CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Formalizing domain models of the typed and the untyped lambda calculus in Agda

Master's thesis in Computer science and engineering

David Lidell

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

MASTER'S THESIS 2020

Formalizing domain models of the typed and the untyped lambda calculus in Agda

David Lidell

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Formalizing domain models of the typed and the untyped lambda calculus in Agda
David Lidell

Supervisor: Peter Dybjer, Department of Computer Science and Engineering
Examiner: Andreas Abel, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Formalizing domain models of the typed and the untyped lambda calculus in Agda

David Lidell
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We present a domain interpretation of the simply typed and the untyped lambda calculus. The interpretations are constructed using the notion of category with families, with added structure. Specifically, for the simply typed case we construct a simply typed category with families of (a version of) neighborhood systems with structures supporting binary product types and function types. For the untyped case, we construct a unityped category with families of neighborhood systems, with added lambda structure.

The work is completely formalized in the dependently typed programming language and proof assistant Agda. The categories with families with added structure are formalized as records and then instantiated with neighborhood systems as objects and approximable mappings as morphisms. In constructing the appropriate neighborhood system for the untyped model, we make use of Agda's sized types; this feature enables us to prove transitivity of the ordering relation between untyped neighborhoods.

Keywords: Agda, categories with families, domain interpretation, lambda calculus, sized types

Acknowledgements

I want to thank Peter Dybjer for supervising this project on his own time. His knowledge, encouragement, and enthusiasm for the project has been invaluable. I also want to thank him, along with Thierry Coquand, for holding the course "Types for Programs and Proofs", where I first became familiar with Agda and type theory. Agda is such a joy to work with, so I want to thank everyone involved in its creation. Particularly Ulf Norell for designing it, and Andreas Abel for introducing sized types; the latter proved to be very useful in this project. I also want to thank Andreas Abel, along with my thesis opponents Warrick Macmillan and Gustav Örtenberg, for their constructive criticism that helped me improve the thesis. The project owes a great deal to Michael Hedberg, and I thank him for the work he did as a PhD student. Finally, thanks to my friends and family for their support, and especially to my wife and daughter for putting up with me working evenings and weekends.

David Lidell, Gothenburg, October 2020

Contents

1

Introduction

Martin-Löf type theory [10] is a dependent type theory designed to serve as an alternative foundation for constructive mathematics. Due to the Curry-Howard correspondence, it may also be viewed as a programming language [12]. Its fundamental building blocks are types and terms, along with contexts (assignments of types to free variables). It is formally defined in terms of judgments and inference rules, using a natural deduction-style sequent calculus.

In order to formally study a program, it needs to be given a semantics. One of the major approaches is that of *denotational* semantics, where each phrase of the language studied is taken to denote a mathematical object. Naively, one might think of using sets and functions as denotations. However, as was discovered while searching for a denotational semantics of the lambda calculus, some programs simply cannot be given a denotation in such terms.

Around 1970, with the goal of assigning meaning to such programs, Dana Scott [15] laid the foundation for *domain theory*, which studies very particular sets with additional structure, and functions between them, well-suited for use in denotational semantics. Scott later presented his theory in alternative ways, first using the language of *neighborhood systems* [17], and later *information systems* [16]. In the 1980s, Martin-Löf [11] worked on giving a denotational semantics to his type theory, using a notion of formal neighborhoods adapted from Scott's neighborhood systems. This work was later expanded on by Palmgren and Stoltenberg-Hansen [14] using Scott's notion of domains.

Also building on Martin-Löf's work, Hedberg [7][8] constructs a domain model of a fragment of partial type theory (type theory extended with general recursion) *inside* total type theory. He does this by formalizing a cartesian closed category of domains in a proof assistant based on type theory. His PhD thesis [7] contains an overview of the construction and some brief and informal proofs. Unfortunately, it does not contain the implementation, and we were not able to retrieve it elsewhere.

In this thesis, we adapt Hedberg's work and give a different domain model of the simply typed lambda calculus using the notion of categories with families (cwfs). We go beyond his work both by introducing a consistency relation to our neighborhood systems, and by extending the result to a model of the untyped lambda calculus. In doing so, we take another step toward a full model of partial type theory within total type theory.

In chapter 2, we present some of the necessary background, introducing categories with families, the prerequisite notions of domain theory, and the language in which our work is formalized. Chapter 3 describes the sets with structure and the mappings between them that we will use for denotations. In chapter 4 and 5, respectively, we present domain models of the typed and the untyped lambda calculus. We end with a short summary in chapter 6.

When discussing proofs, we will be quite informal, and refer the reader who is interested in the details to the appendix, where the code is found. It is also available at **https://github.com/DoppeD/ConsistentCwfsOfDomains**. We use conventional notation in the text as much as possible, but the notation used in the code is occasionally somewhat idiosyncratic.

2

Background

2.1 Categories with families

A well known categorical model of dependent type theories is that given by *locally cartesian closed categories* (lcccs). An alternative model is provided by *categories with families* (cwfs), first described in Dybjer [6]. In comparison to lcccs, their syntax is closer to that of Martin-Löf type theory. Moreover, extensional equality is not built-in, but an optional addition, and the same is true of Σ-types and Π-types. Clairambault and Dybjer [5] show that the 2-category of cwfs equipped with structures supporting Σ, Π, and extensional identity types is biequivalent to the 2-category of lcccs. At their core, cwfs provide a foundation for basic reasoning with dependent types. For reference, we present their definition as given in Castellan et al. [4], almost verbatim. The category Fam, of families of sets, is first defined:

Definition 2.1.1. The objects of Fam are families $(U_x)_{x\in X}$. A morphism with source $(U_x)_{x\in X}$ and target $(V_y)_{y\in Y}$ is a pair consisting of a reindexing function $f : X \to Y$, and a family $(g_x)_{x\in X}$ where for each $x \in X, g_x : U_x \to V_{f(x)}$ is a function.

Definition 2.1.2. A category with families (cwf) consists of the following:

- A category $\mathscr{C}$ with a terminal object 1.
  Notation and terminology. We use Γ, Δ, etc, to range over objects of $\mathscr{C}$, and refer to those as "contexts". Likewise, we use δ, γ, etc, to range over morphisms, and refer to those as "substitutions". We refer to 1 as the *empty context*. We write $\langle\rangle_\Gamma \in \mathscr{C}(\Gamma, 1)$ for the terminal map, representing the empty substitution.

- A Fam-valued presheaf $T : \mathscr{C}^{\mathrm{op}} \to \mathrm{Fam}$.
  Notation and terminology. If $T(\Gamma) = (U_x)_{x\in X}$, we write $X = \mathrm{Ty}(\Gamma)$ and refer to its elements as *types over* Γ - we use A, B, C to range over such "types". For $A \in X = \mathrm{Ty}(\Gamma)$, we write $U_A = \mathrm{Tm}_A(\Gamma)$ and refer to its elements as *terms of type A in context* Γ. Finally, for $\gamma : \Delta \to \Gamma$, the functorial action yields

  $$T(\gamma) : (\mathrm{Tm}_A(\Gamma))_{A\in\mathrm{Ty}(\Gamma)} \to (\mathrm{Tm}_B(\Delta))_{B\in\mathrm{Ty}(\Delta)}$$

  consisting of a pair of a reindexing function $\_[\gamma] : \mathrm{Ty}(\Gamma) \to \mathrm{Ty}(\Delta)$ referred to

as *substitution on types*, and for each $A \in \mathrm{Ty}(\Gamma)$ a function $\_[\gamma] : \mathrm{Tm}_A(\Gamma) \to \mathrm{Tm}_{A[\gamma]}(\Delta)$ referred to as *substitution on terms*.

- A *context comprehension operation* which to a given context $\Gamma \in \mathscr{C}_0$ assigns a context $\Gamma \cdot A$ and two projections

$$\mathrm{p}_{\Gamma,A} : \Gamma \cdot A \to \Gamma \qquad \mathrm{q}_{\Gamma,A} \in \mathrm{Tm}_{A[\mathrm{p}_{\Gamma,A}]}(\Gamma \cdot A)$$

satisfying the following universal property: for all $\gamma : \Delta \to \Gamma$, for all $a \in \mathrm{Tm}_A[\gamma](\Delta)$ there is a unique $\langle \gamma, a \rangle : \Delta \to \Gamma \cdot A$ such that

$$\mathrm{p}_{\Gamma,A} \circ \langle \gamma, a \rangle = \gamma \qquad \mathrm{q}_{\Gamma,A}[\langle \gamma, a \rangle] = a.$$

We say that $(\Gamma \cdot A, \mathrm{p}_{\Gamma,A}, \mathrm{q}_{\Gamma,A})$ is a *context comprehension* of Γ and A.

The "notation and terminology" annotations strongly suggest how these components relate to Martin-Löf type theory, particularly the version with explicit substitutions.

Perhaps the most mysterious looking part is the context comprehension operations. We find the following diagram helpful for understanding the projection p:

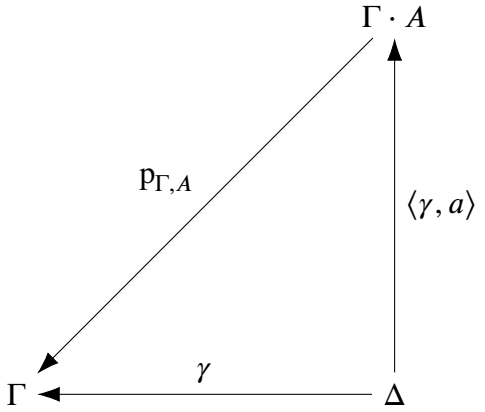


Figure 2.1: Schematic view of the projection p

The equation $\mathrm{p}_{\Gamma,A} \circ \langle \gamma, a \rangle = \gamma$ says that the morphism p acts as a projection in the expected way, by "shaving off" the last introduced type. When replacing terms in Δ for the free variables of $\Gamma \cdot A$, what results is a term in Δ, reflected by the substitution-on-terms-function. The term q with its related equation can be thought of as playing the role of the variable introduced via context comprehension.

In Dybjer [6], it is shown that cwfs can also be defined algebraically as a collection of sorts, operations on these sorts, and rules for the operations, in what is known as a *generalized algebraic theory* (GAT) - see Cartmell [3]. Using such a description is a convenient way to formalize cwfs, since it makes all of their components explicit. As we will focus on the simpler constructions that are the *simply typed* cwfs (scwfs) and *unityped* cwfs (ucwfs), we will present the generalized algebraic theory of cwfs as it specializes to these constructions in chapter 4 and 5, respectively.

2.2 Domain theory

The basic idea of domain theory is to describe a (potentially infinite) computation as being the limit of a sequence of partial, finite approximating computations. In this setting, types are interpreted as sets of elements ordered by information content, typically complete partial orders[1]. Programs are interpreted as functions between domains, satisfying a particular notion of continuity.

Of particular interest are those elements that represent finite data, usually called the compact or finite elements. Historically, one began with choosing a domain, such as a complete partial order, to represent a data type, and then defined the compact elements of that domain. In modern formulations of domain theory, such as those using neighborhood systems or information systems, one instead works directly with partially ordered sets of compact elements, and with *approximable mappings* between them. These mappings are functions from the elements of one set of compact elements to the ideals of another. From such a set of compact elements a domain is generated from its ideal completion. Approximable mappings similarly generate continuous functions between domains.

Definition 2.2.1. A nonempty poset $(S, \sqsubseteq)$ is *directed* if, for any $x, y \in S$, there exists a $z \in S$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$.

Definition 2.2.2. An *ideal* of a poset $(S, \sqsubseteq)$ is a directed subset $I \subseteq S$ that is also a lower set: for any $y \in I$ and any $x \in S$, if $x \sqsubseteq y$ then $y \in I$.

Definition 2.2.3. For any element x of a poset $(S, \sqsubseteq)$, the smallest ideal that contains x is the *principal ideal generated by x*.

Definition 2.2.4. The set of all ideals over a poset $(S, \sqsubseteq)$, ordered by set inclusion, is called the *ideal completion* of S.

To give some intuition for the structures we will work with, we visualize the type of natural numbers as the following tree of partial, finite approximations:

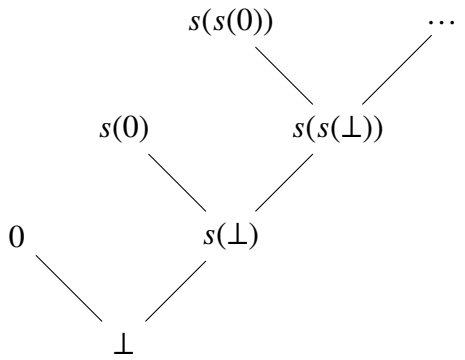


Figure 2.2: The (lazy) natural numbers.

[1] Initially, Scott [15] proposed using complete lattices as domains.

The leaves $0, s(0), s(s(0)), \ldots$ represent the corresponding natural numbers. The symbol $\bot$ represents an element with no information content. More interesting is the partial information provided by $s(\bot)$, which represents knowing that we are dealing with a natural number greater than 0, but not which one. We take the boolean values as another example:

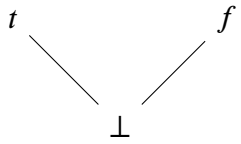


Figure 2.3: The boolean values.

As the images indicate, the elements should be ordered in a consistent way, such that an element x is only below an element y if x contains no information not also contained in y. In the natural numbers example, 0 is not below $s(0)$ since the two elements provide conflicting information.

Martin-Löf's [11] work on developing a denotational semantics for his type theory was done in terms of *formal neighborhoods*; special kinds of compact elements suitable for approximating the programs of type theory. Since our thesis builds on Hedberg's PhD thesis, which was directly inspired by Martin-Löf's notes and uses the neighborhood terminology, we will follow suit and refer to the elements of our structures as "neighborhoods". We will refer to the structures themselves as "neighborhood systems", due to their similarity to those defined by Scott [17]. In defining his version of neighborhood systems, Hedberg [7] made the simplifying assumption that every pair of neighborhoods is consistent, i.e. has a least upper bound. The resulting structure is a pointed upper semilattice, the ideal completion of which is a complete semilattice. This put a restriction on what data types he could model—it excludes, for example, the natural numbers and the booleans. We will remove this restriction by introducing a consistency relation between neighborhoods.

2.3 Agda

We carry out our formalization in the functional dependently typed language Agda [13], developed in Gothenburg. The language is syntactically similar to Haskell. It is based on intuitionistic type theory, and due to the Curry-Howard correspondence, it serves as a proof assistant for constructive mathematics. It is a total language, which means that all functions must be shown to be terminating. Nonetheless, Agda offers the option of enabling specific features, among them one that allows non-terminating functions to be type checked. All such features can be disabled by using the **safe** pragma, which every module in our code uses.

The type of small types in Agda is Set. In order to avoid paradoxes similar to Russell's paradox, it is not the case that Set : Set. Instead, Agda introduces a type Set_1 of large types, and defines $\mathrm{Set} : \mathrm{Set}_1$. For the same reason, there is also an even larger type Set_2, with $\mathrm{Set}_1 : \mathrm{Set}_2$, and so on. These types of types are called *universes*, and the subscripts their

levels. We keep universe levels fixed in our code, as we consider universe polymorphism to offer few benefits while making the code more difficult to read. The main consequence of this choice is that it forces our formalized sets of neighborhoods to be small types. Lifting this restriction, if desired, is completely straightforward.

Many arguments to functions in the presented formalization are missing; these are either implicit arguments hidden through Agda's variable generalization feature, or passed as parameters to modules. All code is type checked with Agda version 2.6.1. No external libraries are used.

2.4 Related work

The formalization of the abstract definitions of the different categories with families, with related structures, is adapted from Brilakis [2]. The definitions of neighborhood systems, approximable mappings, and the arrow neighborhood system are adapted from Hedberg [7]. However, we have reversed the information-content ordering of neighborhood systems, so that we are working with upper semilattices, ideals, monotone mappings, and so on. This is for intuitive considerations, and is more in line with standard domain theory. As with the definition of cwfs in section 2.1, the definitions of scwfs, ucwfs, and related structures are taken almost verbatim from Castellan et al. [4].

3

Formalization of neighborhood systems and approximable mappings

3.1 Neighborhood systems

We define our neighborhood systems as partially ordered sets with a least element. To model types such as the natural numbers or the boolean values, we must introduce the machinery needed to be able to differentiate between those neighborhoods that are consistent, and those that are not. As previously mentioned, the neighborhoods 0 and $s(0)$ of the natural numbers are not consistent, in that they provide conflicting information about the result of a computation. As such, the supremum operator is defined only for consistent pairs.

We will diverge from Martin-Löf's [11] work in adding a consistency relation. While he defines formal intersections—binary suprema in our version—for *arbitrary* pairs of formal neighborhoods, and then defines a consistency predicate that singles out the ones that are consistent, we find it more natural to embed a consistency relation between neighborhoods in the definition of neighborhood systems:

```
record NbhSys : Set₁ where
  field
    Nbh : Set
    _⊑_ : Nbh → Nbh → Set
    Con : Nbh → Nbh → Set
    _⊔_[_] : (x y : Nbh) → Con x y → Nbh
    ⊥ : Nbh

    Con-⊔ : ∀ {x y z} → x ⊑ z → y ⊑ z → Con x y

    ⊑-refl : ∀ {x} → x ⊑ x
    ⊑-trans : ∀ {x y z} → x ⊑ y → y ⊑ z → x ⊑ z
    ⊑-⊥   : ∀ {x} → ⊥ ⊑ x
    ⊑-⊔   : ∀ {x y z} → y ⊑ x → z ⊑ x → (con : Con y z) →
            (y ⊔ z [ con ]) ⊑ x
    ⊑-⊔-fst : ∀ {x y} → (con : Con x y) → x ⊑ (x ⊔ y [ con ])
    ⊑-⊔-snd : ∀ {x y} → (con : Con x y) → y ⊑ (x ⊔ y [ con ])
```

The reader may note that this definition does not include the axiom of antisymmetry. The reason is that the notion of equivalence of interest here is not identity. Instead, we consider two neighborhoods x and y equivalent iff $x \sqsubseteq y \wedge y \sqsubseteq x$. With respect to this equivalence relation $\sqsubseteq$ is a partial order.

In order to access a field in a record, one must specify the particular relevant instance of the record. This makes the syntax cumbersome for mixfix operators, in this case $\sqsubseteq$ and $\sqcup$. For this reason, we introduce the following short-hand:

```
-- Some simplifying syntax.
[_]_⊑_ : (D : NbhSys) → (x y : NbhSys.Nbh D) → Set
[ A ] x ⊑ y = NbhSys._⊑_ A x y

[_]_⊔_ : (D : NbhSys) → (x y : NbhSys.Nbh D) → NbhSys.Nbh D
[ A ] x ⊔ y = NbhSys._⊔_ A x y
```

When there is risk of ambiguity in using the symbols Nbh, $\sqsubseteq$, $\sqcup$ and $\bot$, we will denote them with the subscript D, where D is the neighborhood system in question.

Other structures, such as the product of neighborhood systems, will also need to be defined. These are in our case specific to either scwfs or ucwfs, so we introduce them as needed in the corresponding chapters.

3.2 Approximable mappings

Since domains are generated from the ideals of a neighborhood system's neighborhoods, we should define approximable mappings in a way such that they map neighborhoods of one neighborhood system to ideals of neighborhoods of another. We will formalize these mappings as binary relations.

3.2.1 Defining approximable mappings

Recall that an ideal is a directed lower subset of the target neighborhood system's elements. Let γ denote a would-be approximable mapping. To ensure that its image is a lower set, we require that it is downwards closed. To ensure that any two consistent elements in it have a common upper bound, we require it to be upwards directed. Moreover, we want γ to map to the least element, which also ensures that its image is not the empty set. More informative input should yield more informative output, which we express as monotonicity. Finally, an approximable mapping should take consistent pairs to consistent pairs:

```
record Appmap (D D' : NbhSys) : Set₁ where
  field
    -- The mapping itself.
    _↦_ : NbhSys.Nbh D → NbhSys.Nbh D' → Set

    -- Axioms for the mapping.
    ↦-mono : ∀ {x y z} → [ D ] x ⊑ y → x ↦ z → y ↦ z
```

```
↦-bottom : ∀ {x} → x ↦ NbhSys.⊥ D'
↦-↓closed : ∀ {x y z} → [ D' ] y ⊑ z → x ↦ z → x ↦ y
↦-↑directed : ∀ {x y z} → x ↦ y → x ↦ z → (con : NbhSys.Con D' y z) →
                x ↦ ([ D' ] y ⊔ z [ con ])
↦-con : ∀ {x y x' y'} → x ↦ y → x' ↦ y' → NbhSys.Con D x x' →
          NbhSys.Con D' y y'
```

In this text we will use the notation $\mathbf{x} \overset{\gamma}{\mapsto} \mathbf{y}$ to express that $(\mathbf{x}, \mathbf{y})$ is in the relation specified by γ. We will say that "γ maps $\mathbf{x}$ to $\mathbf{y}$", and ask the reader to bear in mind that γ is a relation and may map $\mathbf{x}$ to other neighborhoods as well.

Note that we do not explicitly include totality as an axiom, as it follows immediately from **↦-bottom**:

```
↦-total : (γ : Appmap D D') → ∀ {x} →
            Σ (NbhSys.Nbh D') λ y → [ γ ] x ↦ y
↦-total {D' = D'} γ = NbhSys.⊥ D' , Appmap.↦-bottom γ
```

Other derived mappings one might expect, such as the composition of mappings, will come later. The reason for this is that we soon will be working with tuples of neighborhoods, and would rather not have to define every such derivation twice.

3.2.2 Equivalence of approximable mappings

Working in type theory, we need to explicitly define extensional equality of approximable mappings. In order to keep the code independent of external libraries, we adapt the formalization of equivalence relations from the standard library:

```
-- The below code is adapted from the standard library.
-- The point is to remove any dependencies on libraries.
-- For the purpose of the project, universe levels can be fixed.

Rel : (A : Set₁) → Set₂
Rel A = A → A → Set₁

Reflexive : Rel A → Set₁
Reflexive _≈_ = ∀ {x} → x ≈ x

Symmetric : Rel A → Set₁
Symmetric _≈_ = ∀ {x y} → x ≈ y → y ≈ x

Transitive : Rel A → Set₁
Transitive _≈_ = ∀ {x y z} → x ≈ y → y ≈ z → x ≈ z

record IsEquivalence (_≈_ : Rel A) : Set₁ where
  field
    refl : Reflexive _≈_
    sym : Symmetric _≈_
    trans : Transitive _≈_
```

We then define:

```
data _≼_ : Rel (Appmap D D') where
  ≼-intro : {γ δ : Appmap D D'} →
            (∀ {x y} → [ γ ] x ↦ y → [ δ ] x ↦ y) → γ ≼ δ

data _≈_ : Rel (Appmap D D') (lsuc lzero) where
  ≈-intro : {γ δ : Appmap D D'} → γ ≼ δ → δ ≼ γ → γ ≈ δ
```

Showing that _≈_ defines an equivalence relation is trivial.

4

A domain model of the typed lambda calculus

An scwf is a special case of a cwf; one where the Fam-valued presheaf of types is constant, which corresponds to types being independent of terms. We present this construction first, and later adapt the results obtained here to the untyped version in chapter 5.

We begin with discussing a *plain* scwf of neighborhood systems. Plain scwfs formalize theories of functions of several variables. We add structures supporting unit types and binary product types in section 4.2, and a structure supporting function types in section 4.3.

The connection between scwfs with structure and cartesian closed categories - the classical categorical model of the simply typed lambda calculus - is explored in Castellan et al. [4].

4.1 A plain scwf of neighborhood systems

We present the definition of plain scwfs, and a formalization thereof. We then present our implementation's types, contexts and morphisms, and the additional components needed in a plain scwf.

4.1.1 Abstract definition

We first give the definition of a plain scwf:

Definition 4.1.1. An *scwf* consists of the following:

- A category $\mathscr{C}$ with a terminal object 1.
- A set Ty.
- A family of presheaves $\mathrm{Tm}_{\mathscr{A}} : \mathscr{C}^{\mathrm{op}} \to \mathrm{Set}$ for $\mathscr{A} \in \mathrm{Ty}$.
- A *context comprehension* operation which to $\Gamma \in \mathscr{C}_0$ and $\mathscr{A} \in \mathrm{Ty}$ assigns a context

$\Gamma \cdot \mathscr{A}$ and two projections

$$\mathrm{p}_{\Gamma,\mathscr{A}} : \Gamma \cdot \mathscr{A} \to \Gamma \qquad \mathrm{q}_{\Gamma,\mathscr{A}} \in \mathrm{Tm}_{\mathscr{A}}(\Gamma \cdot \mathscr{A})$$

satisfying the following universal property: for all $\gamma : \Delta \to \Gamma$, for all $a \in \mathrm{Tm}_{\mathscr{A}}(\Delta)$, there is a unique $\langle \gamma, a \rangle : \Delta \to \Gamma \cdot \mathscr{A}$ such that

$$\mathrm{p}_{\Gamma,\mathscr{A}} \circ \langle \gamma, a \rangle = \gamma \qquad \mathrm{q}_{\Gamma,\mathscr{A}}[\langle \gamma, a \rangle] = a.$$

We will formalize a *contextual* scwf. These admit a length function from their contexts to the set of natural numbers that ensures that all contexts are inductively created via context comprehension, up to isomorphism:

```
record Scwf : Set₂ where
  field
    Ty : Set₁
    Ctx : Nat → Set₁
    Tm : Ctx n → Ty → Set₁
    Sub : Ctx m → Ctx n → Set₁

    _≈_ : ∀ {Γ 𝒜} → Rel (Tm {n} Γ 𝒜) (lsuc lzero)
    _≅_ : ∀ {Γ Δ} → Rel (Sub {m} {n} Γ Δ) (lsuc lzero)

    isEquivT : ∀ {Γ 𝒜} → IsEquivalence (_≈_ {n} {Γ} {𝒜})
    isEquivS : ∀ {Γ Δ} → IsEquivalence (_≅_ {m} {n} {Γ} {Δ})

    ◇ : Ctx zero

    _•_ : Ctx n → Ty → Ctx (suc n)

    q : (Γ : Ctx n) → (𝒜 : Ty) → Tm (Γ • 𝒜) 𝒜
    _[_] : ∀ {𝒜 Γ Δ} → Tm {n} Δ 𝒜 → Sub {m} Γ Δ → Tm Γ 𝒜

    id : (Γ : Ctx n) → Sub Γ Γ
    _∘_ : ∀ {Γ Δ Θ} → Sub {n} {o} Δ Θ → Sub {m} Γ Δ → Sub Γ Θ
    ⟨⟩ : {Γ : Ctx n} → Sub Γ ◇
    ⟨_,_⟩ : ∀ {Γ Δ 𝒜} → Sub {n} {m} Δ Γ → Tm Δ 𝒜 → Sub Δ (Γ • 𝒜)
    p : (Γ : Ctx n) → (𝒜 : Ty) → Sub (Γ • 𝒜) Γ
```

The axioms of the generalized algebraic theory of cwfs simplify to the following:

```
idL : ∀ {Γ Δ} → (γ : Sub {n} {m} Δ Γ) → ((id Γ) ∘ γ) ≅ γ
idR : ∀ {Γ Δ} → (γ : Sub {n} {m} Δ Γ) → (γ ∘ (id Δ)) ≅ γ
subAssoc : ∀ {Γ Δ Θ Λ} → (γ : Sub {m} {n} Γ Δ) →
           (δ : Sub {n} {o} Δ Θ) → (θ : Sub {o} {r} Θ Λ) →
           ((θ ∘ δ) ∘ γ) ≅ (θ ∘ (δ ∘ γ))

idSub : ∀ {Γ 𝒜} → (t : Tm {n} Γ 𝒜) → (t [ id Γ ]) ≈ t
compSub : ∀ {Γ Δ Θ 𝒜} → (t : Tm {n} Δ 𝒜) →
```

```
                (γ : Sub {m} Γ Δ) → (δ : Sub {o} Θ Γ) →
                (t [ γ ∘ δ ]) ≈ ((t [ γ ]) [ δ ])

id₀ : id ⋄ ≅ ⟨⟩
<>-zero : ∀ {Γ Δ} → (γ : Sub {m} {n} Γ Δ) → (⟨⟩ ∘ γ) ≅ ⟨⟩

pCons : ∀ {𝒜 Γ Δ} → (γ : Sub {n} {m} Δ Γ) → (t : Tm Δ 𝒜) →
        ((p Γ 𝒜) ∘ ⟨ γ , t ⟩) ≅ γ
qCons : ∀ {𝒜 Γ Δ} → (γ : Sub {n} {m} Δ Γ) → (t : Tm Δ 𝒜) →
        ((q Γ 𝒜) [ ⟨ γ , t ⟩ ]) ≈ t
idExt : ∀ {𝒜 Γ} → (id (_•_ {m} Γ 𝒜)) ≅ ⟨ p Γ 𝒜 , q Γ 𝒜 ⟩
compExt : ∀ {Γ Δ 𝒜} → (t : Tm Δ 𝒜) →
                (γ : Sub {n} {m} Δ Γ) → (δ : Sub Γ Δ) →
                (⟨ γ , t ⟩ ∘ δ) ≅ ⟨ γ ∘ δ , t [ δ ] ⟩
```

Since we are working with more general notions of equivalence than definitional equality, we explicitly encode that operators should take equivalent terms to equivalent terms as the following congruence rules:

```
subCong : ∀ {𝒜 Γ Δ} → {t t' : Tm {m} Γ 𝒜} →
                {γ γ' : Sub {n} Δ Γ} → t ≈ t' →
                γ ≅ γ' → (t [ γ ]) ≈ (t' [ γ' ])
<,>-cong : ∀ {𝒜 Γ Δ} → {t t' : Tm {m} Γ 𝒜} →
                 {γ γ' : Sub {m} {n} Γ Δ} → t ≈ t' →
                 γ ≅ γ' → ⟨ γ , t ⟩ ≅ ⟨ γ' , t' ⟩
∘-cong : ∀ {Γ Δ Θ} → {γ δ : Sub {n} {o} Δ Θ} →
        {γ' δ' : Sub {m} Γ Δ} → γ ≅ δ →
        γ' ≅ δ' → (γ ∘ γ') ≅ (δ ∘ δ')
```

4.1.2 The types and objects

The objects of our scwf are lists of types indexed by length. They are formalized in the usual manner, with a **nil**-constructor and a **cons**-constructor:

```
data List : Nat → Set₁ → Set₁ where
  [] : List 0 A
  _::_ : A → List n A → List (suc n) A

-- Types are neighborhood systems.
Ty : Set₁
Ty = NbhSys

-- A context is a list of types.
Ctx : Nat → Set₁
Ctx n = List n Ty
```

The terminal object is simply the empty list [].

4.1.3 The morphisms

Since the morphisms of a category with families represent simultaneous substitutions of variables for terms, we need a notion of tuples of neighborhoods, and what it means for them to be well-typed in contexts. We call tuples that are well-typed in contexts *valuations*. Their definition is very similar to that of lists:

```
data Valuation : Ctx n → Set where
  ⟨⟨⟩⟩ : Valuation []
  ⟨⟨_,,_⟩⟩ : NbhSys.Nbh 𝒜 → Valuation Γ → Valuation (𝒜 :: Γ)
```

The two commas in the second constructor are there to avoid parsing errors when we later introduce pairs. Note the usage of Agda's variable generalization feature: the **n** in **Ctx n** and the Γ have been specified elsewhere as denoting a natural number and a context, respectively. We use the following notation for valuations of single typed contexts:

```
⟨⟨_⟩⟩ : ∀ x → Valuation (𝒜 :: [])
⟨⟨ x ⟩⟩ = ⟨⟨ x ,, ⟨⟨⟩⟩ ⟩⟩
```

In the text, we will simply omit the angled brackets when dealing with such valuations and write **x** instead of ⟨⟨**x**⟩⟩.

The empty tuple is the only valuation of the empty context, and a valuation of a context Γ, extended with a neighborhood x of type $\mathscr{A}$, is a valuation of $\mathscr{A}$ :: Γ.

By defining ⊑, ⊔, ⊥, and consistency for valuations component-wise, we form a neighborhood system. Note that the functions **ctHead** and **ctTail** serve the same purpose for valuations as **head** and **tail** do for lists:

```
data ⊑ᵥ : (Γ : Ctx n) → (x y : Valuation Γ) →
          Set where
  ⊑ᵥ-nil : ⊑ᵥ [] ⟨⟨⟩⟩ ⟨⟨⟩⟩
  ⊑ᵥ-cons : (Γ : Ctx (suc n)) → ∀ {x y} →
            [ head Γ ] (ctHead x) ⊑ (ctHead y) →
            ⊑ᵥ (tail Γ) (ctTail x) (ctTail y) →
            ⊑ᵥ Γ x y

data ValCon : (Γ : Ctx n) → (x y : Valuation Γ) → Set where
  con-nil : ValCon [] ⟨⟨⟩⟩ ⟨⟨⟩⟩
  con-tup : ∀ {Γ : Ctx n} → ∀ {x y x y} →
            NbhSys.Con 𝒜 x y → ValCon Γ x y →
            ValCon (𝒜 :: Γ) ⟨⟨ x ,, x ⟩⟩ ⟨⟨ y ,, y ⟩⟩

_⊔ᵥ_[_] : (x : Valuation Γ) → (y : Valuation Γ) → ValCon Γ x y →
          Valuation Γ
_⊔ᵥ_[_] ⟨⟨⟩⟩ ⟨⟨⟩⟩ _ = ⟨⟨⟩⟩
_⊔ᵥ_[_] {Γ = h :: _} ⟨⟨ x ,, x ⟩⟩ ⟨⟨ y ,, y ⟩⟩ (con-tup conxy conxy)
  = ⟨⟨ [ h ] x ⊔ y [ conxy ] ,, x ⊔ᵥ y [ conxy ] ⟩⟩
```

```
⊥ᵥ : Valuation Γ
⊥ᵥ {Γ = []} = ⟨⟨⟩⟩
⊥ᵥ {Γ = h :: _} = ⟨⟨ NbhSys.⊥ h ,, ⊥ᵥ ⟩⟩
```

Proving that these satisfy the neighborhood system axioms is easily done inductively, by making use of the corresponding proofs for the underlying types. We name the neighborhood system **ValNbhSys**. We will in the text use the notation $\mathbf{x} \in \mathrm{Val}(\Gamma)$ to denote that $\mathbf{x}$ is a valuation of Γ. Note that we use boldface to differentiate between ordinary neighborhoods x, y, z and valuation neighborhoods $\mathbf{x}, \mathbf{y}, \mathbf{z}$.

A morphism from a context Γ of length m to a context Δ of length n is thus an n-tuple of m-place approximable mappings, or in other words an approximable mapping from $\mathrm{Val}(\Gamma)$ to $\mathrm{Val}(\Delta)$. We use the following simplifying notation:

```
tAppmap : (Γ : Ctx m) → (Δ : Ctx n) → Set₁
tAppmap Γ Δ = Appmap (ValNbhSys Γ) (ValNbhSys Δ)
```

In the remainder of this chapter, whenever we talk about approximable mappings - or just "mappings" - between contexts, we refer to this definition.

4.1.3.1 The terminal morphism

The morphism to the terminal object is the approximable mapping with the following relation:

```
data _empty↦_ : Valuation Γ → Valuation [] → Set where
  empty↦-intro : {x : Valuation Γ} → x empty↦ ⟨⟨⟩⟩
```

This is a rather obvious definition, since there is only one valuation of the empty context. Proving that the above relation satisfies the required axioms is entirely trivial. We name the mapping **emptyMap**.

