1 \approx A_1 \in \mathscr{U}_i$, and $a_1 \approx a_2 \in \textbf{\textit{El}}_i(\mathscr{D}_1)$ and $a_2 \approx a_3 \in \textbf{\textit{El}}_i(\mathscr{D}_2)$, then $a_1 \approx a_3 \in \textbf{\textit{El}}_i(\mathscr{D}_3)$.*

In addition to $\mathscr{D}_3$ which is used in $\mathbf{El}_i(\mathscr{D}_3)$ in conclusion, reflexivity of A_1, $A_1 \approx A_1 \in \mathscr{U}_i$, is also a required assumption. This is again due to the function case. Prior to establishing transitivity, we do not have reflexivity, so it is easier to make $A_1 \approx A_1 \in \mathscr{U}_i$ an extra assumption. Again, Agda admits our proof just by decreasing structures.

6.2.3 Monotonicity

As explained in Section 5.2, our PERs must respect UMoTs, i.e., are monotonic. Monotonicity ensures that the Kripke structure is successfully internalized, so subsequent proofs are unaware of the exact modality we are handling, making our proof structure very general. Due to the composition of UMoTs, all properties describing the Kripke structure eventually boil down to monotonicity.

Lemma 6.4 (Monotonicity). *Given $\mathscr{D} :: A \approx B \in \mathscr{U}_i$ and a UMoT κ,*

- *$A[\kappa] \approx B[\kappa] \in \mathscr{U}_i$;*
- *if $\mathscr{E} :: A[\kappa] \approx B[\kappa] \in \mathscr{U}_i$ and $a \approx b \in \textbf{\textit{El}}_i(\mathscr{D})$, then $a[\kappa] \approx b[\kappa] \in \textbf{\textit{El}}_i(\mathscr{E})$.*

Similar to symmetry and transitivity, we also require $\mathscr{E}$ in the second statement to ease termination checking in Agda.

6.2.4 Cumulativity and lowering

While there is existing work on NbE proofs of dependent type theories (Abel, 2013; Abel *et al.*, 2017; Gratzer *et al.*, 2019) which considers full cumulative universe hierarchies, they all slightly oversimplify the proofs of the cumulativity of their PER models. Cumulativity states that if two types or values are related at level i, then they are also related at level $1 + i$. At first glance, this property is too intuitive to be worth looking into. However, if we

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

carefully consider the function case, we see that in the proof, we assume two related values at level $1 + i$, but our premise requires them to be related at level i, and we are stuck. To correctly prove cumulativity, we must mutually prove a *lowering* lemma:

Lemma 6.5 (Cumulativity and lowering). *If $\mathscr{D} :: A \approx B \in \mathscr{U}_i$,*

- *then $\mathscr{D}' :: A \approx B \in \mathscr{U}_{i+1}$;*
- *if $a \approx b \in \boldsymbol{El}_i(\mathscr{D})$, then $a \approx b \in \boldsymbol{El}_{i+1}(\mathscr{D}')$;*
- *if $a \approx b \in \boldsymbol{El}_{i+1}(\mathscr{D}')$, then $a \approx b \in \boldsymbol{El}_i(\mathscr{D})$.*

Note that according to the last statement, we can lower from $\mathbf{El}_{1+i}$ to $\mathbf{El}_i$ if we know A and B are related at level i. In general, lowering is necessary for type constructors in which types can occur in negative positions. It is this lowering property that is being somewhat glossed over in prior work.

6.2.5 Realizability

Following Abel (2013), we prove the realizability theorem, which states that related values are read back equal. Realizability of the PER model is essential to establish the completeness and the soundness theorems. More formally,

Theorem 6.6 (Realizability). *Given $\mathscr{D} ::\ A \approx B \in \mathscr{U}_i$,*

- *$A \approx B \in Ty$;*
- *if $c \approx c' \in Ne$, then $\uparrow^A (c) \approx \uparrow^B (c') \in \boldsymbol{El}_i(\mathscr{D})$;*
- *if $a \approx b \in \boldsymbol{El}_i(\mathscr{D})$, then $\downarrow^A (c) \approx \downarrow^B (c') \in Nf$.*

Proof Induction on i and $\mathscr{D}$. ■

The first statement says that if A and B are related, then they are always read back as an equal normal type in syntax. The third statement says that if a and b are related, then they are always read back as an equal normal form in syntax. Combined with the fundamental theorems which we are about to give in Section 6.4, we prove completeness.

6.3 PER model for context stacks and environments

To define semantic judgments that are used to state fundamental theorems, we extend the PER model to context stacks and environments.

Definition 6.2. The equivalence of context stacks $\mathscr{D} :: \vDash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}$ and the equivalence of evaluation environments $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathscr{D}]\!]$ are defined inductive-recursively as follows:

- $$\mathscr{D} := \frac{}{\vDash \varepsilon; \cdot \approx \varepsilon; \cdot}$$

  Then $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathscr{D}]\!]$ is always true.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

- $$\mathcal{D} := \frac{\mathcal{J} :: \vDash \vec{\Gamma} \approx \vec{\Gamma}'}{\vDash \vec{\Gamma}; \cdot \approx \vec{\Gamma}'; \cdot}$$

  Then $\vec{\rho} \approx \vec{\rho}' \in [\![\mathcal{D}]\!]$ iff

  - truncations of $\vec{\rho}$ and $\vec{\rho}'$ by one are recursively related: $\vec{\rho} \mid 1 \approx \vec{\rho}' \mid 1 \in [\![\mathcal{J}]\!]$, and
  - first modal offsets are equal: $n = n'$ where $(n, \_) := \vec{\rho}(0)$ and $(n', \_) := \vec{\rho}'(0)$.

- $$\mathcal{D} := \frac{\mathcal{J}_1 :: \vDash \vec{\Gamma}; \Gamma \approx \vec{\Gamma}'; \Gamma' \qquad \mathcal{J}_2 :: \forall \vec{\rho} \approx \vec{\rho}' \in [\![\mathcal{J}_1]\!].[\![T]\!](\vec{\rho}) \approx [\![T']\!](\vec{\rho}') \in \mathcal{U}_i}{\vDash \vec{\Gamma}; \Gamma.T \approx \vec{\Gamma}'; \Gamma'.T}$$

  Then $\vec{\rho} \approx \vec{\rho}' \in [\![\mathcal{D}]\!]$ iff

  - $\texttt{drop}(\vec{\rho})$ and $\texttt{drop}(\vec{\rho}')$ are recursively related: $\mathcal{E} :: \texttt{drop}(\vec{\rho}) \approx \texttt{drop}(\vec{\rho}') \in [\![\mathcal{J}_1]\!]$, and
  - the topmost values are related by the evaluations of T: $\rho(0) \approx \rho'(0) \in \mathbf{El}_i(\mathcal{J}_2(\mathcal{E}))$ where $(\_, \rho) := \vec{\rho}(0)$ and $(\_, \rho') := \vec{\rho}'(0)$.

We can extend properties in Section 6.2 to context stacks and environments as well, e.g., irrelevance, symmetry, transitivity, and monotonicity.

6.4 Semantic judgments, fundamental theorems, and completeness

Having defined all PER models, we are ready to define the semantic judgments:

Definition 6.3. We define semantic judgments as follows:

$$\frac{\vDash \vec{\Gamma} \approx \vec{\Gamma}}{\vDash \vec{\Gamma}} \qquad \frac{\vec{\Gamma} \vDash t \approx t : T}{\vec{\Gamma} \vDash t : T} \qquad \frac{\vec{\Gamma} \vDash \vec{\sigma} \approx \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vDash \vec{\sigma} : \vec{\Delta}}$$

where $\vec{\Gamma} \vDash t \approx t' : T$ iff

- $\vec{\Gamma}$ is semantically well formed: $\mathcal{D} :: \vDash \vec{\Gamma}$, and
- there exists a universe level i, such that for any related $\vec{\rho}$ and $\vec{\rho}'$, i.e., $\mathcal{E} :: \vec{\rho} \approx \vec{\rho}' \in [\![\mathcal{D}]\!]$,

  - evaluations of T are related types: $\mathcal{J} :: [\![T]\!](\vec{\rho}) \approx [\![T]\!](\vec{\rho}') \in \mathcal{U}_i$, and
  - evaluations of t and t' are related values: $[\![t]\!](\vec{\rho}) \approx [\![t']\!](\vec{\rho}') \in \mathbf{El}_i(\mathcal{J})$;

and $\vec{\Gamma} \vDash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Delta}$ iff

- $\vec{\Gamma}$ and $\vec{\Delta}$ are semantically well formed: $\mathcal{D} :: \vDash \vec{\Gamma}$ and $\mathcal{E} :: \vDash \vec{\Delta}$, and

- for any related $\vec{\rho}$ and $\vec{\rho}'$, i.e., $\vec{\rho} \approx \vec{\rho}' \in [\![\mathscr{D}]\!]$, $\vec{\sigma}$ and $\vec{\sigma}'$ evaluate to related environments: $[\![\vec{\sigma}]\!](\vec{\rho}) \approx [\![\vec{\sigma}']\!](\vec{\rho}') \in [\![\mathscr{E}]\!]$.

The fundamental theorem states that the semantic judgments are sound w.r.t. the syntactic judgments:

Theorem 6.7 (Fundamental).

- *If* $\vdash \vec{\Gamma}$, *then* $\vDash \vec{\Gamma}$.
- *If* $\vec{\Gamma} \vdash t : T$, *then* $\vec{\Gamma} \vDash t : T$.
- *If* $\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}$, *then* $\vec{\Gamma} \vDash \vec{\sigma} : \vec{\Delta}$.
- *If* $\vdash \vec{\Gamma} \approx \vec{\Gamma}'$, *then* $\vDash \vec{\Gamma} \approx \vec{\Gamma}'$.
- *If* $\vec{\Gamma} \vdash t \approx t' : T$, *then* $\vec{\Gamma} \vDash t \approx t' : T$.
- *If* $\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Delta}$, *then* $\vec{\Gamma} \vDash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Delta}$.

Proof Mutual induction on all premises (see the mechanization for full details) ■

The completeness theorem states that syntactic equivalent terms evaluate to equal normal forms:

Theorem 6.8 (Completeness). *If* $\vec{\Gamma} \vdash t \approx t' : T$, *then* $\mathsf{nbe}^{T}_{\vec{\Gamma}}(t) = \mathsf{nbe}^{T}_{\vec{\Gamma}}(t')$.

Proof By the fundamental theorems, we have $\vec{\Gamma} \vDash t \approx t' : T$ from which we know $\mathscr{D} :: \vDash \vec{\Gamma}$. We have that the initial environment is related: $\uparrow^{\vec{\Gamma}} \approx \uparrow^{\vec{\Gamma}} \in [\![\mathscr{D}]\!]$. We further learn that $[\![t]\!](\uparrow^{\vec{\Gamma}})$ and $[\![t']\!](\uparrow^{\vec{\Gamma}})$ are related, and due to realizability, they have equal normal forms. ■

6.5 Consequences of completeness

From the fundamental theorems of the PER model, we can derive a few consequences, which are useful but very challenging to prove syntactically:

Lemma 6.9 (Equal universe levels). *If* $\vec{\Gamma} \vdash \mathtt{Ty}_i \approx \mathtt{Ty}_j : \mathtt{Ty}_k$, *then* $i = j$.

Proof Completeness says that Ty_i and Ty_j have equal normal form, which implies $i = j$. ■

Due to the previous lemma, we can prove the following "exact inversion" lemma:

Lemma 6.10 (Exact inversion).

- *If* $\vec{\Gamma} \vdash \square T : \mathtt{Ty}_i$, *then* $\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i$.
- *If* $\vec{\Gamma}; \Gamma \vdash \Pi S.T : \mathtt{Ty}_i$, *then* $\vec{\Gamma}; \Gamma \vdash S : \mathtt{Ty}_i$ *and* $\vec{\Gamma}; \Gamma.S \vdash T : \mathtt{Ty}_i$.

If we proceed by induction on the premise, the universe levels in the conclusion cannot be made exactly i. However, this is now possible with the fundamental theorems.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

7 Kripke gluing model and soundness

In the previous section, we have established the completeness theorem. In this section, we present the soundness proof for the NbE algorithm, which states that a well-typed term is equivalent to its evaluated normal form. Central to the proof is a *Kripke* gluing model. The model glues the syntax and the semantics (Coquand & Dybjer, 1997) and is Kripke because it is monotonic w.r.t. a special class of K-substitutions, restricted weakenings. Similar to completeness, the proof of soundness also has two steps: (1) the fundamental theorems and (2) the realizability theorem stating that a term is equivalent to the readback of its related value. We first give the definitions of restricted weakenings and the gluing model.

7.1 Restricted weakening

Restricted weakenings are a special class of K-substitutions, which characterize the possible changes in context stacks *during evaluation*. Therefore, if a term and a value are related, their relation must be stable under restricted weakenings, hence introducing a Kripke structure to the gluing model.

Definition 7.1. A K-substitution $\vec{\sigma}$ is a restricted weakening if it inductively satisfies

$$\frac{\Gamma \vdash \vec{\sigma} \approx \vec{I} : \vec{\Delta}}{\vec{\Gamma} \vdash_r \vec{\sigma} : \vec{\Delta}} \qquad \frac{\vec{\Gamma} \vdash_r \vec{\sigma}' : \vec{\Delta}; \Delta.T \qquad \Gamma \vdash \vec{\sigma} \approx \mathsf{wk} \circ \vec{\sigma}' : \vec{\Delta}; \Delta}{\vec{\Gamma} \vdash_r \vec{\sigma} : \vec{\Delta}; \Delta}$$

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

Effectively, a restricted weakening can only do either local weakenings (wk) or modal extensions (_; $\Uparrow^n$), because only these two cases are possible during evaluation. Since restricted weakenings form a category, we require the gluing model to respect them.

7.2 Gluing model

Restricted weakenings are already K-substitutions, so they naturally apply to syntactic terms. However, to define the gluing model, we must also apply them to domain values. This is achieved by UMoT extraction ($\mathtt{mt}(\_)$),[5] which extracts a UMoT from a K-substitution (not just a restricted weakening).

$$\begin{aligned} \mathtt{mt}(\_) &:: \mathsf{Substs} \to \text{UMoT} \\ \mathtt{mt}(\vec{I}\,) &:= \vec{1} \\ \mathtt{mt}(\vec{\sigma}, t) &:= \mathtt{mt}(\vec{\sigma}) \\ \mathtt{mt}(\mathsf{wk}) &:= \vec{1} \\ \mathtt{mt}(\vec{\sigma}; \Uparrow^n) &:= \mathtt{mt}(\vec{\sigma}); \Uparrow^n \\ \mathtt{mt}(\vec{\sigma} \circ \vec{\delta}\,) &:= \mathtt{mt}(\vec{\sigma}) \circ \mathtt{mt}(\vec{\delta}\,) \end{aligned}$$

[5] $\mathtt{mt}()$ stands for “we modal transform a K-substitution into a UMoT.”

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Moreover, for conciseness, given a K-substitution $\vec{\sigma}$ and a domain value a, we write $a[\vec{\sigma}]$ for $a[\mathtt{mt}(\vec{\sigma})]$ unless the distinction is worth emphasizing. Next, we define the gluing model that relates syntactic and domain terms and types. For a PER P, we write $p \in P$ for $p \approx p \in P$:

Definition 7.2. Given $\mathscr{D} :: A \approx B \in \mathscr{U}_i$,

- $\vec{\Gamma} \vdash T \circledR_i \mathscr{D}$ says that T is related to domain types A and B in $\vec{\Gamma}$ as a type at level i.
- $\vec{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathscr{D})$ says that in $\vec{\Gamma}$, t of type T is related to domain value a in $\mathbf{El}_i(\mathscr{D})$.

$\vec{\Gamma} \vdash T \circledR_i \mathscr{D}$ and $\vec{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathscr{D})$ are defined mutually by first well-founded recursion on i and then recursion on $\mathscr{D}$:

- $$\mathscr{D} := \frac{C \approx C' \in Ne}{\uparrow^{A}(C) \approx \uparrow^{A'}(C') \in \mathscr{U}_i}$$

  $\vec{\Gamma} \vdash T \circledR_i \mathscr{D}$ iff

  - T is a type at level i: $\Gamma \vdash T : \mathtt{Ty}_i$.
  - For any restricted weakening $\vec{\sigma}$, that is $\vec{\Delta} \vdash_r \vec{\sigma} : \vec{\Gamma}$, s.t.
  $$\vec{\Delta} \vdash T[\vec{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\vec{z}}(C[\vec{\sigma}]) : \mathtt{Ty}_i$$

  $\vec{\Gamma} \vdash t : T \circledR_i \uparrow^{A''}(c) \in \mathbf{El}_i(\mathscr{D})$ iff

  - $c \in Ne$.
  - T is a type at level i: $\Gamma \vdash T : \mathtt{Ty}_i$.
  - t is well typed: $\Gamma \vdash t : T$.
  - For any restricted weakening $\vec{\sigma}$, that is $\vec{\Delta} \vdash_r \vec{\sigma} : \vec{\Gamma}$, s.t.
  $$\vec{\Delta} \vdash T[\vec{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\vec{z}}(C[\vec{\sigma}]) : \mathtt{Ty}_i$$
  and
  $$\vec{\Delta} \vdash t[\vec{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\vec{z}}(c[\vec{\sigma}]) : T[\vec{\sigma}]$$

- $$\mathscr{D} := \frac{j < i}{\mathsf{U}_j \approx \mathsf{U}_j \in \mathscr{U}_i}$$

  $\vec{\Gamma} \vdash T \circledR_i \mathscr{D}$ iff $\vec{\Gamma} \vdash T \approx \mathtt{Ty}_j : \mathtt{Ty}_i$.

  $\vec{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathscr{D})$ iff

  - t is well typed: $\Gamma \vdash t : T$.
  - T is equivalent to $\mathtt{Ty}_j$: $\vec{\Gamma} \vdash T \approx \mathtt{Ty}_j : \mathtt{Ty}_i$.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

- a is in PER $\mathscr{U}_j$: $\mathscr{E} :: a \in \mathscr{U}_j$.
- t and a are recursively related as types by well-foundedness: $\overrightarrow{\Gamma} \vdash t \circledR_j \mathscr{E}$.

Note that $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathscr{D})$ refers to $\overrightarrow{\Gamma} \vdash t \circledR_j \mathscr{E}$, so the definition requires a well-founded recursion on i.

- $$\mathscr{D} := \frac{\mathscr{E} :: \forall \kappa\,.\, A'[\kappa] \approx A''[\kappa] \in \mathscr{U}_i}{\blacksquare A' \approx \blacksquare A'' \in \mathscr{U}_i}$$

  $\overrightarrow{\Gamma} \vdash T \circledR_i \mathscr{D}$ iff there exists some T', such that

  - T is equivalent to $\square T'$: $\overrightarrow{\Gamma} \vdash T \approx \square T' : \mathtt{Ty}_i$.
  - For any $\overrightarrow{\Delta}'$ such that $\vdash \overrightarrow{\Delta} ; \overrightarrow{\Delta}'$ and any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}$, $T'[\overrightarrow{\sigma} ; \Uparrow^{|\overrightarrow{\Delta}'|}]$ and $\mathscr{E}(\overrightarrow{\sigma} ; \Uparrow^{|\overrightarrow{\Delta}'|})$ are recursively related as types:
  $$\overrightarrow{\Delta} ; \overrightarrow{\Delta}' \vdash T'[\overrightarrow{\sigma} ; \Uparrow^{|\overrightarrow{\Delta}'|}] \circledR_i \mathscr{E}(\overrightarrow{\sigma} ; \Uparrow^{|\overrightarrow{\Delta}'|})$$

  $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathscr{D})$ iff there exists some T', such that

  - t is well typed: $\overrightarrow{\Gamma} \vdash t : T$.
  - a is in $\mathbf{El}_i(\mathscr{D})$: $a \in \mathbf{El}_i(\mathscr{D})$.
  - T is equivalent to $\square T'$: $\overrightarrow{\Gamma} \vdash T \approx \square T' : \mathtt{Ty}_i$.
  - For any $\overrightarrow{\Delta}'$ such that $\vdash \overrightarrow{\Delta} ; \overrightarrow{\Delta}'$ and any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}$, the results of eliminating $t[\overrightarrow{\sigma}]$ and $a[\overrightarrow{\sigma}]$ are recursively related as terms:
  $$\overrightarrow{\Delta} ; \overrightarrow{\Delta}' \vdash \mathtt{unbox}_{|\overrightarrow{\Delta}'|}\, (t[\overrightarrow{\sigma}]) : T'[\overrightarrow{\sigma} ; \Uparrow^{|\overrightarrow{\Delta}'|}] \circledR_i \mathtt{unbox} \cdot (|\overrightarrow{\Delta}'|, a[\overrightarrow{\sigma}]) \in \mathbf{El}_i(\mathscr{E}[\mathtt{mt}(\overrightarrow{\sigma}); \Uparrow^{|\overrightarrow{\Delta}'|}])$$

- $$\mathscr{D} := \frac{\begin{array}{c}\mathscr{E}_1 :: \forall \kappa\,.\, A'[\kappa] \approx A''[\kappa] \in \mathscr{U}_i \\ \mathscr{E}_2 :: \forall \kappa, a \approx a' \in \mathbf{El}_i(\mathscr{E}_1[\kappa]).\ [\![T']\!](\mathtt{lext}(\overrightarrow{\rho}'[\kappa], a)) \approx [\![T'']\!](\mathtt{lext}(\overrightarrow{\rho}''[\kappa], a') \in \mathscr{U}_i\end{array}}{Pi(A', T', \overrightarrow{\rho}') \approx Pi(A'', T'', \overrightarrow{\rho}'') \in \mathscr{U}_i}$$

  $\overrightarrow{\Gamma} ; \Gamma \vdash T \circledR_i \mathscr{D}$ iff there exist T_1 and T_2, such that

  - T and $\Pi T_1.T_2$ are equivalent: $\overrightarrow{\Gamma} ; \Gamma \vdash T \approx \Pi T_1.T_2 : \mathtt{Ty}_i$.
  - T_2 is well typed: $\overrightarrow{\Gamma} ; \Gamma.T_1 \vdash T_2 : \mathtt{Ty}_i$.
  - For any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma} ; \Gamma$,
    - $T_1[\overrightarrow{\sigma}]$ and $\mathscr{E}_1(\overrightarrow{\sigma})$ are recursively related: $\overrightarrow{\Delta} \vdash T_1[\overrightarrow{\sigma}] \circledR_i \mathscr{E}_1(\overrightarrow{\sigma})$, and
    - For any related s and b, i.e. $\overrightarrow{\Delta} \vdash s : T_1[\overrightarrow{\sigma}] \circledR_i b \in \mathbf{El}_i(\mathscr{E}_1(\overrightarrow{\sigma}))$, and $\mathscr{E}_3 :: b \in \mathbf{El}_i(\mathscr{E}_1(\overrightarrow{\sigma}))$, $T_2[\overrightarrow{\sigma}, s]$ and $\mathscr{E}_2(\overrightarrow{\sigma}, \mathscr{E}_3)$ are recursively related as types:
    $$\overrightarrow{\Delta} \vdash T_2[\overrightarrow{\sigma}, s] \circledR_i \mathscr{E}_2(\overrightarrow{\sigma}, \mathscr{E}_3)$$

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Note that here we require $\mathcal{E}_3 :: b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$ as an assumption, which technically we can derive from $\overrightarrow{\Delta} \vdash s : T_1[\overrightarrow{\sigma}] \circledR_i b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$. However, this fact requires a proof and thus cannot be used at the time of definition. Adding this assumption simplifies our definition.

$\overrightarrow{\Gamma}; \Gamma \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathcal{D})$ iff there exist T_1 and T_2, such that

- t is well typed: $\overrightarrow{\Gamma}; \Gamma \vdash t : T$.
- a is in the PER $\mathbf{El}_i(\mathcal{D})$: $a \in \mathbf{El}_i(\mathcal{D})$.
- T and $\Pi T_1.T_2$ are equivalent: $\overrightarrow{\Gamma}; \Gamma \vdash T \approx \Pi T_1.T_2 : \mathtt{Ty}_i$.
- T_2 is well typed: $\overrightarrow{\Gamma}; \Gamma.T_1 \vdash T_2 : \mathtt{Ty}_i$.
- For any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}; \Gamma$,
    * $T_1[\overrightarrow{\sigma}]$ and $\mathcal{E}_1(\overrightarrow{\sigma})$ are recursively related: $\overrightarrow{\Delta} \vdash T_1[\overrightarrow{\sigma}] \circledR_i \mathcal{E}_1(\overrightarrow{\sigma})$, and
    * For any related s and b, i.e., $\overrightarrow{\Delta} \vdash s : T_1[\overrightarrow{\sigma}] \circledR_i b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$, and $\mathcal{E}_3 :: b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$, the results of eliminating $t[\overrightarrow{\sigma}]$ and $a[\overrightarrow{\sigma}]$ are recursively related as terms :

$$\overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}]\, s : T_2[\overrightarrow{\sigma}, s] \circledR_i a[\overrightarrow{\sigma}] \cdot b \in \mathbf{El}_i(\mathcal{E}_2(\overrightarrow{\sigma}, \mathcal{E}_3))$$

7.3 Properties of gluing model

In this section, we discuss some properties of the gluing model. For conciseness, we focus on monotonicity, realizability, and cumulativity.