4.1.3.2 Identity morphisms

The identity morphism for an object Γ is defined as the approximable mapping that takes a valuation $\mathbf{x}$ the principal ideal generated by it:

```
data _id↦_ : Valuation Γ → Valuation Γ → Set where
  id↦-intro : ∀ {x y} → ⊑ᵥ Γ y x → x id↦ y
```

The proofs of the approximable mapping axioms are very simple, following immediately from the domain axioms of the types of the underlying context. We name the mapping **idMap**.

4.1.3.3 Morphism composition

Morphism composition is the usual composition of relations: if γ is an approximable mapping from Γ to Δ, and δ one from Δ to Θ, then $\delta \circ \gamma$ maps $\mathbf{x} \in \mathrm{Val}(\Gamma)$ to $\mathbf{z} \in \mathrm{Val}(\Theta)$ if and only there exists $\mathbf{y} \in \mathrm{Val}(\Delta)$ such that $\mathbf{x} \overset{\gamma}{\mapsto} \mathbf{y}$ and $\mathbf{y} \overset{\delta}{\mapsto} \mathbf{z}$.

```
data _∘↦_ (δ : tAppmap Δ Θ) (γ : tAppmap Γ Δ) :
          Valuation Γ → Valuation Θ → Set where
  ∘↦-intro : ∀ {x y z} → [ γ ] x ↦ y → [ δ ] y ↦ z →
             _∘↦_ δ γ x z
```

The proofs of most of the axioms for approximable mappings easily follow from the corresponding proofs for the mappings γ and δ. Showing that the relation is upwards directed is only slightly more involved:

For appropriate tuples $\mathbf{x}, \mathbf{y}, \mathbf{z}$, proofs that $\mathbf{x} \overset{\delta \circ \gamma}{\mapsto} \mathbf{y}$ and $\mathbf{x} \overset{\delta \circ \gamma}{\mapsto} \mathbf{z}$ consist of the sub-proofs,

$$\mathbf{x} \overset{\gamma}{\mapsto} \mathbf{y}' \quad \mathbf{y}' \overset{\delta}{\mapsto} \mathbf{y} \quad \text{and} \quad \mathbf{x} \overset{\gamma}{\mapsto} \mathbf{z}' \quad \mathbf{z}' \overset{\delta}{\mapsto} \mathbf{z},$$

for some $\mathbf{y}', \mathbf{z}' \in \mathrm{Val}(\Delta)$. From the upwards directed property of γ, it follows that $\mathbf{x} \overset{\gamma}{\mapsto} \mathbf{y}' \sqcup \mathbf{z}'$. From the monotonicity of δ, we get $\mathbf{y}' \sqcup \mathbf{z}' \overset{\delta}{\mapsto} \mathbf{y}$ and $\mathbf{y}' \sqcup \mathbf{z}' \overset{\delta}{\mapsto} \mathbf{z}$. Finally, the upwards directed property of δ yields $\mathbf{y}' \sqcup \mathbf{z}' \overset{\delta}{\mapsto} \mathbf{y} \sqcup \mathbf{z}$. It follows that $\mathbf{x} \overset{\delta \circ \gamma}{\mapsto} \mathbf{y} \sqcup \mathbf{z}$. Note that the consistency of $\mathbf{y}'$ and $\mathbf{z}'$ follows from the consistency of γ.

4.1.4 Presheaves of terms

The family of presheaves of terms is formalized abstractly as the function **Tm** in section 4.1.1. It associates to any context Γ and any type $\mathscr{A}$ the set of terms of type $\mathscr{A}$ that are closed under Γ. Since $\mathscr{A}$ contains all the information needed about these terms, it is natural that **Tm** returns an approximable mapping from the context to the type. So terms, like substitutions, are approximable mappings, and we can reuse the definitions and proofs of section 3.2.

In order to keep using the **tAppmap** notation, we identify the type $\mathscr{A}$ with the context $[\mathscr{A}]$, and define the terms of our implemented scwf - in pseudocode - in this way:

```
Tm Γ 𝒜 = tAppmap Γ [ 𝒜 ]
```

Explicit substitution on terms _[_] is defined as composition of approximable mappings. This should make intuitive sense: consider contexts Γ and Δ, a type $\mathscr{A}$, and an approximable mapping $\gamma : \Gamma \to \Delta$. Recall that γ represents a substitution of terms in Γ for free variables in Δ, and that performing such a substitution results in a term in Γ. The more information we have about a valuation of Γ, the more information we have about a corresponding valuation of Δ, related to by γ. In turn, the more information we have about this valuation, the more information we have about a corresponding term of type $\mathscr{A}$ in Δ, which is also a term in Γ.

4.1.5 Context comprehension

Context comprehension is defined by list **cons**, so that $\Gamma \cdot \mathscr{A} = \mathscr{A} :: \Gamma$.

Since both substitutions and terms have been defined as approximable mappings, the projections **p** and **q** and their respective proofs of the mapping axioms are almost identical.

For $\mathbf{x} \in \mathrm{Val}(\Gamma)$ and a neighborhood a, $\mathbf{p}$ should act as the identity mapping for $\mathbf{x}$, while $\mathbf{q}$ should act as the identity mapping for a. Their underlying relations are hence defined as:

```
data _p↦_ : Valuation (𝒜 :: Γ) → Valuation Γ → Set where
  p↦-intro : {x : Valuation (𝒜 :: Γ)} → ∀ {y} →
             ⊑ᵥ Γ y (ctTail x) → x p↦ y

data _q↦_ : Valuation (𝒜 :: Γ) → Valuation [ 𝒜 ] → Set where
  q↦-intro : {x : Valuation (𝒜 :: Γ)} →
             {y : Valuation [ 𝒜 ]} →
             [ 𝒜 ] (ctHead y) ⊑ (ctHead x) → x q↦ y
```

Proving that these relations satisfy the required axioms is done by making direct use of the axioms for the valuation neighborhood system, and requires no further explanation.

4.1.5.1 Substitution extension

The last definition we require for our plain scwf is that of the substitution extension morphism $\langle \_,\_ \rangle$. Bearing in mind the definitions of $\mathbf{p}$ and $\mathbf{q}$, the equations involving them and $\langle \_,\_ \rangle$ that must hold, and the discussion in section 2.1, we define its relation as:

```
data ⟨⟩↦ (γ : tAppmap Δ Γ) (t : tAppmap Δ [ 𝒜 ]) :
         Valuation Δ → Valuation (𝒜 :: Γ) → Set where
  ⟨⟩↦-intro : ∀ {x y} → [ γ ] x ↦ (ctTail y) →
              [ t ] x ↦ ⟨⟨ ctHead y ⟩⟩ → ⟨⟩↦ γ t x y

-- Some simplifying notation.
[⟨_,_⟩]_↦_ : (γ : tAppmap Δ Γ) → (t : tAppmap Δ [ 𝒜 ]) →
             Valuation Δ → Valuation (𝒜 :: Γ) → Set
[⟨ γ , t ⟩] x ↦ y = ⟨⟩↦ γ t x y
```

Proving the axioms is, again, straightforward.

4.1.6 Proving the axioms of the GAT of scwfs

Most of the proofs of the axioms of the generalized algebraic theory of scwfs are immediately obvious. The main exception is one of the directions of **compExt**:

```
compExtLemma₂ : ∀ {x y} → [ ⟨ γ ∘ δ , t ∘ δ ⟩ ] x ↦ y →
                [ ⟨ γ , t ⟩ ∘ δ ] x ↦ y
```

Let $\mathbf{y} = \langle\langle y, , \mathbf{y}' \rangle\rangle$. From the assumptions we get

$$\mathbf{x} \overset{\delta}{\mapsto} \mathbf{z} \quad \mathbf{z} \overset{\gamma}{\mapsto} \mathbf{y}' \quad \text{and} \quad \mathbf{x} \overset{\delta}{\mapsto} \mathbf{w} \quad \mathbf{w} \overset{t}{\mapsto} y$$

for some $\mathbf{z}, \mathbf{w} \in \mathrm{Val}(\Delta)$. Since δ is a consistent approximable mapping, it follows that $\mathbf{z}$ and $\mathbf{w}$ are consistent. From the monotonicity of γ and t we get

$$(\mathbf{z} \sqcup \mathbf{w}) \overset{\gamma}{\mapsto} \mathbf{y}' \quad \text{and} \quad (\mathbf{z} \sqcup \mathbf{w}) \overset{t}{\mapsto} y.$$

The definition of substitution extension then gives us

$$(\mathbf{z} \sqcup \mathbf{w}) \overset{\langle \gamma, t \rangle}{\longmapsto} \langle\langle y, , \mathbf{y}' \rangle\rangle \quad \text{i.e.} \quad (\mathbf{z} \sqcup \mathbf{w}) \overset{\langle \gamma, t \rangle}{\longmapsto} \mathbf{y}.$$

Finally, we make use of the upwards directed property of δ to get $\mathbf{x} \overset{\delta}{\mapsto} (\mathbf{z} \sqcup \mathbf{w})$, and compose the two results.

4.2 Adding a product type

We define and show our implementation of two structures related to finite product types. They are variants of the $\times$- and $\mathbb{N}_1$-structures described in Castellan et al. [4].

4.2.1 Definition of a weak $\times$-structure

We begin by presenting the definition of a $\times$-structure:

Definition 4.2.1. A $\times$-*structure* on an scwf $\mathscr{C}$ consists of, for each $\Gamma \in \mathscr{C}_0$ and $\mathscr{A}, \mathscr{B} \in \mathrm{Ty}$, a type $\mathscr{A} \times \mathscr{B} \in \mathrm{Ty}$ and term formers

$$\begin{aligned} \mathrm{fst}_{\Gamma,\mathscr{A},\mathscr{B}}(-) &: \mathrm{Tm}_{\mathscr{A}\times\mathscr{B}}(\Gamma) \to \mathrm{Tm}_{\mathscr{A}}(\Gamma) \\ \mathrm{snd}_{\Gamma,\mathscr{A},\mathscr{B}}(-) &: \mathrm{Tm}_{\mathscr{A}\times\mathscr{B}}(\Gamma) \to \mathrm{Tm}_{\mathscr{B}}(\Gamma) \\ \langle -, - \rangle &: \mathrm{Tm}_{\mathscr{A}}(\Gamma) \times \mathrm{Tm}_{\mathscr{B}}(\Gamma) \to \mathrm{Tm}_{\mathscr{A}\times\mathscr{B}}(\Gamma) \end{aligned}$$

such that, for appropriate γ, a, b, c

$$\mathrm{fst}(\langle a, b \rangle) = a \tag{4.1}$$
$$\mathrm{snd}(\langle a, b \rangle) = b \tag{4.2}$$
$$\langle \mathrm{fst}(c), \mathrm{snd}(c) \rangle = c \tag{4.3}$$
$$\langle a, b \rangle[\gamma] = \langle a[\gamma], b[\gamma] \rangle \tag{4.4}$$

Equation 4.3 is the rule of surjective pairing. We will follow Martin-Löf and introduce a specific bottom element $\bot_x$ when defining the binary product type, in order to model lazy evaluation of pairs. As a result, surjective pairing will not hold, as we differentiate between $\bot_x$ and the pair $\langle \bot_{\mathscr{A}}, \bot_{\mathscr{B}} \rangle$. The former neighborhood corresponds to not knowing anything about the evaluation of a term, while the latter corresponds to knowing that it is a pair, but nothing more. Note, however, that the rule holds when $c \neq \bot_x$. We will call a $\times$-structure without surjective pairing a *weak* $\times$-*structure*. We formalize it as a record with added congruence rules:

```
record Prod-scwf : Set₂ where
  field
    scwf : Scwf
  open Scwf scwf public
  field
```

```
_×_ : Ty → Ty → Ty

fst : ∀ {Γ 𝒜 ℬ} → Tm {m} Γ (𝒜 × ℬ) → Tm Γ 𝒜
snd : ∀ {Γ 𝒜 ℬ} → Tm {m} Γ (𝒜 × ℬ) → Tm Γ ℬ
<_,_> : ∀ {Γ 𝒜 ℬ} → Tm {m} Γ 𝒜 → Tm Γ ℬ → Tm Γ (𝒜 × ℬ)

fstAxiom : ∀ {Γ 𝒜 ℬ} → {t : Tm {m} Γ 𝒜} → {u : Tm Γ ℬ} →
             fst < t , u > ≈ t
sndAxiom : ∀ {Γ 𝒜 ℬ} → {t : Tm {m} Γ 𝒜} → {u : Tm Γ ℬ} →
             snd < t , u > ≈ u
pairSub : ∀ {Γ Δ 𝒜 ℬ} → {t : Tm Γ 𝒜} → {u : Tm Γ ℬ} →
            {γ : Sub {n} {m} Δ Γ} →
            (< t , u > [ γ ]) ≈ < t [ γ ] , u [ γ ] >

fstCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ (𝒜 × ℬ)} → t ≈ t' →
            fst t ≈ fst t'
sndCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ (𝒜 × ℬ)} → t ≈ t' →
            snd t ≈ snd t'
pairCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ 𝒜} → {u u' : Tm Γ ℬ} →
             t ≈ t' → u ≈ u' →
             < t , u > ≈ < t' , u' >
```

4.2.2 Definition of a weak $\mathbb{N}_1$-structure

An $\mathbb{N}_1$ structure is defined in [4] as:

Definition 4.2.2. An $\mathbb{N}_1$-*structure* on an scwf consists of a type $\mathbb{N}_1 \in \mathrm{Ty}$, and for each Γ a term $0_1 \in \mathrm{Tm}_{\mathbb{N}_1}(\Gamma)$ such that for all $c \in \mathrm{Tm}_{\mathbb{N}_1}(\Gamma), 0_1 = c$.

In our case, we have to alter the definition somewhat. Since our type must contain one information-carrying element in addition to the bottom element in order to be meaningful, there are generally several distinct terms in $\mathrm{Tm}_{\mathbb{N}_1}(\Gamma)$. The property specified in the definition above is derived by analogy from the axioms of the $\times$-structure, by thinking of the $\mathbb{N}_1$-structure as a nullary product, and removing rules 4.1 and 4.2. We further remove rule 4.3 and arrive at the following definition:

Definition 4.2.3. A weak $\mathbb{N}_1$-*structure* on an scwf of domains consists of a type $\mathbb{N}_1 \in \mathrm{Ty}$, and for each Γ a term $0_1 \in \mathrm{Tm}_{\mathbb{N}_1}(\Gamma)$ such that for all $c \in \mathrm{Tm}_{\mathbb{N}_1}(\Gamma)$ and all $\gamma \in \mathscr{C}(\Delta, \Gamma), 0_1[\gamma] = 0_1$.

We add to the record defined in section 4.2.1 the fields of the weak $\mathbb{N}_1$-structure:

```
-- We merge the ℕ₁-structure with the ×-structure.
ℕ₁ : Ty
0₁ : ∀ {Γ} → Tm {m} Γ ℕ₁
ℕ₁-sub : ∀ {Γ Δ} → {γ : Sub {n} {m} Δ Γ} →
           (0₁ [ γ ]) ≈ 0₁
```

4.2.3 Implementing the weak ×-structure

To implement the weak ×-structure, we must define the product of neighborhood systems and the related morphisms, and prove that these satisfy the given axioms.

4.2.3.1 Defining the product of neighborhood systems

We define the set of neighborhoods of the product of two neighborhood systems D and D' as consisting of the least element $\bot_x$ and pairs of neighborhoods of D and D' using the ordinary definition of pairs:

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

We use the unusual symbol ⊠ as we want to reserve the conventional × for the product neighborhood system itself.

The domain is defined very similarly to the neighborhood system of valuations, with the ordering relation being defined component-wise in terms of the ordering relations of the corresponding underlying neighborhood systems, and similarly for the supremum operator and the consistency relation. This also makes the proofs of the axioms almost identical.

```
data ProdNbh : Set where
  ⊥x : ProdNbh
  <_,_> : NbhSys.Nbh D → NbhSys.Nbh D' → ProdNbh

data _⊑x_ : ProdNbh → ProdNbh → Set where
  ⊑x-intro1 : ∀ {x} → ⊥x ⊑x x
  ⊑x-intro2 : ∀ {x y x' y'} → [ D ] x ⊑ y →
                [ D' ] x' ⊑ y' →
                < x , x' > ⊑x < y , y' >

data ProdCon : ProdNbh → ProdNbh → Set where
  conx-⊥1 : ∀ {x} → ProdCon x ⊥x
  conx-⊥2 : ∀ {x} → ProdCon ⊥x x
  con-pair : ∀ {x1 x2 x'1 x'2} → NbhSys.Con D x1 x'1 →
               NbhSys.Con D' x2 x'2 →
               ProdCon < x1 , x2 > < x'1 , x'2 >

_⊔x_ : ProdNbh → ProdNbh → ProdNbh
⊥x ⊔x ⊥x = ⊥x
⊥x ⊔x < x1 , x2 > = < x1 , x2 >
< x1 , x2 > ⊔x ⊥x = < x1 , x2 >
< x1 , x2 > ⊔x < x'1 , x'2 >
  = < [ D ] x1 ⊔ x'1 , [ D' ] x2 ⊔ x'2 >
```

We visualize the product of the boolean values and boolean values below to help provide intuition for the ordering relation of the product type:

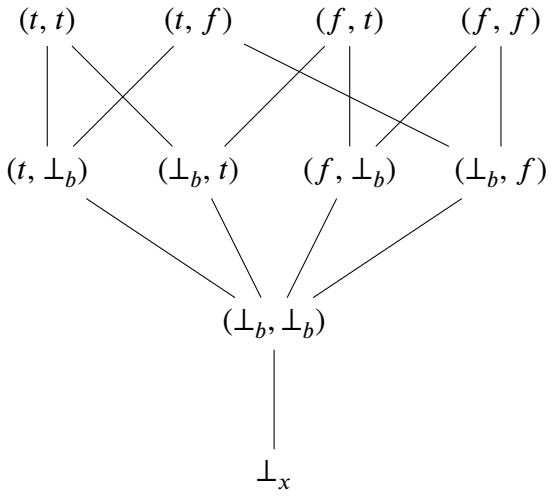


Figure 4.1: The product of booleans. The elements are pairs of boolean neighborhoods, ordered component-wise. The neighborhood $\bot_b$ represents the least element of the boolean neighborhood system.

4.2.3.2 The morphisms

The relations for the three morphisms **fst**, **snd**, and **<_,_>** are defined in fairly obvious ways. If a mapping $t \in \mathrm{Tm}_{\mathscr{A}\times\mathscr{B}}(\Gamma)$ maps $\mathbf{x} \in \mathrm{Val}(\Gamma, \mathscr{A})$ to the pair $(y_1, y_2) \in \mathscr{A} \times \mathscr{B}$, only then do we have that $\mathbf{x} \overset{\mathbf{fst}(t)}{\mapsto} y_1$, and similarly for **snd** and y_2.

For $t \in \mathrm{Tm}_{\mathscr{A}}(\Gamma)$, $u \in \mathrm{Tm}_{\mathscr{B}}(\Gamma)$, and $\mathbf{x} \in \mathrm{Val}(\Gamma, \mathscr{A})$, the pairing mapping **<_,_>** maps **x** to $(y_1, y_2) \in \mathscr{A} \times \mathscr{B}$ if and only if both $\mathbf{x} \overset{t}{\mapsto} y_1$ and $\mathbf{x} \overset{u}{\mapsto} y_2$:

```
data <>↦ (t : tAppmap Γ [ 𝒜 ]) (u : tAppmap Γ [ ℬ ]) :
         Valuation Γ → Valuation [ 𝒜 × ℬ ] → Set where
  <>↦-intro₁ : ∀ {x} → <>↦ t u x ⟨⟨ ⊥ₓ ⟩⟩
  <>↦-intro₂ : ∀ {x y₁ y₂} → [ t ] x ↦ ⟨⟨ y₁ ⟩⟩ →
               [ u ] x ↦ ⟨⟨ y₂ ⟩⟩ →
               <>↦ t u x ⟨⟨ < y₁ , y₂ > ⟩⟩

data fst↦ (t : tAppmap Γ [ 𝒜 × ℬ ]) :
          Valuation Γ → Valuation [ 𝒜 ] → Set where
  fst-intro₁ : ∀ {x y} → [ 𝒜 ] y ⊑ NbhSys.⊥ 𝒜 → fst↦ t x ⟨⟨ y ⟩⟩
  fst-intro₂ : ∀ {x y₁ y₂} → [ t ] x ↦ ⟨⟨ < y₁ , y₂ > ⟩⟩ →
               fst↦ t x ⟨⟨ y₁ ⟩⟩

data snd↦ (t : tAppmap Γ [ 𝒜 × ℬ ]) :
          Valuation Γ → Valuation [ ℬ ] → Set where
  snd-intro₁ : ∀ {x y} → [ ℬ ] y ⊑ NbhSys.⊥ ℬ → snd↦ t x ⟨⟨ y ⟩⟩
  snd-intro₂ : ∀ {x y₁ y₂} → [ t ] x ↦ ⟨⟨ < y₁ , y₂ > ⟩⟩ →
               snd↦ t x ⟨⟨ y₂ ⟩⟩
```

As the definitions suggest, proving the mapping axioms for these relations is done directly using those of their constituent mappings. Note that the rule of surjective pairing would hold were it not for the distinguished least element.

In order to be able to prove the axioms of **fst** and **snd**, it is not enough to specify that they map to the least elements of the respective domains. Instead, their first constructors state that they map to any y such that $y \sqsubseteq \bot$. Of course, since $\bot \sqsubseteq y$, this means that any such y is equivalent to $\bot$.

4.2.3.3 Proving the neighborhood system axioms

Proving **fstAxiom** and **sndAxiom** is simple. Proving one direction of **pairSub** is similiar to proving that mapping composition is upwards directed:

```
pairSubLemma₂ : {γ : tAppmap Δ Γ} → ∀ {x y} →
                [ < t ∘ γ , u ∘ γ > ] x ↦ y →
                [ < t , u > ∘ γ ] x ↦ y
```

From the assumption $\mathbf{x} \overset{\langle t \circ \gamma, u \circ \gamma \rangle}{\longmapsto} (y_1, y_2)$ we get

$$\mathbf{x} \overset{\gamma}{\mapsto} \mathbf{z} \quad \mathbf{z} \overset{t}{\mapsto} y_1 \quad \text{and} \quad \mathbf{x} \overset{\gamma}{\mapsto} \mathbf{w} \quad \mathbf{w} \overset{u}{\mapsto} y_2$$

for some $\mathbf{z}, \mathbf{w} \in \mathrm{Val}(\Gamma)$. From γ being consistent we get that $\mathbf{z}$ and $\mathbf{w}$ also are. Since γ is upwards directed, we get $\mathbf{x} \overset{\gamma}{\mapsto} \mathbf{z} \sqcup \mathbf{w}$, and from the monotonicity of t and u, we get $\mathbf{z} \sqcup \mathbf{w} \overset{t}{\mapsto} y_1$ and $\mathbf{z} \sqcup \mathbf{w} \overset{u}{\mapsto} y_2$. Hence $\mathbf{z} \sqcup \mathbf{w} \overset{\langle t, u \rangle}{\mapsto} (y_1, y_2)$, and the result follows.

4.2.4 Implementing the weak $\mathbb{N}_1$-structure

As alluded to in section 4.2.2, the unit neighborhood system consists of the bottom element and an element with information content that we call 0_1, not to be confused with the term $0_1 \in \mathrm{Tm}_{\mathbb{N}_1}(\Gamma)$ that we will define in a moment.

```
data UnitNbh : Set where
  ⊥₁ : UnitNbh
  0₁ : UnitNbh

data _⊑₁_ : UnitNbh → UnitNbh → Set where
  ⊥₁-bot : ∀ {x} → ⊥₁ ⊑₁ x
  0₁-refl : 0₁ ⊑₁ 0₁

data UnitCon : UnitNbh → UnitNbh → Set where
  allCon : ∀ {x y} → UnitCon x y

_⊔₁_[_] : (x : UnitNbh) → (y : UnitNbh) → UnitCon x y → UnitNbh
⊥₁ ⊔₁ y [ _ ] = y
0₁ ⊔₁ y [ _ ] = 0₁
```

The term $0_1 \in \mathrm{Tm}_{\mathbb{N}_1}(\Gamma)$ can only be defined in one way: it's the mapping that maps any valuation of Γ to both the element 0_1 and $\bot_1$:

```
data _0₁↦_ {Γ : Ctx n} : Valuation Γ → Valuation [ ℕ₁ ] →
                          Set where
  0₁↦∀ : ∀ {x y} → x 0₁↦ y
```

The type can be visualized this way:

Figure 4.2: The unit type, consisting only of the least element and a canonical element 0_1.

4.3 Adding a function type

We define and show our implementation of a structure supporting function types. Our particular implementation does not support η-equality. As with the $\times-$ and $\mathbb{N}_1$-structure, this is because of the introduction of a distinguished least element to the related neighborhood system.

4.3.1 Definition of a weak $\Rightarrow$-structure

The definition of a $\Rightarrow$-structure is as follows:

Definition 4.3.1. A $\Rightarrow$-structure on an scwf $\mathscr{C}$ consists of, for each $\Gamma \in \mathscr{C}_0$ and $\mathscr{A}, \mathscr{B} \in \mathrm{Ty}$, a type $\mathscr{A} \Rightarrow \mathscr{B}$ along with term formers

$$\begin{aligned} \lambda_{\Gamma,\mathscr{A},\mathscr{B}} &: \mathrm{Tm}_{\Gamma.\mathscr{A}}(\mathscr{B}) \to \mathrm{Tm}_{\Gamma}(\mathscr{A} \Rightarrow \mathscr{B}) \\ \mathrm{ap}_{\Gamma,\mathscr{A},\mathscr{B}} &: \mathrm{Tm}_{\Gamma}(\mathscr{A} \Rightarrow \mathscr{B}) \times \mathrm{Tm}_{\Gamma}(\mathscr{A}) \to \mathrm{Tm}_{\Gamma}(\mathscr{B}) \end{aligned}$$

such that, for $a \in \mathrm{Tm}_{\Gamma}(\mathscr{A})$, $b \in \mathrm{Tm}_{\Gamma.\mathscr{A}}(\mathscr{B})$, $c \in \mathrm{Tm}_{\Gamma}(\mathscr{A} \Rightarrow \mathscr{B})$, and $\gamma \in \mathscr{C}(\Delta, \Gamma)$:

$$\begin{aligned} \lambda_{\Gamma,\mathscr{A},\mathscr{B}}(b)[\gamma] &= \lambda_{\Delta,\mathscr{A},\mathscr{B}}(b[\langle \gamma \circ \mathrm{p}_{\Delta,\mathscr{A}}, \mathrm{q}_{\Delta,\mathscr{A}} \rangle]) \\ \mathrm{ap}_{\Gamma,\mathscr{A},\mathscr{B}}(c, a)[\gamma] &= \mathrm{ap}_{\Delta,\mathscr{A},\mathscr{B}}(c[\gamma], a[\gamma]) \\ \mathrm{ap}_{\Gamma,\mathscr{A},\mathscr{B}}(\lambda_{\Gamma,\mathscr{A},\mathscr{B}}(b), a) &= b[\langle \mathrm{id}_{\Gamma}, a \rangle] \\ \lambda_{\Gamma,\mathscr{A},\mathscr{B}}(\mathrm{ap}_{\Gamma.\mathscr{A},\mathscr{A},\mathscr{B}}(c[\mathrm{p}_{\Gamma,\mathscr{A}}], \mathrm{q}_{\Gamma,\mathscr{A}})) &= c \end{aligned}$$

The last rule is that of η-equality. Like we did with the product type, we will introduce a distinguished least element to the arrow neighborhood system, which will invalidate this

rule. The rule does hold, however, if this element is removed. We will call a ⇒-structure without η-equality a *weak ⇒-structure*.

Although the three structures are independent of one another, we choose to formalize an scwf with added weak ×-, $\mathbb{N}_1$-, and ⇒-structures, along with new congruence rules for the latter:

```
record ProductArrow-scwf : Set₂ where
  field
    prod-scwf : Prod-scwf
  open Prod-scwf prod-scwf public
  field
    _⇒_ : Ty → Ty → Ty

    lam : ∀ {Γ 𝒜 ℬ} → Tm (_•_ {m} Γ 𝒜) ℬ → Tm Γ (𝒜 ⇒ ℬ)
    ap  : ∀ {Γ 𝒜 ℬ} → Tm Γ (𝒜 ⇒ ℬ) → Tm {m} Γ 𝒜 → Tm Γ ℬ

    lamSub : ∀ {Γ Δ 𝒜 ℬ} → (γ : Sub {n} {m} Δ Γ) →
             (t : Tm (Γ • 𝒜) ℬ) →
             (lam t [ γ ]) ≈ (lam (t [ ⟨ γ ∘ p Δ 𝒜 , q Δ 𝒜 ⟩ ]))
    apSub : ∀ {Γ Δ 𝒜 ℬ} → (γ : Sub {n} {m} Δ Γ) →
             (t : Tm Γ (𝒜 ⇒ ℬ)) → (u : Tm Γ 𝒜) →
             (ap (t [ γ ]) (u [ γ ])) ≈ (ap t u [ γ ])

    β : ∀ {Γ 𝒜 ℬ} → {t : Tm {m} Γ 𝒜} → {u : Tm (Γ • 𝒜) ℬ} →
        (ap (lam u) t) ≈ (u [ ⟨ id Γ , t ⟩ ])

    lamCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm (_•_ {m} Γ 𝒜) ℬ} →
              t ≈ t' → (lam t) ≈ (lam t')
    apCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ (𝒜 ⇒ ℬ)} →
             ∀ {u u'} → t ≈ t' → u ≈ u' →
             (ap t u) ≈ (ap t' u')
```

4.3.2 The arrow neighborhood system

In order to construct a weak ⇒-structure for our scwf, we must first define the type $\mathscr{A} \Rightarrow \mathscr{B}$, for $\mathscr{A}, \mathscr{B} \in \mathrm{Ty}$. The finite pieces of partial information that we will use to approximate terms of this type, i.e. functions from $\mathscr{A}$ to $\mathscr{B}$, are *finite functions*.

A function can be represented as a set of input/output pairs, and so an element of $\mathscr{A} \Rightarrow \mathscr{B}$ is a finite set of pairs. A set of this kind uniquely specifies an approximable mapping: the mapping such that all input/output pairs of the set belong to it, along with whatever else is required for it to satisfy the approximable mapping axioms, and nothing more. Such a mapping is the *smallest* approximable mapping *containing* a set, and we will refer to these sets as finite functions.

Let $\mathscr{A}, \mathscr{B} \in \mathrm{Ty}$, and $f, f' \in \mathrm{Nbh}_{\mathscr{A}\Rightarrow\mathscr{B}}$. Let γ be the smallest mapping containing f, and γ' be the smallest mapping containing f'. Our goal is to define the information ordering

of $\mathscr{A} \Rightarrow \mathscr{B}$ such that $f \sqsubseteq f'$ iff

$$\forall x \forall y (x \overset{\gamma}{\mapsto} y \Rightarrow x \overset{\gamma'}{\mapsto} y)$$

Thus f' consistently extends the information content of f in the sense that γ' maps x to y whenever γ maps x to y.

Before we present the formalization of this ordering, we will informally derive its definition. Using the variables introduced above, assume that $x \overset{\gamma}{\mapsto} y$. What property must hold of f' in order for $x \overset{\gamma'}{\mapsto} y$ to hold? It certainly holds if $(x, y) \in f'$. Because of the monotonicity axiom, it suffices that $(x', y) \in f'$, for some $x' \sqsubseteq x$. Morever, due to the downwards closed axiom, we only require that $(x', y') \in f'$, for some $y' \sqsupseteq y$.

Finally, since either of x' and y' could be the supremum of neighborhoods, the upwards directed axiom allows us to infer that there must exist some subset of f' such that the multiary supremum of its first components is x' and the multiary supremum of its second components is y'. Of course, since x' and y' are any arbitrary elements satisfying $x' \sqsubseteq x$ and $y' \sqsupseteq y$, we infer the following:

Definition 4.3.2. For two finite functions f and f', we have $f \sqsubseteq f'$ iff there for every $(x, y) \in f$ exists a subset

$$\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \subseteq f'$$

such that $x_1 \sqcup x_2 \sqcup \cdots \sqcup x_n$ and $y \sqsubseteq y_1 \sqcup y_2 \sqcup \cdots \sqcup y_n$ exist and satisfy

$$x_1 \sqcup x_2 \sqcup \cdots \sqcup x_n \sqsubseteq x \quad \text{and} \quad y \sqsubseteq y_1 \sqcup y_2 \sqcup \cdots \sqcup y_n.$$

We will now formally define finite functions and related objects, followed by the arrow neighborhood system itself.

4.3.2.1 Formalizing finite functions

We formalize finite functions as lists of pairs, and will often use the terminology of sets when discussing them:

```
data FinFun (A B : Set) : Set where
  ∅ : FinFun A B
  _::_ : A ⨯ B → FinFun A B → FinFun A B
```

The following notation is convenient:

```
NbhFinFun : Ty → Ty → Set
NbhFinFun 𝒜 ℬ = FinFun (NbhSys.Nbh 𝒜) (NbhSys.Nbh ℬ)
```

We must place some restrictions on our finite functions to ensure that they give rise to approximable mappings. In particular, we must be careful to ensure that the consistency

axiom is satisfied. As we discussed above, a mapping $x \overset{\gamma}{\mapsto} y$ can be generated from a set of pairs such that the multiary supremum of their first components is smaller than x, and that of their second components is greater than y. We require that this latter multiary supremum exists:

Definition 4.3.3. A *consistent* finite function f is a finite function such that, for any subset

$$\{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\} \subseteq f$$

for which the multiary supremum $x_1 \sqcup x_2 \sqcup \cdots \sqcup x_n$ exists, so does the multiary supremum $y_1 \sqcup y_2 \sqcup \cdots \sqcup y_n$.

Taking a cue from Hedberg [7], who calls the multiary supremum operators *pre* and *post*, we will call a finite function for which the multiary supremum of all first or second components exists *preable* or *postable*, respectively. We formalize these properties mutually with the multiary supremum operators:

```
data Postable : NbhFinFun 𝒜 ℬ → Set
post : (f : NbhFinFun 𝒜 ℬ) → Postable f → NbhSys.Nbh ℬ

data Postable where
  post-nil : Postable ∅
  post-cons : ∀ {x y f} → (postablef : Postable f) →
              NbhSys.Con ℬ y (post f postablef) → Postable ((x , y) :: f)

post ∅ _ = NbhSys.⊥ ℬ
post ((x , y) :: f) (post-cons postablef conxpostf)
  = [ ℬ ] y ⊔ post f postablef [ conxpostf ]

data Preable : NbhFinFun 𝒜 ℬ → Set
pre : (f : NbhFinFun 𝒜 ℬ) → Preable f → NbhSys.Nbh 𝒜

data Preable where
  pre-nil : Preable ∅
  pre-cons :  ∀ {x y f} → (preablef : Preable f) →
              NbhSys.Con 𝒜 x (pre f preablef) → Preable ((x , y) :: f)

pre ∅ _ = NbhSys.⊥ 𝒜
pre ((x , y) :: f) (pre-cons preablef conxpref)
  = [ 𝒜 ] x ⊔ pre f preablef [ conxpref ]
```

We can now define consistency of finite functions:

```
data ConFinFun (f : NbhFinFun 𝒜 ℬ) : Set where
  cff : (∀ {f'} → f' ⊆ f → Preable f' → Postable f') →
        ConFinFun f
```

We define set membership, the subset relation, and the set union operator:

```
data _∈_ {A B : Set} : A ⨯ B → FinFun A B → Set where
  here : ∀ {x f} → x ∈ (x :: f)
  there : ∀ {x x' f} → x ∈ f → x ∈ (x' :: f)

_⊆_ : (f f' : FinFun A B) → Set
f ⊆ f' = ∀ {x} → (x ∈ f → x ∈ f')

_∪_ : FinFun A B → FinFun A B → FinFun A B
(x :: f) ∪ f' = x :: (f ∪ f')
∅ ∪ f' = f'
```

We also prove a number of useful properties of the above, such as the reflexivity and transitivity of the subset relation, that the empty set is a subset of any set, and that $(x, y) \in \emptyset$ is an absurdity.