7.3.1 Monotonicity

Monotonicity ensures that the gluing model is stable under restricted weakenings. Restricted weakenings apply to both syntax and semantics because they contain modal extensions to instruct both sides to travel among Kripke worlds. Following a similar strategy as in the PER model, we add the additional typing derivations $\mathcal{D}$ and $\mathcal{E}$, which characterize $A \approx B$ and $A[\overrightarrow{\sigma}] \approx B[\overrightarrow{\sigma}]$, resp., to expose more clearly the proof structure and simplify the termination argument.

Lemma 7.1 (Monotonicity). *Given a restricted weakening $\overrightarrow{\Gamma} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Delta}$, $\mathcal{D} :: A \approx B \in \mathcal{U}_i$ and $\mathcal{E} :: A[\overrightarrow{\sigma}] \approx B[\overrightarrow{\sigma}] \in \mathcal{U}_i$,*

- *if $\overrightarrow{\Delta} \vdash T \circledR_i \mathcal{D}$, then $\overrightarrow{\Gamma} \vdash T[\overrightarrow{\sigma}] \circledR_i \mathcal{E}$;*
- *if $\overrightarrow{\Delta} \vdash t : T \circledR_i a \in \boldsymbol{El}_i(\mathcal{D})$, then $\overrightarrow{\Gamma} \vdash t[\overrightarrow{\sigma}] : T[\overrightarrow{\sigma}] \circledR_i a[\overrightarrow{\sigma}] \in \boldsymbol{El}_i(\mathcal{E})$.*

7.3.2 Realizability

In completeness, realizability morally states that **El** is subsumed by *Nf*. In soundness, realizability has a similar structure. We first need to give three definitions, which serve a purpose similar to *Ty*, *Ne*, and *Nf* in completeness:

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Definition 7.3. Given $\mathscr{D} :: A \approx B \in \mathscr{U}_i$,

$$\frac{\overrightarrow{\Gamma} \vdash T : \mathtt{Ty}_i \qquad A \approx B \in \mathit{Ty} \qquad \forall \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma} \, . \, \overrightarrow{\Delta} \vdash T[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ty}}_{\mathtt{map}(\Delta \mapsto |\Delta|, \overrightarrow{\Delta})}(A[\overrightarrow{\sigma}]) : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash T \, \overline{\circledR}_i \, \mathscr{D}}$$

$$\frac{\begin{array}{c} \overrightarrow{\Gamma} \vdash t : T \qquad \overrightarrow{\Gamma} \vdash T \circledR_i \mathscr{D} \\ c \in \mathit{Ne} \quad \forall \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma} \, . \, \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\mathtt{map}(\Delta \mapsto |\Delta|, \overrightarrow{\Delta})}(c[\overrightarrow{\sigma}]) : T[\overrightarrow{\sigma}] \end{array}}{\overrightarrow{\Gamma} \vdash t : T \, \underline{\circledR}_i \, c \in \mathbf{El}_i(\mathscr{D})}$$

$$\frac{\begin{array}{c} \overrightarrow{\Gamma} \vdash t : T \qquad \overrightarrow{\Gamma} \vdash T \circledR_i \mathscr{D} \\ \downarrow^A (a) \approx \downarrow^B (a) \in \mathit{Nf} \qquad \forall \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma} \, . \, \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Nf}}_{\mathtt{map}(\Delta \mapsto |\Delta|, \overrightarrow{\Delta})}(\downarrow^A (a)[\overrightarrow{\sigma}]) : T[\overrightarrow{\sigma}] \end{array}}{\overrightarrow{\Gamma} \vdash t : T \, \overline{\circledR}_i \, a \in \mathbf{El}_i(\mathscr{D})}$$

The first judgment $\overrightarrow{\Gamma} \vdash T \, \overline{\circledR}_i \, \mathscr{D}$ states that T is equivalent to the readback of A (or equally the readback of B) under all valid restricted weakenings. Similarly, the third judgment $\overrightarrow{\Gamma} \vdash t : T \, \overline{\circledR}_i \, a \in \mathbf{El}_i(\mathscr{D})$ states that t is equivalent to the readback of a under all valid restricted weakenings. The realizability theorem states that the gluing models of types and terms are subsumed by these two judgments, respectively, which constitutes our second step to the soundness proof. Then, we state the realizability for the gluing model:

Theorem 7.2 (Realizability)**.** *Given $\mathscr{D} :: A \approx B \in \mathscr{U}_i$,*

- *if $\overrightarrow{\Gamma} \vdash T \circledR_i \mathscr{D}$, then $\overrightarrow{\Gamma} \vdash T \, \overline{\circledR}_i \, \mathscr{D}$.*
- *if $\overrightarrow{\Gamma} \vdash t : T \, \underline{\circledR}_i \, c \in \boldsymbol{El}_i(\mathscr{D})$, then $\overrightarrow{\Gamma} \vdash t : T \circledR_i \uparrow^A (c) \in \boldsymbol{El}_i(\mathscr{D})$;*
- *if $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \boldsymbol{El}_i(\mathscr{D})$, then $\overrightarrow{\Gamma} \vdash t : T \, \overline{\circledR}_i \, a \in \boldsymbol{El}_i(\mathscr{D})$;*

7.3.3 Cumulativity and lowering

Similar to the PER model, cumulativity of the gluing model also requires a lowering statement to handle the function cases and contravariant occurrences in type constructors in general:

Lemma 7.3 (Cumulativity and lowering)**.** *Given $\mathscr{D} :: A \approx B \in \mathscr{U}_i$ and $\mathscr{E} :: A \approx B \in \mathscr{U}_{1+i}$ which we obtain by applying Lemma 6.5 to $\mathscr{D}$,*

- *if $\overrightarrow{\Gamma} \vdash T \circledR_i \mathscr{D}$, then $\overrightarrow{\Gamma} \vdash T \circledR_{1+i} \mathscr{E}$;*
- *if $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \boldsymbol{El}_i(\mathscr{D})$, then $\overrightarrow{\Gamma} \vdash t : T \circledR_{1+i} a \in \boldsymbol{El}_{1+i}(\mathscr{E})$;*
- *if $\overrightarrow{\Gamma} \vdash t : T \circledR_{1+i} a \in \boldsymbol{El}_{1+i}(\mathscr{E})$ and $\overrightarrow{\Gamma} \vdash T \circledR_i \mathscr{D}$, then $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \boldsymbol{El}_i(\mathscr{D})$.*

The first two statements are just cumulativity. However, there is one more complication here in the lowering: we cannot simply derive $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathscr{D})$ from $\overrightarrow{\Gamma} \vdash t : T \circledR_{1+i} a \in \mathbf{El}_{1+i}(\mathscr{E})$! This is because the gluing model contains syntactic information about types so the lowering statement must in addition have an assumption about T and $\mathscr{D}$'s relation at a lower level, which is introduced by $\overrightarrow{\Gamma} \vdash T \circledR_i \mathscr{D}$. With this assumption, we successfully establish the cumulativity for the gluing model.

7.4 Gluing model for K-substitutions and environments

In this section, we generalize the gluing model to K-substitutions and evaluation environments. Our gluing model is in fact more complex than existing proofs on paper (Abel, 2013; Abel *et al.*, 2017; Gratzer *et al.*, 2019), in that our gluing model is again defined through induction–recursion, while existing proofs directly proceed by recursion on the structure of the domain contexts (or context stacks in our case). This existing proof technique will not work in mechanization, because this technique heavily relies on cumulativity to bring the gluing model for terms and values to a limit, so the generalization to K-substitutions and environments does not care about universe levels. Evidently, in Agda, we cannot really take limits, so we must always keep track of universe levels. In our definition, we have an inductive definition $\Vdash \overrightarrow{\Gamma}$ of semantic well formedness of context stacks, in which universe levels are maintained. We speculate that this extra inductive predicate allows an easy adaptation to non-cumulative settings, an NbE proof of which remains unseen to date.

Definition 7.4. The semantic well formedness of context stacks $\mathscr{D} :: \Vdash \overrightarrow{\Gamma}$ and the gluing model for K-substitutions and environments $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \mathscr{D} \circledR \overrightarrow{\rho}$ are defined inductive-recursively:

- $$\mathscr{D} := \frac{}{\Vdash \varepsilon; \cdot}$$

  $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \mathscr{D} \circledR \overrightarrow{\rho}$ iff $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \varepsilon; \cdot$.

- $$\mathscr{D} := \frac{\mathscr{E} :: \Vdash \overrightarrow{\Gamma}}{\Vdash \overrightarrow{\Gamma}; \cdot}$$

  $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \mathscr{D} \circledR \overrightarrow{\rho}$ iff

  - $\overrightarrow{\sigma}$ is well typed: $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}; \cdot$.
  - There exists a K-substitution $\overrightarrow{\sigma}'$ and an modal offset n, such that
    - $\overrightarrow{\sigma}'$ is $\overrightarrow{\sigma}$'s truncation: $\overrightarrow{\Delta} \mid n \vdash \overrightarrow{\sigma} \mid 1 \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}$,
    - modal offsets are equal: $\mathscr{O}(\overrightarrow{\sigma}, 1) = \mathscr{O}(\overrightarrow{\rho}, 1) = n$, and
    - $\overrightarrow{\sigma}'$ and $\overrightarrow{\rho} \mid 1$ are recursively related: $\overrightarrow{\Delta} \mid n \vdash \overrightarrow{\sigma}' : \mathscr{E} \circledR \overrightarrow{\rho} \mid 1$.

- $$\mathscr{D} := \frac{\begin{array}{c}\exists i \qquad \mathscr{E} :: \Vdash \overrightarrow{\Gamma}; \Gamma \\ \forall \overrightarrow{\Delta} \vdash \overrightarrow{\delta} : \mathscr{D} \circledR \overrightarrow{\rho}.\ \Sigma(\mathscr{J} :: [\![T]\!](\overrightarrow{\rho}) \in \mathscr{U}_i).\ \overrightarrow{\Delta} \vdash T[\overrightarrow{\delta}] \circledR_i \mathscr{J}\end{array}}{\Vdash \overrightarrow{\Gamma}; \Gamma.T}$$

$\vec{\Delta} \vdash \vec{\sigma} : \mathscr{D} \circledR \vec{\rho}$ iff

- $\vec{\sigma}$ is well typed: $\vec{\Delta} \vdash \vec{\sigma} : \vec{\Gamma}\,; \Gamma.T$.
- There exists a K-substitution $\vec{\sigma}\,'$ and t, such that
  - $\vec{\sigma}\,'$ is $\vec{\sigma}$ with the topmost term dropped: $\vec{\Delta} \vdash \mathsf{wk} \circ \vec{\sigma} \approx \vec{\sigma}\,' : \vec{\Gamma}\,; \Gamma$,
  - t is that topmost term: $\vec{\Delta} \vdash v_0[\vec{\sigma}\,] \approx t : T[\vec{\sigma}\,']$.
  - T evaluates in $\mathtt{drop}(\vec{\rho}\,)$ and the result is in $\mathscr{U}_i$:
  $$\mathscr{J}\,' :: [\![T]\!](\mathtt{drop}(\vec{\rho}\,)) \in \mathscr{U}_i$$
  - t and $\vec{\rho}\,(0)$ are related at level i: $\vec{\Delta} \vdash t : T[\vec{\sigma}\,'] \circledR_i \rho(0) \in \mathbf{El}_i(\mathscr{J}\,')$, where $(\_, \rho) := \vec{\rho}\,(0)$,
  - $\vec{\sigma}\,'$ and $\mathtt{drop}(\vec{\rho}\,)$ are recursively related: $\vec{\Delta} \vdash \vec{\sigma}\,' : \mathscr{E} \circledR \mathtt{drop}(\vec{\rho}\,)$.

We can then prove properties like monotonicity, which we omit here in favor of space. We refer the readers to our Agda development. We then give the definitions for semantic judgments.

Definition 7.5. Semantic judgments for soundness are defined as follows:

$$\frac{\exists i \qquad \mathscr{D} :: \Vdash \vec{\Gamma} \qquad \forall \vec{\Delta} \vdash \vec{\sigma} : \mathscr{D} \circledR \vec{\rho}\,.\ \vec{\Delta} \vdash t[\vec{\sigma}\,] : T[\vec{\sigma}\,] \circledR_i [\![t]\!](\vec{\rho}\,) \in \mathbf{El}_i([\![T]\!](\vec{\rho}\,))}{\vec{\Gamma} \Vdash t : T}$$

$$\frac{\mathscr{D}_1 :: \Vdash \vec{\Gamma} \qquad \mathscr{D}_2 :: \Vdash \vec{\Gamma}\,' \qquad \forall \vec{\Delta} \vdash \vec{\sigma} : \mathscr{D}_1 \circledR \vec{\rho}\,.\ \vec{\Delta} \vdash \vec{\delta} \circ \vec{\sigma} : \mathscr{D}_2 \circledR [\![\vec{\delta}\,]\!](\vec{\rho}\,)}{\vec{\Gamma} \Vdash \vec{\delta} : \vec{\Gamma}\,'}$$

7.5 Fundamental theorems and soundness

Finally, we are ready to establish the fundamental theorems and soundness:

Theorem 7.4 (Fundamental).

- *If* $\vdash \vec{\Gamma}$, *then* $\Vdash \vec{\Gamma}$.
- *If* $\vec{\Gamma} \vdash t : T$, *then* $\vec{\Gamma} \Vdash t : T$.
- *If* $\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}$, *then* $\vec{\Gamma} \Vdash \vec{\sigma} : \vec{\Delta}$.

Theorem 7.5 (Soundness). *If* $\vec{\Gamma} \vdash t : T$, *then* $\vec{\Gamma} \vdash t \approx \mathsf{nbe}^{T}_{\vec{\Gamma}}(t) : T$.

Proof We first apply the fundamental theorems and obtain $\vec{\Gamma} \Vdash t : T$. Moreover, we know $\vec{\Gamma} \vdash \vec{I} : \vec{\Gamma} \circledR \uparrow^{\vec{\Gamma}}$, t and $[\![t]\!](\uparrow^{\vec{\Gamma}})$ are related. The goal is concluded by further applying realizability. ■

7.6 Consequences

The fundamental theorems for the gluing model allow us to establish a few more consequences, which are difficult to prove syntactically. We show that the standard injectivity and canonicity hold in MINT.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Lemma 7.6 (Injectivity of Type Constructors).

- *If* $\overrightarrow{\Gamma} \vdash \square T_1 \approx \square T_2 : \mathit{Ty}_i$, *then* $\overrightarrow{\Gamma}; \cdot \vdash T_1 \approx T_2 : \mathit{Ty}_i$.
- *If* $\overrightarrow{\Gamma}; \Gamma \vdash \Pi S_1.T_1 \approx \Pi S_2.T_2 : \mathit{Ty}_i$, *then* $\overrightarrow{\Gamma}; \Gamma \vdash S_1 \approx S_2 : \mathit{Ty}_i$ *and* $\overrightarrow{\Gamma}; \Gamma.S_1 \vdash T_1 \approx T_2 : \mathit{Ty}_i$.

Lemma 7.7 (Canonicity of N). *If* $\varepsilon; \cdot \vdash t : N$, *then* $\varepsilon; \cdot \vdash t \approx \mathtt{succ}^n\ \mathtt{zero} : N$ *for some number n.*

The following lemma about universe levels is also interesting. It says that if two types are equivalent and they are well typed at a different level, then they are equivalent also at that level. This lemma is intuitive but very challenging to prove syntactically.

Lemma 7.8 (Type equivalence). *If* $\overrightarrow{\Gamma} \vdash T_1 \approx T_2 : \mathit{Ty}_i$, $\overrightarrow{\Gamma} \vdash T_1 : \mathit{Ty}_j$ *and* $\overrightarrow{\Gamma} \vdash T_2 : \mathit{Ty}_j$, *then* $\overrightarrow{\Gamma} \vdash T_1 \approx T_2 : \mathit{Ty}_j$.

Proof From $\overrightarrow{\Gamma} \vdash T_1 \approx T_2 : \mathtt{Ty}_i$ and completeness, we know T_1 and T_2 have equal normal form. We conclude the goal by soundness and transitivity. ■

As the final and conclusive theorem, we show that MINT is consistent.

Theorem 7.9 (Consistency). *There is no closed term of type* $\Pi\, \mathit{Ty}_i.v_0$. *That is, there is no t such that the following judgment holds:*

$$\varepsilon; \cdot \vdash t : \Pi\, \mathit{Ty}_i.v_0$$

In Agda's syntax, the type is written as `(A : Set`$_i$`) → A` and consistency requires that this type does not have a closed inhabitant. In other words, there is no generic way to construct a term for an arbitrary type. If MINT has a bottom type, then the consistency proof is immediately reduced to the consistency of our meta-language (i.e. Agda) and hence significantly simplifies the proof. For the current definition of MINT, we can still prove this theorem with a bit more technical setup.

Proof Note that consistency is equivalent to proving that there is no t' such that

$$\varepsilon; \mathtt{Ty}_i \vdash t' : v_0$$

Note that, by soundness, t' must be equivalent to some neutral term u by NbE, because its type (v_0) is neutral. So our goal is to show that this u also does not exist:

$$\varepsilon; \mathtt{Ty}_i \vdash u : v_0$$

Now we do induction on u. We show that in $\varepsilon; \mathtt{Ty}_i$, there does not exist a neutral term of any type other than $\mathtt{Ty}_j$ (i and j are not necessarily equal due to cumulativity), and thus it is impossible for u to have type v_0. Otherwise, that would require $\mathtt{Ty}_i$ and v_0 to be equivalent, which can be rejected by completeness as they do not evaluate to the same normal form. ■

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

8 Extraction of the NbE algorithm

To use our NbE algorithm as a verified kernel of a proof assistant, we want to execute our mechanization. Even though we cannot directly execute our mechanization in Agda, we can obtain a Haskell implementation of our NbE algorithm from it using Agda's extraction facility.

To illustrate how to use and run this extracted algorithm, we provide an example of executable code of the ubiquitous power function (see Appendix B). This Agda file contains the representation of the power function, its typing derivation, and use cases in MINT. We also apply the NbE algorithm to these use cases. Once extracted into Haskell, we can run the NbE algorithm on these use cases to obtain their normal forms and print them. For ease of use and reproducibility, we also provide the extracted Haskell code itself.

This shows that our formalization in principle can serve as an implementation of a verified kernel of a proof assistant equipped with the necessity modality.

9 Related work

9.1 Applications of modalities in multi-staged programming

As previously mentioned, the modal system $S4$ corresponds to multi-staged programming under Curry–Howard correspondence (Davies & Pfenning, 2001) and MINT provides a dependently typed variant of $S4$. There are other applications of dependent typed systems to multi-staged programming. Kawata & Igarashi (2019) study λ^{MD}, a logical framework with stages. λ^{MD} uses stage variables to keep track of stages, while MINT uses context stacks and `unbox` levels for the same purpose. Pasalic *et al.* (2002) propose Meta-D to remove administrative redexes introduced in a staged interpreter. Brady & Hammond (2006) improve Pasalic *et al.* (2002) by also extending MLTT with stages. Fundamentally, the type theory in Brady & Hammond (2006) is the dependently typed system T with cross-stage persistence. Extending the T variant of MINT with cross-stage persistence would constitute Brady & Hammond (2006)' system. Though the authors claim that their type theory is strongly normalizing, they do not provide any proof, whereas MINT's normalization proof naturally adapts to all subsystems of $S4$, including T.

9.2 Modal type theories

One of the first "propositions-as-types"-interpretations of modal logics was given by Borghuis (1994) in his PhD thesis, which studies the modal natural deduction with the modality S4 and modal pure type system. His work, similar to Pfenning & Wong (1995) and Davies & Pfenning (2001), also introduces a context stack structure (called generalized contexts). However, in contrast to Davies & Pfenning (2001) where the elimination rule ($\mathtt{unbox}_n$) integrates both modal and local weakening, Borghuis' elimination rule has explicit rules for both local weakening and modal weakening. As a consequence, weakening is not a property of the overall system in Borghuis' work. Further, Borghuis studies strong normalization via a translation of the modal pure type system to a pure type system. In contrast, we give a direct normalization proof that yields an algorithm for normalizing objects in MINT.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Most closely related to our work is the line of research by Clouston (2018), Gratzer *et al.* (2019), Birkedal *et al.* (2020*b*), and Valliappan *et al.* (2022). This line of work explores various modal dependent type theories where different stages are separated by locks (🔒). Kripke- and Lock-style (sometimes also referred to as Fitch-style) systems are fundamentally the same, although we manage the context stack structure slightly differently. Syntactically, the Kripke style simply uses a stack of contexts, while the Lock-style uses locks (🔒) to segment one context into sections. Therefore, Lock-style systems typically rely on external side conditions such as "delete all of the locks that occur in the context" or "no lock occurs in context Γ". These checks and operations on contexts with locks correspond in fact to modal structural properties in the Kripke-style formulation. For example, "deleting all of the locks" corresponds to the property of modal fusion. The condition that "no locks occur in context Γ" is naturally captured simply by the stack structure of the context stack in Kripke-style systems. Hence, while the lock-based treatment of contexts and the use of context stacks are related (see also Clouston, 2018), context stacks reflect more directly the underlying Kripke semantics. As a consequence, structural properties of context stacks are more clearly formulated as general logical principles on those context stacks. This also allows us to characterize the context stack structure by a *truncoid* algebra and *internalize* it in our semantics. One consequence of that approach is that it allows us to directly adapt existing NbE algorithms such as Abel (2013). The end result is a clean and general NbE proof that captures all four subsystems of $S4$ without change. The fact that our NbE algorithm can be compactly and elegantly mechanized in Agda further emphasizes the benefit of the Kripke-style structure of context stacks and its corresponding semantic characterizations.

In comparison, Valliappan *et al.* (2022) give four (similar but) different formulations for all four subsystems of simply typed $S4$ and provide their normalization proofs separately. Our previous work (Hu & Pientka, 2022*a*) considers the Kripke style and gives one NbE proof based on a presheaf model that handles all four subsystems. The NbE proof given by Gratzer *et al.* (2019) for dependently typed, idempotent $S4$ cannot be easily adapted to all four subsystems of $S4$ and it is also not fully mechanized.

Last, the work by Gratzer (2022) gives a NbE proof for a richer MTT. However, its relationship to Kripke-style formulations is less clear, as it uses a very different formulation of the $\square$ elimination rule. As such, it is closer in spirit to the dual-context formulation of modal $S4$ given by Davies & Pfenning (2001) and further developed by Jang *et al.* (2022) (see also discussion below). Disregarding the differences in the underlying formulation of the modal elimination rule, the NbE proof described in Gratzer (2022) builds on the idea of synthetic Tait's computability introduced by Sterling (2022). This framework is much less understood than, for example, Abel (2013). In particular, it is less clear how one would mechanize the NbE proof given by Gratzer (2022) or how one would extract an algorithm that can be implemented in a conventional programming language. Hence, while Mint is less powerful than MTT, our work has direct practical benefits: it simply relies on well-understood techniques and approaches and we can extract an NbE algorithm from our Agda mechanization.

An alternative formulation of modal type systems is the dual-context formulation, which also goes back to Davies & Pfenning (2001) and where the authors gave a translation between the *implicit* (or Kripke-style) modal λ-calculus using context stacks and the *explicit* using dual contexts to distinguish between global assumptions and local assumptions. This translation shows that both styles have the same logical strength. However, the translation does not give a correspondence between the equivalence relations of both styles. In fact, we speculate that the translation from the Kripke style to the dual-context style does not preserve equivalence. This further implies that the translation cannot be scaled to dependent types, because of the mutual definition of typing and equivalence judgments, and that both styles may not have the same expressive power in the dependently typed setting. In recent years, modal type systems in the dual-context style have found a wide range of seemingly unconnected applications: from reasoning about universes in homotopy type theory (Licata *et al*., 2018; Shulman, 2018) to mechanizing meta-theory (Pientka, 2008), to reasoning about effects (Zyuzin & Nanevski, 2021), and meta-programming (Jang *et al*., 2022). It would hence be interesting to further explore the relationship between implicit and explicit formulations of modal type theories in the future.