4.3.2.2 Defining the neighborhood system

For types $\mathscr{A}$ and $\mathscr{B}$, the neighborhood system $\mathscr{A} \Rightarrow \mathscr{B}$ consists of the bottom element $\bot_e$, and of consistent finite functions of neighborhoods of $\mathscr{A}$ and $\mathscr{B}$. We define consistency of pairs of finite functions as consistency of their union:

```
data ArrNbh : Set where
  ⊥e : ArrNbh
  F : (f : NbhFinFun 𝒜 ℬ) → ConFinFun f → ArrNbh

data ArrCon : ArrNbh → ArrNbh → Set where
  cone-⊥1 : ∀ {x} → ArrCon x ⊥e
  cone-⊥2 : ∀ {x} → ArrCon ⊥e x
  con-∪ : ∀ {f f'} → (conf : ConFinFun f) → (conf' : ConFinFun f') →
          ConFinFun (f ∪ f') → ArrCon (F f conf) (F f' conf')

_⊔e_[_] : (x : ArrNbh) → (y : ArrNbh) → ArrCon x y → ArrNbh
⊥e ⊔e ⊥e [ _ ] = ⊥e
⊥e ⊔e (F f' conf') [ _ ] = F f' conf'
(F f conf) ⊔e ⊥e [ _ ] = F f conf
F f _ ⊔e F f' _ [ con-∪ _ _ conf□ ] = F (f ∪ f') conf□
```

We now define the relation ordering of finite functions in the way described in the beginning of section 4.3.2. The ordering relation $\sqsubseteq_e$ is defined in terms of the auxiliary record **$\sqsubseteq_e$-proof**:

```
record ⊑e-proof (f : NbhFinFun 𝒜 ℬ) (isCon : ConFinFun f)
                (x : NbhSys.Nbh 𝒜) (y : NbhSys.Nbh ℬ) :
                Set where
  field
    sub : NbhFinFun 𝒜 ℬ
    sub⊆f : sub ⊆ f
    preablesub : Preable sub
    postablesub : Postable sub
```

```
    y⊑post : NbhSys._⊑_ ℬ y (post sub postablesub)
    pre⊑x : NbhSys._⊑_ 𝒜 (pre sub preablesub) x

data _⊑ₑ_ : ArrNbh → ArrNbh → Set where
  ⊑ₑ-intro₁ : ∀ {x} → ⊥ₑ ⊑ₑ x
  ⊑ₑ-intro₂ : ∀ {f f'} → (conf : ConFinFun f) → (conf' : ConFinFun f') →
              (∀ {x y} → (x , y) ∈ f → ⊑ₑ-proof f' conf' x y) →
              (F f conf) ⊑ₑ (F f' conf')
```

4.3.2.3 Containment and smallest mappings

An approximable mapping γ from $\mathscr{A}$ to $\mathscr{B}$ is defined to contain a finite function f if $x \overset{\gamma}{\mapsto} y$ for any pair $(x, y) \in f$:

```
data _∈_ (f : NbhFinFun 𝒜 ℬ) (γ : Appmap 𝒜 ℬ) :
         Set where
  ∈-intro : (∀ {x y} → (x , y) ∈ f → [ γ ] x ↦ y) →
            f ∈ γ
```

A mapping γ containing a finite function f is the smallest such mapping if, whenever $x \overset{\gamma}{\mapsto} y$, either $(x, y) \in f$ or the relation $x \overset{\gamma}{\mapsto} y$ is required in order for γ to satisfy the axioms for approximable mappings. We define this mapping's relation as follows:

```
-- A pair (x, y) is in this relation iff (x, y) ∈ f, or if
-- it can be derived from the approximable mapping axioms.
data AppmapClosure (f : NbhFinFun 𝒜 ℬ) : ∀ x y → Set where
  ig-inset : ∀ {x y} → < x , y > ∈ f →
             AppmapClosure f x y
  ig-bot : ∀ {x} →
             AppmapClosure f x (NbhSys.⊥ ℬ)
  ig-mono : ∀ {x x' y} → [ 𝒜 ] x' ⊑ x → AppmapClosure f x' y →
             AppmapClosure f x y
  ig-↓clo : ∀ {x y y'} → [ ℬ ] y ⊑ y' → AppmapClosure f x y' →
             AppmapClosure f x y
  ig-↑dir : ∀ {x y y'} → AppmapClosure f x y →
             AppmapClosure f x y' →
             AppmapClosure f x ([ ℬ ] y ⊔ y')
```

We give the mapping with this relation the name **SmallestAppmap**. Its axioms are easy to prove.

We now prove that the definition of $F\ f \sqsubseteq_e F\ f'$ is logically equivalent to the condition that the smallest mapping that contains f' also contains f. We note that for a mapping γ that contains a finite function f, we have $\mathbf{pre}(f) \overset{\gamma}{\mapsto} \mathbf{post}(f)$. We prove this as a lemma:

```
-- If f is contained in the mapping γ, then γ maps (pre f)
-- to (post f)
```

```
pre↦post : (f : NbhFinFun 𝒜 ℬ) → (preablef : Preable f) →
           (postablef : Postable f) → (γ : Appmap 𝒜 ℬ) →
           f ∈ γ → [ γ ] (pre f preablef) ↦ (post f postablef)
```

First, we show the implication one way;

```
exp⇒smallest : (f f' : NbhFinFun 𝒜 ℬ) →
               ∀ {conf conf'} →
               F f conf ⊑e F f' conf' →
               f ∈ SmallestAppmap f' conf'
```

For any $(x, y) \in f$, we obtain the $\sqsubseteq_e$-**proof**-record given by the proof that $f \sqsubseteq_e f'$. Since γ contains f', it must also contain **sub**, so from **pre↦post** we get **pre(sub)** $\overset{\gamma}{\mapsto}$ **post(sub)**. From this, we have a proof that **pre(sub)** $\sqsubseteq_{\mathscr{A}} x$, which means that $x \overset{\gamma}{\mapsto}$ **post(sub)**, since γ is monotone. Similarly, we use the downwards closed property of γ and the proof that $y \sqsubseteq_{\mathscr{B}}$ **post(sub)** to obtain $x \overset{\gamma}{\mapsto} y$, as desired.

We now contend with the other direction:

```
smallest⇒exp : (f f' : NbhFinFun 𝒜 ℬ) →
               (conf : ConFinFun f) →
               (conf' : ConFinFun f') →
               f ∈ SmallestAppmap f' conf' →
               F f conf ⊑e F f' conf'
```

To prove this, we must for any $(x, y) \in f$ construct a $\sqsubseteq_e$-**proof**-record. We make use of pattern matching to obtain a constructor of the data type **AppmapClosure** f'. We take the neighborhoods supplied by the constructor as subset, and use the accompanying relations or equivalences to prove $y \sqsubseteq_{\mathscr{B}}$ **post**(f') and **pre**$(f') \sqsubseteq_{\mathscr{A}} x$.

For example, with the constructor **ig-mono** we are supplied with a neighborhood x' and a proof that $x' \sqsubseteq_{\mathscr{A}} x$. We take the singleton set $\{(x', y)\}$ as subset. That $y \sqsubseteq_{\mathscr{B}} (y \sqcup \bot)$ is immediately obvious, and that $(x' \sqcup \bot) \sqsubseteq_{\mathscr{A}} x$ follows from $x' \sqsubseteq_{\mathscr{A}} x$. The proofs are no more difficult for the other constructors.

4.3.2.4 Proving transitivity of the arrow neighborhood system's order

Proving most of the axioms for our arrow neighborhood system is fairly easy. One exception is proving transitivity—that $x \sqsubseteq y$ and $y \sqsubseteq z$ together imply that $x \sqsubseteq z$—is difficult when x, y, and z are not $\bot_e$. We do it by making use of the following record, which is variant of $\sqsubseteq_e$-**proof**:

```
-- This can be derived from F f ⊑e F f', and makes proving
-- transitivity very simple.
record ⊑e-proof₂ (f f' : NbhFinFun 𝒜 ℬ) (preablef : Preable f)
                 (postablef : Postable f) : Set where
  field
```

```
sub : NbhFinFun 𝒜 ℬ
preablesub : Preable sub
postablesub : Postable sub
pf⊑post : [ ℬ ] (post f postablef) ⊑ (post sub postablesub)
pre⊑pf : [ 𝒜 ] (pre sub preablesub) ⊑ (pre f preablef)
sub⊆f' : sub ⊆ f'
```

As stated in the comment in the above code, such a record can be derived from a $\sqsubseteq_e$-**proof**-record. For any $(x, y) \in f$, let $\omega_{x,y}$ denote the corresponding subset of f' obtained from the $\sqsubseteq_e$-**proof**. The union of any number of such sets is preable; since they are bounded by $\mathbf{pre}(f)$, the proof follows from the axiom **Con-⊔** of $\mathscr{A}$. Their union is also postable, since f' is consistent. We observe that, for any two pairs $(x, y), (x', y') \in f$, we have

$$\begin{aligned}(y \sqcup y') &\sqsubseteq_{\mathscr{B}} \mathbf{post}(\omega_{x,y} \cup \omega_{x',y'}) \\ \mathbf{pre}(\omega_{x,y} \cup \omega_{x',y'}) &\sqsubseteq_{\mathscr{A}} (x \sqcup x').\end{aligned}$$

With this in mind, let

$$\Omega(f) = \bigcup_{(x,y)\in f} \omega_{x,y}.$$

The set $\Omega(f)$ serves as the **sub** of our $\sqsubseteq_e$-$\mathbf{proof}_2$. We extend the above observation to arbitrary numbers of pairs get

$$\begin{aligned}\mathrm{post}(f) &\sqsubseteq_{\mathscr{B}} \mathrm{post}(\Omega_f) \\ \mathrm{pre}(\Omega_f) &\sqsubseteq_{\mathscr{A}} \mathrm{pre}(f).\end{aligned}$$

We implement this proof using structural induction on f. For the recursive call, we need to show the following, which is easily done:

```
-- If f ⊆ f' and f' ⊑e f'', then we can adapt the ⊑e-proof
-- of f' and f'' to one for f and f''.
shrinkExp : ∀ {conf conf' conf''} →
            f ⊆ f' → (F f' conf') ⊑e (F f'' conf'') →
            (F f conf) ⊑e (F f'' conf'')
```

For any $(x, y) \in f$, the desired subset of f'' is $\Omega(\omega_{x,y})$, since

$$\begin{aligned}y \sqsubseteq_{\mathscr{B}} \mathbf{post}(\omega_{x,y}) &\sqsubseteq_{\mathscr{B}} \mathbf{post}(\Omega(\omega_{x,y})) \\ \mathbf{pre}(\Omega(\omega_{x,y})) &\sqsubseteq_{\mathscr{A}} \mathbf{pre}(\omega_{x,y}) \sqsubseteq_{\mathscr{A}} x.\end{aligned}$$

The proof follows from the transitivity of $\sqsubseteq$ as defined for $\mathscr{A}$ and $\mathscr{B}$.

4.3.2.5 Proving consistency of the arrow neighborhood system's order

To prove **Con-⊔** for the arrow neighborhood's system, we must for any neighborhoods x, x', and y, such that $x \sqsubseteq y$ and $x' \sqsubseteq y$, show that x and x' are consistent. The proof is trivial if any neighborhood is the bottom element, so let us assume that $x = Ff, x' = Ff'$, and $y = Ff''$.

Showing that Ff and Ff' are consistent amounts to showing that $f \cup f'$ is a consistent finite function, which amounts to showing that any preable subset of $f \cup f'$ is also postable. Much like we did when proving transitivity, we introduce a variation of $\sqsubseteq_e$**-proof**, and the actual proof is similar:

```
record ⊑e-proof3 (f : NbhFinFun 𝒜 ℬ) (isCon : ConFinFun f)
                 (f' : NbhFinFun 𝒜 ℬ) (preablef' : Preable f') :
                 Set where
  field
    sub : NbhFinFun 𝒜 ℬ
    sub⊆f : sub ⊆ f
    preablesub : Preable sub
    postablesub : Postable sub
    ybound : ∀ {x y} → ( x , y ) ∈f' → [ ℬ ] y ⊑ (post sub postablesub)
    pre⊑pref' : [ 𝒜 ] (pre sub preablesub) ⊑ (pre f' preablef')
```

A finite function such that every second component has the same upper bound can be shown to be postable, by appealing to the axiom **Con-⊔** of $\mathscr{B}$. Given a preable subset $g \subseteq f \cup f'$, we will create an instance of $\sqsubseteq_e$**-proof**$_3$, with g in place of the record's f'.

Take two arbitrary $(x, y) \in g$ and $(x', y') \in g$. From $Ff \sqsubseteq Ff''$ and $Ff' \sqsubseteq Ff''$, we get preable and postable subsets $sub_1 \subseteq f''$ and $sub_2 \subseteq f''$ such that

$$\begin{array}{cc} \mathbf{pre}(sub_1) \sqsubseteq x & \mathbf{pre}(sub_2) \sqsubseteq x' \\ y \sqsubseteq \mathbf{post}(sub_1) & y' \sqsubseteq \mathbf{post}(sub_2) \end{array}$$

Since $\mathbf{pre}(sub_1) \sqsubseteq x \sqsubseteq \mathbf{pre}(g)$ and $\mathbf{pre}(sub_2) \sqsubseteq x' \sqsubseteq \mathbf{pre}(g)$, we get that $sub_1 \cup sub_2$ is preable. A union of subsets is still a subset, so from the consistency of f'' we derive that $sub_1 \cup sub_2$ is also postable. We then use **Con-⊔** of $\mathscr{B}$ with $y \sqsubseteq \mathbf{post}(sub_1) \sqsubseteq \mathbf{post}(sub_1 \cup sub_2)$ and $y' \sqsubseteq \mathbf{post}(sub_2) \sqsubseteq \mathbf{post}(sub_1 \cup sub_2)$ to get that y and y' are consistent, which implies that $\{(x, y), (x', y')\}$ is a postable set.

By extending this reasoning to the entirety of g using induction, we are done.

With everything in place, we name the resulting neighborhood system **ArrNbhSys**.

4.3.3 The mapping ap

In a context Γ, given a term $t \in \mathrm{Tm}_{\mathscr{A}\Rightarrow\mathscr{B}}(\Gamma)$ and a term $u \in \mathrm{Tm}_{\mathscr{A}}(\Gamma)$, the mapping $\mathbf{ap}\, t\, u$ is to represent applying t, viewed as a function, to the argument u. Of course, $\mathbf{ap}\ t\, u$ must also map anything to the least element of $\mathscr{B}$.

Assume that t maps a valuation $\mathbf{x} \in \mathrm{Val}(\Gamma)$ to f, and that u maps that same valuation to x. Thinking of f as corresponding to a function, and of x as input to that function, we want $\mathbf{ap}\, t\, u$ to map $\mathbf{x}$ to any element that the function corresponding to f maps x to. In other words, if the smallest approximable mapping generated by f maps x to y, then $\mathbf{ap}\, t\, u$ maps $\mathbf{x}$ to y. We use the notation **[_,_]_ap↦_** so that we can write **[t , u] x ap↦ y** instead of e.g. **ap↦ t u x y**.

```
data [_,_]_ap↦_ (t : tAppmap Γ [ ArrNbhSys 𝒜 ℬ ])
                (u : tAppmap Γ [ 𝒜 ]) (x : Valuation Γ) :
                Valuation [ ℬ ] → Set where
  ap↦-intro₁ : ∀ {x} → [ ℬ ] x ⊑ NbhSys.⊥ ℬ →
               [ t , u ] x ap↦ ⟨⟨ x ⟩⟩
  ap↦-intro₂ : ∀ {x y f} conf conxy →
               [ t ] x ↦ ⟨⟨ F f conf ⟩⟩ → [ u ] x ↦ ⟨⟨ x ⟩⟩ →
               [ ArrNbhSys 𝒜 ℬ ] (F ((x , y) :: ∅) conxy) ⊑ (F f conf) →
               [ t , u ] x ap↦ ⟨⟨ y ⟩⟩
```

The more involved proofs of the mapping axioms are those involving the upwards directed property and consistency, and they are almost identical. We remind us of the definition of being upwards directed:

```
ap↦-↑directed : ∀ {x y z} →
                [ t , u ] x ap↦ y → [ t , u ] x ap↦ z →
                (conyz : ValCon _ y z) →
                [ t , u ] x ap↦ (y ⊔ᵥ z [ conyz ])
```

The proof is simple when **ap↦intro**$_1$ is involved, so let's assume that

$$\mathbf{x} \overset{t}{\mapsto} Ff \quad \mathbf{x} \overset{u}{\mapsto} x \quad F\{(x,y)\} \sqsubseteq Ff$$
$$\mathbf{x} \overset{t}{\mapsto} Ff' \quad \mathbf{x} \overset{u}{\mapsto} x' \quad F\{(x',y')\} \sqsubseteq Ff'$$

for some $x, x' \in \mathrm{Nbh}_{\mathscr{A}}$, $y, y' \in \mathrm{Nbh}_{\mathscr{B}}$, and consistent finite functions f and f'. For the first three (implicit) parameters of the constructor **ap↦-intro**, we supply $x \sqcup x'$, $y \sqcup y'$, and $f \cup f'$, respectively. The required proofs that $\mathbf{x} \overset{t}{\mapsto} F(f \cup f')$ and $\mathbf{x} \overset{u}{\mapsto} x \sqcup x'$ are consequences of t and u being upwards directed, and we get consistency of f and f' and of x and x' from the consistency of t and u, respectively. What remains is to supply a proof that $F\{(x \sqcup x', y \sqcup y')\} \sqsubseteq F(f \cup f')$, which amounts to finding a subset $g \subseteq f \cup f'$ such that

$$y \sqcup y' \sqsubseteq_{\mathscr{B}} \mathbf{post}(g)$$
$$\mathbf{pre}(g) \sqsubseteq_{\mathscr{A}} x \sqcup x'.$$

We take as g the union of the two subsets provided by the proofs $F\{(x,y)\} \sqsubseteq Ff$ and $F\{(x',y')\} \sqsubseteq Ff'$, and prove the two above equations by making use of the fact that $\mathbf{pre}(f \cup f')$ is equivalent to $\mathbf{pre}(f) \sqcup \mathbf{pre}(f')$, and analogously for post.

Turning to the proof of consistency:

```
ap↦-con : ∀ {x y x' y'} → [ t , u ] x ap↦ y →
          [ t , u ] x' ap↦ y' → ValCon _ x x' →
          ValCon _ y y'
```

The proof is almost identical to the above: we construct the same subset $g \subseteq f \cup f'$, which satisfies $y \sqsubseteq_{\mathscr{B}} \mathbf{post}(g)$ and $y' \sqsubseteq_{\mathscr{B}} \mathbf{post}(g)$. This, together with **Con-⊔** for $\mathscr{B}$, gives us the desired result.

4.3.4 The mapping lam

A term t of type B, in a context $\Gamma \bullet \mathscr{A}$ is abstracted by the mapping **lam** into a term of type $\mathscr{A} \Rightarrow \mathscr{B}$ in the context Γ.

Given a valuation $\mathbf{x} \in \mathrm{Val}(\Gamma)$, which finite function should **lam** t map to? Since it is to be an abstraction of the term t, and as we are binding a variable term of the type $\mathscr{A}$ by abstracting, we want it to map to the finite function f that contains exactly those pairs (x, y) such that t maps $\langle\langle\, x\,,,\, \mathbf{x}\,\rangle\rangle$ to y. Additionally, it should also map to the least element:

```
data [_]_lam↦_ (t : tAppmap (𝒜 :: Γ) [ ℬ ]) :
               Valuation Γ → Valuation [ ArrNbhSys 𝒜 ℬ ] →
               Set where
  lam↦-intro₁ : ∀ {x} → [ t ] x lam↦ ⟨⟨ ⊥ₑ ⟩⟩
  lam↦-intro₂ : ∀ {x} → {f : NbhFinFun 𝒜 ℬ} →
               (conf : ConFinFun f) →
               (∀ {x y} → (x , y) ∈ f →
               [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
               [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩
```

To help with proving the mapping axioms, we have the following useful lemma:

```
shrinkLam : ∀ {x conf conf'} → f ⊆ f' →
            [ t ] x lam↦ ⟨⟨ F f' conf' ⟩⟩ →
            [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩
```

Proving the downwards closed property involves some work:

```
lam↦-↓closed : ∀ {x y z} →
               ⊑ᵤ [ ArrNbhSys 𝒜 ℬ ] y z →
               [ t ] x lam↦ z → [ t ] x lam↦ y
```

The involved part is when $\mathbf{y} = F\,f$ and $\mathbf{z} = F\,f'$. We need to show that $\langle\langle x, , \mathbf{x}\rangle\rangle \overset{t}{\mapsto} y$ for any $(x, y) \in f$. To do this, we first prove a lemma similar to **pre↦post**:

```
↓closedLemma : {x : Valuation Γ} →
               ∀ conf preablef postablef →
               [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩ →
               [ t ] ⟨⟨ pre f preablef ,, x ⟩⟩ ↦ ⟨⟨ post f postablef ⟩⟩
```

We now do something similar to what we did when proving **exp⇒smallest** in section 4.3.2.2; for any $(x, y) \in f$, we obtain the $\sqsubseteq_e$-**proof**-record given by the assumption that $F\,f \sqsubseteq F\,f'$. Using g to denote the corresponding subset of f' given by the record, we then use **↓closedLemma** with **shrinkLam** to show that $\langle\langle\, \mathbf{pre}(g), , \mathbf{x}\,\rangle\rangle \overset{t}{\mapsto} \mathbf{post}(g)$. This, together with the fact that t is monotone and downwards closed, leads us to the conclusion that $\langle\langle x, , \mathbf{x}\rangle\rangle \overset{t}{\mapsto} y$.

We now turn to proving the upwards directed property:

```
lam↦-↑directed : ∀ {x y z} →
                 [ t ] x lam↦ y → [ t ] x lam↦ z →
                 (conyz : ValCon _ y z) →
                 [ t ] x lam↦ (y ⊔ᵥ z [ conyz ])
```

The more interesting part is when $\mathbf{y} = F\,f$ and $\mathbf{z} = F\,f'$. In this case, we make use of the logical *or* operator, and a lemma:

```
data _∨_ (A B : Set) : Set where
  inl : A → A ∨ B
  inr : B → A ∨ B

∪-lemma₂ : ∀ {x} → x ∈ (f ∪ f') → (x ∈ f) ∨ (x ∈ f')
```

We recall that the supremum of two neighborhoods in the arrow neighborhood system is their union. For any $(x, y) \in (f \cup f')$, we pattern match on **∪-lemma**$_2$. If **inl** is matched, then we get a proof that $(x, y) \in f$. From the assumption $\mathbf{x} \overset{\mathbf{lam}\,t}{\longmapsto} F\,f$, we get a function that takes our proof that $(x, y) \in f$ and returns a proof that $\langle\langle\, x\,,\,\mathbf{x}\,\rangle\rangle \overset{t}{\longmapsto} y$, which is exactly what we need. Similarly, if **inr** is matched, we get a proof that $(x, y) \in f'$, and use the assumption $\mathbf{x} \overset{\mathbf{lam}\,t}{\longmapsto} F\,f'$ in the same way.

Finally, we prove that **lam↦** is consistent:

```
lam↦-con : ∀ {x y x' y'} → [ t ] x lam↦ y →
           [ t ] x' lam↦ y' → ValCon _ x x' →
           ValCon _ y y'
```

As usual, the non-trivial case in when $\mathbf{y} = F\,f$ and $\mathbf{y}' = F\,f'$. The goal is then to show that $f \cup f'$ is a consistent finite function. So let us assume that $g \subseteq f \cup f'$ is a preable set. We prove that it is postable by constructing the following record, with g in place of the record's f:

```
record ⊑ₑ-proof₄ (f : NbhFinFun 𝒜 ℬ) (preablef : Preable f)
                 (x : Valuation Γ) : Set where
  field
    postablef : Postable f
    tpre↦post : [ t ] ⟨⟨ pre f preablef ,, x ⟩⟩ ↦ ⟨⟨ post f postablef ⟩⟩
```

From the assumptions we have proofs that

$$\forall (x, y) \in f \left(\langle\langle\, x\,,\,\mathbf{x}\,\rangle\rangle \overset{t}{\mapsto} y \right)$$
$$\forall (x, y) \in f' \left(\langle\langle\, x\,,\,\mathbf{x}'\,\rangle\rangle \overset{t}{\mapsto} y \right)$$

From the upwards directed property of t and from the assumed consistency of $\mathbf{x}$ and $\mathbf{x}'$, we can turn this into a proof that

$$\forall (x, y) \in f \cup f' \left(\langle\langle\, x\,,\,\mathbf{x} \sqcup \mathbf{x}'\,\rangle\rangle \overset{t}{\mapsto} y \right)$$

For any two $(x, y), (x', y') \in g$, we hence have that $\langle\langle\, x\, ,, \mathbf{x} \sqcup \mathbf{x}'\, \rangle\rangle \overset{t}{\mapsto} y$ and $\langle\langle\, x'\, ,, \mathbf{x} \sqcup \mathbf{x}'\, \rangle\rangle \overset{t}{\mapsto} y'$. From the fact that $x \sqsubseteq \mathbf{pre}(g)$ and $x' \sqsubseteq \mathbf{pre}(g)$, we get that x and x' are consistent. Since t is consistent, we get that y and y' are consistent, which means that $\{(x, y), (x', y')\}$ is postable.

As usual, we extend this to the entirety of g via induction.

4.3.5 Proving the weak ⇒-structure axioms

The proofs for the congruence axioms and **apSub** are very simple, so we will focus our attention on the other two below. We will remind ourselves of their definitions as we go.

4.3.5.1 lamSub

The definition:

```
lamSub : ∀ {Γ : Ctx n} → (γ : tAppmap Δ Γ) → ∀ t →
         (lam t ∘ γ) ≈ lam (t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩)
```

The more difficult direction to prove is the following:

```
lamSubLemma₂ : ∀ {x y} →
               [ t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩ ] x lam↦ y →
               [ lam t ∘ γ ] x ↦ y
```

To help us with this proof in the case when $y = F\, f$, we define a record:

```
-- From a proof that t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩ maps
-- x to ⟨⟨ F f ⟩⟩, we can find a valuation y such that
-- γ maps x to y, and t maps ⟨⟨ x , y ⟩⟩ to ⟨⟨ y ⟩⟩ for any
-- (x , y) ∈ f.
record P-Struct (γ : tAppmap Δ Γ) (t : tAppmap (𝒜 :: Γ) [ ℬ ])
                (x : Valuation Δ) (f : NbhFinFun 𝒜 ℬ) :
                Set where
  field
    y : Valuation Γ
    γx↦y : [ γ ] x ↦ y
    λty : ∀ {x y} → (x , y) ∈ f → [ t ] ⟨⟨ x ,, y ⟩⟩ ↦ ⟨⟨ y ⟩⟩
```

Once we have such a record, the proof is trivial. For any $(x, y) \in f$, we have from the assumption:

$$\mathbf{x} \overset{\langle \gamma \circ \mathbf{p}, \mathbf{q} \rangle}{\longmapsto} \langle\langle z, , \mathbf{z} \rangle\rangle \quad \langle\langle z, , \mathbf{z} \rangle\rangle \overset{t}{\mapsto} y$$

for some $\langle\langle z, , \mathbf{z} \rangle\rangle \in \mathrm{Val}(\Gamma \cdot \mathscr{A})$, where we have omitted the arguments to $\mathbf{p}$ and $\mathbf{q}$ for the sake of readability. We take as the valuation of our record the supremum of all such $\mathbf{z}$; let's denote this supremum $\bar{\mathbf{z}}$. That this supremum exists follows from the consistency of γ. From $\mathbf{x} \overset{\langle \gamma \circ \mathbf{p}, \mathbf{q} \rangle}{\longmapsto} \langle\langle z, , \mathbf{z} \rangle\rangle$ we get

$$\mathbf{y} \overset{\gamma}{\mapsto} \mathbf{z} \quad \mathbf{y} \sqsubseteq \mathbf{x} \quad z \sqsubseteq x$$

for some $\mathbf{y} \in \mathrm{Val}(\Delta)$. We find that $\mathbf{x} \overset{\gamma}{\mapsto} \mathbf{z}$, since γ is monotone, and $\mathbf{y} \sqsubseteq \mathbf{x}$. From this, the upwards directed property of γ shows that $\mathbf{x} \overset{\gamma}{\mapsto} \overline{\mathbf{z}}$.

From $\langle\langle z, , \mathbf{z}\rangle\rangle \overset{t}{\mapsto} y$ and $z \sqsubseteq x$ we get $\langle\langle x, , \mathbf{z}\rangle\rangle \overset{t}{\mapsto} y$ by using the monotonicity of t. The same property, together with $\mathbf{z} \sqsubseteq \overline{\mathbf{z}}$, implies the final required part of the record: that $\langle\langle x, , \overline{\mathbf{z}}\rangle\rangle \overset{t}{\mapsto} y$.

Formalizing this proof in Agda requires more effort than is indicated here, since one must use structural induction on f. See appendix B for details.

4.3.5.2 β-equality

The definition:

```
β-equal : {t : tAppmap Γ [ 𝒜 ]} →
          {u : tAppmap (𝒜 :: Γ) [ ℬ ]} →
          ap (lam u) t ≈ (u ∘ ⟨ idMap Γ , t ⟩)
```

One direction is more involved to prove than the other:

```
β-lemma₁ : {t : tAppmap Γ [ 𝒜 ]} →
           {u : tAppmap (𝒜 :: Γ) [ ℬ ]} → ∀ {x y} →
           [ ap (lam u) t ] x ↦ y →
           [ u ∘ ⟨ idMap Γ , t ⟩ ] x ↦ y
```

When $y = Ff$, we have as part of the assumption a finite function f, neighborhoods $x \in \mathrm{Nbh}_{\mathscr{A}}$ and $y \in \mathrm{Nbh}_{\mathscr{B}}$. We also have proofs that $\mathbf{x} \overset{\mathbf{lam}\ u}{\longmapsto} Ff$, that $\mathbf{x} \overset{t}{\mapsto} x$, and that $F\{(x, y)\} \sqsubseteq Ff$. From the latter we pick out the $\sqsubseteq_e$-**proof**-record for x and y, so that we have a preable and postable subset $g \subseteq f$ such that

$$y \sqsubseteq \mathbf{post}(g) \quad \text{and} \quad \mathbf{pre}(g) \sqsubseteq x.$$

It is immediately clear that $\mathbf{x} \overset{\mathbf{id}_\Gamma}{\mapsto} \mathbf{x}$, so combined with $\mathbf{x} \overset{t}{\mapsto} x$ above we get $\mathbf{x} \overset{\langle\, \mathbf{id}_\Gamma\, , t\, \rangle}{\longmapsto} \langle\langle x, , \mathbf{x}\rangle\rangle$. We then make use of $\downarrow$**closedLemma**, defined in section 4.3.4, to show that $\langle\langle \mathbf{pre}(g), , \mathbf{x}\rangle\rangle \overset{u}{\mapsto} \mathbf{post}(g)$, and use the monotonicity and downwards closed property of u to get $\langle\langle x, , \mathbf{x}\rangle\rangle \overset{u}{\mapsto} y$. This completes the proof.

5

A domain model of the untyped lambda calculus

By removing type information from an scwf, we obtain a unityped cwf. In a contextual ucwf, contexts can be thought of as natural numbers, according to the number of assumptions they contain. Contextual ucwfs are in [4] shown to be equivalent to cartesian operads and Lawvere theories [9].

Hedberg [7] mentions that extending his work to a domain interpretation of the untyped lambda calculus would require solving a recursive domain equation,

$$\mathscr{D} \cong (\mathscr{D} \Rightarrow \mathscr{D})_\perp$$

Of this, he presciently writes: "One way to construct a semilattice solution to such an equation is to use the technique employed by Scott and Martin-Löf, and interpret the equation as an inductive definition of formal neighborhoods and of formal inclusion. The proof of transitivity is then not a straight forward structural induction, but will depend on a measure on neighborhoods relating to the depth of nesting constructors. The proof will also repeat a major part of the proof of exponential transitivity."

We will in section 5.1.2 directly construct the compact elements of $\mathscr{D} = (\mathscr{D} \Rightarrow \mathscr{D})_\perp$, which we will call the universal type. The proof of transitivity will indeed be very much like the proof in section 4.3.2.4, and we will make use of sized types to track the depth of nesting constructors of a particular kind.

5.1 A plain ucwf of neighborhood systems

We present the definition of plain ucwfs, and a formalization thereof. Our implementation will reuse most of the definitions and proofs of the plain scwf of neighborhood systems. The key difference is that we construct a neighborhood system corresponding to the universal type in a way such that it is isomorphic to its own lifted function space.