9.3 Mechanization of normalization for dependent type theory

From the 1990s, the question of how to mechanize the normalization proof for dependent type theory has been fundamental to gain trust in the type-theoretic foundation proof assistants such as Coq or Agda are built on. One of the earliest works is, for example, by Barras & Werner (1997). Barras and Werner formalized the strong normalization for the calculus of construction in Coq using reducibility candidates. More recently, Abel *et al*. (2017) mechanize a normalization proof for Martin-Löf logical framework in Agda. Instead of NbE, they first reduce terms to weak head normal forms and then use a type-directed algorithmic convertibility checking algorithm to check the equivalence of terms. In their development, they also rely on induction–recursion to give a semantics to dependent types. The algorithmic convertibility checking algorithm is shown to be complete and sound w.r.t. the equivalence rules. The completeness and soundness proofs are parameterized by a relation, reducing the size of the proofs. Pujet & Tabareau (2022) extend the work by Abel *et al*. (2017) by mechanizing observational equality and two-level cumulative universe hierarchy. In their work, they did not need a lowering lemma for related cumulative properties. Instead, they make the following adjustments:

- In their logical relations, they added an extra case to represent the embedding from the lower universe to the higher one. This adjustment turns the cumulativity lemmas into a case in the logical relations.
- Moreover, in addition to the level 0 and level 1 universes, they have an extra level ∞ universe, which subsumes both level 0 and 1 universes. Then, the proof can simply lift all types to the ∞ level and completely avoid explicit discussions of universe levels entirely. This treatment resembles the typical paper proof.

This work is further superseded by Pujet & Tabareau (2023), which mechanizes impredicative observational equality. In this work, however, cumulativity is removed from the universe hierarchy.

In contrast, mechanizations of NbE algorithms in dependent type theory are less common and remain challenging. Danielsson (2006) presents the first mechanization of NbE

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

for Martin-Löf logical framework using induction–recursion in AgdaLight. As pointed out by Chapman (2009), Danielsson (2006)'s formulation contains non-strictly positive predicates, which compromise the trust in this work.

Chapman (2009) formalizes Martin-Löf logical framework in the style of categories with families (Dybjer, 1996) in Agda and presents a sound normalizer. However, the normalizer is not shown complete whereas our paper has shown both completeness and soundness.

Altenkirch & Kaposi (2016*b*,a) and Kaposi & Altenkirch (2017) mechanize an NbE algorithm for Martin-Löf logical framework in Agda and prove completeness and soundness using a presheaf formulation akin to categories with families. Their development explores an advanced combination of intrinsic syntactic representations and involved features like induction–induction (Nordvall Forsberg & Setzer, 2010) and quotient inductive types. In comparison, our development is simpler and only relies on two standard extensions: induction–recursion and functional extensionality. The simplicity leads to a formalization of a full hierarchy of universes. We hope that our mechanization of NbE in Agda, while focused on a modal type theory, also provides more general guidance on how to deal with a full universe hierarchy in the mechanization and implementation of NbE in type theory.

Most work in this area is done in Agda, as it supports induction–recursion, which strengthens the logical power of the meta-language and allows us to define the semantics for universes. Nevertheless, Agda is not the only choice. Wieczorek & Biernacki (2018) mechanize an NbE algorithm for Martin-Löf logical framework in Coq. Since Coq does not support induction–recursion, they universally quantify over the impredicative universe `Prop` in their models. Their algorithm can also be extracted to and run in Haskell or OCaml. One benefit of using Coq is that `Prop` is automatically removed during extraction. Hence, their extraction code is cleaner than the one generated from Agda. While the readability of our extracted code could be improved by adding compiler auxiliary pragmas, the root cause for the verbosity of our extracted code lies in Agda's extraction facilities. In the future, it would be interesting to adapt Wieczorek & Biernacki (2018)'s work to mechanize the NbE algorithm for Mint in Coq, which would allow us to take advantage of Coq's extraction facilities.

10 Conclusions and future work

In this paper, we introduce Mint, a foundation for dependently typed variants of Systems K, T, $K4$, and $S4$. This work adds a further piece to the landscape of combining modalities and dependent types. To justify Mint, we provide an NbE algorithm and its completeness and soundness proofs. To further strengthen our confidence in the theoretical development, we fully mechanize our constructions in safe Agda, with the standard extensions of functional extensionality and induction–recursion. In our proof, we introduce the notion of truncoids, a special algebra that characterizes structures appearing in both syntax and semantics. It serves as a guiding principle throughout our proof. We further exploit the notion of UMoTs to modularize our constructions so that our normalization result applies to all aforementioned systems without modifications.

As part of future work, we will explore the application of the dependently typed variant of $S4$ to multi-staged and meta-programming. This direction allows us to develop a type

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

theory that is capable of proving and meta-programming in the same language. To achieve this goal, we would like to have the power of doing pattern matching on code as demonstrated by Jang *et al*. (2022). Having pattern matching on code will allow MINT to write meta-programs that construct proofs based on the goal types. As such it would provide a general foundation for boot-strapping proof assistants.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions and feedbacks. This work was funded by the Natural Sciences and Engineering Research Council of Canada (grant number 206263), Fonds de recherche du Québec – Nature et technologies (grant number 253521). Postgraduate Scholarship – Doctoral by the Natural Sciences and Engineering Research Council of Canada was awarded to the first author. A graduate fellowship from Fonds de recherche du Québec – Nature et technologies (grant number 304215) was awarded to the second author.

Conflicts of Interest

None.

References

Abel, A. (2013) *Normalization by Evaluation: Dependent Types and Impredicativity*. Habilitation Thesis. Ludwig-Maximilians-Universität München.

Abel, A., Öhman, J. & Vezzosi, A. (2017) Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* **2**(POPL), 23:1–23:29.

Abel, A., Vezzosi, A. & Winterhalter, T. (2017) Normalization by evaluation for sized dependent types. *Proc. ACM Program. Lang.* **1**(ICFP), 33:1–33:30.

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003) A functional correspondence between evaluators and abstract machines. In Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming. New York, NY, USA: Association for Computing Machinery, pp. 8–19.

Altenkirch, T., Hofmann, M. & Streicher, T. (1995) Categorical reconstruction of a reduction free normalization proof. In Category Theory and Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 182–199.

Altenkirch, T. & Kaposi, A. (2016a) Normalisation by evaluation for dependent types. In 1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016). Dagstuhl, Germany. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, pp. 6:1–6:16. ISSN: 1868-8969.

Altenkirch, T. & Kaposi, A. (2016b) Type theory in type theory using quotient inductive types. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: Association for Computing Machinery, pp. 18–29.

Barras, B. & Werner, B. (1997) Coq in coq.

Berger, U. & Schwichtenberg, H. (1991) An inverse of the evaluation functional for typed lambda-calculus. In 1991 Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science, pp. 203–211.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Bierman, G. M. & de Paiva, V. (2000) On an intuitionistic modal logic. *Studia Logica* **65**(3), 383–416.

Bierman, G. M. & de Paiva, V. C. V. (1996) *Intuitionistic Necessity Revisited*. Technical report. University of Birmingham.

Birkedal, L., Clouston, R., Mannaa, B., Møgelberg, R. E., Pitts, A. M. & Spitters, B. (2020a) Modal dependent type theory and dependent right adjoints. *Math. Struct. Comput. Sci.* **30**(2), 118–138.

Birkedal, L., Clouston, R., Mannaa, B., Møgelberg, R. E., Pitts, A. M. & Spitters, B. (2020b) Modal dependent type theory and dependent right adjoints. *Math. Struct. Comput. Sci.* **30**(2), 118–138. Publisher: Cambridge University.

Borghuis, V. A. J. (1994) *Coming to Terms with Modal Logic: On the Interpretation of Modalities in Typed Lambda-Calculus*. PhD Thesis. Mathematics and Computer Science.

Brady, E. C. & Hammond, K. (2006) A verified staged interpreter is a verified compiler. In 5th International Conference on Generative Programming and Component Engineering (GPCE'06). ACM, pp. 111–120.

Chapman, J. (2009) Type theory should eat itself. In International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08). Elsevier, pp. 21–36.

Clouston, R. (2018) Fitch-style modal lambda calculi. In *Foundations of Software Science and Computation Structures*. Cham: Springer International Publishing, pp. 258–275.

Clouston, R., Bizjak, A., Grathwohl, H. B. & Birkedal, L. (2015) Programming and reasoning with guarded recursion for coinductive types. In *Foundations of Software Science and Computation Structures*, Pitts, A. (ed)., vol. 9034. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 407–421. Available at: http://link.springer.com/10.1007/978-3-662-46678-0_26. Series Title: Lecture Notes in Computer Science.

Coquand, T. & Dybjer, P. (1997) Intuitionistic model constructions and normalization proofs. *Math. Struct. Comput. Sci.* **7**(1), 75–94. Publisher: Cambridge University.

Danielsson, N. A. (2006) A formalisation of a dependently typed language as an inductive-recursive family. In TYPES. Springer, pp. 93–109.

Davies, R. & Pfenning, F. (2001) A modal analysis of staged computation. *J. ACM*. **48**(3), 555–604.

Dybjer, P. (1996) Internal type theory. In Types for Proofs and Programs. Berlin, Heidelberg: Springer, pp. 120–134.

Dybjer, P. (2000) A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Logic* **65**(2), 525–549. Publisher: Cambridge University.

Gratzer, D. (2022) Normalization for multimodal type theory. In Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science. Association for Computing Machinery.

Gratzer, D., Kavvos, G. A., Nuyts, A. & Birkedal, L. (2020) Multimodal dependent type theory. In Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. New York, NY, USA: Association for Computing Machinery, pp. 492–506.

Gratzer, D., Sterling, J. & Birkedal, L. (2019) Implementing a modal dependent type theory. *Proc. ACM Program. Lang.* **3**(ICFP), 107:1–107:29.

Harper, R. & Pfenning, F. (2005) On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Logic* **6**(1), 61–101.

Hu, J. Z. S. & Pientka, B. (2022a) A categorical normalization proof for the modal lambda-calculus. In Proceedings 38th Conference on Mathematical Foundations of Programming Semantics, MFPS 2022.

Hu, J. Z. S. & Pientka, B. (2022b) An Investigation of Kripke-style Modal Type Theories. Number: arXiv:2206.07823 arXiv:2206.07823 [cs].

Jang, J., Gélineau, S., Monnier, S. & Pientka, B. (2022) Moebius: Metaprogramming using contextual types – the stage where system f can pattern match on itself. *Proc. ACM Program. Lang. (PACMPL)*. (POPL).

Kaposi, A. & Altenkirch, T. (2017) Normalisation by evaluation for type theory, in type theory. *Logical Methods Comput. Sci.* **13**(4). Publisher: Episciences.org.

Kavvos, G. A. (2017) Dual-context calculi for modal logic. In 2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–12.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Kawata, A. & Igarashi, A. (2019) A dependently typed multi-stage calculus. In 17th Asian Symposium on Programming Languages and Systems (APLAS'19). Springer, pp. 53–72.

Kripke, S. A. (1963) Semantical analysis of modal logic I mormal modal propositional calculi. *Math. Logic Q.* **9**(5-6), 67–96. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19630090502.

Licata, D. R., Orton, I., Pitts, A. M. & Spitters, B. (2018) Internal universes in models of homotopy type theory. In 3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018). Dagstuhl, Germany. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, pp. 22:1–22:17. ISSN: 1868-8969.

Lindley, S. (2007) Extensional rewriting with sums. In Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26–28, 2007, Proceedings. Springer, pp. 255–271.

Martin-Löf, P. (1975) An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, Rose, H. E. & Shepherdson, J. C. (eds). Logic Colloquium'73, vol. 80. Elsevier, pp. 73–118. Available at: https://www.sciencedirect.com/science/article/pii/S0049237X08719451.

Martini, S. & Masini, A. (1996) A computational interpretation of modal proofs. In *Proof Theory of Modal Logic*, Wansing, H. (eds). Applied Logic Series. Dordrecht: Springer Netherlands, pp. 213–241. Available at: https://doi.org/10.1007/978-94-017-2798-3_12.

Nakano, H. (2000) A modality for recursion. In Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332), pp. 255–266. ISSN: 1043-6871.

Nordvall Forsberg, F. & Setzer, A. (2010) Inductive-Inductive definitions. In Computer Science Logic. Berlin, Heidelberg: Springer, pp. 454–468.

Palmgren, E. (1998) On universes in type theory. In *Twenty Five Years of Constructive Type Theory*. Oxford University. Available at: https://oxford.universitypressscholarship.com/view/10.1093/oso/9780198501275.001.0001/isbn-9780198501275-book-part-12.

Pasalic, E., Taha, W. & Sheard, T. (2002) Tagless staged interpreters for typed languages. In Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02). ACM, pp. 218–229.

Pfenning, F. & Davies, R. (2001) A judgmental reconstruction of modal logic. *Math. Struct. Comput. Sci.* **11**(04), 511–540.

Pfenning, F. & Wong, H. (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. Elsevier, pp. 515–534.

Pientka, B. (2008) A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08). ACM, pp. 371–382.

Pientka, B., Abel, A., Ferreira, F., Thibodeau, D. & Zucchini, R. (2019) A type theory for defining logics and proofs. In 34th IEEE/ACM Symposium on Logic in Computer Science (LICS'19). IEEE Computer Society, pp. 1–13.

Pujet, L. & Tabareau, N. (2022) Observational equality: Now for good. *Proc. ACM Program. Lang.* **6**(POPL), 1–27.

Pujet, L. & Tabareau, N. (2023) Impredicative observational equality. *Proc. ACM Program. Lang.* **7**(POPL), 2171–2196.

Reynolds, J. C. (1998) Definitional interpreters for higher-order programming languages. *Higher-Order Symb. Comput.* **11**(4), 363–397.

Schürmann, C., Despeyroux, J. & Pfenning, F. (2001) Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.* **266**(1-2), 1–57.

Shulman, M. (2018) Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Math. Struct. Comput. Sci.* **28**(6), 856–941. Publisher: Cambridge University.

Sterling, J. (2022) *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD Thesis. Carnegie Mellon University, USA.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Taha, W. (2000) A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of metaml is non-trivial (extended abstract). In PEPM. ACM, pp. 34–43.

Taha, W. & Sheard, T. (1997) Multi-stage programming with explicit annotations. In Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. New York, NY, USA: Association for Computing Machinery, pp. 203–217.

Valliappan, N., Ruch, F. & Tomé Cortiñas, C. (2022) Normalization for Fitch-style modal calculi. *Proc. ACM Program. Lang.* **6**(ICFP), 772–798.

Wieczorek, P. & Biernacki, D. (2018) A Coq formalization of normalization by evaluation for Martin-Löf type theory. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. New York, NY, USA: Association for Computing Machinery, pp. 266–279.

Zyuzin, N. & Nanevski, A. (2021) Contextual modal types for algebraic effects and handlers. *Proc. ACM Program. Lang.* **5**(ICFP), 1–29.

A Full set of rules for Mint

$\boxed{\vdash \overrightarrow{\Gamma}}$: Context stack $\overrightarrow{\Gamma}$ is well formed.

$$\frac{}{\vdash \varepsilon ; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma}}{\vdash \overrightarrow{\Gamma} ; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma} ; \Gamma \qquad \overrightarrow{\Gamma} ; \Gamma \vdash T : \texttt{Ty}_i}{\vdash \overrightarrow{\Gamma} ; \Gamma . T}$$

$\boxed{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}$: $\overrightarrow{\Gamma}$ and $\overrightarrow{\Delta}$ are equivalent context stacks.

$$\frac{}{\vdash \varepsilon ; \cdot \approx \varepsilon ; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}{\vdash \overrightarrow{\Gamma} ; \cdot \approx \overrightarrow{\Delta} ; \cdot}$$

$$\frac{\begin{array}{c} \vdash \overrightarrow{\Gamma} ; \Gamma \approx \overrightarrow{\Delta} ; \Delta \qquad \overrightarrow{\Gamma} ; \Gamma \vdash T \approx T' : \texttt{Ty}_i \\ \overrightarrow{\Delta} ; \Delta \vdash T \approx T' : \texttt{Ty}_i \qquad \overrightarrow{\Gamma} ; \Gamma \vdash T : \texttt{Ty}_i \qquad \overrightarrow{\Delta} ; \Delta \vdash T' : \texttt{Ty}_i \end{array}}{\vdash \overrightarrow{\Gamma} ; \Gamma . T \approx \overrightarrow{\Delta} ; \Delta . T'}$$

$\boxed{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}$: $\overrightarrow{\sigma}$ is a K-substitution substituting terms in $\overrightarrow{\Delta}$ into ones in $\overrightarrow{\Gamma}$.

$$\frac{\vdash \overrightarrow{\Gamma}}{\overrightarrow{\Gamma} \vdash \overrightarrow{I} : \overrightarrow{\Gamma}} \qquad \frac{\vdash \overrightarrow{\Gamma} ; \Gamma . T}{\overrightarrow{\Gamma} ; \Gamma . T \vdash \mathsf{wk} : \overrightarrow{\Gamma} ; \Gamma}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}' ; \Gamma \qquad \overrightarrow{\Gamma}' ; \Gamma \vdash T : \texttt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : T[\overrightarrow{\sigma}]}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} , t : \overrightarrow{\Gamma}' ; \Gamma . T} \qquad \frac{\begin{array}{c} \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \\ \vdash \overrightarrow{\Gamma} ; \overrightarrow{\Gamma}' \qquad |\overrightarrow{\Gamma}'| = n \end{array}}{\overrightarrow{\Gamma} ; \overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} ; \Uparrow^n : \overrightarrow{\Delta} ; \cdot}$$

$$\frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'' \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}'}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \circ \overrightarrow{\delta} : \overrightarrow{\Gamma}''} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \qquad \vdash \overrightarrow{\Delta} \approx \overrightarrow{\Delta}'}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}'}$$

$\boxed{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\delta} : \overrightarrow{\Delta}}$: $\overrightarrow{\sigma}$ and $\overrightarrow{\delta}$ are equivalent in K-substituting terms in $\overrightarrow{\Delta}$ into ones in $\overrightarrow{\Gamma}$.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Congruence rules:

$$\frac{\vdash \overrightarrow{\Gamma}}{\overrightarrow{\Gamma} \vdash \overrightarrow{I} \approx \overrightarrow{I} : \overrightarrow{\Gamma}} \qquad \frac{\vdash \overrightarrow{\Gamma}\,; \Gamma.T}{\overrightarrow{\Gamma}\,; \Gamma.T \vdash \mathsf{wk} \approx \mathsf{wk} : \overrightarrow{\Gamma}\,; \Gamma}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}'; \Gamma \qquad \overrightarrow{\Gamma}'; \Gamma \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t \approx t' : T[\overrightarrow{\sigma}]}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}, t \approx \overrightarrow{\sigma}, t' : \overrightarrow{\Gamma}'; \Gamma.T}$$

$$\frac{\begin{array}{c}\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta} \\ \vdash \overrightarrow{\Gamma}\,; \overrightarrow{\Gamma}' \qquad |\overrightarrow{\Gamma}'| = n\end{array}}{\overrightarrow{\Gamma}\,; \overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma}\,; \Uparrow^n \approx \overrightarrow{\sigma}'; \Uparrow^n : \overrightarrow{\Delta}\,; \cdot} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}'' \\ \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} \approx \overrightarrow{\delta}' : \overrightarrow{\Gamma}'\end{array}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \circ \overrightarrow{\delta} \approx \overrightarrow{\sigma}' \circ \overrightarrow{\delta}' : \overrightarrow{\Gamma}''}$$

Categorical rules:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \circ \overrightarrow{I} \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \overrightarrow{I} \circ \overrightarrow{\sigma} \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}}$$

$$\frac{\overrightarrow{\Gamma}'' \vdash \overrightarrow{\sigma}'' : \overrightarrow{\Gamma}''' \qquad \overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma}' : \overrightarrow{\Gamma}'' \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'}{\overrightarrow{\Gamma} \vdash (\overrightarrow{\sigma}'' \circ \overrightarrow{\sigma}') \circ \overrightarrow{\sigma} \approx \overrightarrow{\sigma}'' \circ (\overrightarrow{\sigma}' \circ \overrightarrow{\sigma}) : \overrightarrow{\Gamma}'''}$$

Other rules:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}' \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta} \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}' \approx \overrightarrow{\sigma}'' : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}'' : \overrightarrow{\Delta}}$$

$$\frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}''; \Gamma \qquad \overrightarrow{\Gamma}'; \Gamma \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma}' \vdash t : T[\overrightarrow{\sigma}] \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}'}{\overrightarrow{\Gamma} \vdash (\overrightarrow{\sigma}, t) \circ \overrightarrow{\delta} \approx (\overrightarrow{\sigma} \circ \overrightarrow{\delta}), t[\overrightarrow{\delta}] : \overrightarrow{\Gamma}''; \Gamma.T}$$

$$\frac{\begin{array}{c}\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \\ \overrightarrow{\Delta} \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : T[\overrightarrow{\sigma}]\end{array}}{\overrightarrow{\Gamma} \vdash \mathsf{wk} \circ (\overrightarrow{\sigma}, t) \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}' \qquad \overrightarrow{\Gamma}'' \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}\,; \overrightarrow{\Delta} \\ |\overrightarrow{\Delta}| = n \qquad \vdash \overrightarrow{\Gamma}\,; \overrightarrow{\Delta}\end{array}}{\overrightarrow{\Gamma}'' \vdash (\overrightarrow{\sigma}\,; \Uparrow^n) \circ \overrightarrow{\delta} \approx (\overrightarrow{\sigma} \circ \overrightarrow{\delta} \mid n); \Uparrow^{\mathscr{O}(\overrightarrow{\delta}, n)} : \overrightarrow{\Gamma}'; \cdot}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}\,; \cdot \qquad |\overrightarrow{\Delta}| > 0}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma} \mid 1; \Uparrow^{\mathscr{O}(\overrightarrow{\sigma}, 1)} : \overrightarrow{\Delta}\,; \cdot} \qquad \frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}\,; \Gamma.T}{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} \approx (\mathsf{wk} \circ \overrightarrow{\sigma}), v_0[\overrightarrow{\sigma}] : \overrightarrow{\Gamma}\,; (\Gamma.T)}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta} \qquad \vdash \overrightarrow{\Delta} \approx \overrightarrow{\Delta}'}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}'}$$

To define the variable rule for the typing judgment, we need to define the lookup judgment $x : T \in \overrightarrow{\Gamma}$:

$$\frac{}{0 : T[\mathsf{wk}] \in \overrightarrow{\Gamma}\,; \Gamma.T} \qquad \frac{x : T \in \overrightarrow{\Gamma}\,; \Gamma}{1 + x : T[\mathsf{wk}] \in \overrightarrow{\Gamma}\,; \Gamma.S}$$

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

$\boxed{\vec{\Gamma} \vdash t : T}$: Term t has type T in context stack $\vec{\Gamma}$.

$$\frac{\vdash \vec{\Gamma}\,;\Gamma \qquad x : T \in \vec{\Gamma}\,;\Gamma}{\vec{\Gamma}\,;\Gamma \vdash v_x : T} \qquad \frac{\vec{\Gamma} \vdash t : T \qquad \vec{\Gamma} \vdash T \approx T' : \mathtt{Ty}_i}{\vec{\Gamma} \vdash t : T'}$$

$$\frac{\vec{\Delta} \vdash t : T \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash t[\vec{\sigma}\,] : T[\vec{\sigma}\,]} \qquad \frac{\vec{\Gamma}\,;\Gamma \vdash S : \mathtt{Ty}_i \qquad \vec{\Gamma}\,;\Gamma.S \vdash T : \mathtt{Ty}_i}{\vec{\Gamma}\,;\Gamma \vdash \Pi S.T : \mathtt{Ty}_i}$$

$$\frac{\begin{array}{c}\vec{\Gamma}\,;\Gamma \vdash S : \mathtt{Ty}_i \\ \vec{\Gamma}\,;\Gamma.S \vdash t : T\end{array}}{\vec{\Gamma}\,;\Gamma \vdash \lambda t : \Pi S.T} \qquad \frac{\begin{array}{cc}\vec{\Gamma}\,;\Gamma \vdash S : \mathtt{Ty}_i & \vec{\Gamma}\,;\Gamma.S \vdash T : \mathtt{Ty}_i \\ \vec{\Gamma}\,;\Gamma \vdash t : \Pi S.T & \vec{\Gamma}\,;\Gamma \vdash s : S\end{array}}{\vec{\Gamma}\,;\Gamma \vdash t\ s : T[\vec{I}\,, s]} \qquad \frac{\vdash \vec{\Gamma}}{\vec{\Gamma} \vdash \mathtt{Nat} : \mathtt{Ty}_i}$$

$$\frac{\vdash \vec{\Gamma}}{\vec{\Gamma} \vdash \mathtt{zero} : \mathtt{Nat}} \qquad \frac{\vec{\Gamma} \vdash t : \mathtt{Nat}}{\vec{\Gamma} \vdash \mathtt{succ}\ \ t : \mathtt{Nat}}$$

$$\frac{\begin{array}{cc}\vec{\Gamma}\,;\Gamma.\mathtt{Nat} \vdash M : \mathtt{Ty}_i & \vec{\Gamma}\,;\Gamma \vdash s : M[\vec{I}\,, \mathtt{zero}] \\ \vec{\Gamma}\,;\Gamma.\mathtt{Nat}.M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathtt{succ}\ \ v_1] & \vec{\Gamma}\,;\Gamma \vdash t : \mathtt{Nat}\end{array}}{\vec{\Gamma}\,;\Gamma \vdash \mathtt{elim}\, M\ \ s\, s'\, t : M[\vec{I}\,, t]} \qquad \frac{\vdash \vec{\Gamma}}{\vec{\Gamma} \vdash \mathtt{Ty}_i : \mathtt{Ty}_{1+i}}$$

$$\frac{\vec{\Gamma} \vdash T : \mathtt{Ty}_i}{\vec{\Gamma} \vdash T : \mathtt{Ty}_{1+i}} \qquad \frac{\vec{\Gamma}\,;\cdot \vdash T : \mathtt{Ty}_i}{\vec{\Gamma} \vdash \square T : \mathtt{Ty}_i} \qquad \frac{\vec{\Gamma}\,;\cdot \vdash t : T}{\vec{\Gamma} \vdash \mathtt{box}\ t : \square T}$$