5.1.1 Abstract definition

Definition 5.1.1. A *ucwf* consists of the following:

- A category $\mathscr{C}$ with a terminal object 0.
- A presheaf $\mathrm{Tm} : \mathscr{C}^{\mathrm{op}} \to \mathrm{Set}$
- A *context comprehension operation* which to a given context $n \in \mathscr{C}_0$ assigns a context $s(n) \in \mathscr{C}_0$ along with two projections

$$\mathrm{p}_n : s(n) \to n \qquad \mathrm{q}_n \in \mathrm{Tm}(s(n))$$

  satisfying the following universal property: for all $\gamma : m \to n$, for all $a \in \mathrm{Tm}(m)$, there is a unique $\langle \gamma, a \rangle : m \to s(n)$ such that

$$\mathrm{p}_n \circ \langle \gamma, a \rangle = \gamma \qquad \mathrm{q}_n[\langle \gamma, a \rangle] = a$$

We adapt the formalization of plain scwfs according to the above definition:

```
record Ucwf : Set₂ where
  field
    Tm : Nat → Set₁
    Sub : Nat → Nat → Set₁

    _≈_ : Rel (Tm n) (lsuc lzero)
    _≅_ : Rel (Sub m n) (lsuc lzero)

    isEquivT : IsEquivalence (_≈_ {n})
    isEquivS : IsEquivalence (_≅_ {m} {n})

    q : Tm (suc n)
    _[_] : Tm n → Sub m n → Tm m

    id : Sub n n
    _∘_ : Sub n o → Sub m n → Sub m o
    ⟨⟩ : Sub n 0
    ⟨_,_⟩ : Sub m n → Tm m → Sub m (suc n)
    p : Sub (suc n) n

    idL : (γ : Sub n m) → (id ∘ γ) ≅ γ
    idR : (γ : Sub n m) → (γ ∘ id) ≅ γ
    subAssoc : (γ : Sub m n) → (δ : Sub n o) →
               (θ : Sub o r) →
               ((θ ∘ δ) ∘ γ) ≅ (θ ∘ (δ ∘ γ))

    idSub : (t : Tm n) → (t [ id ]) ≈ t
    compSub : (t : Tm n) → (γ : Sub m n) →
              (δ : Sub o m) →
              (t [ (γ ∘ δ) ]) ≈ ((t [ γ ]) [ δ ])

    id₀ : id ≅ ⟨⟩
```

```
<>-zero : (γ : Sub m n) → (⟨⟩ ∘ γ) ≅ ⟨⟩

pCons : (γ : Sub n m) → (t : Tm n) →
        (p ∘ ⟨ γ , t ⟩) ≅ γ
qCons : (γ : Sub n m) → (t : Tm n) →
        (q [ ⟨ γ , t ⟩ ]) ≅ t
idExt : (id {suc m}) ≅ ⟨ p , q ⟩
compExt : (t : Tm n) → (γ : Sub n m) → (δ : Sub m n) →
          (⟨ γ , t ⟩ ∘ δ) ≅ ⟨ γ ∘ δ , t [ δ ] ⟩

subCong : {t t' : Tm m} → {γ γ' : Sub n m} →
          t ≅ t' → γ ≅ γ' →
          (t [ γ ]) ≅ (t' [ γ' ])
<,>-cong : {t t' : Tm m} → {γ γ' : Sub m n} →
           t ≅ t' → γ ≅ γ' →
           ⟨ γ , t ⟩ ≅ ⟨ γ' , t' ⟩
∘-cong : {γ δ : Sub n o} → {γ' δ' : Sub m n} →
         γ ≅ δ →
         γ' ≅ δ' → (γ ∘ γ') ≅ (δ ∘ δ')
```

5.1.2 The universal type

The difficulty of formalizing a ucwf of neighborhood systems arises from trying to construct the neighborhood system that represents the universal type. What guides us in defining its neighborhoods is thinking of untyped theories of computable functions that deal with abstraction and application, particularly the untyped lambda calculus. With this in mind, our neighborhood system will be very similar to the arrow neighborhood system described in section 4.3.2, but it will be isomorphic to its own lifted function space. Where we for two types $\mathscr{A}$ and $\mathscr{B}$ defined neighborhoods of $\mathscr{A} \Rightarrow \mathscr{B}$ as sets of pairs (x, y) for $x \in \mathscr{A}$, $y \in \mathscr{B}$, we will for the universal type inductively define neighborhoods as sets of pairs of structurally smaller elements of the universal type itself.

A first attempt to define the neighborhoods of the universal type might be the following:

```
data UniNbh : Set where
  ⊥ᵤ : UniNbh
  λᵤ : FinFun UniNbh UniNbh → UniNbh,
```

with definitions of $\sqcup_u$ and $\sqsubseteq_u$ analogous to those of the arrow neighborhood system. This works well up to a point, but we encounter a problem when proving transitivity using the same kind of proof that we used in section 4.3.2.4. There, we made use of the following record, here shown as adapted to our attempted universal type:

```
record ⊑ₑ-proof₂ (f f' : NbhFinFun 𝒜 ℬ) (preablef : Preable f)
                 (postablef : Postable f) : Set where
  field
    sub : FinFun UniNbh UniNbh
```

preablesub : Preable sub
postablesub : Postable sub
pf⊑post : (post f) $\sqsubseteq_u$ (post sub)
pre⊑pf : (pre sub) $\sqsubseteq_u$ (pre f)
sub⊆f' : sub $\subseteq f'$

Assume $x = \lambda_u f$, $y = \lambda_u f'$, $z = \lambda_u f''$, and that we are trying to show that $x \sqsubseteq y$ and $y \sqsubseteq z$ together imply that $x \sqsubseteq z$. Recall that, for any $(x, y) \in f$, we showed that $y \sqsubseteq \mathbf{post}(\Omega(\omega_{x,y}))$ and $\mathbf{pre}(\Omega(\omega_{x,y})) \sqsubseteq x$, where $\omega_{x,y} \subseteq f'$ and $\Omega(\omega_{x,y}) \subseteq f''$, by making use of the following:

$$y \sqsubseteq \mathbf{post}(\omega_{x,y}) \sqsubseteq \mathbf{post}(\Omega(\omega_{x,y}))$$
$$\mathbf{pre}(\Omega(\omega_{x,y})) \sqsubseteq \mathbf{pre}(\omega_{x,y}) \sqsubseteq x,$$

and appealing to the transitivity of the underlying types. The underlying types in the present case is the type we are currently trying to construct! Fortunately, we can in principle make use of recursion in our proof. Unfortunately, Agda can not infer that the recursive call is done on structurally smaller arguments, and so does not accept the recursion as being well founded. Put another way, Agda can not see that the function terminates for all input. To convince Agda of this, we make use of *sized types* [1]. These allow us to associate to each data type a size, in an abstract sense. Agda then accepts recursive calls on elements of smaller size.

5.1.2.1 Defining the neighborhood system using sized types

What we will use as a notion of "size" in describing the universal type is the maximum number of nested lambda abstractions in a term, i.e. applications of the constructor λ_u. We base our formalization on the following ideas:

- If two neighborhoods x, y contain a maximum of i lambda abstractions, then so does the pair (x, y).
- For any i, the empty set contains a maximum of i lambda abstractions.
- If a pair of neighborhoods x and a finite function f contain a maximum of i lambda abstractions, then so does $x :: f$.
- For any i, the least element $\bot_e$ contains a maximum of i lambda abstractions.
- Most importantly: if f is a finite function containing a maximum of i lambda abstractions, then $\lambda_u f$ contains *more* lambda abstractions than f. This is denoted $\uparrow i$.

The above points indicate that we need to define sized versions of pairs, finite functions, and the type's neighborhoods, in terms of one another:

data $\times_s$: {i : Size} → Set
data FinFun$_s$: {i : Size} → Set

```
data UniNbh : {i : Size} → Set

data ×ₛ where
  _,_ : ∀ {i} → (x y : UniNbh {i}) → ×ₛ {i}

data FinFunₛ where
  ∅ : ∀ {i} → FinFunₛ {i}
  _::_ : ∀ {i} → ×ₛ {i} → FinFunₛ {i} → FinFunₛ {i}

data UniNbh where
  ⊥ᵤ : ∀ {i} → UniNbh {i}
  -- Note that λᵤ increases the size!
  λᵤ : ∀ {i} → FinFunₛ {i} → UniNbh {↑ i}
```

We also define the supremum of neighborhoods, and the union of sized sets, noting that these operations preserve sizes. We consider all elements pairwise consistent:

```
data UniCon : UniNbh → UniNbh → Set where
  con-all : ∀ {x y} → UniCon x y

_∪ₛ_ : ∀ {i} → FinFunₛ {i} → FinFunₛ {i} → FinFunₛ {i}
∅ ∪ₛ f' = f'
(x :: f) ∪ₛ f' = x :: (f ∪ₛ f')

_⊔ᵤ_[_] : ∀ {i} → (x y : UniNbh {i}) →
          UniCon x y → UniNbh {i}
⊥ᵤ ⊔ᵤ ⊥ᵤ [ _ ] = ⊥ᵤ
⊥ᵤ ⊔ᵤ (λᵤ f) [ _ ] = λᵤ f
(λᵤ f) ⊔ᵤ ⊥ᵤ [ _ ] = λᵤ f
(λᵤ f) ⊔ᵤ (λᵤ f') [ _ ] = λᵤ (f ∪ₛ f')
```

We define sized versions of the multiary supremum **pre** and **post**; these also preserve sizes. Since all neighborhoods are consistent, we require no notion of preable and postable finite functions, nor of consistent finite functions:

```
pre : ∀ {i} → FinFunₛ {i} → UniNbh {i}
pre ∅ = ⊥ᵤ
pre ((x , y) :: f) = x ⊔ᵤ pre f

post : ∀ {i} → FinFunₛ {i} → UniNbh {i}
post ∅ = ⊥ᵤ
post ((x , y) :: f) = y ⊔ᵤ post f
```

Finally, the ordering is defined inductively:

```
record ⊑ᵤ-proof {i j : Size} (f : FinFunₛ {j})
                (x y : UniNbh {i}) : Set

data _⊑ᵤ_ : UniNbh {i} → UniNbh {j} → Set
```

```
record ⊑ᵤ-proof f x y where
  inductive
  field
    sub : FinFunₛ
    y⊑ᵤpost : y ⊑ᵤ (post sub)
    pre⊑ᵤx : (pre sub) ⊑ᵤ x
    sub⊆f' : sub ⊆ₛ f

data _⊑ᵤ_ where
  ⊑ᵤ-intro₁ : ∀ {i j} → {x : UniNbh {j}} →
              (⊥ᵤ {i}) ⊑ᵤ x
  ⊑ᵤ-intro₂ : ∀ {i j} → (f : FinFunₛ {i}) →
              (f' : FinFunₛ {j}) →
              ((x y : UniNbh {i}) → (x , y) ∈ₛ f →
              ⊑ᵤ-proof {i} {j} f' x y) →
              _⊑ᵤ_ {↑ i} {↑ j} (λᵤ f) (λᵤ f')
```

5.1.2.2 Proving the neighborhood system axioms

As with the arrow neighborhood system of section 4.3.2, the transitivity proof is the most challenging. We implement the function Ω in a different way, in that we make use of equational reasoning and rewriting instead of the transitivity of the underlying types. See the code in appendix B for details.

To recapitulate, we need to show that the assumptions $x \sqsubseteq y$ and $y \sqsubseteq z$ together imply that $x \sqsubseteq z$, and the case that requires recursion is when $x = \lambda_u f$, $y = \lambda_u f'$, and $z = \lambda_u f''$. We need to find a subset $\mathbf{sub}'' \subseteq f''$ such that, for any pair $(x, y) \in f$, we have $y \sqsubseteq \mathbf{post}(\mathbf{sub}'')$ and $\mathbf{pre}(\mathbf{sub}'') \sqsubseteq x$. For a given $(x, y) \in f$, with definitions of $\omega_{x,y}$ and $\Omega(\omega_{x,y})$ entirely analogous to those of section 4.3.2, we get,

$$y \sqsubseteq \mathbf{post}(\omega_{x,y}) \sqsubseteq \mathbf{post}(\Omega(\omega_{x,y}))$$
$$\mathbf{pre}(\Omega(\omega_{x,y})) \sqsubseteq \mathbf{pre}(\omega_{x,y}) \sqsubseteq x.$$

Due to using sized types and to how we have defined our data types and operators, Agda can now infer that, if $y = \lambda_u f'$ then $\omega_{x,y}$, being a subset of f', is of the same size as f'. Moreover, so is $\mathbf{post}(\omega_{x,y})$, since **post** preserves size. The same argument applies to $\mathbf{pre}(\omega_{x,y})$. The proof is inductive, in the following way:

```
⊑ᵤ-trans : ∀ {i x} → {y : UniNbh {i}} → ∀ {z} →
           x ⊑ᵤ y → y ⊑ᵤ z → x ⊑ᵤ z

⊑ᵤ-trans' : ∀ {i} → ∀ f → (f' : FinFunₛ {i}) → ∀ f'' →
            (λᵤ f) ⊑ᵤ (λᵤ f') → (λᵤ f') ⊑ᵤ (λᵤ f'') →
            ∀ x' y' → (x' , y') ∈ₛ f → ⊑ᵤ-proof f'' x' y'
```

Note that we track the size of the second (explicit) arguments of both functions.

From $\sqsubseteq_u$-**trans**, if $x = \lambda_u f$, $y = \lambda_u f'$, and $z = \lambda_u f''$, we call $\sqsubseteq_u$-**trans'** with f' as the second argument. In turn, the function $\sqsubseteq_u$-**trans'** calls $\sqsubseteq_u$-**trans** twice; once with $\mathbf{pre}(\omega_{x,y})$ as second argument, and once with $\mathbf{post}(\omega_{x,y})$. As mentioned, both of these are of the same size as $\omega_{x,y}$, which is strictly smaller than the initial argument $\lambda_u f'$! Hence, the recursion is recognized as being well founded by Agda. With all axioms now proved, we give this neighborhood system the name **UniType**.

5.1.3 Morphisms and proofs

With the universal type in place, we define valuations over contexts in our ucwf in terms of those in the scwf, and similarly for approximable mappings:

```
-- In a ucwf contexts are simply natural numbers.
-- As we want to use approximable mappings as initially
-- defined for scwfs, we define a function that "translates"
-- natural numbers to scwf-contexts.
nToCtx : ∀ (n) → Ctx n
nToCtx zero = []
nToCtx (suc n) = UniType :: (nToCtx n)

-- Notation for valuations of contexts in the ucwf.
uValuation : Nat → Set
uValuation n = Valuation (nToCtx n)

-- Notation for approximable mappings between
-- neighborhoods of the universal type.
uAppmap : Nat → Nat → Set₁
uAppmap m n = tAppmap (nToCtx m) (nToCtx n)
```

We define substitutions as **uAppmap**s from contexts of length m to contexts of length n, and terms as **uAppmap**s from contexts of length n to contexts of length 1. This allows us to reuse the definitions of—and proofs involving—context comprehension, mapping composition, and so on, from our scwf.

5.2 A $\lambda\beta$-ucwf of domains

To ucwfs we may add a structure analogous to the $\Rightarrow$-structure of scwfs. We formulate a definition without the rule of η-equality, since adding a distinguished least element invalidates this rule.

5.2.1 Abstract definition

Definition 5.2.1. A $\lambda\beta$-ucwf is a ucwf with two more operations:

$$\lambda_n : \mathrm{Tm}(s(n)) \to \mathrm{Tm}(n)$$
$$\mathrm{ap}_n : \mathrm{Tm}(n) \times \mathrm{Tm}(n) \to \mathrm{Tm}(n)$$

for all $n \in \mathscr{C}_0$, and three more equations:

$$\begin{aligned} \lambda(b)[\gamma] &= \lambda_m(b[\langle \gamma \circ \mathrm{p}_m, \mathrm{q}_m \rangle]) \\ \mathrm{ap}_n(c, a)[\gamma] &= \mathrm{ap}_m(c[\gamma], a[\gamma]) \\ \mathrm{ap}_n(\lambda_n(b), a) &= b[\langle \mathrm{id}_n, a \rangle] \end{aligned}$$

This leads to the following formalization:

```
record λβ-ucwf : Set₂ where
  field
    ucwf : Ucwf
  open Ucwf ucwf public
  field
    lam : Tm (suc m) → Tm m
    ap  : Tm m → Tm m → Tm m

    lamSub : (γ : Sub n m) → (t : Tm (suc m)) →
             (lam t [ γ ]) ≈ (lam (t [ ⟨ γ ∘ p , q ⟩ ]))
    apSub : (γ : Sub n m) → (t u : Tm m) →
            ap (t [ γ ]) (u [ γ ]) ≈ (ap t u [ γ ])

    β : {t : Tm m} → {u : Tm (suc m)} →
        ap (lam u) t ≈ (u [ ⟨ id , t ⟩ ])

    lamCong : ∀ {t t' : Tm (suc m)} → t ≈ t' →
              lam t ≈ lam t'
    apCong : {t t' : Tm m} → ∀ {u u'} →
             t ≈ t' → u ≈ u' →
             ap t u ≈ ap t' u'
```

5.2.2 The mappings ap and lam

Given the similarity of the universal type to the arrow neighborhood system, it should come as no surprise that the relations of these mappings are almost identical to those defined in section 4.3.3 and 4.3.4:

```
data [_,_]_ap↦_ (t u : uAppmap n 1) (x : uValuation n) :
                uValuation 1 → Set where
  ap↦-intro₁ : [ t , u ] x ap↦ ⟨⟨ ⊥ᵤ ⟩⟩
  ap↦-intro₂ : ∀ {x y f} →
               [ t ] x ↦ ⟨⟨ λᵤ f ⟩⟩ → [ u ] x ↦ ⟨⟨ x ⟩⟩ →
               (λᵤ ((x , y) :: ∅)) ⊑ᵤ (λᵤ f) →
               [ t , u ] x ap↦ ⟨⟨ y ⟩⟩


data [_]_lam↦_ (t : uAppmap (suc n) 1) :
               uValuation n → uValuation 1 → Set where
```

$$
\begin{aligned}
&\text{lam}{\mapsto}\text{-intro}_1 : \forall\ \{\mathbf{x}\} \to [\ \mathbf{t}\ ]\ \mathbf{x}\ \text{lam}{\mapsto}\ \langle\langle\ \bot_u\ \rangle\rangle \\
&\text{lam}{\mapsto}\text{-intro}_2 : \forall\ \{\mathbf{x}\ f\} \to \\
&\qquad\qquad (\forall\ \{x\ y\} \to (x\ ,\ y) \in_s f \to \\
&\qquad\qquad [\ \mathbf{t}\ ]\ \langle\langle\ x\ ,,\ \mathbf{x}\ \rangle\rangle \mapsto \langle\langle\ y\ \rangle\rangle) \to \\
&\qquad\qquad [\ \mathbf{t}\ ]\ \mathbf{x}\ \text{lam}{\mapsto}\ \langle\langle\ \lambda_u\ f\ \rangle\rangle
\end{aligned}
$$

Notice the difference in the first constructor of the **ap**-relation compared to when defining it for the $\Rightarrow$-structure; when proving the mapping axioms, Agda identifies that there is no constructor of **UniNbh** that can create a neighborhood x such that $x \sqsubseteq \bot$.

Proving the mapping axioms and the three equations of definition 5.2.1 is done in exactly the same way as in section 4.3.3, 4.3.4, and 4.3.5.

6

Summary

As a step toward giving an domain interpretation of partial type theory inside total type theory, we have constructed a simply typed and a unityped category with families of neighborhood systems, which we have formalized in the dependently typed language Agda.

The project is partly an adaptation of Hedberg's work, in which he constructed a cartesian closed category of domains, to a version making use of the notion of simply typed categories with families. We have gone beyond his work both by introducing a consistency relation to our formalization, so that we can model types such as the natural numbers, and also by extending the result to a domain interpretation of the untyped lambda calculus. We did this by directly constructing the compact elements of the universal type, and by making use of Agda's sized types to prove transitivity of its underlying ordering relation.

Initially, the aim of this project was to construct a full cwf of neighborhood systems. We decided early on to formalize all results in Agda, because we find that having such a formalization is much more valuable than having an informal construction because of the added confidence in its correctness it gives. Formalizing mathematics is a much more time consuming endeavor than writing out informal mathematics, and we found it necessary to limit the scope of the project and construct a simply typed and unityped cwf of neighborhood systems.

The obvious direction in which to extend the work presented in this thesis, then, is to extend it to a formalization of a full cwf of neighborhood systems, with extra structure for modeling e.g. Π-types and Σ-types. The immediate difficulty in doing this is defining neighborhood systems where the neighborhoods correspond to partial information about types, and specifying the interplay between types and terms when only partial information is available for both.

Bibliography

[1] Andreas Abel. Miniagda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010*, volume 43 of *EPTCS*, pages 14–28, 2010.

[2] Konstantinos Brilakis. On initial categories with families. Master's thesis, Chalmers University of Technology, 2018.

[3] John Cartmell. Generalised algebraic theories and contextual categories. *Ann. Pure Appl. Log.*, 32:209–243, 1986.

[4] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019.

[5] Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. In *International Conference on Typed Lambda Calculi and Applications*, pages 91–106. Springer, 2011.

[6] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995.

[7] Michael Hedberg. *Type Theory and the External Logic of Programs*. PhD thesis, Chalmers University of Technology, 1994.

[8] Michael Hedberg. A type-theoretic interpretation of constructive domain theory. *J. Autom. Reasoning*, 16(3):369–425, 1996.

[9] F. William Lawvere. *Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963.

[10] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[11] Per Martin-Löf. The domain interpretation of type theory, lecture notes. In Kent Karlsson and Kent Petersson, editors, *Workshop on Semantics of Programming Lan-*

guages, Abstracts and Notes, Chalmers University of Technology and University of Göteborg, August 1983. Programming Methodology Group.

[12] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 1982.

[13] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[14] Erik Palmgren and Viggo Stoltenberg-Hansen. Domain interpretations of Martin-Löf's partial type theory. *Ann. Pure Appl. Log.*, 48(2):135–196, 1990.

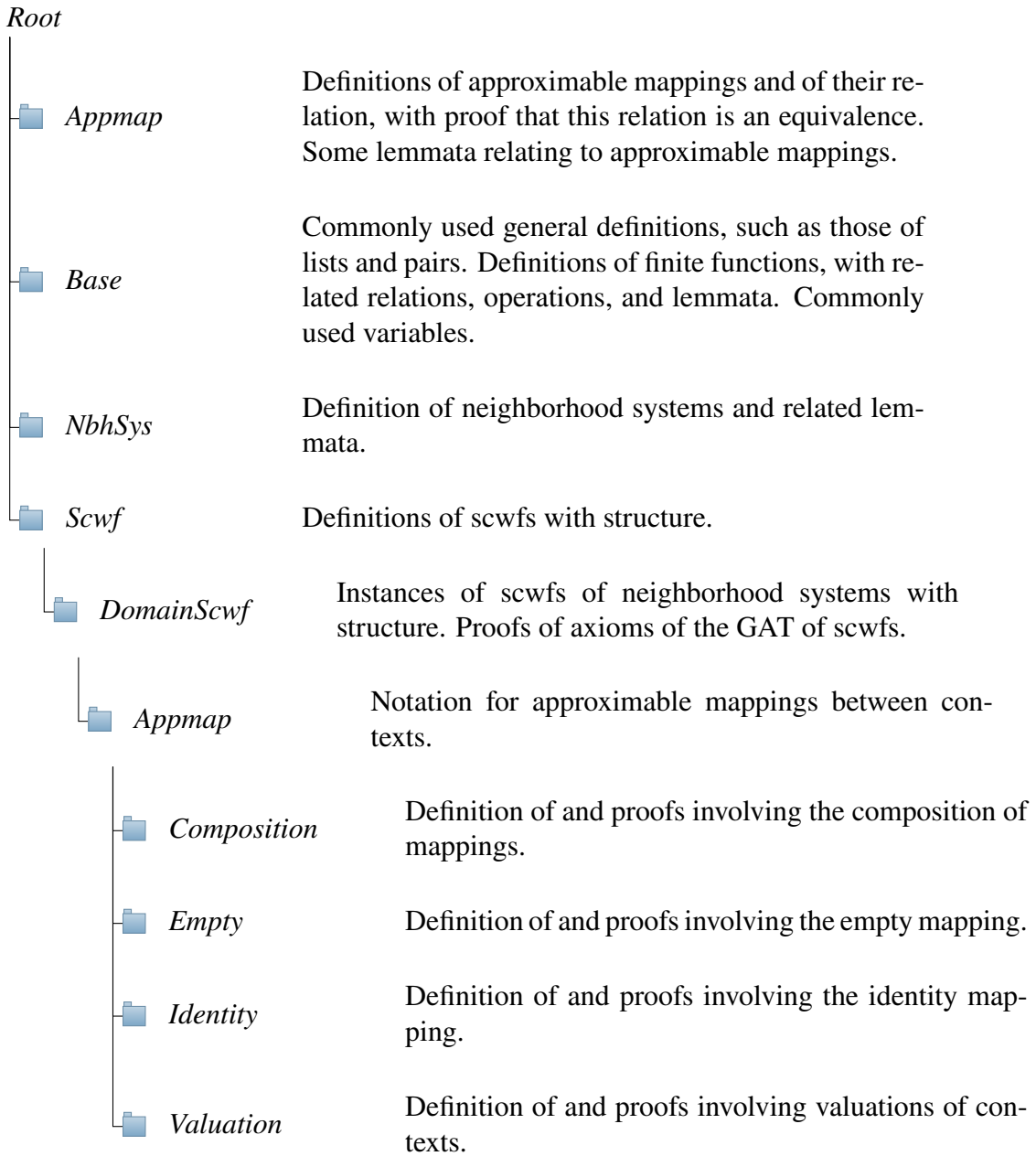
[15] Dana S. Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

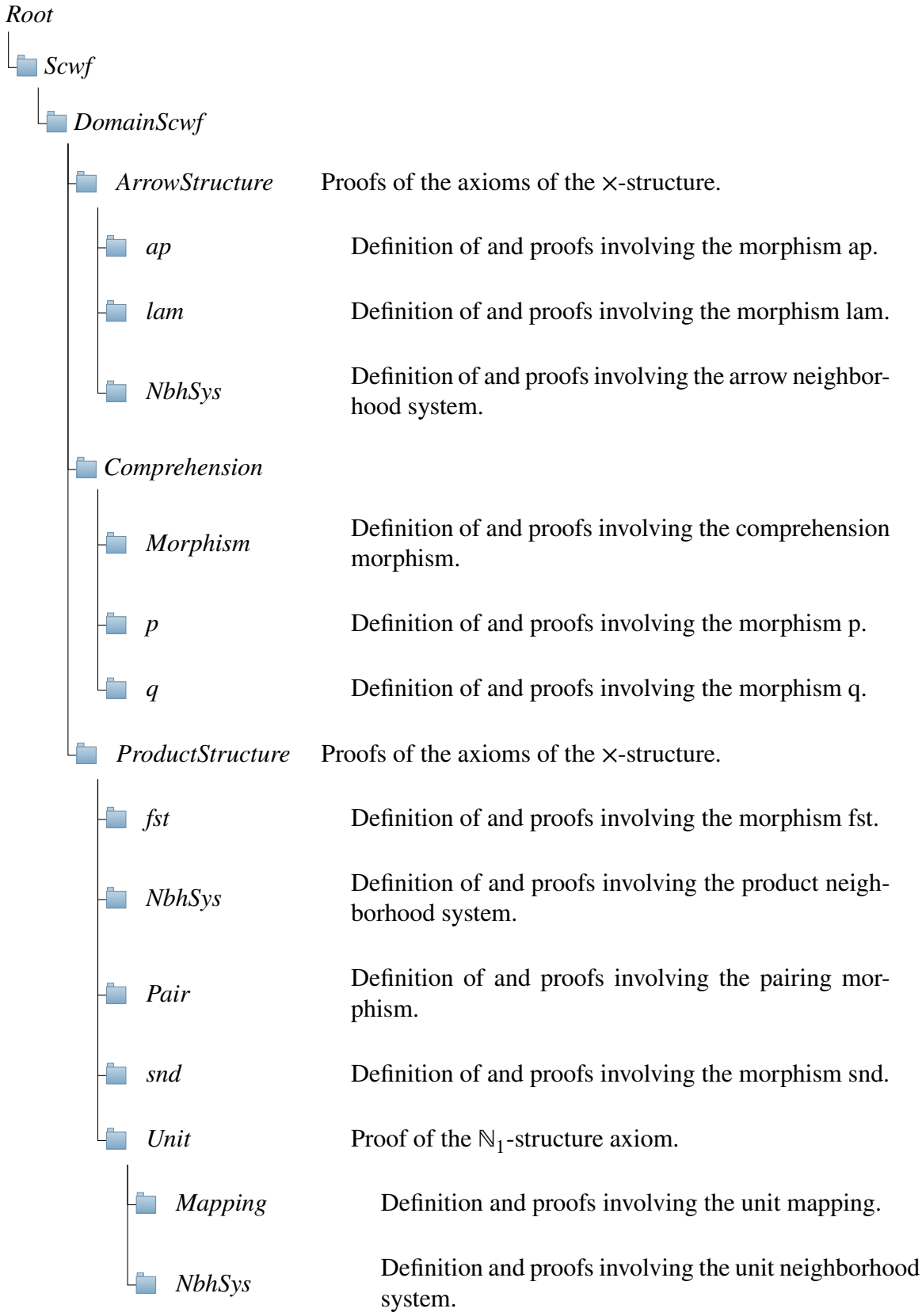
[16] Dana S. Scott. Domains for denotational semantics. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer, 1982.

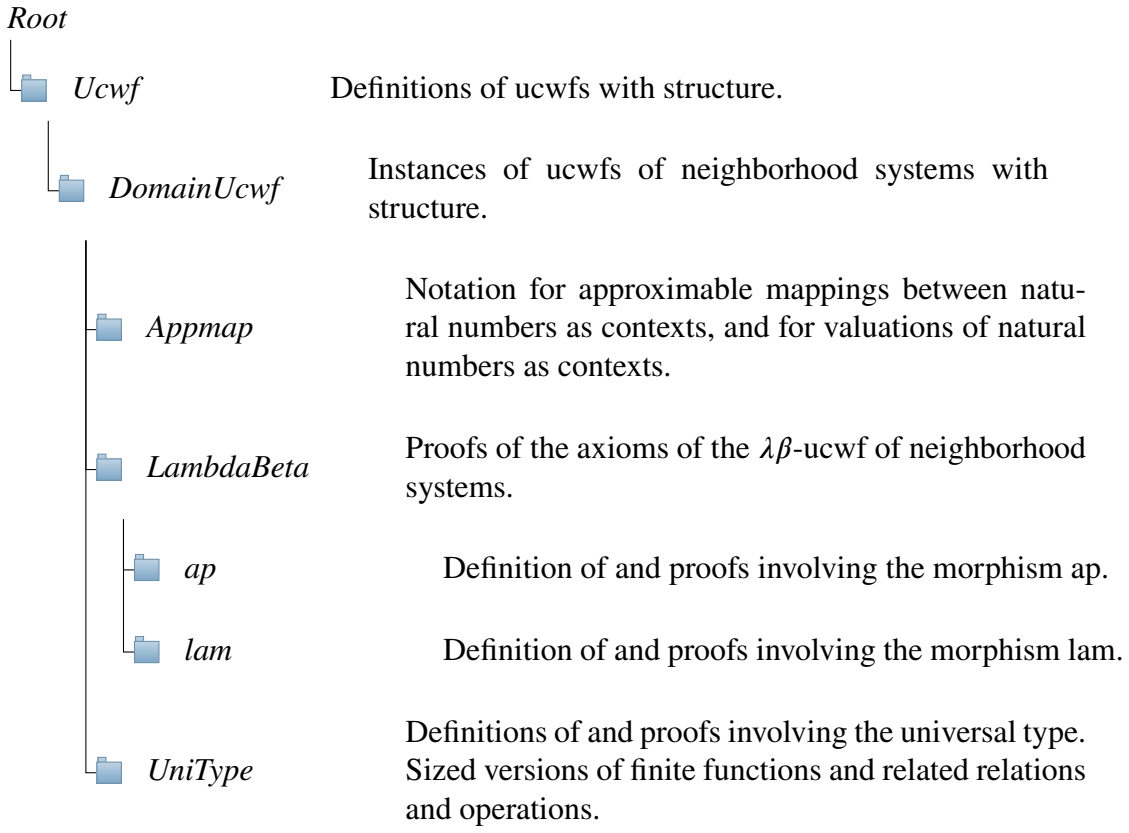
[17] Dana S Scott. Lectures on a mathematical theory of computation. In *Theoretical Foundations of Programming Methodology*, pages 145–292. Springer, 1982.

A

Code structure




Root
Scwf
DomainScwf
ArrowStructure
Proofs of the axioms of the ×-structure.
ap
Definition of and proofs involving the morphism ap.
lam
Definition of and proofs involving the morphism lam.
NbhSys
Definition of and proofs involving the arrow neighborhood system.
Comprehension
Morphism
Definition of and proofs involving the comprehension morphism.
p
Definition of and proofs involving the morphism p.
q
Definition of and proofs involving the morphism q.
ProductStructure
Proofs of the axioms of the ×-structure.
fst
Definition of and proofs involving the morphism fst.
NbhSys
Definition of and proofs involving the product neighborhood system.
Pair
Definition of and proofs involving the pairing morphism.
snd
Definition of and proofs involving the morphism snd.
Unit
Proof of the $\mathbb{N}_1$-structure axiom.
Mapping
Definition and proofs involving the unit mapping.
NbhSys
Definition and proofs involving the unit neighborhood system.


Root
Ucwf
Definitions of ucwfs with structure.
DomainUcwf
Instances of ucwfs of neighborhood systems with structure.
Appmap
Notation for approximable mappings between natural numbers as contexts, and for valuations of natural numbers as contexts.
LambdaBeta
Proofs of the axioms of the λβ-ucwf of neighborhood systems.
ap
Definition of and proofs involving the morphism ap.
lam
Definition of and proofs involving the morphism lam.
UniType
Definitions of and proofs involving the universal type.
Sized versions of finite functions and related relations and operations.

B

Code

B.0.1 Appmap/Definition

```
{-# OPTIONS --safe #-}

module Appmap.Definition where

open import NbhSys.Definition

open import Agda.Builtin.Sigma

private
  variable
    D D' : NbhSys

record Appmap (D D' : NbhSys) : Set₁ where
  field
    -- The mapping itself.
    _↦_ : NbhSys.Nbh D → NbhSys.Nbh D' → Set

    -- Axioms for the mapping.
    ↦-mono : ∀ {x y z} → [ D ] x ⊑ y → x ↦ z → y ↦ z
    ↦-bottom : ∀ {x} → x ↦ NbhSys.⊥ D'
    ↦-↓closed : ∀ {x y z} → [ D' ] y ⊑ z → x ↦ z → x ↦ y
    ↦-↑directed : ∀ {x y z} → x ↦ y → x ↦ z → (con : NbhSys.Con D' y z) →
                  x ↦ ([ D' ] y ⊔ z [ con ])
    ↦-con : ∀ {x y x' y'} → x ↦ y → x' ↦ y' → NbhSys.Con D x x' →
            NbhSys.Con D' y y'

-- Some simplifying syntax.
[_]_↦_ : Appmap D D' → NbhSys.Nbh D → NbhSys.Nbh D' → Set
[ γ ] x ↦ y = Appmap._↦_ γ x y

-- A (trivial) proof that approximable mappings are total.
↦-total : (γ : Appmap D D') → ∀ {x} →
          Σ (NbhSys.Nbh D') λ y → [ γ ] x ↦ y
↦-total {D' = D'} γ = NbhSys.⊥ D' , Appmap.↦-bottom γ
```

B.0.2 Appmap/Equivalence

```
{-# OPTIONS --safe #-}

module Appmap.Equivalence where

open import Appmap.Definition
open import Base.Core
open import NbhSys.Definition

private
  variable
    D D' : NbhSys

data _≼_ : Rel (Appmap D D') where
  ≼-intro : {γ δ : Appmap D D'} →
            (∀ {x y} → [ γ ] x ↦ y → [ δ ] x ↦ y) → γ ≼ δ

-- Two binary relations are equivalent iff they contain exactly
-- the same pairs.
data _≈_ : Rel (Appmap D D') where
  ≈-intro : {γ δ : Appmap D D'} → γ ≼ δ → δ ≼ γ → γ ≈ δ

≈Reflexive : Reflexive (_≈_ {D} {D'})
≈Reflexive = ≈-intro (≼-intro λ γx↦y → γx↦y)
                     (≼-intro λ γx↦y → γx↦y)

≈Symmetric : Symmetric (_≈_ {D} {D'})
≈Symmetric (≈-intro (≼-intro p) (≼-intro q))
  = ≈-intro (≼-intro q) (≼-intro p)

≈Transitive : Transitive (_≈_ {D} {D'})
≈Transitive (≈-intro (≼-intro p₁) (≼-intro q₁))
            (≈-intro (≼-intro p₂) (≼-intro q₂))
  = ≈-intro (≼-intro λ kx↦y → p₂ (p₁ kx↦y))
            (≼-intro λ kx↦y → q₁ (q₂ kx↦y))

≈IsEquiv : IsEquivalence (_≈_ {D} {D'})
≈IsEquiv = record { refl = ≈Reflexive
                  ; sym = ≈Symmetric
                  ; trans = ≈Transitive
                  }
```

B.0.3 Appmap/Lemmata

```
{-# OPTIONS --safe #-}

module Appmap.Lemmata where
```

```
open import Appmap.Definition
open import NbhSys.Definition
open import NbhSys.Lemmata

private
  variable
    D D' : NbhSys

appmapLemma₁ : {γ : Appmap D D'} → ∀ {x y z} →
                 (con : NbhSys.Con D x y) →
                 [ γ ] x ↦ z → [ γ ] ([ D ] x ⊔ y [ con ]) ↦ z
appmapLemma₁ {D} {γ = γ} con γx↦z
  = Appmap.↦-mono γ (NbhSys.⊑-⊔-fst D con) γx↦z

appmapLemma₂ : {γ : Appmap D D'} → ∀ {x y z} →
                 (con : NbhSys.Con D x y) →
                 [ γ ] y ↦ z → [ γ ] ([ D ] x ⊔ y [ con ]) ↦ z
appmapLemma₂ {D} {γ = γ} con γy↦z
  = Appmap.↦-mono γ (NbhSys.⊑-⊔-snd D con) γy↦z

appmapLemma₃ : {γ : Appmap D D'} → ∀ x y z w →
                 (conxy : NbhSys.Con D x y) →
                 (conzw : NbhSys.Con D' z w) →
                 [ γ ] x ↦ z → [ γ ] y ↦ w →
                 [ γ ] ([ D ] x ⊔ y [ conxy ]) ↦ ([ D' ] z ⊔ w [ conzw ])
appmapLemma₃ {γ = γ} x y z w conxy conzw γx↦z γy↦w
  = Appmap.↦-↑directed γ γ⊔↦z γ⊔↦w conzw
  where γ⊔↦z = appmapLemma₁ {γ = γ} conxy γx↦z
        γ⊔↦w = appmapLemma₂ {γ = γ} conxy γy↦w
```

B.0.4 Base/Core

```
{-# OPTIONS --safe #-}

module Base.Core where

open import NbhSys.Definition

open import Agda.Builtin.Nat

private
  variable
    A : Set□
    n : Nat

-- Standard implementation of a list.
data List : Nat → Set□ → Set□ where
  [] : List 0 A
```

```
  _::_ : A → List n A → List (suc n) A

-- Notation for lists with one element.
[_] : A → List 1 A
[ x ] = x :: []

head : List (suc n) A → A
head (x :: _) = x

tail : List (suc n) A → List n A
tail (_ :: xs) = xs

-- Standard implementation of a tuple.
-- We reserve the symbol × for other definitions.
data _⊠_ (A B : Set) : Set where
  _,_ : A → B → A ⊠ B

-- Logical or.
data _∨_ (A B : Set) : Set where
  inl : A → A ∨ B
  inr : B → A ∨ B

-- Types are neighborhood systems.
Ty : Set□
Ty = NbhSys

-- A context is a list of types.
Ctx : Nat → Set□
Ctx n = List n Ty

-- The below code is adapted from the standard library.
-- The point is to remove any dependencies on libraries.
-- For the purpose of the project, universe levels can be fixed.

Rel : (A : Set□) → Set□
Rel A = A → A → Set□

Reflexive : Rel A → Set□
Reflexive _≈_ = ∀ {x} → x ≈ x

Symmetric : Rel A → Set□
Symmetric _≈_ = ∀ {x y} → x ≈ y → y ≈ x

Transitive : Rel A → Set□
Transitive _≈_ = ∀ {x y z} → x ≈ y → y ≈ z → x ≈ z

record IsEquivalence (_≈_ : Rel A) : Set□ where
  field
    refl : Reflexive _≈_
    sym : Symmetric _≈_
    trans : Transitive _≈_
```

B.0.5 Base/FinFun

```
{-# OPTIONS --safe #-}

module Base.FinFun where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition

-- Finite functions are lists of pairs.
data FinFun (A B : Set) : Set where
  ∅ : FinFun A B
  _::_ : A ⊠ B → FinFun A B → FinFun A B

private
  variable
    f f' f'' f''' : FinFun A B

-- Short-hand when dealing with neighborhood systems.
NbhFinFun : Ty → Ty → Set
NbhFinFun 𝒜 ℬ = FinFun (NbhSys.Nbh 𝒜) (NbhSys.Nbh ℬ)

-- Set membership relation.
data _∈_ {A B : Set} : A ⊠ B → FinFun A B → Set where
  here : ∀ {x f} → x ∈ (x :: f)
  there : ∀ {x x' f} → x ∈ f → x ∈ (x' :: f)

-- Subset relation.
_⊆_ : (f f' : FinFun A B) → Set
f ⊆ f' = ∀ {x} → (x ∈ f → x ∈ f')

⊆-refl : f ⊆ f
⊆-refl x∈f = x∈f

⊆-trans : f ⊆ f' → f' ⊆ f'' → f ⊆ f''
⊆-trans f⊆f' f'⊆f'' x∈f = f'⊆f'' (f⊆f' x∈f)

⊆-lemma₁ : ∀ {x} → (x :: f') ⊆ f → (x :: ∅) ⊆ f
⊆-lemma₁ {x = x} xf'⊆f here = xf'⊆f here

⊆-lemma₂ : ∀ {x} → (x :: f') ⊆ f → f' ⊆ f
⊆-lemma₂ xf'⊆f y∈f' = xf'⊆f (there y∈f')

⊆-lemma₃ : ∀ {x} → f ⊆ (x :: f)
⊆-lemma₃ y∈f = ⊆-lemma₂ ⊆-refl y∈f

⊆-lemma₄ : ∀ {x} → x ∈ f → f' ⊆ f → (x :: f') ⊆ f
⊆-lemma₄ x∈f _ here = x∈f
⊆-lemma₄ x∈f f'⊆f (there y∈f) = f'⊆f y∈f
```

```
-- Set union.
_∪_ : FinFun A B → FinFun A B → FinFun A B
(x :: f) ∪ f' = x :: (f ∪ f')
∅ ∪ f' = f'

-- The empty set is a subset of any set.
∅-isSubset : ∅ ⊆ f
∅-isSubset ()

∪-lemma₁ : f ⊆ f'' → f' ⊆ f'' → (f ∪ f') ⊆ f''
∪-lemma₁ {f = ∅} f⊆f'' f'⊆f'' y∈f∪f' = f'⊆f'' y∈f∪f'
∪-lemma₁ {f = x :: _} f⊆f'' f'⊆f'' here = f⊆f'' here
∪-lemma₁ {f = x :: f'''} f⊆f'' f'⊆f'' (there y∈f∪f')
  = ∪-lemma₁ (⊆-trans ⊆-lemma₃ f⊆f'') f'⊆f'' y∈f∪f'

∪-lemma₂ : ∀ {x} → x ∈ (f ∪ f') → (x ∈ f) ∨ (x ∈ f')
∪-lemma₂ {f = ∅} here = inr here
∪-lemma₂ {f = ∅} (there x∈xs) = inr (there x∈xs)
∪-lemma₂ {f = x :: _} here = inl here
∪-lemma₂ {f = x :: f''} (there y∈∪) with (∪-lemma₂ y∈∪)
∪-lemma₂ (there y∈∪) | inl y∈f'' = inl (there y∈f'')
∪-lemma₂ (there y∈∪) | inr y∈f' = inr y∈f'

∪-lemma₃ : ∀ {x} → x ∈ f → x ∈ (f ∪ f')
∪-lemma₃ {f = x :: f''} here = here
∪-lemma₃ {f = x :: f''} {x = y} (there y∈f'') = ⊆-lemma₃ y∈f''∪f'
  where y∈f''∪f' = ∪-lemma₃ y∈f''

∪-lemma₄ : ∀ {x} → x ∈ f' → x ∈ (f ∪ f')
∪-lemma₄ {f = ∅} x∈f' = x∈f'
∪-lemma₄ {f = x :: f''} {x = y} y∈f' = ⊆-lemma₃ y∈f''∪f'
  where y∈f''∪f' = ∪-lemma₄ y∈f'

∪-lemma₅ : f ⊆ f'' → f' ⊆ f''' → (f ∪ f') ⊆ (f'' ∪ f''')
∪-lemma₅ _ _ x∈f∪f' with (∪-lemma₂ x∈f∪f')
∪-lemma₅ {f'' = f''} {f''' = f'''} f⊆f'' _ x∈f∪f' | inl x∈f
  = ∪-lemma₃ (f⊆f'' x∈f)
∪-lemma₅ _ f'⊆f''' x∈f∪f' | inr x∈f'
  = ∪-lemma₄ (f'⊆f''' x∈f')

∪-lemma₆ : f ⊆ (f ∪ f')
∪-lemma₆ x∈f = ∪-lemma₃ x∈f

∪-lemma□ : f' ⊆ (f ∪ f')
∪-lemma□ x∈f = ∪-lemma₄ x∈f

-- From a proof that a pair of neighborhoods is in the
-- empty set, anything.
xy∈∅-abs : {p : Set} → ∀ {x y} →
             _∈_ {A} {B} (x , y) ∅ → p
xy∈∅-abs ()
```

B.0.6 Base/Variables

```
{-# OPTIONS --safe #-}

module Base.Variables where

open import Base.Core

open import Agda.Builtin.Nat

variable
  m n o r : Nat
  Γ : Ctx m
  Δ : Ctx n
  Θ : Ctx o
  Λ : Ctx r
  𝒜 ℬ : Ty
  A B : Set
```

B.0.7 NbhSys/Definition

```
{-# OPTIONS --safe #-}

module NbhSys.Definition where

record NbhSys : Set₁ where
  field
    Nbh : Set
    _⊑_ : Nbh → Nbh → Set
    Con : Nbh → Nbh → Set
    _⊔_[_] : (x y : Nbh) → Con x y → Nbh
    ⊥ : Nbh

    Con-⊔ : ∀ {x y z} → x ⊑ z → y ⊑ z → Con x y

    ⊑-refl  : ∀ {x} → x ⊑ x
    ⊑-trans : ∀ {x y z} → x ⊑ y → y ⊑ z → x ⊑ z
    ⊑-⊥     : ∀ {x} → ⊥ ⊑ x
    ⊑-⊔     : ∀ {x y z} → y ⊑ x → z ⊑ x → (con : Con y z) →
              (y ⊔ z [ con ]) ⊑ x
    ⊑-⊔-fst : ∀ {x y} → (con : Con x y) → x ⊑ (x ⊔ y [ con ])
    ⊑-⊔-snd : ∀ {x y} → (con : Con x y) → y ⊑ (x ⊔ y [ con ])

  -- Some simplifying syntax.
  [_]_⊑_ : (D : NbhSys) → (x y : NbhSys.Nbh D) → Set
  [ A ] x ⊑ y = NbhSys._⊑_ A x y

  [_]_⊔_[_] : (D : NbhSys) → (x y : NbhSys.Nbh D) → NbhSys.Con D x y →
              NbhSys.Nbh D
  [ A ] x ⊔ y [ con ] = NbhSys._⊔_[_] A x y con
```

B.0.8 NbhSys/Lemmata

```
{-# OPTIONS --safe #-}

open import NbhSys.Definition

module NbhSys.Lemmata (D : NbhSys) where

private
  variable
    x y z w : NbhSys.Nbh D

conRefl : ∀ {x} → NbhSys.Con D x x
conRefl = NbhSys.Con-⊔ D (NbhSys.⊑-refl D) (NbhSys.⊑-refl D)

conSym : NbhSys.Con D x y → NbhSys.Con D y x
conSym {x} {y} conxy
  = NbhSys.Con-⊔ D (NbhSys.⊑-⊔-snd D conxy) (NbhSys.⊑-⊔-fst D conxy)
  where x⊔y = [ D ] x ⊔ y [ conxy ]

con⊥₁ : NbhSys.Con D (NbhSys.⊥ D) x
con⊥₁ = NbhSys.Con-⊔ D (NbhSys.⊑-⊥ D) (NbhSys.⊑-refl D)

con⊥₂ : NbhSys.Con D x (NbhSys.⊥ D)
con⊥₂ = NbhSys.Con-⊔ D (NbhSys.⊑-refl D) (NbhSys.⊑-⊥ D)

⊑-⊔-lemma₁ : (con : NbhSys.Con D y z) → [ D ] ([ D ] y ⊔ z [ con ]) ⊑ x →
               [ D ] y ⊑ x
⊑-⊔-lemma₁ con y⊔z⊑x =
  NbhSys.⊑-trans D (NbhSys.⊑-⊔-fst D con) y⊔z⊑x

⊑-⊔-lemma₂ : (con : NbhSys.Con D y z) → [ D ] ([ D ] y ⊔ z [ con ]) ⊑ x →
               [ D ] z ⊑ x
⊑-⊔-lemma₂ con y⊔z⊑x =
  NbhSys.⊑-trans D (NbhSys.⊑-⊔-snd D con) y⊔z⊑x

⊑-⊔-lemma₃ : (conxy : NbhSys.Con D x y) → (conzw : NbhSys.Con D z w) →
               [ D ] x ⊑ z → [ D ] y ⊑ w →
               [ D ] ([ D ] x ⊔ y [ conxy ]) ⊑ ([ D ] z ⊔ w [ conzw ])
⊑-⊔-lemma₃ conxy conzw x⊑z y⊑w = NbhSys.⊑-⊔ D x⊑z⊔w y⊑z⊔w conxy
  where x⊑z⊔w = NbhSys.⊑-trans D x⊑z (NbhSys.⊑-⊔-fst D conzw)
        y⊑z⊔w = NbhSys.⊑-trans D y⊑w (NbhSys.⊑-⊔-snd D conzw)

⊑-⊔-lemma₄ : [ D ] x ⊑ y → (con : NbhSys.Con D y z) →
               [ D ] x ⊑ ([ D ] y ⊔ z [ con ])
⊑-⊔-lemma₄ x⊑y con = NbhSys.⊑-trans D x⊑y (NbhSys.⊑-⊔-fst D con)

⊑-⊔-lemma₅ : [ D ] x ⊑ z → (con : NbhSys.Con D y z) →
               [ D ] x ⊑ ([ D ] y ⊔ z [ con ])
⊑-⊔-lemma₅ x⊑z con = NbhSys.⊑-trans D x⊑z (NbhSys.⊑-⊔-snd D con)
```

```
⊔-ass₁ : (conxy : NbhSys.Con D x y) →
         (conyz : NbhSys.Con D y z) →
         (conxy⊔z : NbhSys.Con D x ([ D ] y ⊔ z [ conyz ])) →
         (conx⊔yz : NbhSys.Con D ([ D ] x ⊔ y [ conxy ]) z) →
         [ D ] ([ D ] x ⊔ ([ D ] y ⊔ z [ conyz ]) [ conxy⊔z ]) ⊑ w →
         [ D ] [ D ] ([ D ] x ⊔ y [ conxy ]) ⊔ z [ conx⊔yz ] ⊑ w
⊔-ass₁ conxy conyz conxy⊔z conx⊔yz p
  = NbhSys.⊑-⊔ D (NbhSys.⊑-⊔ D (⊑-⊔-lemma₁ _ p)
    (⊑-⊔-lemma₁ _ (⊑-⊔-lemma₂ _ p)) _)
    (⊑-⊔-lemma₂ _ (⊑-⊔-lemma₂ _ p)) _

⊔-ass₂ : (conxy : NbhSys.Con D x y) →
         (conyz : NbhSys.Con D y z) →
         (conxy⊔z : NbhSys.Con D x ([ D ] y ⊔ z [ conyz ])) →
         (conx⊔yz : NbhSys.Con D ([ D ] x ⊔ y [ conxy ]) z) →
         [ D ] [ D ] ([ D ] x ⊔ y [ conxy ]) ⊔ z [ conx⊔yz ] ⊑ w →
         [ D ] ([ D ] x ⊔ ([ D ] y ⊔ z [ conyz ]) [ conxy⊔z ]) ⊑ w
⊔-ass₂ conxy conyz conxy⊔z conx⊔yz p
  = NbhSys.⊑-⊔ D (⊑-⊔-lemma₁ _ (⊑-⊔-lemma₁ _ p))
    (NbhSys.⊑-⊔ D (⊑-⊔-lemma₂ _ (⊑-⊔-lemma₁ _ p))
    (⊑-⊔-lemma₂ _ p) _) _
```

B.0.9 Scwf/Plain

```
{-# OPTIONS --safe #-}

module Scwf.Plain where

open import Base.Core using (Rel ; IsEquivalence)
open import Base.Variables

open import Agda.Builtin.Nat

record Scwf : Set₂ where
  field
    Ty : Set₁
    Ctx : Nat → Set₁
    Tm : Ctx n → Ty → Set₁
    Sub : Ctx m → Ctx n → Set₁

    _≈_ : ∀ {Γ 𝒜} → Rel (Tm {n} Γ 𝒜)
    _≅_ : ∀ {Γ Δ} → Rel (Sub {m} {n} Γ Δ)

    isEquivT : ∀ {Γ 𝒜} → IsEquivalence (_≈_ {n} {Γ} {𝒜})
    isEquivS : ∀ {Γ Δ} → IsEquivalence (_≅_ {m} {n} {Γ} {Δ})

    ⋄ : Ctx zero
```

```
_•_ : Ctx n → Ty → Ctx (suc n)

q : (Γ : Ctx n) → (𝒜 : Ty) → Tm (Γ • 𝒜) 𝒜
_[_] : ∀ {𝒜 Γ Δ} → Tm {n} Δ 𝒜 → Sub {m} Γ Δ → Tm Γ 𝒜

id : (Γ : Ctx n) → Sub Γ Γ
_∘_ : ∀ {Γ Δ Θ} → Sub {n} {o} Δ Θ → Sub {m} Γ Δ → Sub Γ Θ
⟨⟩ : {Γ : Ctx n} → Sub Γ ◇
⟨_,_⟩ : ∀ {Γ Δ 𝒜} → Sub {n} {m} Δ Γ → Tm Δ 𝒜 → Sub Δ (Γ • 𝒜)
p : (Γ : Ctx n) → (𝒜 : Ty) → Sub (Γ • 𝒜) Γ

idL : ∀ {Γ Δ} → (γ : Sub {n} {m} Δ Γ) → ((id Γ) ∘ γ) ≅ γ
idR : ∀ {Γ Δ} → (γ : Sub {n} {m} Δ Γ) → (γ ∘ (id Δ)) ≅ γ
subAssoc : ∀ {Γ Δ Θ Λ} → (γ : Sub {m} {n} Γ Δ) →
           (δ : Sub {n} {o} Δ Θ) → (θ : Sub {o} {r} Θ Λ) →
           ((θ ∘ δ) ∘ γ) ≅ (θ ∘ (δ ∘ γ))

idSub : ∀ {Γ 𝒜} → (t : Tm {n} Γ 𝒜) → (t [ id Γ ]) ≈ t
compSub : ∀ {Γ Δ Θ 𝒜} → (t : Tm {n} Δ 𝒜) →
          (γ : Sub {m} Γ Δ) → (δ : Sub {o} Θ Γ) →
          (t [ γ ∘ δ ]) ≈ ((t [ γ ]) [ δ ])

id₀ : id ◇ ≅ ⟨⟩
<>-zero : ∀ {Γ Δ} → (γ : Sub {m} {n} Γ Δ) → (⟨⟩ ∘ γ) ≅ ⟨⟩

pCons : ∀ {𝒜 Γ Δ} → (γ : Sub {n} {m} Δ Γ) → (t : Tm Δ 𝒜) →
        ((p Γ 𝒜) ∘ ⟨ γ , t ⟩) ≅ γ
qCons : ∀ {𝒜 Γ Δ} → (γ : Sub {n} {m} Δ Γ) → (t : Tm Δ 𝒜) →
        ((q Γ 𝒜) [ ⟨ γ , t ⟩ ]) ≈ t
idExt : ∀ {𝒜 Γ} → (id (_•_ {m} Γ 𝒜)) ≅ ⟨ p Γ 𝒜 , q Γ 𝒜 ⟩
compExt : ∀ {Γ Δ 𝒜} → (t : Tm Δ 𝒜) →
          (γ : Sub {n} {m} Δ Γ) → (δ : Sub Γ Δ) →
          (⟨ γ , t ⟩ ∘ δ) ≅ ⟨ γ ∘ δ , t [ δ ] ⟩

subCong : ∀ {𝒜 Γ Δ} → {t t' : Tm {m} Γ 𝒜} →
          {γ γ' : Sub {n} Δ Γ} → t ≈ t' →
          γ ≅ γ' → (t [ γ ]) ≈ (t' [ γ' ])
<,>-cong : ∀ {𝒜 Γ Δ} → {t t' : Tm {m} Γ 𝒜} →
           {γ γ' : Sub {m} {n} Γ Δ} → t ≈ t' →
           γ ≅ γ' → ⟨ γ , t ⟩ ≅ ⟨ γ' , t' ⟩
∘-cong : ∀ {Γ Δ Θ} → {γ δ : Sub {n} {o} Δ Θ} →
         {γ' δ' : Sub {m} Γ Δ} → γ ≅ δ →
         γ' ≅ δ' → (γ ∘ γ') ≅ (δ ∘ δ')
```

B.0.10 Scwf/Product

```
{-# OPTIONS --safe #-}
```

```
module Scwf.Product where

open import Base.Variables
open import Scwf.Plain

open import Agda.Builtin.Nat

record Prod-scwf : Set₂ where
  field
    scwf : Scwf
  open Scwf scwf public
  field
    _×_ : Ty → Ty → Ty

    fst : ∀ {Γ 𝒜 ℬ} → Tm {m} Γ (𝒜 × ℬ) → Tm Γ 𝒜
    snd : ∀ {Γ 𝒜 ℬ} → Tm {m} Γ (𝒜 × ℬ) → Tm Γ ℬ
    <_,_> : ∀ {Γ 𝒜 ℬ} → Tm {m} Γ 𝒜 → Tm Γ ℬ → Tm Γ (𝒜 × ℬ)

    fstAxiom : ∀ {Γ 𝒜 ℬ} → {t : Tm {m} Γ 𝒜} → {u : Tm Γ ℬ} →
                 fst < t , u > ≈ t
    sndAxiom : ∀ {Γ 𝒜 ℬ} → {t : Tm {m} Γ 𝒜} → {u : Tm Γ ℬ} →
                 snd < t , u > ≈ u
    pairSub : ∀ {Γ Δ 𝒜 ℬ} → {t : Tm Γ 𝒜} → {u : Tm Γ ℬ} →
                {γ : Sub {n} {m} Δ Γ} →
                (< t , u > [ γ ]) ≈ < t [ γ ] , u [ γ ] >

    fstCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ (𝒜 × ℬ)} → t ≈ t' →
                fst t ≈ fst t'
    sndCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ (𝒜 × ℬ)} → t ≈ t' →
                snd t ≈ snd t'
    pairCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ 𝒜} → {u u' : Tm Γ ℬ} →
                 t ≈ t' → u ≈ u' →
                 < t , u > ≈ < t' , u' >

    -- We merge the ℕ₁-structure with the ×-structure.
    ℕ₁ : Ty
    0₁ : ∀ {Γ} → Tm {m} Γ ℕ₁
    ℕ₁-sub : ∀ {Γ Δ} → {γ : Sub {n} {m} Δ Γ} →
               (0₁ [ γ ]) ≈ 0₁
```

B.0.11 Scwf/ProductArrow

```
{-# OPTIONS --safe #-}

module Scwf.ProductArrow where

open import Base.Variables
open import Scwf.Product
```

```
record ProductArrow-scwf : Set₂ where
  field
    prod-scwf : Prod-scwf
  open Prod-scwf prod-scwf public
  field
    _⇒_ : Ty → Ty → Ty

    lam : ∀ {Γ 𝒜 ℬ} → Tm (_•_ {m} Γ 𝒜) ℬ → Tm Γ (𝒜 ⇒ ℬ)
    ap  : ∀ {Γ 𝒜 ℬ} → Tm Γ (𝒜 ⇒ ℬ) → Tm {m} Γ 𝒜 → Tm Γ ℬ

    lamSub : ∀ {Γ Δ 𝒜 ℬ} → (γ : Sub {n} {m} Δ Γ) →
             (t : Tm (Γ • 𝒜) ℬ) →
             (lam t [ γ ]) ≈ (lam (t [ ⟨ γ ∘ p Δ 𝒜 , q Δ 𝒜 ⟩ ]))
    apSub : ∀ {Γ Δ 𝒜 ℬ} → (γ : Sub {n} {m} Δ Γ) →
             (t : Tm Γ (𝒜 ⇒ ℬ)) → (u : Tm Γ 𝒜) →
             (ap (t [ γ ]) (u [ γ ])) ≈ (ap t u [ γ ])

    β : ∀ {Γ 𝒜 ℬ} → {t : Tm {m} Γ 𝒜} → {u : Tm (Γ • 𝒜) ℬ} →
        (ap (lam u) t) ≈ (u [ ⟨ id Γ , t ⟩ ])

    lamCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm (_•_ {m} Γ 𝒜) ℬ} →
              t ≈ t' → (lam t) ≈ (lam t')
    apCong : ∀ {Γ 𝒜 ℬ} → {t t' : Tm {m} Γ (𝒜 ⇒ ℬ)} →
              ∀ {u u'} → t ≈ t' → u ≈ u' →
              (ap t u) ≈ (ap t' u')
```

B.0.12 Scwf/DomainScwf/PlainAxiomProofs

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.PlainAxiomProofs where

open import Appmap.Equivalence
open import Appmap.Lemmata
open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Appmap.Empty.Instance
open import Scwf.DomainScwf.Appmap.Empty.Relation
open import Scwf.DomainScwf.Appmap.Identity.Instance
open import Scwf.DomainScwf.Appmap.Identity.Relation
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
```

```
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.Comprehension.p.Instance
open import Scwf.DomainScwf.Comprehension.p.Relation
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Relation
open import Scwf.DomainScwf.Comprehension.q.Instance
open import Scwf.DomainScwf.Comprehension.q.Relation

private
  variable
    γ γ' : tAppmap Γ Δ
    δ δ' : tAppmap Δ Θ
    θ : tAppmap Θ Λ
    t t' : tAppmap Δ [ 𝒜 ]

subAssocLemma₁ : ∀ {x y} → [ (θ ∘ δ) ∘ γ ] x ↦ y →
                           [ θ ∘ (δ ∘ γ) ] x ↦ y
subAssocLemma₁ (∘↦-intro γx↦z (∘↦-intro δz↦w θw↦y))
  = ∘↦-intro (∘↦-intro γx↦z δz↦w) θw↦y

subAssocLemma₂ : ∀ {x y} → [ θ ∘ (δ ∘ γ) ] x ↦ y →
                           [ (θ ∘ δ) ∘ γ ] x ↦ y
subAssocLemma₂ (∘↦-intro (∘↦-intro γx↦w δw↦z) θz↦y)
  = ∘↦-intro γx↦w (∘↦-intro δw↦z θz↦y)

subAssoc : (γ : tAppmap Γ Δ) → (δ : tAppmap Δ Θ) →
           (θ : tAppmap Θ Λ) →
           ((θ ∘ δ) ∘ γ) ≈ (θ ∘ (δ ∘ γ))
subAssoc γ δ θ = ≈-intro (≼-intro subAssocLemma₁)
                         (≼-intro subAssocLemma₂)

pConsLemma₁ : ∀ {x y} → [ p Γ 𝒜 ∘ ⟨ γ , t ⟩ ] x ↦ y →
                        [ γ ] x ↦ y
pConsLemma₁ {γ = γ} (∘↦-intro (⟨⟩↦-intro γx↦z _) (p↦-intro y⊑z))
  = Appmap.↦-↓closed γ y⊑z γx↦z

pConsLemma₂ : ∀ {x y} → [ γ ] x ↦ y →
                        [ p Γ 𝒜 ∘ ⟨ γ , t ⟩ ] x ↦ y
pConsLemma₂ {γ = γ} {𝒜 = 𝒜} {t} γx↦y
  = ∘↦-intro γtx↦⊥y p⊥y↦y
  where tx↦⊥ = Appmap.↦-bottom t
        γtx↦⊥y = ⟨⟩↦-intro {y = ⟨⟨ NbhSys.⊥ 𝒜 ,, _ ⟩⟩} γx↦y tx↦⊥
        p⊥y↦y = p↦-intro (NbhSys.⊑-refl (ValNbhSys _))

pCons : (γ : tAppmap Δ Γ) → (t : tAppmap Δ [ 𝒜 ]) →
        (p Γ 𝒜 ∘ ⟨ γ , t ⟩) ≈ γ
pCons γ t = ≈-intro (≼-intro pConsLemma₁)
```

```
                (≼-intro pConsLemma₂)

qConsLemma₁ : ∀ {x y} → [ q Γ 𝒜 ∘ ⟨ γ , t ⟩ ] x ↦ y →
                [ t ] x ↦ y
qConsLemma₁ {𝒜 = 𝒜} {t = t} {y = ⟨⟨ y ,, ⟨⟩ ⟩⟩}
  (∘↦-intro (⟨⟩↦-intro _ tx↦z) (q↦-intro y⊑z))
  = Appmap.↦-↓closed t tup-y⊑z tx↦z
  where tup-y⊑z = ⊑ᵥ-cons [ 𝒜 ] y⊑z ⊑ᵥ-nil

qConsLemma₂ : ∀ {x y} → [ t ] x ↦ y →
                [ q Γ 𝒜 ∘ ⟨ γ , t ⟩ ] x ↦ y
qConsLemma₂ {𝒜 = 𝒜} {γ = γ} {y = ⟨⟨ y ,, ⟨⟩ ⟩⟩} tx↦y =
  ∘↦-intro γtx↦y⊥ qy⊥↦y
  where γx↦⊥ = Appmap.↦-bottom γ
        qy⊥↦y = q↦-intro (NbhSys.⊑-refl 𝒜)
        γtx↦y⊥ = ⟨⟩↦-intro {y = ⟨⟨ y ,, ⊥ᵥ ⟩⟩} γx↦⊥ tx↦y

qCons : (γ : tAppmap Δ Γ) → (t : tAppmap Δ [ 𝒜 ]) →
        ((q Γ 𝒜) ∘ ⟨ γ , t ⟩) ≈ t
qCons γ t = ≈-intro (≼-intro qConsLemma₁)
                    (≼-intro qConsLemma₂)

idExtLemma₁ : ∀ {x y} → x id↦ y → ⟨⟩↦ (p Γ 𝒜) (q Γ 𝒜) x y
idExtLemma₁ (id↦-intro (⊑ᵥ-cons _ y⊑x y⊑x))
  = ⟨⟩↦-intro px x↦y qx x↦y
  where px x↦y = p↦-intro y⊑x
        qx x↦y = q↦-intro y⊑x

idExtLemma₂ : ∀ {x y} → ⟨⟩↦ (p Γ 𝒜) (q Γ 𝒜) x y →
                x id↦ y
idExtLemma₂ {Γ = Γ} {𝒜 = 𝒜}
  (⟨⟩↦-intro (p↦-intro y⊑x) (q↦-intro y⊑x))
  = id↦-intro yy⊑xx
  where yy⊑xx = ⊑ᵥ-cons (𝒜 :: Γ) y⊑x y⊑x

idExt : idMap (𝒜 :: Γ) ≈ ⟨ p Γ 𝒜 , q Γ 𝒜 ⟩
idExt = ≈-intro (≼-intro idExtLemma₁)
                (≼-intro idExtLemma₂)

idLLemma₁ : ∀ {x y} → [ idMap Γ ∘ γ ] x ↦ y →
                [ γ ] x ↦ y
idLLemma₁ {γ = γ} (∘↦-intro γx↦z (id↦-intro y⊑z))
  = Appmap.↦-↓closed γ y⊑z γx↦z

idLLemma₂ : ∀ {x y} → [ γ ] x ↦ y →
                [ idMap Γ ∘ γ ] x ↦ y
idLLemma₂ x↦y = ∘↦-intro x↦y (id↦-intro y⊑y)
  where y⊑y = NbhSys.⊑-refl (ValNbhSys _)
```

```
idL : (γ : tAppmap Δ Γ) → (idMap Γ ∘ γ) ≈ γ
idL γ = ≈-intro (≼-intro idLLemma₁) (≼-intro idLLemma₂)

idRLemma₁ : ∀ {x y} → [ γ ∘ idMap Δ ] x ↦ y →
                      [ γ ] x ↦ y
idRLemma₁ {γ = γ} (∘↦-intro (id↦-intro z⊑x) γz↦y)
  = Appmap.↦-mono γ z⊑x γz↦y

idRLemma₂ : ∀ {x y} → [ γ ] x ↦ y →
                      [ γ ∘ idMap Δ ] x ↦ y
idRLemma₂ x↦y
  = ∘↦-intro (id↦-intro x⊑x) x↦y
  where x⊑x = NbhSys.⊑-refl (ValNbhSys _)

idR : (γ : tAppmap Δ Γ) → (γ ∘ idMap Δ) ≈ γ
idR γ = ≈-intro (≼-intro idRLemma₁) (≼-intro idRLemma₂)

id₀Lemma₁ : ∀ {x y} → x id↦ y → x empty↦ y
id₀Lemma₁ {⟨⟩} {⟨⟩} idx↦y = empty↦-intro

id₀Lemma₂ : ∀ {x y} → x empty↦ y → x id↦ y
id₀Lemma₂ {⟨⟩} {⟨⟩} emx↦y = id↦-intro ⊑ᵥ-nil

id₀ : idMap [] ≈ emptyMap
id₀ = ≈-intro (≼-intro id₀Lemma₁) (≼-intro id₀Lemma₂)

<>-zeroLemma₁ : ∀ {x y} → [ emptyMap ∘ γ ] x ↦ y →
                          x empty↦ y
<>-zeroLemma₁ {y = ⟨⟩} _ = empty↦-intro

<>-zeroLemma₂ : ∀ {x y} → x empty↦ y →
                          [ emptyMap ∘ γ ] x ↦ y
<>-zeroLemma₂ {γ = γ} {y = ⟨⟩} empty↦-intro
  = ∘↦-intro γx↦⊥ empty↦-intro
    where γx↦⊥ = Appmap.↦-bottom γ

<>-zero : (γ : tAppmap Γ Δ) → (emptyMap ∘ γ) ≈ emptyMap
<>-zero γ = ≈-intro (≼-intro <>-zeroLemma₁)
                    (≼-intro <>-zeroLemma₂)

idSubLemma₁ : ∀ {x y} → [ t ∘ idMap Γ ] x ↦ y →
                        [ t ] x ↦ y
idSubLemma₁ {t = t} (∘↦-intro (id↦-intro z⊑x) tz↦y)
  = Appmap.↦-mono t z⊑x tz↦y

idSubLemma₂ : ∀ {x y} → [ t ] x ↦ y →
                        [ t ∘ idMap Γ ] x ↦ y
idSubLemma₂ {t = t} tx↦y
```