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

$\boxed{\vec{\Gamma} \vdash t \approx s : T}$: Terms t and s of type T are equivalent in context stack $\vec{\Gamma}$.
Congruence rules:

$$\frac{\vdash \vec{\Gamma}\,;\Gamma \qquad x : T \in \vec{\Gamma}\,;\Gamma}{\vec{\Gamma}\,;\Gamma \vdash v_x \approx v_x : T} \qquad \frac{\begin{array}{c}\vec{\Gamma}\,;\Gamma \vdash S : \mathtt{Ty}_i \\ \vec{\Gamma}\,;\Gamma \vdash S \approx S' : \mathtt{Ty}_i \qquad \vec{\Gamma}\,;\Gamma.S \vdash T \approx T' : \mathtt{Ty}_i\end{array}}{\vec{\Gamma}\,;\Gamma \vdash \Pi S.T \approx \Pi S'.T' : \mathtt{Ty}_i}$$

$$\frac{\vec{\Delta} \vdash t \approx t' : T \qquad \vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma}\,' : \vec{\Delta}}{\vec{\Gamma} \vdash t[\vec{\sigma}\,] \approx t'[\vec{\sigma}\,'] : T[\vec{\sigma}\,]} \qquad \frac{\vec{\Gamma}\,;\Gamma \vdash S : \mathtt{Ty}_i \qquad \vec{\Gamma}\,;\Gamma.S \vdash t \approx t' : T}{\vec{\Gamma}\,;\Gamma \vdash \lambda t \approx \lambda t' : \Pi S.T}$$

$$\frac{\vec{\Gamma}\,;\Gamma \vdash S : \mathtt{Ty}_i \qquad \vec{\Gamma}\,;\Gamma.S \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma}\,;\Gamma \vdash t \approx t' : \Pi S.T \qquad \vec{\Gamma}\,;\Gamma \vdash s \approx s' : S}{\vec{\Gamma}\,;\Gamma \vdash t\ s \approx t'\ s' : T[\vec{I}\,, s]}$$

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

$$\frac{\vdash \vec{\Gamma}}{\vec{\Gamma} \vdash \mathtt{zero} \approx \mathtt{zero} : \mathtt{Nat}} \qquad \frac{\vec{\Gamma} \vdash t \approx t' : \mathtt{Nat}}{\vec{\Gamma} \vdash \mathtt{succ}\ \ t \approx \mathtt{succ}\ \ t' : \mathtt{Nat}}$$

$$\frac{\begin{array}{c}\vec{\Gamma}; \Gamma.\mathtt{Nat} \vdash M \approx M' : \mathtt{Ty}_i \qquad \vec{\Gamma}; \Gamma \vdash s_1 \approx s_1' : M[\vec{I}, \mathtt{zero}] \\ \vec{\Gamma}; \Gamma.\mathtt{Nat}.M \vdash s_2 \approx s_2' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathtt{succ}\ \ v_1] \qquad \vec{\Gamma}; \Gamma \vdash t \approx t' : \mathtt{Nat}\end{array}}{\vec{\Gamma}; \Gamma \vdash \mathtt{elim}\, M\ s_1\ s_2\ t \approx \mathtt{elim}\, M'\ s_1'\ s_2'\ t' : M[\vec{I}, t]}$$

$$\frac{\vec{\Gamma}; \cdot \vdash T \approx T' : \mathtt{Ty}_i}{\vec{\Gamma} \vdash \square T \approx \square T' : \mathtt{Ty}_i} \qquad \frac{\vec{\Gamma}; \cdot \vdash t \approx t' : T}{\vec{\Gamma} \vdash \mathtt{box}\ t \approx \mathtt{box}\ t' : \square T}$$

$$\frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t \approx t' : \square T \qquad \vdash \vec{\Gamma}; \vec{\Delta} \qquad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \mathtt{unbox}_n\ t \approx \mathtt{unbox}_n\ t' : T[\vec{I}; \Uparrow^n]}$$

β and η rules:

$$\frac{\begin{array}{c}\vec{\Gamma}; \Gamma \vdash S : \mathtt{Ty}_i \qquad \vec{\Gamma}; \Gamma.S \vdash T : \mathtt{Ty}_i \\ \vec{\Gamma}; \Gamma.S \vdash t : T \qquad \vec{\Gamma}; \Gamma \vdash s : S\end{array}}{\vec{\Gamma}; \Gamma \vdash (\lambda t)\ s \approx t[\vec{I}, s] : T[\vec{I}, s]} \qquad \frac{\begin{array}{c}\vec{\Gamma}; \Gamma \vdash S : \mathtt{Ty}_i \qquad \vec{\Gamma}; \Gamma.S \vdash T : \mathtt{Ty}_i \\ \vec{\Gamma}; \Gamma \vdash t : \Pi S.T\end{array}}{\vec{\Gamma}; \Gamma \vdash t \approx \lambda(t[\mathsf{wk}]\ v_0) : \Pi S.T}$$

$$\frac{\begin{array}{c}\vec{\Gamma}; \Gamma.\mathtt{Nat} \vdash M : \mathtt{Ty}_i \\ \vec{\Gamma}; \Gamma \vdash s : M[\vec{I}, \mathtt{zero}] \qquad \vec{\Gamma}; \Gamma.\mathtt{Nat}.M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathtt{succ}\ \ v_1]\end{array}}{\vec{\Gamma}; \Gamma \vdash \mathtt{elim}\, M\ \ s\ s'\mathtt{zero} \approx s : M[\vec{I}, \mathtt{zero}]}$$

$$\frac{\begin{array}{c}\vec{\Gamma}; \Gamma.\mathtt{Nat} \vdash M : \mathtt{Ty}_i \qquad \vec{\Gamma}; \Gamma \vdash s : M[\vec{I}, \mathtt{zero}] \\ \vec{\Gamma}; \Gamma.\mathtt{Nat}.M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathtt{succ}\ \ v_1] \qquad \vec{\Gamma}; \Gamma \vdash t : \mathtt{Nat}\end{array}}{\vec{\Gamma}; \Gamma \vdash \mathtt{elim}\, M\ s\ s'\ (\mathtt{succ}\ \ t) \approx s'[\vec{I}, t, \mathtt{elim}\, M\ \ s\ s' t] : M[\vec{I}, \mathtt{succ}\ \ t]}$$

$$\frac{\begin{array}{c}\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma}; \cdot \vdash t : T \\ \vdash \vec{\Gamma}; \vec{\Delta} \qquad |\vec{\Delta}| = n\end{array}}{\vec{\Gamma}; \vec{\Delta} \vdash \mathtt{unbox}_n\ (\mathtt{box}\ t) \approx t[\vec{I}; \Uparrow^n] : T[\vec{I}; \Uparrow^n]} \qquad \frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t : \square T}{\vec{\Gamma} \vdash t \approx \mathtt{box}\ (\mathtt{unbox}_1\ t) : \square T}$$

General K-substitution rules:

$$\frac{\vec{\Gamma} \vdash t : T}{\vec{\Gamma} \vdash t[\vec{I}] \approx t : T} \qquad \frac{\vec{\Gamma}' \vdash \vec{\sigma} : \vec{\Gamma}'' \qquad \vec{\Gamma} \vdash \vec{\delta} : \vec{\Gamma}' \qquad \vec{\Gamma}'' \vdash t : T}{\vec{\Gamma} \vdash t[\vec{\sigma} \circ \vec{\delta}] \approx t[\vec{\sigma}][\vec{\delta}] : T[\vec{\sigma} \circ \vec{\delta}]}$$

Variable rules:

$$\frac{\vdash \vec{\Gamma}; \Gamma.T \qquad x : T' \in \vec{\Gamma}; \Gamma}{\vec{\Gamma}; \Gamma \vdash v_x[\mathsf{wk}] \approx v_{1+x} : T'[\mathsf{wk}]}$$

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}; \Delta \qquad \vec{\Delta}; \Delta \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t : T[\vec{\sigma}]}{\vec{\Gamma} \vdash v_0[\vec{\sigma}, t] \approx t : T[\vec{\sigma}]}$$

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}; \Delta \qquad \vec{\Delta}; \Delta \vdash T' : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t : T'[\vec{\sigma}] \qquad x : T \in \vec{\Delta}; \Delta}{\vec{\Gamma} \vdash v_{1+x}[\vec{\sigma}, t] \approx v_x[\vec{\sigma}] : T[\vec{\sigma}]}$$

Π rules:

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

To describe how K-substitutions are propagated into different constructs, we need to define the weakening of a K-substitution:

$$q_T(\overrightarrow{\sigma}) := (\overrightarrow{\sigma} \circ \mathsf{wk}), v_0$$

where the subscript T is needed for the following typing rule:

$$\frac{\overrightarrow{\Gamma}\,; \Gamma \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}\,; \Delta \qquad \overrightarrow{\Delta}\,; \Delta \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma}\,; \Gamma.T[\overrightarrow{\sigma}] \vdash q_T(\overrightarrow{\sigma}) : \overrightarrow{\Delta}\,; \Delta.T}$$

We often omit this subscript when it can be inferred from the context.

$$\frac{\overrightarrow{\Gamma}\,; \Gamma \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}\,; \Delta \qquad \overrightarrow{\Delta}\,; \Delta \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Delta}\,; \Delta.S \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma}\,; \Gamma \vdash (\Pi S.T)[\overrightarrow{\sigma}] \approx \Pi S[\overrightarrow{\sigma}].(T[q(\overrightarrow{\sigma})]) : \mathtt{Ty}_i}$$

$$\frac{\overrightarrow{\Gamma}\,; \Gamma \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}\,; \Delta \qquad \overrightarrow{\Delta}\,; \Delta.S \vdash t : T}{\overrightarrow{\Gamma}\,; \Gamma \vdash (\lambda t)[\overrightarrow{\sigma}] \approx \lambda(t[q(\overrightarrow{\sigma})]) : (\Pi S.T)[\overrightarrow{\sigma}]}$$

$$\frac{\begin{array}{c} \overrightarrow{\Delta}\,; \Delta \vdash S : \mathtt{Ty}_i \\ \overrightarrow{\Delta}\,; \Delta.S \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}\,; \Delta \qquad \overrightarrow{\Delta}\,; \Delta \vdash s : \Pi S.T \qquad \overrightarrow{\Delta}\,; \Delta \vdash t : S \end{array}}{\overrightarrow{\Gamma} \vdash s\ t[\overrightarrow{\sigma}] \approx (s[\overrightarrow{\sigma}])\ (t[\overrightarrow{\sigma}]) : T[\overrightarrow{\sigma}, t[\overrightarrow{\sigma}]]}$$