```
  = ∘↦-intro (id↦-intro x⊑x) tx↦y
  where x⊑x = NbhSys.⊑-refl (ValNbhSys _)

idSub : (t : tAppmap Γ [ 𝒜 ]) →
        (t ∘ idMap Γ) ≈ t
idSub t = ≈-intro (≼-intro idSubLemma₁)
                  (≼-intro idSubLemma₂)

compSubLemma₁ : ∀ {x y} → [ t ∘ (γ ∘ δ) ] x ↦ y →
                          [ (t ∘ γ) ∘ δ ] x ↦ y
compSubLemma₁ (∘↦-intro (∘↦-intro δx↦w γw↦z) tz↦y)
  = ∘↦-intro δx↦w (∘↦-intro γw↦z tz↦y)

compSubLemma₂ : ∀ {x y} → [ (t ∘ γ) ∘ δ ] x ↦ y →
                          [ t ∘ (γ ∘ δ) ] x ↦ y
compSubLemma₂ (∘↦-intro δx↦z (∘↦-intro γz↦w tw↦y))
  = ∘↦-intro (∘↦-intro δx↦z γz↦w) tw↦y

compSub : (t : tAppmap Δ [ 𝒜 ]) → (γ : tAppmap Γ Δ) →
          (δ : tAppmap Θ Γ) →
          (t ∘ (γ ∘ δ)) ≈ ((t ∘ γ) ∘ δ)
compSub t γ δ = ≈-intro (≼-intro compSubLemma₁)
                        (≼-intro compSubLemma₂)

compExtLemma₁ : ∀ {x y} → [ ⟨ γ , t ⟩ ∘ δ ] x ↦ y →
                          [ ⟨ γ ∘ δ , t ∘ δ ⟩ ] x ↦ y
compExtLemma₁ (∘↦-intro δx↦z (⟨⟩↦-intro γz↦y tz↦y))
  = ⟨⟩↦-intro (∘↦-intro δx↦z γz↦y) (∘↦-intro δx↦z tz↦y)

compExtLemma₂ : ∀ {x y} → [ ⟨ γ ∘ δ , t ∘ δ ⟩ ] x ↦ y →
                          [ ⟨ γ , t ⟩ ∘ δ ] x ↦ y
compExtLemma₂ {γ = γ} {δ = δ} {t = t}
  (⟨⟩↦-intro (∘↦-intro δx↦z γz↦y) (∘↦-intro δx↦w tw↦y))
    = ∘↦-intro δx↦z⊔w ⟨γ,t⟩↦
      where conzw = Appmap.↦-con δ δx↦z δx↦w valConRefl
            δx↦z⊔w = Appmap.↦-↑directed δ δx↦z δx↦w conzw
            γz⊔w↦y = appmapLemma₁ {γ = γ} conzw γz↦y
            tz⊔w↦y = appmapLemma₂ {γ = t} conzw tw↦y
            ⟨γ,t⟩↦ = ⟨⟩↦-intro γz⊔w↦y tz⊔w↦y

compExt : (t : tAppmap Δ [ 𝒜 ]) → (γ : tAppmap Δ Γ) →
          (δ : tAppmap Γ Δ) →
          (⟨ γ , t ⟩ ∘ δ) ≈ ⟨ γ ∘ δ , t ∘ δ ⟩
compExt t γ δ = ≈-intro (≼-intro compExtLemma₁)
                        (≼-intro compExtLemma₂)

<,>-congLemma : t ≈ t' → γ ≈ γ' → ∀ {x y} → ⟨⟩↦ γ t x y →
                          ⟨⟩↦ γ' t' x y
```

```
<,>-congLemma (≈-intro (≼-intro t'x↦y) _)
  (≈-intro (≼-intro γ'x↦y) _) (⟨⟩↦-intro γx↦y tx↦y)
  = ⟨⟩↦-intro (γ'x↦y γx↦y) (t'x↦y tx↦y)

<,>-cong : t ≈ t' → γ ≈ γ' → ⟨ γ , t ⟩ ≈ ⟨ γ' , t' ⟩
<,>-cong t≈t' γ≈γ' = ≈-intro γt≼γ't' γ't'≼γt
  where γt≼γ't' = ≼-intro (<,>-congLemma t≈t' γ≈γ')
        t'≈t = ≈Symmetric t≈t'
        γ'≈γ = ≈Symmetric γ≈γ'
        γ't'≼γt = ≼-intro (<,>-congLemma t'≈t γ'≈γ)

∘-congLemma : γ ≈ δ → γ' ≈ δ' → ∀ {x y} → [ γ ∘ γ' ] x ↦ y →
                [ δ ∘ δ' ] x ↦ y
∘-congLemma (≈-intro (≼-intro t'z↦y) _)
  (≈-intro (≼-intro γ'x↦z) _) (∘↦-intro γx↦z tz↦y)
  = ∘↦-intro (γ'x↦z γx↦z) (t'z↦y tz↦y)

∘-cong : γ ≈ δ → γ' ≈ δ' → (γ ∘ γ') ≈ (δ ∘ δ')
∘-cong γ≈δ γ'≈δ'
  = ≈-intro γ∘γ'≼δ∘δ' δ∘δ'≼γ∘γ'
  where γ∘γ'≼δ∘δ' = ≼-intro (∘-congLemma γ≈δ γ'≈δ')
        δ≈γ = ≈Symmetric γ≈δ
        δ'≈γ' = ≈Symmetric γ'≈δ'
        δ∘δ'≼γ∘γ' = ≼-intro (∘-congLemma δ≈γ δ'≈γ')
```

B.0.13 Scwf/DomainScwf/PlainInstance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.PlainInstance where

open import Appmap.Equivalence
open import Base.Core
open import Scwf.Plain
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Empty.Instance
open import Scwf.DomainScwf.Appmap.Identity.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.p.Instance
open import Scwf.DomainScwf.Comprehension.q.Instance
open import Scwf.DomainScwf.PlainAxiomProofs

domScwf : Scwf
Scwf.Ty domScwf        = Ty
Scwf.Ctx domScwf       = Ctx
Scwf.Tm domScwf Γ 𝒜    = tAppmap Γ [ 𝒜 ]
```

```
Scwf.Sub domScwf        = tAppmap
Scwf._≈_ domScwf        = _≈_
Scwf._≋_ domScwf        = _≈_
Scwf.isEquivT domScwf = ≈IsEquiv
Scwf.isEquivS domScwf = ≈IsEquiv
Scwf.⋄ domScwf          = []
Scwf._•_ domScwf Γ 𝒜  = 𝒜 :: Γ
Scwf.q domScwf          = q
Scwf._[_] domScwf       = _∘_
Scwf.id domScwf         = idMap
Scwf._∘_ domScwf        = _∘_
Scwf.⟨⟩ domScwf         = emptyMap
Scwf.⟨_,_⟩ domScwf      = ⟨_,_⟩
Scwf.p domScwf          = p
Scwf.idL domScwf        = idL
Scwf.idR domScwf        = idR
Scwf.subAssoc domScwf = subAssoc
Scwf.idSub domScwf      = idSub
Scwf.compSub domScwf = compSub
Scwf.id₀ domScwf        = id₀
Scwf.<>-zero domScwf  = <>-zero
Scwf.pCons domScwf      = pCons
Scwf.qCons domScwf      = qCons
Scwf.idExt domScwf      = idExt
Scwf.compExt domScwf  = compExt
Scwf.subCong domScwf  = ∘-cong
Scwf.<,>-cong domScwf = <,>-cong
Scwf.∘-cong domScwf     = ∘-cong
```

B.0.14 Scwf/DomainScwf/ProdArrInstance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProdArrInstance where

open import Scwf.DomainScwf.ArrowStructure.ap.Instance
open import Scwf.DomainScwf.ArrowStructure.apCong
open import Scwf.DomainScwf.ArrowStructure.apSub
open import Scwf.DomainScwf.ArrowStructure.beta
open import Scwf.DomainScwf.ArrowStructure.lam.Instance
open import Scwf.DomainScwf.ArrowStructure.lamCong
open import Scwf.DomainScwf.ArrowStructure.lamSub
open import Scwf.DomainScwf.ArrowStructure.NbhSys.DefProof
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.ProdInstance
open import Scwf.ProductArrow
```

```
domProductArrowScwf : ProductArrow-scwf
ProductArrow-scwf.prod-scwf domProductArrowScwf
  = domProdScwf
ProductArrow-scwf._⇒_ domProductArrowScwf
  = ArrNbhSys
ProductArrow-scwf.lam domProductArrowScwf
  = lam
ProductArrow-scwf.ap domProductArrowScwf
  = ap
ProductArrow-scwf.lamSub domProductArrowScwf {𝒜 = 𝒜} {ℬ}
  = lamSub 𝒜 ℬ
ProductArrow-scwf.apSub domProductArrowScwf {𝒜 = 𝒜} {ℬ}
  = apSub 𝒜 ℬ
ProductArrow-scwf.β domProductArrowScwf {𝒜 = 𝒜} {ℬ}
  = β-equal 𝒜 ℬ
ProductArrow-scwf.lamCong domProductArrowScwf {𝒜 = 𝒜} {ℬ}
  = lamCong 𝒜 ℬ
ProductArrow-scwf.apCong domProductArrowScwf {𝒜 = 𝒜} {ℬ}
  = apCong 𝒜 ℬ
```

B.0.15 Scwf/DomainScwf/ProdInstance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProdInstance where

open import Scwf.DomainScwf.PlainInstance
open import Scwf.DomainScwf.ProductStructure.AxiomProofs
open import Scwf.DomainScwf.ProductStructure.fst.Instance
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance
open import Scwf.DomainScwf.ProductStructure.Pair.Instance
open import Scwf.DomainScwf.ProductStructure.snd.Instance
open import Scwf.DomainScwf.ProductStructure.Unit.Mapping.Instance
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Instance
open import Scwf.DomainScwf.ProductStructure.Unit.NSub
open import Scwf.Product

domProdScwf : Prod-scwf
Prod-scwf.scwf domProdScwf                    = domScwf
Prod-scwf._×_ domProdScwf                     = _×_
Prod-scwf.fst domProdScwf                     = fst
Prod-scwf.snd domProdScwf                     = snd
Prod-scwf.<_,_> domProdScwf                   = <_,_>
Prod-scwf.fstAxiom domProdScwf {𝒜 = 𝒜} {ℬ} = fstAxiom 𝒜 ℬ
Prod-scwf.sndAxiom domProdScwf {𝒜 = 𝒜} {ℬ} = sndAxiom 𝒜 ℬ
Prod-scwf.pairSub domProdScwf {𝒜 = 𝒜} {ℬ}  = pairSub 𝒜 ℬ
```

```
Prod-scwf.fstCong domProdScwf {𝒜 = 𝒜} {ℬ}   = fstCong 𝒜 ℬ
Prod-scwf.sndCong domProdScwf {𝒜 = 𝒜} {ℬ} = sndCong 𝒜 ℬ
Prod-scwf.pairCong domProdScwf {𝒜 = 𝒜} {ℬ} = pairCong 𝒜 ℬ
Prod-scwf.𝕟₁ domProdScwf                    = 𝕟₁
Prod-scwf.0₁ domProdScwf                     = 0₁
Prod-scwf.𝕟₁-sub domProdScwf                 = 𝕟₁-sub
```

B.0.16 Scwf/DomainScwf/Appmap/Definition

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Definition where

open import Appmap.Definition public
open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance

-- Some simplifying notation for approximable mappings in
-- our scwf.
tAppmap : (Γ : Ctx m) → (Δ : Ctx n) → Set₁
tAppmap Γ Δ = Appmap (ValNbhSys Γ) (ValNbhSys Δ)
```

B.0.17 Scwf/DomainScwf/Appmap/Composition/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition

module Scwf.DomainScwf.Appmap.Composition.AxiomProofs
  (δ : tAppmap Δ Θ) (γ : tAppmap Γ Δ) where

open import Base.Core
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Relation

∘↦-mono : ∀ {x y z} → ⊑ᵥ Γ x y →
          _∘↦_ δ γ x z → _∘↦_ δ γ y z
∘↦-mono {y = y} {z} x⊑y (∘↦-intro γx↦y δy↦z)
  = ∘↦-intro (Appmap.↦-mono γ x⊑y γx↦y) δy↦z
```

```
∘↦-bottom : ∀ {x} → _∘↦_ δ γ x ⊥ᵥ
∘↦-bottom {x = x}
  = ∘↦-intro (Appmap.↦-bottom γ) (Appmap.↦-bottom δ)

∘↦-↓closed : ∀ {x z w} → ⊑ᵥ Θ z w →
                _∘↦_ δ γ x w → _∘↦_ δ γ x z
∘↦-↓closed {x = x} {z} z⊑w (∘↦-intro γx↦y δy↦w)
  = ∘↦-intro γx↦y (Appmap.↦-↓closed δ z⊑w δy↦w)

∘↦-↑directed : ∀ {x z w} → _∘↦_ δ γ x z → _∘↦_ δ γ x w →
                  (con : ValCon Θ z w) →
                  _∘↦_ δ γ x (z ⊔ᵥ w [ con ])
∘↦-↑directed (∘↦-intro γx↦y δy↦z)
  (∘↦-intro γx↦y' δy'↦w) conzw
  = ∘↦-intro γx↦y⊔y' δy⊔y'↦z⊔w
    where conyy' = Appmap.↦-con γ γx↦y γx↦y' valConRefl
          γx↦y⊔y' = Appmap.↦-↑directed γ γx↦y γx↦y' conyy'
          y⊑y⊔y' = NbhSys.⊑-⊔-fst (ValNbhSys Δ) conyy'
          δy⊔y'↦z = Appmap.↦-mono δ y⊑y⊔y' δy↦z
          y'⊑y⊔y' = NbhSys.⊑-⊔-snd (ValNbhSys Δ) conyy'
          δy⊔y'↦w = Appmap.↦-mono δ y'⊑y⊔y' δy'↦w
          δy⊔y'↦z⊔w = Appmap.↦-↑directed δ δy⊔y'↦z δy⊔y'↦w conzw

∘↦-con : ∀ {x y x' y'} → _∘↦_ δ γ x y → _∘↦_ δ γ x' y' →
           ValCon Γ x x' → ValCon Θ y y'
∘↦-con {y = ⟨⟩} {y' = ⟨⟩} _ _ _ = con-nil
∘↦-con {y = ⟨ y ,, y ⟩} {y' = ⟨ y' ,, y' ⟩}
  (∘↦-intro γx↦z δz↦y) (∘↦-intro γx'↦z' δz'↦y') conxx'
  = Appmap.↦-con δ δz↦y δz'↦y' conzz'
  where conzz' = Appmap.↦-con γ γx↦z γx'↦z' conxx'
```

B.0.18 Scwf/DomainScwf/Appmap/Composition/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Composition.Instance where

open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Composition.AxiomProofs
open import Scwf.DomainScwf.Appmap.Composition.Relation

_∘_ : tAppmap Δ Θ → tAppmap Γ Δ → tAppmap Γ Θ
Appmap._↦_ (δ ∘ γ)         = _∘↦_ δ γ
Appmap.↦-mono (δ ∘ γ)      = ∘↦-mono δ γ
Appmap.↦-bottom (δ ∘ γ)    = ∘↦-bottom δ γ
Appmap.↦-↓closed (δ ∘ γ)   = ∘↦-↓closed δ γ
Appmap.↦-↑directed (δ ∘ γ) = ∘↦-↑directed δ γ
Appmap.↦-con (δ ∘ γ)       = ∘↦-con δ γ
```

B.0.19 Scwf/DomainScwf/Appmap/Composition/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Composition.Relation where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition

data _∘↦_ (δ : tAppmap Δ Θ) (γ : tAppmap Γ Δ) :
            Valuation Γ → Valuation Θ → Set where
  ∘↦-intro : ∀ {x y z} → [ γ ] x ↦ y → [ δ ] y ↦ z →
               _∘↦_ δ γ x z
```

B.0.20 Scwf/DomainScwf/Appmap/Empty/AxiomProofs

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Empty.AxiomProofs where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Empty.Relation
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Relation

empty↦-mono : ∀ {x y z} → ⊑ᵥ Γ x y → x empty↦ z →
                y empty↦ z
empty↦-mono {z = ⟨⟩} _ _ = empty↦-intro

empty↦-↓closed : {x : Valuation Γ} → ∀ {y z} →
                   ⊑ᵥ [] y z → x empty↦ z →
                   x empty↦ y
empty↦-↓closed {y = ⟨⟩} _ _ = empty↦-intro

empty↦-↑directed : {x : Valuation Γ} → ∀ {y z} →
                     x empty↦ y → x empty↦ z → (con : ValCon [] y z) →
                     x empty↦ (y ⊔ᵥ z [ con ])
empty↦-↑directed {y = ⟨⟩} {⟨⟩} _ _ _ = empty↦-intro

empty↦-con : {x : Valuation Γ} → ∀ {y x' y'} →
               x empty↦ y → x' empty↦ y' → ValCon Γ x x' →
               ValCon [] y y'
empty↦-con {y = ⟨⟩} {y' = ⟨⟩} x x₁ x₂ = con-nil
```

B.0.21 Scwf/DomainScwf/Appmap/Empty/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Empty.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Empty.AxiomProofs
open import Scwf.DomainScwf.Appmap.Empty.Relation
open import Scwf.DomainScwf.Appmap.Definition

emptyMap : tAppmap Γ []
Appmap._↦_ (emptyMap)          = _empty↦_
Appmap.↦-mono (emptyMap)       = empty↦-mono
Appmap.↦-bottom (emptyMap)     = empty↦-intro
Appmap.↦-↓closed (emptyMap)    = empty↦-↓closed
Appmap.↦-↑directed (emptyMap)  = empty↦-↑directed
Appmap.↦-con (emptyMap)        = empty↦-con
```

B.0.22 Scwf/DomainScwf/Appmap/Empty/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Empty.Relation where

open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Base.Core
open import Base.Variables

data _empty↦_ : Valuation Γ → Valuation [] → Set where
  empty↦-intro : {x : Valuation Γ} → x empty↦ ⟨⟨⟩⟩
```

B.0.23 Scwf/DomainScwf/Appmap/Identity/AxiomProofs

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Identity.AxiomProofs where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Valuation.AxiomProofs
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.Appmap.Identity.Relation
```

```
private
  variable
    x y z w : Valuation Γ

id↦-mono : ⊑ᵥ Γ x y → x id↦ z → y id↦ z
id↦-mono {Γ = Γ} {y = y} {z} x⊑y (id↦-intro z⊑x)
  = id↦-intro (NbhSys.⊑-trans (ValNbhSys _) z⊑x x⊑y)

id↦-bottom : x id↦ ⊥ᵥ
id↦-bottom {x = x}
  = id↦-intro (NbhSys.⊑-⊥ (ValNbhSys _))

id↦-↓closed : ⊑ᵥ Γ y z → x id↦ z → x id↦ y
id↦-↓closed {y = y} {x = x} y⊑z (id↦-intro z⊑x)
  = id↦-intro (NbhSys.⊑-trans (ValNbhSys _) y⊑z z⊑x)

id↦-↑directed : x id↦ y → x id↦ z → (con : ValCon Γ y z) →
                x id↦ (y ⊔ᵥ z [ con ])
id↦-↑directed {x = x} {y} {z} (id↦-intro y⊑x)
  (id↦-intro z⊑x) con
  = id↦-intro (NbhSys.⊑-⊔ (ValNbhSys _) y⊑x z⊑x con)

id↦-con : x id↦ z → y id↦ w → ValCon Γ x y → ValCon Γ z w
id↦-con (id↦-intro z⊑x) (id↦-intro w⊑y) con
  = Con-⊔ᵥ z⊑x⊔y w⊑x⊔y
  where z⊑x⊔y = ⊑ᵥ-trans z⊑x (⊑ᵥ-⊔-fst con)
        w⊑x⊔y = ⊑ᵥ-trans w⊑y (⊑ᵥ-⊔-snd con)
```

B.0.24 Scwf/DomainScwf/Appmap/Identity/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Identity.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Identity.AxiomProofs
open import Scwf.DomainScwf.Appmap.Identity.Relation

idMap : (Γ : Ctx n) → tAppmap Γ Γ
Appmap._↦_ (idMap Γ)         = _id↦_
Appmap.↦-mono (idMap Γ)      = id↦-mono
Appmap.↦-bottom (idMap Γ)    = id↦-bottom
Appmap.↦-↓closed (idMap Γ)   = id↦-↓closed
Appmap.↦-↑directed (idMap Γ) = id↦-↑directed
Appmap.↦-con (idMap Γ)       = id↦-con
```

B.0.25 Scwf/DomainScwf/Appmap/Identity/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Identity.Relation where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Relation

data _id↦_ : Valuation Γ → Valuation Γ → Set where
  id↦-intro : ∀ {x y} → ⊑ᵥ Γ y x → x id↦ y
```

B.0.26 Scwf/DomainScwf/Appmap/Valuation/AxiomProofs

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Valuation.AxiomProofs where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Relation

Con-⊔ᵥ : ∀ {x y z} → ⊑ᵥ Γ x z → ⊑ᵥ Γ y z → ValCon Γ x y
Con-⊔ᵥ ⊑ᵥ-nil ⊑ᵥ-nil = con-nil
Con-⊔ᵥ {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩} {⟨⟨ y ,, y ⟩⟩}
  (⊑ᵥ-cons _ x⊑z x⊑z) (⊑ᵥ-cons _ y⊑z y⊑z)
  = con-tup conxy conxy
  where conxy = NbhSys.Con-⊔ 𝒜 x⊑z y⊑z
        conxy = Con-⊔ᵥ x⊑z y⊑z

⊑ᵥ-refl : ∀ {x} → ⊑ᵥ Γ x x
⊑ᵥ-refl {x = ⟨⟩}= ⊑ᵥ-nil
⊑ᵥ-refl {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩}
  = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-refl 𝒜) ⊑ᵥ-refl

⊑ᵥ-trans : ∀ {x y z} → ⊑ᵥ Γ x y → ⊑ᵥ Γ y z → ⊑ᵥ Γ x z
⊑ᵥ-trans {x = ⟨⟩} {⟨⟩} {⟨⟩} _ _ = ⊑ᵥ-nil
⊑ᵥ-trans {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩} {z = ⟨⟨ z ,, z ⟩⟩}
  (⊑ᵥ-cons _ x⊑y x⊑y) (⊑ᵥ-cons _ y⊑z y⊑z)
  = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-trans 𝒜 x⊑y y⊑z)
    (⊑ᵥ-trans x⊑y y⊑z)

⊑ᵥ-⊥ : ∀ {x} → ⊑ᵥ Γ ⊥ᵥ x
⊑ᵥ-⊥ {x = ⟨⟩} = ⊑ᵥ-nil
⊑ᵥ-⊥ {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩}
```

```
  = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-⊥ 𝒜) (⊑ᵥ-⊥)

⊑ᵥ-⊔ : ∀ {x y z} → ⊑ᵥ Γ y x → ⊑ᵥ Γ z x → (con : ValCon Γ y z) →
       ⊑ᵥ Γ (y ⊔ᵥ z [ con ]) x
⊑ᵥ-⊔ {x = ⟨⟩} {⟨⟩} {⟨⟩} _ _ _ = ⊑ᵥ-nil
⊑ᵥ-⊔ {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩} {⟨⟨ y ,, y ⟩⟩} {⟨⟨ z ,, z ⟩⟩}
  (⊑ᵥ-cons _ y⊑x y⊑x) (⊑ᵥ-cons _ z⊑x z⊑x)
  (con-tup conyz conyz) =
  ⊑ᵥ-cons (𝒜 :: Γ) y⊔z⊑x (⊑ᵥ-⊔ y⊑x z⊑x conyz)
  where y⊔z⊑x = NbhSys.⊑-⊔ 𝒜 y⊑x z⊑x conyz

⊑ᵥ-⊔-fst : ∀ {x y} → (con : ValCon Γ x y) → ⊑ᵥ Γ x (x ⊔ᵥ y [ con ])
⊑ᵥ-⊔-fst {x = ⟨⟩} {⟨⟩} _ = ⊑ᵥ-nil
⊑ᵥ-⊔-fst {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩} {⟨⟨ y ,, y ⟩⟩}
  (con-tup conxy conxy)
  = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-⊔-fst 𝒜 conxy) (⊑ᵥ-⊔-fst conxy)

⊑ᵥ-⊔-snd : ∀ {x y} → (con : ValCon Γ x y) → ⊑ᵥ Γ y (x ⊔ᵥ y [ con ])
⊑ᵥ-⊔-snd {x = ⟨⟩} {⟨⟩} _ = ⊑ᵥ-nil
⊑ᵥ-⊔-snd {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩} {⟨⟨ y ,, y ⟩⟩}
  (con-tup conxy conxy)
  = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-⊔-snd 𝒜 conxy) (⊑ᵥ-⊔-snd conxy)
```

B.0.27 Scwf/DomainScwf/Appmap/Valuation/Definition

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Valuation.Definition where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition

-- Valuations of contexts are tuples of appropriately
-- typed neighborhoods.
data Valuation : Ctx n → Set where
  ⟨⟩ : Valuation []
  ⟨_,,_⟩ : NbhSys.Nbh 𝒜 → Valuation Γ → Valuation (𝒜 :: Γ)

-- Notation for 1-tuples.
⟨⟨_⟩⟩ : ∀ x → Valuation (𝒜 :: [])
⟨⟨ x ⟩⟩ = ⟨⟨ x ,, ⟨⟩ ⟩⟩

data ValCon : (Γ : Ctx n) → (x y : Valuation Γ) → Set where
  con-nil : ValCon [] ⟨⟩ ⟨⟩
  con-tup : ∀ {Γ : Ctx n} → ∀ {x y x y} →
            NbhSys.Con 𝒜 x y → ValCon Γ x y →
            ValCon (𝒜 :: Γ) ⟨⟨ x ,, x ⟩⟩ ⟨⟨ y ,, y ⟩⟩
```

```
-- The supremum of valuations is defined component-wise.
_⊔ᵥ_[_] : (x : Valuation Γ) → (y : Valuation Γ) → ValCon Γ x y →
          Valuation Γ
_⊔ᵥ_[_] ⟨⟨⟩⟩ ⟨⟨⟩⟩ _ = ⟨⟨⟩⟩
_⊔ᵥ_[_] {Γ = h :: _} ⟨⟨ x ,, x ⟩⟩ ⟨⟨ y ,, y ⟩⟩ (con-tup conxy conxy)
  = ⟨⟨ [ h ] x ⊔ y [ conxy ] ,, x ⊔ᵥ y [ conxy ] ⟩⟩

⊥ᵥ : Valuation Γ
⊥ᵥ {Γ = []} = ⟨⟨⟩⟩
⊥ᵥ {Γ = h :: _} = ⟨⟨ NbhSys.⊥ h ,, ⊥ᵥ ⟩⟩

-- Analogous to head, but for valuations.
ctHead : Valuation Γ → NbhSys.Nbh (head Γ)
ctHead ⟨⟨ x ,, _ ⟩⟩ = x

-- Analogous to tail for lists.
ctTail : Valuation Γ → Valuation (tail Γ)
ctTail ⟨⟨ _ ,, x ⟩⟩ = x

toValCon : ∀ {D x y} → (conxy : NbhSys.Con D x y) →
           ValCon [ D ] ⟨⟨ x ⟩⟩ ⟨⟨ y ⟩⟩
toValCon conxy = con-tup conxy con-nil

fromValCon : ∀ {D x y} → (conxy : ValCon [ D ] ⟨⟨ x ⟩⟩ ⟨⟨ y ⟩⟩) →
             NbhSys.Con D x y
fromValCon (con-tup conxy _) = conxy
```

B.0.28 Scwf/DomainScwf/Appmap/Valuation/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Valuation.Instance where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Valuation.AxiomProofs
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Relation

ValNbhSys : (Γ : Ctx n) → NbhSys
NbhSys.Nbh (ValNbhSys Γ)     = Valuation Γ
NbhSys._⊑_ (ValNbhSys Γ)     = ⊑ᵥ Γ
NbhSys.Con (ValNbhSys Γ)     = ValCon Γ
NbhSys._⊔_[_] (ValNbhSys Γ) = _⊔ᵥ_[_]
NbhSys.⊥ (ValNbhSys Γ)       = ⊥ᵥ
NbhSys.Con-⊔ (ValNbhSys Γ)  = Con-⊔ᵥ
```

```
NbhSys.⊑-refl (ValNbhSys Γ)  = ⊑ᵥ-refl
NbhSys.⊑-trans (ValNbhSys Γ) = ⊑ᵥ-trans
NbhSys.⊑-⊥ (ValNbhSys Γ)     = ⊑ᵥ-⊥
NbhSys.⊑-⊔ (ValNbhSys Γ)     = ⊑ᵥ-⊔
NbhSys.⊑-⊔-fst (ValNbhSys Γ) = ⊑ᵥ-⊔-fst
NbhSys.⊑-⊔-snd (ValNbhSys Γ) = ⊑ᵥ-⊔-snd
```

B.0.29 Scwf/DomainScwf/Appmap/Valuation/Lemmata

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Valuation.Lemmata where

open import Base.Core
open import Base.Variables
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition

valConRefl : ∀ {x} → ValCon Γ x x
valConRefl {x = ⟨⟩} = con-nil
valConRefl {Γ = 𝒜 :: Γ} {x = ⟨⟨ x ,, x ⟩⟩}
  = con-tup (conRefl 𝒜) valConRefl
```

B.0.30 Scwf/DomainScwf/Appmap/Valuation/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Appmap.Valuation.Relation where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition

open import Agda.Builtin.Nat

data ⊑ᵥ : (Γ : Ctx n) → (x y : Valuation Γ) →
          Set where
  ⊑ᵥ-nil : ⊑ᵥ [] ⟨⟩ ⟨⟩
  ⊑ᵥ-cons : (Γ : Ctx (suc n)) → ∀ {x y} →
              [ head Γ ] (ctHead x) ⊑ (ctHead y) →
              ⊑ᵥ (tail Γ) (ctTail x) (ctTail y) →
              ⊑ᵥ Γ x y
```

B.0.31 Scwf/DomainScwf/ArrowStructure/apCong

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.apCong (𝒜 ℬ : Ty) where

open import Appmap.Definition
open import Appmap.Equivalence
open import Base.Variables hiding (𝒜 ; ℬ)
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.ArrowStructure.ap.Instance
open import Scwf.DomainScwf.ArrowStructure.ap.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒜 ℬ

private
  variable
    t t' : tAppmap Γ [ ArrNbhSys 𝒜 ℬ ]
    u u' : tAppmap Γ [ 𝒜 ]

apCongLemma : t ≼ t' → u ≼ u' → ∀ {x y} →
                [ ap t u ] x ↦ y → [ ap t' u' ] x ↦ y
apCongLemma (≼-intro t≼t') (≼-intro u≼u') (ap↦-intro₁ p)
  = ap↦-intro₁ p
apCongLemma (≼-intro t≼t') (≼-intro u≼u')
  (ap↦-intro₂ _ _ tx↦f ux↦z zy⊑f)
  = ap↦-intro₂ _ _ (t≼t' tx↦f) (u≼u' ux↦z) zy⊑f

apCong : t ≈ t' → u ≈ u' →
         ap t u ≈ ap t' u'
apCong (≈-intro t≼t' t'≼t) (≈-intro u≼u' u'≼u)
  = ≈-intro (≼-intro (apCongLemma t≼t' u≼u'))
            (≼-intro (apCongLemma t'≼t u'≼u))
```

B.0.32 Scwf/DomainScwf/ArrowStructure/apSub

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.apSub (𝒜 ℬ : Ty) where

open import Appmap.Equivalence
open import Base.FinFun
open import Base.Variables hiding (𝒜 ; ℬ)
```

```
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Appmap.Identity.Relation
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.ap.Instance
open import Scwf.DomainScwf.ArrowStructure.ap.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Relation

private
  variable
    γ : tAppmap Δ Γ
    t : tAppmap Γ [ ArrNbhSys 𝒜 ℬ ]
    u : tAppmap Γ [ 𝒜 ]

apSubLemma₁ : ∀ {x y} → [ ap (t ∘ γ) (u ∘ γ) ] x ↦ y →
                        [ ap t u ∘ γ ] x ↦ y
apSubLemma₁ {t = t} {γ = γ} {u} (ap↦-intro₁ p)
  = Appmap.↦-↓closed (ap t u ∘ γ) tupy⊑⊥ aptu∘γ↦⊥
  where tupy⊑⊥ = ⊑ᵥ-cons [ ℬ ] p ⊑ᵥ-nil
        aptu∘γ↦⊥ = Appmap.↦-bottom (ap t u ∘ γ)
apSubLemma₁ {t = t} {γ = γ} {u}
  (ap↦-intro₂ _ _ (∘↦-intro γx↦y ty↦f)
  (∘↦-intro γx↦z uz↦x) xy⊑f)
  = ∘↦-intro γx↦y⊔z aptuyz↦y
  where conyz = Appmap.↦-con γ γx↦y γx↦z valConRefl
        γx↦y⊔z = Appmap.↦-↑directed γ γx↦y γx↦z conyz
        tyz↦f = Appmap.↦-mono t (NbhSys.⊑-⊔-fst (ValNbhSys _) _) ty↦f
        uyz↦x = Appmap.↦-mono u (NbhSys.⊑-⊔-snd (ValNbhSys _) _) uz↦x
        aptuyz↦y = ap↦-intro₂ _ _ tyz↦f uyz↦x xy⊑f

apSubLemma₂ : ∀ {x y} → [ ap t u ∘ γ ] x ↦ y →
                        [ ap (t ∘ γ) (u ∘ γ) ] x ↦ y
apSubLemma₂ {t = t} {u} {γ = γ} (∘↦-intro γx↦z (ap↦-intro₁ p))
  = Appmap.↦-↓closed (ap (t ∘ γ) (u ∘ γ)) tupy⊑⊥ apt∘γu∘γ↦⊥
  where tupy⊑⊥ = ⊑ᵥ-cons [ ℬ ] p ⊑ᵥ-nil
        apt∘γu∘γ↦⊥ = Appmap.↦-bottom (ap (t ∘ γ) (u ∘ γ))
apSubLemma₂ (∘↦-intro γx↦z
  (ap↦-intro₂ _ _ tz↦f uz↦x xy⊑f))
```

```
  = ap↦-intro₂ _ _ t∘γx↦f u∘γx↦x xy⊑f
  where t∘γx↦f = ∘↦-intro γx↦z tz↦f
        u∘γx↦x = ∘↦-intro γx↦z uz↦x

apSub : {Γ : Ctx n} → (γ : tAppmap Δ Γ) → ∀ t u →
        (ap (t ∘ γ) (u ∘ γ)) ≈ ((ap t u) ∘ γ)
apSub γ t u = ≈-intro (≼-intro apSubLemma₁)
                      (≼-intro apSubLemma₂)
```

B.0.33 Scwf/DomainScwf/ArrowStructure/beta

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.beta (𝒜 ℬ : Ty) where

open import Appmap.Equivalence
open import Base.FinFun
open import Base.Variables hiding (𝒜 ; ℬ)
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Identity.Instance
open import Scwf.DomainScwf.Appmap.Identity.Relation
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.ap.Instance
open import Scwf.DomainScwf.ArrowStructure.ap.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.lam.Instance
open import Scwf.DomainScwf.ArrowStructure.lam.Lemmata 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.lam.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Relation

β-lemma₁ : {t : tAppmap Γ [ 𝒜 ]} →
           {u : tAppmap (𝒜 :: Γ) [ ℬ ]} → ∀ {x y} →
           [ ap (lam u) t ] x ↦ y →
           [ u ∘ ⟨ idMap Γ , t ⟩ ] x ↦ y
β-lemma₁ {Γ = Γ} {t} {u} {x} {⟨⟨ y ,, ⟨⟩⟩ ⟩⟩} (ap↦-intro₁ p)
```

```
    = ∘↦-intro ⟨⟩x↦⊥ u⊥↦y
    where idx↦⊥ = Appmap.↦-bottom (idMap Γ)
          tx↦⊥ = Appmap.↦-bottom t
          ⟨⟩x↦⊥ = ⟨⟩↦-intro {y = ⟨⟨ _ ,, ⊥ᵥ ⟩⟩} idx↦⊥ tx↦⊥
          tupy⊑⊥ = ⊑ᵥ-cons [ ℬ ] p ⊑ᵥ-nil
          u⊥↦⊥ = Appmap.↦-bottom u
          u⊥↦y = Appmap.↦-↓closed u tupy⊑⊥ u⊥↦⊥
  β-lemma₁ (ap↦-intro₂ _ _ _ _ (⊑ₑ-intro₂ _ _ p))
    with (p here)
  β-lemma₁ {Γ = Γ} {u = u}
    (ap↦-intro₂ {x = x} {y} conf _ lamux↦f tx↦x _)
    | record { sub = sub
             ; postablesub = postablesub
             ; preablesub = preablesub
             ; y⊑post = y⊑post
             ; pre⊑x = pre⊑x
             ; sub⊆f = sub⊆f
             }
    = ∘↦-intro (⟨⟩↦-intro {y = ⟨⟨ x ,, _ ⟩⟩} idx↦x tx↦x) uxx↦y
    where idx↦x = id↦-intro (NbhSys.⊑-refl (ValNbhSys _))
          y⊑post' = ⊑ᵥ-cons [ ℬ ] y⊑post ⊑ᵥ-nil
          prex⊑xx = ⊑ᵥ-cons (𝒜 :: Γ) pre⊑x
                    (NbhSys.⊑-refl (ValNbhSys _))
          uprex↦postx = ↓closedLemma u (subsetIsCon conf sub⊆f)
                        preablesub postablesub
                        (shrinkLam {Γ = Γ} u
                        sub⊆f lamux↦f)
          uxx↦post = Appmap.↦-mono u prex⊑xx uprex↦postx
          uxx↦y = Appmap.↦-↓closed u y⊑post' uxx↦post

  β-lemma₂' : {u : tAppmap (𝒜 :: Γ) [ ℬ ]} → ∀ {x x' y'} →
              [ u ] ⟨⟨ x' ,, x ⟩⟩ ↦ ⟨⟨ y' ⟩⟩ →
              ∀ {x y} → (x , y) ∈ ((x' , y') :: ∅) →
              [ u ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
  β-lemma₂' ux'x↦y' here = ux'x↦y'

  β-lemma₂ : {t : tAppmap Γ [ 𝒜 ]} →
             {u : tAppmap (𝒜 :: Γ) [ ℬ ]} →
             ∀ {x y} → [ u ∘ ⟨ idMap Γ , t ⟩ ] x ↦ y →
             [ ap (lam u) t ] x ↦ y
  β-lemma₂ {Γ = Γ} {u = u} {y = ⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩}
    (∘↦-intro (⟨⟩↦-intro (id↦-intro x'⊑x) tx↦x) uxx'↦y)
    = ap↦-intro₂ singletonIsCon singletonIsCon
      lamx↦xy tx↦x xy⊑xy
    where xy⊑xy = NbhSys.⊑-refl (ArrNbhSys 𝒜 ℬ)
          xx'⊑xx = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-refl 𝒜) x'⊑x
          uxx↦y = Appmap.↦-mono u xx'⊑xx uxx'↦y
```

```
    lamx↦xy = lam↦-intro₂ singletonIsCon
                  (β-lemma₂' {u = u} uxx↦y)

β-equal : {t : tAppmap Γ [ 𝒜 ]} →
          {u : tAppmap (𝒜 :: Γ) [ ℬ ]} →
          ap (lam u) t ≈ (u ∘ ⟨ idMap Γ , t ⟩)
β-equal = ≈-intro (≼-intro β-lemma₁) (≼-intro β-lemma₂)
```

B.0.34 Scwf/DomainScwf/ArrowStructure/lamCong

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.lamCong (𝒜 ℬ : Ty) where

open import Appmap.Equivalence
open import Base.Variables hiding (𝒜 ; ℬ)
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ArrowStructure.lam.Instance
open import Scwf.DomainScwf.ArrowStructure.lam.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance

lamCongLemma : {t t' : tAppmap (𝒜 :: Γ) [ ℬ ]} →
               t ≼ t' → ∀ {x y} →
               [ lam t ] x ↦ y → [ lam t' ] x ↦ y
lamCongLemma (≼-intro p₁) lam↦-intro₁
  = lam↦-intro₁
lamCongLemma (≼-intro p₁) (lam↦-intro₂ _ p₂)
  = lam↦-intro₂ _ (λ xy∈f → p₁ (p₂ xy∈f))

lamCong : {t t' : tAppmap (𝒜 :: Γ) [ ℬ ]} → t ≈ t' →
          lam t ≈ lam t'
lamCong (≈-intro t≼t' t'≼t)
  = ≈-intro (≼-intro (lamCongLemma t≼t'))
            (≼-intro (lamCongLemma t'≼t))
```

B.0.35 Scwf/DomainScwf/ArrowStructure/lamSub

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.lamSub (𝒜 ℬ : Ty) where
```

```
open import Appmap.Equivalence
open import Base.FinFun
open import Base.Variables hiding (𝒜 ; ℬ)
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.lam.Instance
open import Scwf.DomainScwf.ArrowStructure.lam.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Relation
open import Scwf.DomainScwf.Comprehension.p.Instance
open import Scwf.DomainScwf.Comprehension.p.Relation
open import Scwf.DomainScwf.Comprehension.q.Instance
open import Scwf.DomainScwf.Comprehension.q.Relation

private
  variable
    γ : tAppmap Δ Γ
    t : tAppmap (𝒜 :: Γ) [ ℬ ]

lamSubLemma₁' : ∀ {x f} → ∀ {conf} →
                  [ lam t ∘ γ ] x ↦ ⟨⟨ F f conf ⟩⟩ →
                  ∀ {x y} → (x , y) ∈ f →
                  [ t ∘ ⟨ γ ∘ (p Δ 𝒜) , q Δ 𝒜 ⟩ ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
lamSubLemma₁' (∘↦-intro γx↦y (lam↦-intro₂ f' p)) xy∈f
  = ∘↦-intro γ∘pq↦ (p xy∈f)
  where q↦ = q↦-intro (NbhSys.⊑-refl 𝒜)
        p↦x = p↦-intro (NbhSys.⊑-refl (ValNbhSys _))
        γ∘p↦ = ∘↦-intro p↦x γx↦y
        γ∘pq↦ = ⟨⟩↦-intro γ∘p↦ q↦

lamSubLemma₁ : ∀ {x y} → [ lam t ∘ γ ] x ↦ y →
                 [ lam (t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩) ] x ↦ y
lamSubLemma₁ {t = t} {Δ = Δ} {γ = γ} {y = ⟨⟨ ⊥e ,, ⟨⟨⟩⟩ ⟩⟩}
  (∘↦-intro γx↦f' lamf'↦f)
  = Appmap.↦-bottom (lam (t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩))
```

```
lamSubLemma₁ {y = ⟨⟨ F f conf ,, ⟨⟨⟩⟩ ⟩⟩} (∘↦-intro γx↦f' lamf'↦f)
  = lam↦-intro₂ _ (lamSubLemma₁' lamx↦f)
  where lamx↦f = ∘↦-intro γx↦f' lamf'↦f

-- From a proof that t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩ maps
-- x to ⟨⟨ F f ⟩⟩, we can find a valuation y such that
-- γ maps x to y, and t maps ⟨⟨ x , y ⟩⟩ to ⟨⟨ y ⟩⟩ for any
-- (x , y) ∈ f.
record P-Struct (γ : tAppmap Δ Γ) (t : tAppmap (𝒜 :: Γ) [ ℬ ])
                (x : Valuation Δ) (f : NbhFinFun 𝒜 ℬ) :
                Set where
  field
    y : Valuation Γ
    γx↦y : [ γ ] x ↦ y
    λty : ∀ {x y} → (x , y) ∈ f → [ t ] ⟨⟨ x ,, y ⟩⟩ ↦ ⟨⟨ y ⟩⟩

getP-Struct' : {γ : tAppmap Δ Γ} →
               ∀ x x y y z → (f : NbhFinFun 𝒜 ℬ) →
               ∀ {conyz conxyf} →
               [ t ∘ ⟨ γ ∘ p Δ 𝒜 , q Δ 𝒜 ⟩ ] x lam↦
               ⟨⟨ F ((x , y) :: f) conxyf ⟩⟩ →
               [ t ] ⟨⟨ x ,, y ⟩⟩ ↦ ⟨⟨ y ⟩⟩ →
               (∀ {x' y'} → (x' , y') ∈ f →
               [ t ] ⟨⟨ x' ,, z ⟩⟩ ↦ ⟨⟨ y' ⟩⟩) →
               ∀ {x' y'} → (x' , y') ∈ ((x , y) :: f) →
               [ t ] ⟨⟨ x' ,, y ⊔ᵥ z [ conyz ] ⟩⟩ ↦ ⟨⟨ y' ⟩⟩
getP-Struct' {Γ = Γ} {t = t} x x y y z f {conyz} _ txy↦y _ here
  = Appmap.↦-mono t xy⊑x⊔ txy↦y
  where y⊑⊔ = NbhSys.⊑-⊔-fst (ValNbhSys _) conyz
        xy⊑x⊔ = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-refl 𝒜) y⊑⊔
getP-Struct' {Γ = Γ} {t = t} x x y y z f {conyz} _ _ p
  (there x'y'∈f)
  = Appmap.↦-mono t x'r⊑x'⊔ (p x'y'∈f)
  where r⊑⊔ = NbhSys.⊑-⊔-snd (ValNbhSys _) conyz
        x'r⊑x'⊔ = ⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-refl 𝒜) r⊑⊔

getP-Struct : {γ : tAppmap Δ Γ} →
              ∀ x → (f : NbhFinFun 𝒜 ℬ) → ∀ {conf} →
              [ t ∘ ⟨ γ ∘ p Δ 𝒜 , q Δ 𝒜 ⟩ ] x lam↦ ⟨⟨ F f conf ⟩⟩ →
              P-Struct γ t x f
getP-Struct {Γ = Γ} {t = t} {γ = γ} x ∅ _
  = record { y = ⊥ᵥ
           ; γx↦y = Appmap.↦-bottom γ
           ; λty = xy∈∅-abs
           }
getP-Struct x ((x , y) :: f) (lam↦-intro₂ _ p)
  with (p here)
```

```
getP-Struct {Γ = Γ} {t = t} {γ = γ} x ((x , y) :: f)
  {conf = conf} (lam↦-intro₂ _ p)
  | ∘↦-intro {y = ⟨⟨ z ,, z ⟩⟩}
    (⟨⟩↦-intro (∘↦-intro (p↦-intro y⊑x) γy↦z)
    (q↦-intro z⊑x)) tzz↦y
  = record { y = big⊔
           ; γx↦y = Appmap.↦-↑directed γ γx↦z rec-γx↦y conzrecy
           ; λty = getP-Struct' x x y z rec-y f {conxyf = conf}
                   (lam↦-intro₂ _ p)
                   txz↦y rec-λty
           }
  where rec = getP-Struct {t = t} {γ = γ} x f
                {subsetIsCon conf ⊆-lemma₃}
                (lam↦-intro₂ _ λ x'y'∈f →
                p (there x'y'∈f))
        rec-y = P-Struct.y rec
        rec-γx↦y = P-Struct.γx↦y rec
        rec-λty = P-Struct.λty rec
        γx↦z = Appmap.↦-mono γ y⊑x γy↦z
        zz⊑xz = ⊑ᵥ-cons (𝒜 :: Γ) z⊑x
                  (NbhSys.⊑-refl (ValNbhSys _))
        txz↦y = Appmap.↦-mono t zz⊑xz tzz↦y
        conyx = NbhSys.Con-⊔ (ValNbhSys _) y⊑x
                  (NbhSys.⊑-refl (ValNbhSys _))
        conzrecy = Appmap.↦-con γ γy↦z rec-γx↦y conyx
        big⊔ = z ⊔ᵥ rec-y [ conzrecy ]

lamSubLemma₂ : ∀ {x y} →
                 [ t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩ ] x lam↦ y →
                 [ lam t ∘ γ ] x ↦ y
lamSubLemma₂ {t = t} {γ = γ} lam↦-intro₁
  = Appmap.↦-bottom (lam t ∘ γ)
lamSubLemma₂ (lam↦-intro₂ conf p)
  with (getP-Struct _ _ {conf = conf} (lam↦-intro₂ _ p))
lamSubLemma₂ {t = t} {γ = γ} (lam↦-intro₂ _ p)
  | record { y = y
           ; γx↦y = γx↦y
           ; λty = λty
           }
  = ∘↦-intro γx↦y (lam↦-intro₂ _ λty)

lamSub : ∀ {Γ : Ctx n} → (γ : tAppmap Δ Γ) → ∀ t →
           (lam t ∘ γ) ≈ lam (t ∘ ⟨ (γ ∘ p Δ 𝒜) , q Δ 𝒜 ⟩)
lamSub γ t = ≈-intro (≼-intro lamSubLemma₁)
               (≼-intro lamSubLemma₂)
```

B.0.36 Scwf/DomainScwf/ArrowStructure/Variables

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.Variables (𝒜 ℬ : Ty) where

open import Base.FinFun

variable
  f f' f'' f''' : NbhFinFun 𝒜 ℬ
```

B.0.37 Scwf/DomainScwf/ArrowStructure/ap/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables using (n)
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.Appmap.Definition

module Scwf.DomainScwf.ArrowStructure.ap.AxiomProofs
  {Γ : Ctx n}
  {𝒜 ℬ : Ty}
  (t : tAppmap Γ [ ArrNbhSys 𝒜 ℬ ])
  (u : tAppmap Γ [ 𝒜 ])
  where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.ap.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ

ap↦-mono : ∀ {x y z} → ⊑ᵥ Γ x y →
                 [ t , u ] x ap↦ z → [ t , u ] y ap↦ z
ap↦-mono _ (ap↦-intro₁ p) = ap↦-intro₁ p
ap↦-mono {x} {y} x⊑y (ap↦-intro₂ _ _ tx↦f ux↦x xy⊑f)
  = ap↦-intro₂ _ _ ty↦f uy↦x xy⊑f
```

```
    where ty↦f = Appmap.↦-mono t x⊑y tx↦f
          uy↦x = Appmap.↦-mono u x⊑y ux↦x
ap↦-bottom : ∀ {x} → [ t , u ] x ap↦ ⟨⟨ NbhSys.⊥ 𝒜 ,, ⟨⟨⟩⟩ ⟩⟩
ap↦-bottom = ap↦-intro₁ (NbhSys.⊑-refl 𝒜)

ap↦-↓closed' : ∀ {f x y y'} → ∀ conxy conf → [ 𝒜 ] y' ⊑ y →
                 [ ArrNbhSys 𝒜 ℬ ] F ((x , y) :: ∅) conxy ⊑ F f conf →
                 ∀ {x'' y''} → (x'' , y'') ∈ ((x , y') :: ∅) →
                 ⊑ₑ-proof f conf x'' y''
ap↦-↓closed' conxy conf y'⊑y (⊑ₑ-intro₂ _ _ p) here
  = record { sub = sub
           ; y⊑post = NbhSys.⊑-trans ℬ y'⊑y y⊑post
           ; pre⊑x = pre⊑x
           ; sub⊆f = sub⊆f
           }
  where paxy = p here
        sub = ⊑ₑ-proof.sub paxy
        pre⊑x = ⊑ₑ-proof.pre⊑x paxy
        y⊑post = ⊑ₑ-proof.y⊑post paxy
        sub⊆f = ⊑ₑ-proof.sub⊆f paxy

ap↦-↓closed : ∀ {x y z} → ⊑ᵥ [ ℬ ] y z →
                [ t , u ] x ap↦ z → [ t , u ] x ap↦ y
ap↦-↓closed {y = ⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ y⊑y' ⊑ᵥ-nil) (ap↦-intro₁ y'⊑⊥)
  = ap↦-intro₁ (NbhSys.⊑-trans ℬ y⊑y' y'⊑⊥)
ap↦-↓closed {y = ⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ y⊑y' ⊑ᵥ-nil)
  (ap↦-intro₂ _ _ tx↦f ux↦x' x'y'⊑f')
  = ap↦-intro₂ _ _ tx↦f ux↦x' x'y⊑f
  where x'y⊑f' = ap↦-↓closed' _ _ y⊑y' x'y'⊑f'
        x'y⊑f = ⊑ₑ-intro₂ singletonIsCon _ x'y⊑f'

ap↦-↑directed''' : ∀ {x y z g cong conxy} → ∀ conyz →
                     [ ArrNbhSys 𝒜 ℬ ] (F ((x , y) :: ∅) conxy) ⊑ (F g cong) →
                     [ ℬ ] z ⊑ NbhSys.⊥ ℬ → ∀ {x' y'} →
                     (x' , y') ∈ ((x , [ ℬ ] y ⊔ z [ conyz ]) :: ∅) →
                     ⊑ₑ-proof g cong x' y'
ap↦-↑directed''' {x = x} {y} _ (⊑ₑ-intro₂ _ _ p) _ here
  with (p here)
ap↦-↑directed''' conyz (⊑ₑ-intro₂ _ _ p) z⊑⊥ here
  | record { sub = sub
           ; y⊑post = y⊑post
           ; pre⊑x = pre⊑x
           ; sub⊆f = sub⊆f
           }
  = record { sub = sub
```

```
                ; y⊑post = NbhSys.⊑-⊔ ℬ y⊑post
                             (NbhSys.⊑-trans ℬ z⊑⊥ (NbhSys.⊑-⊥ ℬ))
                             conyz
                ; pre⊑x = pre⊑x
                ; sub⊆f = sub⊆f
                }

ap↦-↑directed'' : ∀ x y z g → ∀ {cong conxz} → ∀ conyz →
                    [ ArrNbhSys 𝒜 ℬ ] (F ((x , z) :: ∅) conxz) ⊑ (F g cong) →
                    [ ℬ ] y ⊑ NbhSys.⊥ ℬ → ∀ {x' y'} →
                    (x' , y') ∈ ((x , [ ℬ ] y ⊔ z [ conyz ]) :: ∅) →
                    ⊑ₑ-proof g cong x' y'
ap↦-↑directed'' x _ z _ _ (⊑ₑ-intro₂ _ _ p) _ here
  with (p here)
ap↦-↑directed'' x y z _ conyz _ y⊑⊥ here
  | record { sub = sub
           ; y⊑post = y⊑post
           ; pre⊑x = pre⊑x
           ; sub⊆f = sub⊆f
           }
  = record { sub = sub
           ; y⊑post = NbhSys.⊑-⊔ ℬ (NbhSys.⊑-trans ℬ y⊑⊥
                        (NbhSys.⊑-⊥ ℬ)) y⊑post
                        conyz
           ; pre⊑x = pre⊑x
           ; sub⊆f = sub⊆f
           }

ap↦-↑directed' : {f f' : NbhFinFun 𝒜 ℬ} → ∀ {x x' y y' conf conf' con∪} →
                   ∀ conxx' conyy' conxy conx'y' →
                   (F ((x , y) :: ∅) conxy) ⊑ₑ (F f conf) →
                   (F ((x' , y') :: ∅) conx'y') ⊑ₑ (F f' conf') →
                   ∀ {x'' y''} →
                   (x'' , y'') ∈ (([ 𝒜 ] x ⊔ x' [ conxx' ] ,
                       [ ℬ ] y ⊔ y' [ conyy' ]) :: ∅) →
                   ⊑ₑ-proof (f ∪ f') con∪ x'' y''
ap↦-↑directed' {con∪ = cff con∪} conxx' conyy' _ _
  (⊑ₑ-intro₂ _ _ p₁) (⊑ₑ-intro₂ _ _ p₂) here
  = record { sub = p₁sub ∪ p₂sub
           ; y⊑post = NbhSys.⊑-trans ℬ
                        (⊑-⊔-lemma₃ ℬ conyy' conposts p₁y⊑post p₂y⊑post)
                        (postLemma₃ p₁postable p₂postable postable∪ conposts)
           ; pre⊑x = NbhSys.⊑-trans 𝒜
                       (preLemma₃ p₁preable p₂preable preable∪ conpres)
                       (⊑-⊔-lemma₃ 𝒜 conpres conxx' p₁pre⊑x p₂pre⊑x)
           ; sub⊆f = ∪-lemma₅ p₁sub⊆f p₂sub⊆f
           }
```

```
    where p₁xyh = p₁ here
          p₂x'y'h = p₂ here
          p₁sub = ⊑ₑ-proof.sub p₁xyh
          p₂sub = ⊑ₑ-proof.sub p₂x'y'h
          p₁y⊑post = ⊑ₑ-proof.y⊑post p₁xyh
          p₂y⊑post = ⊑ₑ-proof.y⊑post p₂x'y'h
          p₁pre⊑x = ⊑ₑ-proof.pre⊑x p₁xyh
          p₂pre⊑x = ⊑ₑ-proof.pre⊑x p₂x'y'h
          p₁sub⊆f = ⊑ₑ-proof.sub⊆f p₁xyh
          p₂sub⊆f = ⊑ₑ-proof.sub⊆f p₂x'y'h
          p₁postable = ⊑ₑ-proof.postablesub p₁xyh
          p₂postable = ⊑ₑ-proof.postablesub p₂x'y'h
          p₁preable = ⊑ₑ-proof.preablesub p₁xyh
          p₂preable = ⊑ₑ-proof.preablesub p₂x'y'h
          p₁pre⊑x⊔x' = ⊑-⊔-lemma₄ 𝒜 p₁pre⊑x conxx'
          p₂pre⊑x⊔x' = ⊑-⊔-lemma₅ 𝒜 p₂pre⊑x conxx'
          conpres = NbhSys.Con-⊔ 𝒜 p₁pre⊑x⊔x' p₂pre⊑x⊔x'
          preable∪ = preUnionLemma p₁preable p₂preable
                       p₁pre⊑x⊔x' p₂pre⊑x⊔x'
          postable∪ = con∪ (∪-lemma₅ p₁sub⊆f p₂sub⊆f) preable∪
          conposts = NbhSys.Con-⊔ ℬ {z = post (p₁sub ∪ p₂sub) postable∪}
                       (postLemma₁ {postablef = p₁postable} {postable∪})
                       (postLemma₂ {postablef' = p₂postable} {postable∪})

ap↦-↑directed : ∀ {x y z} →
                  [ t , u ] x ap↦ y → [ t , u ] x ap↦ z →
                  (conyz : ValCon _ y z) →
                  [ t , u ] x ap↦ (y ⊔ᵥ z [ conyz ])
ap↦-↑directed (ap↦-intro₁ p₁) (ap↦-intro₁ p₂) (con-tup _ _)
  = ap↦-intro₁ (NbhSys.⊑-⊔ ℬ p₁ p₂ _)

ap↦-↑directed {y = ⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩} {⟨⟨ z ,, ⟨⟨⟩⟩ ⟩⟩} (ap↦-intro₁ p)
  (ap↦-intro₂ cong' conxz tx↦g' ux↦x' x'z⊑g')
  (con-tup _ _)
  = ap↦-intro₂ cong' singletonIsCon tx↦g' ux↦x' x'y⊔z⊑g'
  where x'y⊔z⊑g' = ⊑ₑ-intro₂ _ _
                     (ap↦-↑directed'' _ _ _ _ _ x'z⊑g' p)
ap↦-↑directed {y = ⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩} {⟨⟨ z ,, ⟨⟨⟩⟩ ⟩⟩}
  (ap↦-intro₂ _ _ tx↦g ux↦x xy⊑g) (ap↦-intro₁ p)
  (con-tup _ _)
  = ap↦-intro₂ _ singletonIsCon tx↦g ux↦x xy⊔z⊑g
  where xy⊔z⊑g = ⊑ₑ-intro₂ _ _ (ap↦-↑directed''' _ xy⊑g p)
ap↦-↑directed {y = ⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩} {⟨⟨ z ,, ⟨⟨⟩⟩ ⟩⟩}
  (ap↦-intro₂ _ _ tx↦g ux↦x xy⊑g)
  (ap↦-intro₂ _ _ tx↦g' ux↦x' x'z⊑g')
  (con-tup _ _)
  with (fromValCon (Appmap.↦-con t tx↦g tx↦g' valConRefl))
```

```
... | con-∪ _ _ cong∪g' =
  ap↦-intro₂ cong∪g' singletonIsCon tx↦g∪g' ux↦x⊔x' ⊔⊑∪
  where conxx' = fromValCon (Appmap.↦-con u ux↦x ux↦x' valConRefl)
        tx↦g∪g' = Appmap.↦-↑directed t tx↦g tx↦g'
                    (con-tup (con-∪ _ _ cong∪g') con-nil)
        ux↦x⊔x' = Appmap.↦-↑directed u ux↦x ux↦x'
                    (con-tup conxx' con-nil)
        ⊔⊑∪ = ⊑ₑ-intro₂ _ cong∪g'
                (ap↦-↑directed' conxx' _ _ _ xy⊑g x'z⊑g')
```

B.0.38 Scwf/DomainScwf/ArrowStructure/ap/Consistency

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables using (n)
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.Appmap.Definition

module Scwf.DomainScwf.ArrowStructure.ap.Consistency
  {Γ : Ctx n}
  {𝒜 ℬ : Ty}
  (t : tAppmap Γ [ ArrNbhSys 𝒜 ℬ ])
  (u : tAppmap Γ [ 𝒜 ])
  where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.ap.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ

ap↦-con : ∀ {x y x' y'} → [ t , u ] x ap↦ y →
            [ t , u ] x' ap↦ y' → ValCon _ x x' →
            ValCon _ y y'
ap↦-con {y' = ⟨⟨ y' ,, ⟨⟨⟩⟩ ⟩⟩} (ap↦-intro₁ y⊑⊥) apx'↦y' _
  = NbhSys.Con-⊔ (ValNbhSys [ ℬ ]) y⊑y' y'⊑y'
  where y'⊑y' = NbhSys.⊑-refl (ValNbhSys _)
        y⊑y' = NbhSys.⊑-trans ℬ y⊑⊥ (NbhSys.⊑-⊥ ℬ)
        y⊑y' = ⊑ᵥ-cons _ y⊑y' ⊑ᵥ-nil
```

```
ap↦-con (ap↦-intro₂ _ _ _ _ _) (ap↦-intro₁ y'⊑⊥) _
  = NbhSys.Con-⊔ (ValNbhSys [ ℬ ]) y⊑y y'⊑y
  where y⊑y = NbhSys.⊑-refl (ValNbhSys _)
        y'⊑y = NbhSys.⊑-trans ℬ y'⊑⊥ (NbhSys.⊑-⊥ ℬ)
        y'⊑y = ⊑ᵥ-cons _ y'⊑y ⊑ᵥ-nil
ap↦-con
  (ap↦-intro₂ {x} {y} conf conxy tx↦f ux↦x
  (⊑ₑ-intro₂ _ _ p₁))
  (ap↦-intro₂ {x'} {y'} conf' conx'y' tx'↦f' ux'↦x'
  (⊑ₑ-intro₂ _ _ p₂))
  conxx'
  with (fromValCon (Appmap.↦-con t tx↦f tx'↦f' conxx'))
... | con-∪ _ _ (cff p) = toValCon conyy'
  where p₁proof = p₁ here
        p₂proof = p₂ here
        p₁sub = ⊑ₑ-proof.sub p₁proof
        p₂sub = ⊑ₑ-proof.sub p₂proof
        p₁sub⊆f = ⊑ₑ-proof.sub⊆f p₁proof
        p₂sub⊆f = ⊑ₑ-proof.sub⊆f p₂proof
        p₁y⊑post = ⊑ₑ-proof.y⊑post p₁proof
        p₂y⊑post = ⊑ₑ-proof.y⊑post p₂proof
        p₁pre⊑x = ⊑ₑ-proof.pre⊑x p₁proof
        p₂pre⊑x = ⊑ₑ-proof.pre⊑x p₂proof
        p₁postable = ⊑ₑ-proof.postablesub p₁proof
        p₂postable = ⊑ₑ-proof.postablesub p₂proof
        p₁preable = ⊑ₑ-proof.preablesub p₁proof
        p₂preable = ⊑ₑ-proof.preablesub p₂proof
        conxx' = fromValCon (Appmap.↦-con u ux↦x ux'↦x' conxx')
        p₁pre⊑x⊔x' = ⊑-⊔-lemma₄ 𝒜 p₁pre⊑x conxx'
        p₂pre⊑x⊔x' = ⊑-⊔-lemma₅ 𝒜 p₂pre⊑x conxx'
        preable∪ = preUnionLemma p₁preable p₂preable
                     p₁pre⊑x⊔x' p₂pre⊑x⊔x'
        postable∪ = p (∪-lemma₅ p₁sub⊆f p₂sub⊆f) preable∪
        y⊑post∪ = NbhSys.⊑-trans ℬ p₁y⊑post
                    (postLemma₁ {postablef = p₁postable})
        y'⊑post∪ = NbhSys.⊑-trans ℬ p₂y⊑post
                     (postLemma₂ {postablef' = p₂postable}
                     {postable∪})
        conyy' = NbhSys.Con-⊔ ℬ y⊑post∪ y'⊑post∪
```

B.0.39 Scwf/DomainScwf/ArrowStructure/ap/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ArrowStructure.ap.Instance where
```

```
open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ArrowStructure.ap.AxiomProofs
open import Scwf.DomainScwf.ArrowStructure.ap.Consistency
open import Scwf.DomainScwf.ArrowStructure.ap.Relation
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance

ap : tAppmap Γ [ ArrNbhSys 𝒜 ℬ ] → tAppmap Γ [ 𝒜 ] →
     tAppmap Γ [ ℬ ]
Appmap._↦_ (ap {𝒜 = 𝒜} {ℬ} t u) = [_,_]_ap↦_ 𝒜 ℬ t u
Appmap.↦-mono (ap t u)      = ap↦-mono t u
Appmap.↦-bottom (ap t u)    = ap↦-bottom t u
Appmap.↦-↓closed (ap t u)   = ap↦-↓closed t u
Appmap.↦-↑directed (ap t u) = ap↦-↑directed t u
Appmap.↦-con (ap t u)       = ap↦-con t u
```

B.0.40 Scwf/DomainScwf/ArrowStructure/ap/Relation

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.ap.Relation (𝒜 ℬ : Ty) where

open import Base.FinFun
open import Base.Variables hiding (𝒜 ; ℬ)
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance

data [_,_]_ap↦_ (t : tAppmap Γ [ ArrNbhSys 𝒜 ℬ ])
                (u : tAppmap Γ [ 𝒜 ]) (x : Valuation Γ) :
                Valuation [ ℬ ] → Set where
  ap↦-intro₁ : ∀ {x} → [ ℬ ] x ⊑ NbhSys.⊥ ℬ →
               [ t , u ] x ap↦ ⟨⟨ x ⟩⟩
  ap↦-intro₂ : ∀ {x y f} conf conxy →
               [ t ] x ↦ ⟨⟨ F f conf ⟩⟩ → [ u ] x ↦ ⟨⟨ x ⟩⟩ →
               [ ArrNbhSys 𝒜 ℬ ] (F ((x , y) :: ∅) conxy) ⊑ (F f conf) →
               [ t , u ] x ap↦ ⟨⟨ y ⟩⟩
```

B.0.41 Scwf/DomainScwf/ArrowStructure/lam/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables using (n)
open import Scwf.DomainScwf.Appmap.Definition

module Scwf.DomainScwf.ArrowStructure.lam.AxiomProofs
    {𝒜 ℬ : Ty}
    {Γ : Ctx n}
    (t : tAppmap (𝒜 :: Γ) [ ℬ ]) where

open import Appmap.Lemmata
open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.lam.Lemmata 𝒜 ℬ t
open import Scwf.DomainScwf.ArrowStructure.lam.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒜 ℬ

lam↦-mono : ∀ {x y z} → ⊑ᵥ Γ x y →
                [ t ] x lam↦ z → [ t ] y lam↦ z
lam↦-mono _ lam↦-intro₁ = lam↦-intro₁
lam↦-mono {x = x} {y} x⊑y (lam↦-intro₂ _ p)
    = lam↦-intro₂ _ λ xy∈f → Appmap.↦-mono t
      (⊑ᵥ-cons (𝒜 :: Γ) (NbhSys.⊑-refl 𝒜) x⊑y) (p xy∈f)

lam↦-bottom : ∀ {x} → [ t ] x lam↦ ⟨⟨ ⊥ₑ ⟩⟩
lam↦-bottom = lam↦-intro₁

lam↦-↓closed' : ∀ {x f f' conf conf'} →
                    [ ArrNbhSys 𝒜 ℬ ] F f conf ⊑ F f' conf' →
                    [ t ] x lam↦ ⟨⟨ F f' conf' ⟩⟩ → ∀ {x y} →
                    (x , y) ∈ f →
                    [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
lam↦-↓closed' (⊑ₑ-intro₂ _ _ p) _ xy∈f
    with (p xy∈f)
lam↦-↓closed' {x = x} {conf' = conf'} _ tx↦f' xy∈f
```

```
    | record { sub = sub
            ; preablesub = preablesub
            ; postablesub = postablesub
            ; y⊑post = y⊑post
            ; pre⊑x = pre⊑x
            ; sub⊆f = sub⊆f
            }
    = Appmap.↦-↓closed t y⊑post' txx↦post
    where y⊑post' = ⊑ᵥ-cons [ ℬ ] y⊑post ⊑ᵥ-nil
          pre⊑post = ⊑ᵥ-cons (𝒜 :: Γ) pre⊑x
                     (NbhSys.⊑-refl (ValNbhSys _))
          tprex↦post = ↓closedLemma (subsetIsCon conf' sub⊆f)
                       preablesub postablesub
                       (shrinkLam sub⊆f tx↦f')
          txx↦post = Appmap.↦-mono t pre⊑post tprex↦post

lam↦-↓closed : ∀ {x y z} →
               ⊑ᵥ [ ArrNbhSys 𝒜 ℬ ] y z →
               [ t ] x lam↦ z → [ t ] x lam↦ y
lam↦-↓closed {y = ⟨⟨ _ ,, ⟨⟨⟩⟩ ⟩⟩}
    (⊑ᵥ-cons _ ⊑ₑ-intro₁ ⊑ᵥ-nil) lam↦-intro₁
    = lam↦-intro₁
lam↦-↓closed {y = ⟨⟨ ⊥ₑ ,, ⟨⟨⟩⟩ ⟩⟩}
    (⊑ᵥ-cons _ y⊑f' ⊑ᵥ-nil) (lam↦-intro₂ _ p)
    = lam↦-intro₁
lam↦-↓closed {x = x} {⟨⟨ F f _ ,, ⟨⟨⟩⟩ ⟩⟩}
    (⊑ᵥ-cons _ f⊑f' ⊑ᵥ-nil) (lam↦-intro₂ _ p)
    = lam↦-intro₂ _ (lam↦-↓closed' f⊑f' (lam↦-intro₂ _ p))

lam↦-↑directed' : ∀ {f f' x conf conf'} →
                  [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩ →
                  [ t ] x lam↦ ⟨⟨ F f' conf' ⟩⟩ → ∀ {x y} →
                  (x , y) ∈ (f ∪ f') →
                  [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
lam↦-↑directed' {f = f} _ _ xy∈f⊔f'
    with (∪-lemma₂ {f = f} xy∈f⊔f')
lam↦-↑directed' (lam↦-intro₂ _ p) _ _ | inl xy∈f
    = p xy∈f
lam↦-↑directed' _ (lam↦-intro₂ _ p) _ | inr xy∈f'
    = p xy∈f'

lam↦-↑directed : ∀ {x y z} →
                 [ t ] x lam↦ y → [ t ] x lam↦ z →
                 (conyz : ValCon _ y z) →
                 [ t ] x lam↦ (y ⊔ᵥ z [ conyz ])
lam↦-↑directed {x = x} {z = ⟨⟨ z ,, ⟨⟨⟩⟩ ⟩⟩} lam↦-intro₁ tx↦z
  (con-tup con⊥z _)
```

```
  rewrite (⊥⊔x≡x z {con⊥z}) = tx↦z
lam↦-↑directed {x = x} (lam↦-intro₂ conf p) lam↦-intro₁
  (con-tup confz _)
    rewrite (x⊔⊥≡x (F _ conf) {confz}) = lam↦-intro₂ _ p
lam↦-↑directed {x = x} (lam↦-intro₂ _ p₁) (lam↦-intro₂ _ p₂)
  (con-tup (con-∪ conf conf' _) _)
    = lam↦-intro₂ _ txx↦y
    where txx↦y = lam↦-↑directed' (lam↦-intro₂ conf p₁)
                                  (lam↦-intro₂ conf' p₂)
```