`Nat` rules:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash N[\overrightarrow{\sigma}] \approx N : \mathtt{Ty}_i} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \mathtt{zero}[\overrightarrow{\sigma}] \approx \mathtt{zero} : N}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \qquad \overrightarrow{\Delta} \vdash t : N}{\overrightarrow{\Gamma} \vdash \mathtt{succ}\ \ t[\overrightarrow{\sigma}] \approx \mathtt{succ}\ \ (t[\overrightarrow{\sigma}]) : N}$$

$$\frac{\begin{array}{c} \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}\,; \Delta \qquad \overrightarrow{\Delta}\,; \Delta.N \vdash M : \mathtt{Ty}_i \\ \overrightarrow{\Delta}\,; \Delta \vdash s : M[\overrightarrow{I}, \mathtt{zero}] \qquad \overrightarrow{\Delta}\,; \Delta.N.M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathtt{succ}\ \ v_1] \qquad \overrightarrow{\Delta}\,; \Delta \vdash t : N \end{array}}{\overrightarrow{\Gamma} \vdash (\mathtt{elim}\, M\ \ s\ s'\, t)[\overrightarrow{\sigma}] \approx \mathtt{elim}\, M[q(\overrightarrow{\sigma})]\ \ (s[\overrightarrow{\sigma}])\ s'[q(q(\overrightarrow{\sigma}))](t[\overrightarrow{\sigma}]) : M[\overrightarrow{\sigma}, t[\overrightarrow{\sigma}]]}$$

□ rules:

$$\frac{\overrightarrow{\Delta}\,; \cdot \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \square T[\overrightarrow{\sigma}] \approx \square(T[\overrightarrow{\sigma}\,; \Uparrow^1]) : \mathtt{Ty}_i} \qquad \frac{\overrightarrow{\Delta}\,; \cdot \vdash t : T \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \mathsf{box}\ t[\overrightarrow{\sigma}] \approx \mathsf{box}\ (t[\overrightarrow{\sigma}\,; \Uparrow^1]) : \square T[\overrightarrow{\sigma}]}$$

$$\frac{\overrightarrow{\Delta}\,; \cdot \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Delta} \vdash t : \square T \qquad |\overrightarrow{\Delta}'| = n \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}\,; \overrightarrow{\Delta}'}{\overrightarrow{\Gamma} \vdash \mathtt{unbox}_n\ t[\overrightarrow{\sigma}] \approx \mathtt{unbox}_{\mathcal{O}(\overrightarrow{\sigma}, n)}\ (t[\overrightarrow{\sigma} \mid n]) : T[\overrightarrow{\sigma} \mid n; \Uparrow^{\mathcal{O}(\overrightarrow{\sigma}, n)}]}$$

`Ty` rule:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \mathtt{Ty}_i[\overrightarrow{\sigma}] \approx \mathtt{Ty}_i : \mathtt{Ty}_{1+i}}$$

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

Other rules:

$$\frac{\vec{\Gamma} \vdash t \approx t' : T}{\vec{\Gamma} \vdash t' \approx t : T} \qquad \frac{\vec{\Gamma} \vdash t \approx t' : T \qquad \vec{\Gamma} \vdash t' \approx t'' : T}{\vec{\Gamma} \vdash t \approx t'' : T}$$

$$\frac{\vec{\Gamma} \vdash t \approx t' : T \qquad \vec{\Gamma} \vdash T \approx T' : \mathtt{Ty}_i}{\vec{\Gamma} \vdash t \approx t' : T'} \qquad \frac{\vec{\Gamma} \vdash T \approx T' : \mathtt{Ty}_i}{\vec{\Gamma} \vdash T \approx T' : \mathtt{Ty}_{1+i}}$$

B Generating power functions

In this section, we give one more example for a program in MINT and run our NbE algorithm on it. The program that we use is a quintessential example of staged programming: the staged power function, which generates code of the n-th power of another number that will be given in a later stage. Recall that the $S4$ variant of MINT models staged computation. Following Davies & Pfenning (2001), we use the box modality to ascribe the type □ (Nat → Nat) to generated code. The implementation of the power function in MINT then follows familiar ideas.

```
pow  : Nat → □ (Nat → Nat)
pow zero     = box λ x → 1
pow (succ n) = box λ x → ((unbox₁ (pow n)) x) * x
```

In the zero case, pow returns a constant function always returning 1. Otherwise, if the input is succ n, pow recurses on n to generate code for pow n. To build our final result, i.e., the code for pow (succ n), we need to splice in the code from the recursive call. We use unbox$_1$ (pow n) to obtain code of type Nat → Nat. We then apply this function to x to obtain the code for n-th power of x and then multiply it with x to obtain the final result. The following is an execution of pow 2:

```
  pow 2
= box λ x → ((unbox₁ (pow 1)) x) * x
= box λ x → ((unbox₁ (box λ y → ((unbox₁ (pow 0)) y) * y)) x) * x
= box λ x → ((λ y → ((unbox₁ (pow 0)) y) * y) x) * x
= box λ x → (((unbox₁ (pow 0)) x) * x) * x
= box λ x → (((λ z → 1) x) * x) * x
= box λ x → x * x
```

In the rest of this section, we will run pow 2 using our NbE algorithm. To normalize pow 2, let us redefine pow in MINT syntax:

$$\mathsf{pow} : \Pi\mathtt{Nat}.\square(\Pi\mathtt{Nat}.\mathtt{Nat})$$

$$\mathsf{pow} := \lambda\ \mathtt{elim}\ (\square(\Pi\mathtt{Nat}.\mathtt{Nat}))\ \ (\mathsf{box}\ (\lambda 1))\ (\mathsf{box}\ (\lambda((\mathsf{unbox}_1\ v_1)\ v_0) * v_0))v_0$$

We assume multiplication is defined in the syntax and the semantics in the usual way. In this definition, we rewrite pattern matching into an elimination of a natural number and change named variables to de Bruijn indices. Instead of using explicit names, we associate binders and variables using colors. First, pow is a function, so we start with a λ. It introduces a variable, which we mark in red. Then we eliminate that variable (v_0) using elim. We supply the motive ($\square(\Pi\mathtt{Nat}.\mathtt{Nat})$), the base case (box ($\lambda 1$)), the step case, and the scrutinee (v_0). The step case introduces two more variables marked in orange, one for

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press

the predecessor (which we do not use) and the result of the recursive call. Then, we introduce another λ in box for constructing $\Pi\texttt{Nat.Nat}$. Inside, we first unbox the result of the recursive call (v_1) and obtain a function of type $\Pi\texttt{Nat.Nat}$. We apply that function to v_0 to obtain a Nat and then multiply that by an v_0 again. Note that although there are three v_0's in the code, they refer to two different variables, as distinguished by their colors.

Now let us first examine our evaluation process. For simplicity, we use **0**, **1**, and **2** for ze, su(ze), and su(su(ze)). To simplify the evaluation of $\llbracket \mathsf{pow}\ 2 \rrbracket(\uparrow^{\varepsilon;\cdot})$, we first define a few sub-evaluations. Let $\overrightarrow{\rho}$ be $\texttt{lext}(\uparrow^{\varepsilon;\cdot}, \mathbf{2})$:

$$\begin{aligned} P_0 &:= \llbracket \texttt{elim}\ (\square(\Pi\texttt{Nat.Nat}))\ \ (\texttt{box}\ (\lambda 1))\ (\texttt{box}\ (\lambda((\texttt{unbox}_1\ v_1)\ v_0) * v_0))0 \rrbracket(\overrightarrow{\rho}) \\ &= \texttt{box}(\Lambda(1, \texttt{ext}(\overrightarrow{\rho}))) \end{aligned}$$

Here P_0 represents the evaluation of the body of pow in the case of 0. The λ of the base case is evaluated to a Λ, in which the first component is still a syntactic term of Mint and the second component is the surrounding environment. The environment is extended modally because of the outer box. We define P_1 similarly:

$$\begin{aligned} P_1 &:= \llbracket \texttt{elim}\ (\square(\Pi\texttt{Nat.Nat}))\ \ (\texttt{box}\ (\lambda 1))\ (\texttt{box}\ (\lambda((\texttt{unbox}_1\ v_1)\ v_0) * v_0))1 \rrbracket(\overrightarrow{\rho}) \\ &= \texttt{box}(\Lambda((((\texttt{unbox}_1\ v_1)\ v_0) * v_0), \texttt{ext}(\texttt{lext}(\texttt{lext}(\overrightarrow{\rho}, \mathbf{0}), P_0)))) \end{aligned}$$

Similarly, in the final result, the function body $((\texttt{unbox}_1\ v_1)\ v_0) * v_0$ is captured. The environment is extended with **0** and P_0 because the step case has two open variables: the predecessor and the recursive call. Then, we define $\llbracket \mathsf{pow}\ 2 \rrbracket(\uparrow^{\varepsilon;\cdot})$:

$$\llbracket \mathsf{pow}\ 2 \rrbracket(\uparrow^{\varepsilon;\cdot}) = \texttt{box}(\Lambda((((\texttt{unbox}_1\ v_1)\ v_0) * v_0), \texttt{ext}(\texttt{lext}(\texttt{lext}(\overrightarrow{\rho}, \mathbf{1}), P_1))))$$

Note that this only evaluates the term. To perform NbE, we must also evaluate its type:

$$\llbracket \square(\Pi\texttt{Nat.Nat}) \rrbracket(\uparrow^{\varepsilon;\cdot}) = \blacksquare(Pi(\mathsf{N}, \texttt{Nat}, \uparrow^{\varepsilon;\cdot}))$$

Note that the first component of Pi is evaluated and in the domain while the second one is still syntax. Now, we compute $\mathsf{nbe}^{\square(\Pi\texttt{Nat.Nat})}_{\varepsilon;\cdot}(\mathsf{pow}\ 2)$, which expands to

$$\mathsf{nbe}^{\square(\Pi\texttt{Nat.Nat})}_{\varepsilon;\cdot}(\mathsf{pow}\ 2) = \mathsf{R}^{\mathsf{Nf}}_{\varepsilon;0}(\downarrow^{\llbracket \square(\Pi\texttt{Nat.Nat}) \rrbracket(\uparrow^{\varepsilon;\cdot})} (\llbracket \mathsf{pow}\ 2 \rrbracket(\uparrow^{\varepsilon;\cdot})))$$

We have computed the evaluations above and now move on to the readback:

$$\begin{aligned} &\mathsf{nbe}^{\square(\Pi\texttt{Nat.Nat})}_{\varepsilon;\cdot}(\mathsf{pow}\ 2) \\ = {} &\mathsf{R}^{\mathsf{Nf}}_{\varepsilon;0}(\downarrow^{\blacksquare(Pi(\mathsf{N},\texttt{Nat},\uparrow^{\varepsilon;\cdot}))} (\llbracket \mathsf{pow}\ 2 \rrbracket(\uparrow^{\varepsilon;\cdot}))) \\ = {} &\mathsf{R}^{\mathsf{Nf}}_{\varepsilon;0}(\downarrow^{\blacksquare(Pi(\mathsf{N},\texttt{Nat},\uparrow^{\varepsilon;\cdot}))} (\texttt{box}(\Lambda((((\texttt{unbox}_1\ v_1)\ v_0) * v_0), \texttt{ext}(\texttt{lext}(\texttt{lext}(\overrightarrow{\rho}, \mathbf{1}), P_1)))))) \\ = {} &\texttt{box}\ \mathsf{R}^{\mathsf{Nf}}_{\varepsilon;0;0}(\downarrow^{Pi(\mathsf{N},\texttt{Nat},\uparrow^{\varepsilon;\cdot})} (\Lambda((((\texttt{unbox}_1\ v_1)\ v_0) * v_0), \texttt{ext}(\texttt{lext}(\texttt{lext}(\overrightarrow{\rho}, \mathbf{1}), P_1))))) \\ = {} &\texttt{box}\ \lambda\ \mathsf{R}^{\mathsf{Nf}}_{\varepsilon;0;1}(\downarrow^{\mathsf{N}} (\llbracket((\texttt{unbox}_1\ v_1)\ v_0) * v_0\rrbracket(\texttt{lext}(\texttt{ext}(\texttt{lext}(\texttt{lext}(\overrightarrow{\rho}, \mathbf{1}), P_1)), \uparrow^{\mathsf{N}} (l_0))))) \\ = {} &\texttt{box}\ \lambda\ \mathsf{R}^{\mathsf{Nf}}_{\varepsilon;0;1}(\downarrow^{\mathsf{N}} ((\texttt{unbox} \cdot (1, P_1)) \cdot \uparrow^{\mathsf{N}} (l_0))) * v_0 \\ = {} &\texttt{box}\ \lambda\ v_0 * v_0 \end{aligned}$$

Thus, we obtain a normal form as we previously claimed.

https://doi.org/10.1017/S0956796823000060 Published online by Cambridge University Press