B.0.42 Scwf/DomainScwf/ArrowStructure/lam/Consistency

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables using (n)
open import Scwf.DomainScwf.Appmap.Definition

module Scwf.DomainScwf.ArrowStructure.lam.Consistency
  {𝒜 ℬ : Ty}
  {Γ : Ctx n}
  (t : tAppmap (𝒜 :: Γ) [ ℬ ]) where

open import Base.FinFun
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Valuation.AxiomProofs
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.lam.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ

lamPrePost : ∀ {x y f x} →
             ∀ preablef conxpref postablef conypostf →
             [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩ →
             [ t ] ⟨⟨ pre f preablef ,, x ⟩⟩ ↦ ⟨⟨ post f postablef ⟩⟩ →
             [ t ] ⟨⟨ pre ((x , y) :: f)
                         (pre-cons preablef conxpref) ,, x ⟩⟩ ↦
                 ⟨⟨ post ((x , y) :: f)
                         (post-cons postablef conypostf) ⟩⟩
lamPrePost {x} {y} {f} {x}
  preablef conxpref postablef conypostf txx↦y txx↦postf
  = Appmap.↦-↑directed t txy⊔prefx↦y txy⊔prefx↦postf
```

```
      (toValCon conypostf)
  where x𝐱⊑prexyf𝐱 = ⊑ᵥ-cons _ (NbhSys.⊑-⊔-fst 𝒜 conxpref)
                              ⊑ᵥ-refl
        t𝐱⊔pref𝐱↦y = Appmap.↦-mono 𝐭 x𝐱⊑prexyf𝐱 t𝐱𝐱↦y
        pref𝐱⊑prexyf𝐱 = ⊑ᵥ-cons _ (NbhSys.⊑-⊔-snd 𝒜 conxpref)
                                  ⊑ᵥ-refl
        t𝐱⊔pref𝐱↦postf = Appmap.↦-mono 𝐭 pref𝐱⊑prexyf𝐱 t𝐱𝐱↦postf

record ⊑ₑ-proof₄ (f : NbhFinFun 𝒜 ℬ) (preablef : Preable f)
                 (𝐱 : Valuation Γ) : Set where
  field
    postablef : Postable f
    t𝐩re↦post : [ 𝐭 ] ⟨⟨ pre f preablef ,, 𝐱 ⟩⟩ ↦ ⟨⟨ post f postablef ⟩⟩

lam↦-con'' : ∀ {f 𝐱} →
             (∀ {x y} → (x , y) ∈ f → [ 𝐭 ] ⟨⟨ x ,, 𝐱 ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
             (preablef : Preable f) →
             ⊑ₑ-proof₄ f preablef 𝐱
lam↦-con'' _ pre-nil
  = record { postablef = post-nil
           ; t𝐩re↦post = Appmap.↦-bottom 𝐭
           }
lam↦-con'' {f = (x , y) :: f}
  p (pre-cons preablef conxpref)
  = record { postablef = postablexyf
           ; t𝐩re↦post = lamPrePost preablef _ recpostablef _
                                    (p here) rec𝐭pre↦post
           }
  where rec = lam↦-con'' (λ x'y'∈f → p (there x'y'∈f)) preablef
        recpostablef = ⊑ₑ-proof₄.postablef rec
        rec𝐭pre↦post = ⊑ₑ-proof₄.t𝐩re↦post rec
        conypostf = fromValCon (Appmap.↦-con 𝐭
                                (p here) rec𝐭pre↦post
                                (con-tup conxpref valConRefl))
        postablexyf = post-cons recpostablef conypostf

lam↦-con' : ∀ {f f' 𝐱 𝐱' con𝐱𝐱'} →
            (∀ {x y} → (x , y) ∈ f → [ 𝐭 ] ⟨⟨ x ,, 𝐱 ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
            (∀ {x y} → (x , y) ∈ f' → [ 𝐭 ] ⟨⟨ x ,, 𝐱' ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
            ∀ {x y} → (x , y) ∈ (f ∪ f') →
            [ 𝐭 ] ⟨⟨ x ,, 𝐱 ⊔ᵥ 𝐱' [ con𝐱𝐱' ] ⟩⟩ ↦ ⟨⟨ y ⟩⟩
lam↦-con' {f} {con𝐱𝐱' = con𝐱𝐱'} p₁ p₂ xy∈∪
  with (∪-lemma₂ {f = f} xy∈∪)
... | inl xy∈f = Appmap.↦-mono 𝐭 x𝐱⊑x𝐱⊔𝐱' (p₁ xy∈f)
  where 𝐱⊑𝐱⊔𝐱' = NbhSys.⊑-⊔-fst (ValNbhSys Γ) con𝐱𝐱'
        x𝐱⊑x𝐱⊔𝐱' = ⊑ᵥ-cons _ (NbhSys.⊑-refl 𝒜) 𝐱⊑𝐱⊔𝐱'
... | inr xy∈f' = Appmap.↦-mono 𝐭 x𝐱'⊑x𝐱⊔𝐱' (p₂ xy∈f')
```

```
  where x'⊑x⊔x' = NbhSys.⊑-⊔-snd (ValNbhSys Γ) conxx'
        xx'⊑xx⊔x' = ⊑ᵥ-cons _ (NbhSys.⊑-refl 𝒜) x'⊑x⊔x'

from⊑ₑ-proof₄ : ∀ {f f' x x' sub} →
                (∀ {x y} → (x , y) ∈ f → [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
                (∀ {x y} → (x , y) ∈ f' → [ t ] ⟨⟨ x ,, x' ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
                ValCon _ x x' →
                sub ⊆ (f ∪ f') → Preable sub →
                Postable sub
from⊑ₑ-proof₄ p₁ p₂ conxx' sub⊆∪ preablesub
  = ⊑ₑ-proof₄.postablef (lam↦-con'' (λ xy∈sub →
    lam↦-con' {conxx' = conxx'} p₁ p₂ (sub⊆∪ xy∈sub)) preablesub)

lam↦-con : ∀ {x y x' y'} → [ t ] x lam↦ y →
           [ t ] x' lam↦ y' → ValCon _ x x' →
           ValCon _ y y'
lam↦-con lam↦-intro₁ lam↦-intro₁ _
  = toValCon conₑ-⊥₂
lam↦-con lam↦-intro₁ (lam↦-intro₂ _ _) _
  = toValCon conₑ-⊥₂
lam↦-con (lam↦-intro₂ _ _) lam↦-intro₁ _
  = toValCon conₑ-⊥₁
lam↦-con (lam↦-intro₂ conf p₁)
  (lam↦-intro₂ conf' p₂) conxx'
  = con-tup (con-∪ _ _ conf∪f') con-nil
  where conf∪f' = cff (from⊑ₑ-proof₄ p₁ p₂ conxx')
```

B.0.43 Scwf/DomainScwf/ArrowStructure/lam/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ArrowStructure.lam.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ArrowStructure.lam.AxiomProofs
open import Scwf.DomainScwf.ArrowStructure.lam.Consistency
open import Scwf.DomainScwf.ArrowStructure.lam.Relation
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance

lam : tAppmap (𝒜 :: Γ) [ ℬ ] → tAppmap Γ [ ArrNbhSys 𝒜 ℬ ]
Appmap._↦_ (lam {𝒜} {ℬ = ℬ} t) = [_]_lam↦_ 𝒜 ℬ t
Appmap.↦-mono (lam t)      = lam↦-mono t
Appmap.↦-bottom (lam t)    = lam↦-bottom t
Appmap.↦-↓closed (lam t)   = lam↦-↓closed t
Appmap.↦-↑directed (lam t) = lam↦-↑directed t
Appmap.↦-con (lam t)       = lam↦-con t
```

B.0.44 Scwf/DomainScwf/ArrowStructure/lam/Lemmata

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Scwf.DomainScwf.Appmap.Definition

open import Agda.Builtin.Nat

module Scwf.DomainScwf.ArrowStructure.lam.Lemmata
  (𝒜 ℬ : Ty)
  {n : Nat}
  {Γ : Ctx n}
  (t : tAppmap (𝒜 :: Γ) [ ℬ ]) where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ArrowStructure.lam.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒜 ℬ

open import Agda.Builtin.Equality

shrinkLam : ∀ {x conf conf'} → f ⊆ f' →
            [ t ] x lam↦ ⟨⟨ F f' conf' ⟩⟩ →
            [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩
shrinkLam {f = f} f⊆f' (lam↦-intro₂ _ p)
  = lam↦-intro₂ _ (λ xy∈f → p (f⊆f' xy∈f))

-- The first component of any pair in a FinFun f is smaller
-- than pre f.
preBiggest : ∀ {x y f preablef} → (x , y) ∈ f →
             [ 𝒜 ] x ⊑ pre f preablef
preBiggest {preablef = pre-nil} = xy∈∅-abs
preBiggest {preablef = pre-cons preablef conx'pref} here
  = NbhSys.⊑-⊔-fst 𝒜 conx'pref
preBiggest {preablef = pre-cons preablef conx'pref} (there xy∈f)
  with (preBiggest {preablef = preablef} xy∈f)
... | x⊑pref = ⊑-⊔-lemma₅ 𝒜 x⊑pref conx'pref

↓closedLemma' : {x : Valuation Γ} → ∀ conf preablef →
                [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩ →
```

```
                    ∀ x y → (x , y) ∈ f →
                    [ t ] ⟨⟨ pre f preablef ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
↓closedLemma'  {f = (x :: f')} {x = x} _ preable
  (lam↦-intro₂ _ p) x' y' x'y'∈f
  = Appmap.↦-mono t ax⊑pfx (p x'y'∈f)
  where a⊑pf = preBiggest x'y'∈f
        ax⊑pfx = ⊑ᵥ-cons (𝒜 :: Γ) a⊑pf
                   (NbhSys.⊑-refl (ValNbhSys _))

↓closedLemma : {x : Valuation Γ} →
                    ∀ conf preablef postablef →
                    [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩ →
                    [ t ] ⟨⟨ pre f preablef ,, x ⟩⟩ ↦ ⟨⟨ post f postablef ⟩⟩
↓closedLemma {f = ∅} _ _ _ _ = Appmap.↦-bottom t
↓closedLemma {f = ((x , y) :: f')} {x = x}
  conf (pre-cons preablef' conxpref')
  (post-cons postablef' conypostf') lamtx↦f
  = Appmap.↦-↑directed t tpref'↦y tfx↦pf'
    (con-tup _ con-nil)
  where f' = (x , y) :: f'
        tpref'↦y = ↓closedLemma' _ (pre-cons preablef' conxpref')
                     lamtx↦f x y here
        pf'⊑pf = NbhSys.⊑-⊔-snd 𝒜 conxpref'
        pf'x⊑pfx = ⊑ᵥ-cons (𝒜 :: Γ) pf'⊑pf
                     (NbhSys.⊑-refl (ValNbhSys _))
        tpf'x↦pf' = ↓closedLemma (subsetIsCon conf ⊆-lemma₃)
                      preablef' postablef'
                      (shrinkLam (λ y∈f' → there y∈f') lamtx↦f)
        tfx↦pf' = Appmap.↦-mono t pf'x⊑pfx tpf'x↦pf'

⊥⊔x≡x : ∀ x → ∀ {con⊥x} →
        ⊥ₑ ⊔ₑ x [ con⊥x ] ≡ x
⊥⊔x≡x ⊥ₑ = refl
⊥⊔x≡x (F f _) = refl

x⊔⊥≡x : ∀ x → ∀ {conx⊥} →
        x ⊔ₑ ⊥ₑ [ conx⊥ ] ≡ x
x⊔⊥≡x ⊥ₑ = refl
x⊔⊥≡x (F f _ ) = refl
```

B.0.45 Scwf/DomainScwf/ArrowStructure/lam/Relation

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.lam.Relation (𝒜 ℬ : Ty) where
```

```
open import Base.FinFun
open import Base.Variables hiding (𝒜 ; ℬ)
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Instance

data [_]_lam↦_ (t : tAppmap (𝒜 :: Γ) [ ℬ ]) :
               Valuation Γ → Valuation [ ArrNbhSys 𝒜 ℬ ] →
               Set where
  lam↦-intro₁ : ∀ {x} → [ t ] x lam↦ ⟨⟨ ⊥ₑ ⟩⟩
  lam↦-intro₂ : ∀ {x} → {f : NbhFinFun 𝒜 ℬ} →
               (conf : ConFinFun f) →
               (∀ {x y} → (x , y) ∈ f →
               [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
               [ t ] x lam↦ ⟨⟨ F f conf ⟩⟩
```

B.0.46 Scwf/DomainScwf/ArrowStructure/NbhSys/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.AxiomProofs
  (𝒜 ℬ : Ty) where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒜 ℬ

⊑ₑ-refl : ∀ {x} → x ⊑ₑ x
⊑ₑ-refl {⊥ₑ} = ⊑ₑ-intro₁
⊑ₑ-refl {F f conf} = ⊑ₑ-intro₂ conf conf λ {x} {y} xy∈f →
  record
    { sub = (x , y) :: ∅
    ; sub⊆f = ⊆-lemma₄ xy∈f ∅-isSubset
    ; preablesub = singletonIsPreable
    ; postablesub = singletonIsPostable
```

```
      ; y⊑post = ⊑-⊔-lemma₄ 𝒝 (NbhSys.⊑-refl 𝒝) (con⊥₂ 𝒝)
      ; pre⊑x = NbhSys.⊑-⊔ 𝒜 (NbhSys.⊑-refl 𝒜) (NbhSys.⊑-⊥ 𝒜) (con⊥₂ 𝒜)
      }

⊑ₑ-⊥ₑ : ∀ {x} → ⊥ₑ ⊑ₑ x
⊑ₑ-⊥ₑ = ⊑ₑ-intro₁

⊑ₑ-⊔ₑ' : ∀ {f f' f'' conf conf' conf''} →
          F f' conf' ⊑ₑ F f conf → F f'' conf'' ⊑ₑ F f conf →
          ∀ {x y} → (x , y) ∈ (f' ∪ f'') →
          ⊑ₑ-proof f conf x y
⊑ₑ-⊔ₑ' {f' = f'} _ _ xy∈∪
  with (∪-lemma₂ {f = f'} xy∈∪)
⊑ₑ-⊔ₑ' (⊑ₑ-intro₂ _ _ p) _ xy∈∪ | inl xy∈f'
  = p xy∈f'
⊑ₑ-⊔ₑ' _ (⊑ₑ-intro₂ _ _ p) xy∈∪ | inr xy∈f''
  = p xy∈f''

⊑ₑ-⊔ₑ : ∀ {x y z} → y ⊑ₑ x → z ⊑ₑ x → (conyz : ArrCon y z) →
        (y ⊔ₑ z [ conyz ]) ⊑ₑ x
⊑ₑ-⊔ₑ {y = ⊥ₑ} {⊥ₑ} _ _ _ = ⊑ₑ-intro₁
⊑ₑ-⊔ₑ {y = F _ _} {⊥ₑ} y⊑x _ _ = y⊑x
⊑ₑ-⊔ₑ {y = ⊥ₑ} {F _ _} _ z⊑x _ = z⊑x
⊑ₑ-⊔ₑ {x = ArrNbh.F f _} {ArrNbh.F f' _} {ArrNbh.F f'' _} y⊑x z⊑x
  (ArrCon.con-∪ _ _ _)
  = ⊑ₑ-intro₂ _ _ (⊑ₑ-⊔ₑ' y⊑x z⊑x)

⊑ₑ-⊔ₑ-fst : ∀ {x y} → (conxy : ArrCon x y) → x ⊑ₑ (x ⊔ₑ y [ conxy ])
⊑ₑ-⊔ₑ-fst {⊥ₑ} _ = ⊑ₑ-intro₁
⊑ₑ-⊔ₑ-fst {F f _} {⊥ₑ} _ = ⊑ₑ-refl
⊑ₑ-⊔ₑ-fst {F f _} {F f' _} (ArrCon.con-∪ _ _ _)
  = ⊑ₑ-intro₂ _ _ λ {x} {y} xy∈f →
  record
    { sub = (x , y) :: ∅
    ; sub⊆f = ⊆-lemma₄ (∪-lemma₃ xy∈f) ∅-isSubset
    ; preablesub = singletonIsPreable
    ; postablesub = singletonIsPostable
    ; y⊑post = ⊑-⊔-lemma₄ 𝒝 (NbhSys.⊑-refl 𝒝) (con⊥₂ 𝒝)
    ; pre⊑x = NbhSys.⊑-⊔ 𝒜 (NbhSys.⊑-refl 𝒜) (NbhSys.⊑-⊥ 𝒜)
                (con⊥₂ 𝒜)
    }

⊑ₑ-⊔ₑ-snd : ∀ {x y} → (conxy : ArrCon x y) → y ⊑ₑ (x ⊔ₑ y [ conxy ])
⊑ₑ-⊔ₑ-snd {y = ⊥ₑ} _ = ⊑ₑ-intro₁
⊑ₑ-⊔ₑ-snd {⊥ₑ} {F f _} _ = ⊑ₑ-refl
⊑ₑ-⊔ₑ-snd {F f _} {F f' _} (ArrCon.con-∪ _ _ _)
  = ⊑ₑ-intro₂ _ _ λ {x} {y} xy∈f' →
  record
```

```
    { sub = (x , y) :: ∅
    ; sub⊆f = ⊆-lemma₄ (∪-lemma₄ xy∈f') ∅-isSubset
    ; preablesub = singletonIsPreable
    ; postablesub = singletonIsPostable
    ; y⊑post = ⊑-⊔-lemma₄ ℬ (NbhSys.⊑-refl ℬ) (con⊥₂ ℬ)
    ; pre⊑x = NbhSys.⊑-⊔ 𝒜 (NbhSys.⊑-refl 𝒜) (NbhSys.⊑-⊥ 𝒜)
                (con⊥₂ 𝒜)
    }
```

B.0.47 Scwf/DomainScwf/ArrowStructure/NbhSys/ConFinFun

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun
  (𝒜 ℬ : Ty) where

open import Base.FinFun
open import NbhSys.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ

open import Agda.Builtin.Equality

-- A finite function f is consistent if every preable subset
-- of it is also postable.
data ConFinFun (f : NbhFinFun 𝒜 ℬ) : Set where
  cff : (∀ {f'} → f' ⊆ f → Preable f' → Postable f') →
        ConFinFun f

subsetIsCon : ∀ {f f'} → ConFinFun f' → f ⊆ f' → ConFinFun f
subsetIsCon (cff p) f⊆f'
  = cff (λ f''⊆f preablef'' → p (⊆-trans f''⊆f f⊆f') preablef'')

singletonIsCon'' : ∀ {x y} → {f : NbhFinFun 𝒜 ℬ} →
                   f ⊆ ((x , y) :: ∅) →
                   ∀ {x' y'} → (x' , y') ∈ f →
                   [ ℬ ] y' ⊑ y
singletonIsCon'' f⊆xy x'y'∈f with (f⊆xy x'y'∈f)
... | here = NbhSys.⊑-refl ℬ

singletonIsCon' : ∀ {x y f} → f ⊆ ((x , y) :: ∅) →
                  Preable f → Postable f
singletonIsCon' f⊆xy preablef = boundedPostable (singletonIsCon'' f⊆xy)

singletonIsCon : ∀ {x y} → ConFinFun ((x , y) :: ∅)
singletonIsCon = cff (singletonIsCon')
```

B.0.48 Scwf/DomainScwf/ArrowStructure/NbhSys/Consistency

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.Consistency
  (𝒜 ℬ : Ty) where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒜 ℬ

yboundlemma : {x : NbhSys.Nbh 𝒜} → ∀ {y sub} →
      ∀ postablef postablef' postable∪ →
      [ ℬ ] y ⊑ post f postablef →
      (∀ {x' y'} → (x' , y') ∈ sub → [ ℬ ] y' ⊑ post f' postablef') →
      ∀ {x' y'} → (x' , y') ∈ ((x , y) :: sub) →
      [ ℬ ] y' ⊑ post (f ∪ f') postable∪
yboundlemma {f = f} {f'} postablef _ postable∪ y⊑postf _ here
  = NbhSys.⊑-trans ℬ y⊑postf postf⊑post∪
  where postf⊑post∪ = postLemma₁ {f = f} {f'}
yboundlemma {f = f} {f'} _ postablef' postable∪ _ p (there x'y'∈sub)
  = NbhSys.⊑-trans ℬ (p x'y'∈sub) postf'⊑post∪
  where postf'⊑post∪ = postLemma₂ {f = f} {f'}

record ⊑ₑ-proof₃ (f : NbhFinFun 𝒜 ℬ) (isCon : ConFinFun f)
                 (f' : NbhFinFun 𝒜 ℬ) (preablef' : Preable f') :
                 Set where
  field
    sub : NbhFinFun 𝒜 ℬ
    sub⊆f : sub ⊆ f
    preablesub : Preable sub
    postablesub : Postable sub
    ybound : ∀ {x y} → (x , y) ∈ f' → [ ℬ ] y ⊑ (post sub postablesub)
    pre⊑pref' : [ 𝒜 ] (pre sub preablesub) ⊑ (pre f' preablef')

Con-⊔ₑ'' : ∀ {sub conf conf' conf''} →
           (F f conf) ⊑ₑ (F f'' conf'') →
           (F f' conf') ⊑ₑ (F f'' conf'') →
           sub ⊆ (f ∪ f') → (preable : Preable sub) →
           ⊑ₑ-proof₃ f'' conf'' sub preable
```

```
Con-⊔ₑ'' {sub = ∅} _ _ _ _
  = record
      { sub = ∅
      ; sub⊆f = ∅-isSubset
      ; preablesub = pre-nil
      ; postablesub = post-nil
      ; ybound = xy∈∅-abs
      ; pre⊑pref' = NbhSys.⊑-⊥ 𝒜
      }
Con-⊔ₑ'' {f = f} {sub = (x , y) :: sub} _ _ sub⊆f∪f' _
  with (∪-lemma₂ {f = f} (sub⊆f∪f' here))
Con-⊔ₑ'' {sub = (x , y) :: sub} (⊑ₑ-intro₂ _ _ p) _ _ _
  | inl xy∈f with (p xy∈f)
Con-⊔ₑ'' {sub = (x , y) :: sub} {conf'' = cff p} f⊑f'' f'⊑f''
  sub⊆f∪f' (pre-cons preablesub conxpresub)
  | inl xy∈f
  | record { sub = sub''
           ; sub⊆f = sub''⊆f''
           ; preablesub = preablesub''
           ; postablesub = postablesub''
           ; y⊑post = y⊑post''
           ; pre⊑x = pre''⊑x
           }
  = record
      { sub = sub'' ∪ recsub
      ; sub⊆f = ∪⊆f''
      ; preablesub = preable∪
      ; postablesub = postable∪
      ; ybound = yboundlemma postablesub'' recpostablesub postable∪
                   y⊑post'' recybound
      ; pre⊑pref' = NbhSys.⊑-trans 𝒜 (preLemma₃ preablesub'' recpreablesub
                    preable∪ consub''recsub)
                    (⊑-⊔-lemma₃ 𝒜 consub''recsub conxpresub pre''⊑x
                    recpre⊑pref')
      }
  where rec = Con-⊔ₑ'' f⊑f'' f'⊑f'' (⊆-lemma₂ sub⊆f∪f')
                preablesub
        recsub = ⊑ₑ-proof₃.sub rec
        recsub⊆f'' = ⊑ₑ-proof₃.sub⊆f rec
        recpostablesub = ⊑ₑ-proof₃.postablesub rec
        recpreablesub = ⊑ₑ-proof₃.preablesub rec
        recybound = ⊑ₑ-proof₃.ybound rec
        recpre⊑pref' = ⊑ₑ-proof₃.pre⊑pref' rec
        sub''⊑prexysub = NbhSys.⊑-trans 𝒜 pre''⊑x
                           (NbhSys.⊑-⊔-fst 𝒜 conxpresub)
        recsub⊑prexysub = NbhSys.⊑-trans 𝒜 recpre⊑pref'
```

```
                                  (NbhSys.⊑-⊔-snd 𝒜 _)
            preable∪ = preUnionLemma preablesub'' recpreablesub sub''⊑prexysub
                         recsub⊑prexysub
            ∪⊆f'' = ∪-lemma₁ sub''⊆f'' recsub⊆f''
            postable∪ = p ∪⊆f'' preable∪
            consub''recsub = NbhSys.Con-⊔ 𝒜 {z = pre (sub'' ∪ recsub) preable∪}
                                 (preLemma₁ {preablef = preablesub''} {preable∪})
                                 (preLemma₂ {preablef' = recpreablesub} {preable∪})
Con-⊔ₑ'' {sub = (x , y) :: sub} _ (⊑ₑ-intro₂ _ _ p) _ _
  | inr xy∈f' with (p xy∈f')
Con-⊔ₑ'' {sub = (x , y) :: sub} {conf'' = cff p} f⊑f'' f'⊑f''
  sub⊆f∪f' (pre-cons preablesub conxpresub)
  | inr xy∈f'
  | record { sub = sub''
           ; sub⊆f = sub''⊆f''
           ; preablesub = preablesub''
           ; postablesub = postablesub''
           ; y⊑post = y⊑post''
           ; pre⊑x = pre''⊑x
           }
  = record
      { sub = sub'' ∪ recsub
      ; sub⊆f = ∪⊆f''
      ; preablesub = preable∪
      ; postablesub = postable∪
      ; ybound = yboundlemma postablesub'' recpostablesub postable∪
                   y⊑post'' recybound
      ; pre⊑pref' = NbhSys.⊑-trans 𝒜 (preLemma₃ preablesub'' recpreablesub
                     preable∪ consub''recsub)
                     (⊑-⊔-lemma₃ 𝒜 consub''recsub conxpresub pre''⊑x
                     recpre⊑pref')
      }
  where rec = Con-⊔ₑ'' f⊑f'' f'⊑f'' (⊆-lemma₂ sub⊆f∪f')
                preablesub
        recsub = ⊑ₑ-proof₃.sub rec
        recsub⊆f'' = ⊑ₑ-proof₃.sub⊆f rec
        recpostablesub = ⊑ₑ-proof₃.postablesub rec
        recpreablesub = ⊑ₑ-proof₃.preablesub rec
        recybound = ⊑ₑ-proof₃.ybound rec
        recpre⊑pref' = ⊑ₑ-proof₃.pre⊑pref' rec
        sub''⊑prexysub = NbhSys.⊑-trans 𝒜 pre''⊑x
                           (NbhSys.⊑-⊔-fst 𝒜 conxpresub)
        recsub⊑prexysub = NbhSys.⊑-trans 𝒜 recpre⊑pref'
                            (NbhSys.⊑-⊔-snd 𝒜 _)
        preable∪ = preUnionLemma preablesub'' recpreablesub sub''⊑prexysub
                     recsub⊑prexysub
```

```
        ∪⊆f'' = ∪-lemma₁ sub''⊆f'' recsub⊆f''
        postable∪ = p ∪⊆f'' preable∪
        consub''recsub = NbhSys.Con-⊔ 𝒜 {z = pre (sub'' ∪ recsub) preable∪}
                         (preLemma₁ {preablef = preablesub''} {preable∪})
                         (preLemma₂ {preablef' = recpreablesub} {preable∪})

Con-⊔ₑ' : ∀ {sub conf conf' conf''} →
          (F f conf) ⊑ₑ (F f'' conf'') →
          (F f' conf') ⊑ₑ (F f'' conf'') →
          sub ⊆ (f ∪ f') → (preable : Preable sub) →
          Postable sub
Con-⊔ₑ' f⊑f'' f'⊑f'' sub⊆f∪f' preablesub
  = boundedPostable ybound
  where proof = Con-⊔ₑ'' f⊑f'' f'⊑f'' sub⊆f∪f' preablesub
        sub'' = ⊑ₑ-proof₃.sub proof
        ybound = ⊑ₑ-proof₃.ybound proof

Con-⊔ₑ : ∀ {x y z} → x ⊑ₑ z → y ⊑ₑ z → ArrCon x y
Con-⊔ₑ {⊥ₑ} {y} _ _ = conₑ-⊥₂
Con-⊔ₑ {F f _} {⊥ₑ} _ _ = conₑ-⊥₁
Con-⊔ₑ {F f _} {F f' _} {⊥ₑ} () _
Con-⊔ₑ {F f conf} {F f' conf'} {F f'' conf''} f⊑f'' f'⊑f''
  = ArrCon.con-∪ _ _ (cff λ {f' = sub} sub⊆f∪f' preablesub →
    Con-⊔ₑ' f⊑f'' f'⊑f'' sub⊆f∪f' preablesub)
```

B.0.49 Scwf/DomainScwf/ArrowStructure/NbhSys/Definition

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.Definition
  (𝒜 ℬ : Ty) where

open import Base.FinFun
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ

data ArrNbh : Set where
  ⊥ₑ : ArrNbh
  F : (f : NbhFinFun 𝒜 ℬ) → ConFinFun f → ArrNbh

data ArrCon : ArrNbh → ArrNbh → Set where
  conₑ-⊥₁ : ∀ {x} → ArrCon x ⊥ₑ
  conₑ-⊥₂ : ∀ {x} → ArrCon ⊥ₑ x
  con-∪ : ∀ {f f'} → (conf : ConFinFun f) → (conf' : ConFinFun f') →
          ConFinFun (f ∪ f') → ArrCon (F f conf) (F f' conf')

_⊔ₑ_[_] : (x : ArrNbh) → (y : ArrNbh) → ArrCon x y → ArrNbh
```

```
⊥e ⊔e ⊥e [ _ ] = ⊥e
⊥e ⊔e (F f' conf') [ _ ] = F f' conf'
(F f conf) ⊔e ⊥e [ _ ] = F f conf
F f _ ⊔e F f' _ [ con-∪ _ _ con∪ ] = F (f ∪ f') con∪
```

B.0.50 Scwf/DomainScwf/ArrowStructure/NbhSys/DefProof

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.DefProof
  (𝒜 ℬ : Ty) where

open import Appmap.Lemmata
open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ

-- Contains the proof that F f ⊑ F f' in the arrow
-- neighborhood system if and only if the smallest approximable
-- mapping containing f' also contains f. We show that the
-- two propositions imply one another.

-- The "containment" relation.
data _∈_ (f : NbhFinFun 𝒜 ℬ) (γ : Appmap 𝒜 ℬ) :
         Set where
  ∈-intro : (∀ {x y} → (x , y) ∈ f → [ γ ] x ↦ y) →
              f ∈ γ

-- If an approximable mapping γ contains f, then it
-- contains any subset f' of f.
∈-lemma : (f' f : NbhFinFun 𝒜 ℬ) → f' ⊆ f →
            (γ : Appmap 𝒜 ℬ) →
            f ∈ γ → f' ∈ γ
∈-lemma f' f f'⊆f γ (∈-intro p)
  = ∈-intro λ xy∈f' → p (f'⊆f xy∈f')

-- If f is contained in the mapping γ, then γ maps (pre f)
-- to (post f)
pre↦post : (f : NbhFinFun 𝒜 ℬ) → (preablef : Preable f) →
```

```
                (postablef : Postable f) → (γ : Appmap 𝒜 ℬ) →
                f ⋐ γ → [ γ ] (pre f preablef) ↦ (post f postablef)
pre↦post ∅ _ _ γ _ = Appmap.↦-bottom γ
pre↦post ((x , y) :: f') (pre-cons preablef' conxpref')
  (post-cons postablef' conypostf') γ (⋐-intro p)
  = appmapLemma₃ {γ = γ} x (pre f' preablef') y
    (post f' _) _ _ (p here)
    (pre↦post f' preablef' postablef' γ (⋐-intro (λ x'y'∈f' →
    p (there x'y'∈f'))))

-- A pair (x, y) is in this relation iff (x, y) ∈ f, or if
-- it can be derived from the approximable mapping axioms.
data AppmapClosure (f : NbhFinFun 𝒜 ℬ)
                   (conf : ConFinFun f) : ∀ x y → Set where
  ig-inset : ∀ {x y} → (x , y) ∈ f →
             AppmapClosure f conf x y
  ig-bot   : ∀ {x} →
             AppmapClosure f conf x (NbhSys.⊥ ℬ)
  ig-mono : ∀ {x x' y} → [ 𝒜 ] x' ⊑ x → AppmapClosure f conf x' y →
             AppmapClosure f conf x y
  ig-↓clo : ∀ {x y y'} → [ ℬ ] y ⊑ y' → AppmapClosure f conf x y' →
             AppmapClosure f conf x y
  ig-↑dir : ∀ {x y y'} → AppmapClosure f conf x y →
             AppmapClosure f conf x y' → (con : NbhSys.Con ℬ y y') →
             AppmapClosure f conf x ([ ℬ ] y ⊔ y' [ con ])

smallest⇒exp' : (f' : NbhFinFun 𝒜 ℬ) → {con : ConFinFun f'} →
                ∀ {x y} → AppmapClosure f' con x y →
                ⊑ₑ-proof f' con x y
smallest⇒exp' f' {x = x} {y} (ig-inset xy∈f')
  = record
      { sub = (x , y) :: ∅
      ; sub⊆f = ⊆-lemma₄ xy∈f' ∅-isSubset
      ; preablesub = pre-cons pre-nil (con⊥₂ 𝒜)
      ; postablesub = post-cons post-nil (con⊥₂ ℬ)
      ; y⊑post = NbhSys.⊑-⊔-fst ℬ (con⊥₂ ℬ)
      ; pre⊑x = NbhSys.⊑-⊔ 𝒜 (NbhSys.⊑-refl 𝒜)
                (NbhSys.⊑-⊥ 𝒜) (con⊥₂ 𝒜)
      }
smallest⇒exp' f' ig-bot
  = record
      { sub = ∅
      ; sub⊆f = ∅-isSubset
      ; preablesub = pre-nil
      ; postablesub = post-nil
      ; y⊑post = NbhSys.⊑-⊥ ℬ
      ; pre⊑x = NbhSys.⊑-⊥ 𝒜
```

```
        }
smallest⇒exp' f' {con} {x} {y} (ig-mono {x' = x'} x'⊑x idGen)
  = record
      { sub = ⊑e-proof.sub rec
      ; sub⊆f = ⊑e-proof.sub⊆f rec
      ; preablesub = ⊑e-proof.preablesub rec
      ; postablesub = ⊑e-proof.postablesub rec
      ; y⊑post = ⊑e-proof.y⊑post rec
      ; pre⊑x = NbhSys.⊑-trans 𝒜 (⊑e-proof.pre⊑x rec) x'⊑x
      }
  where rec = smallest⇒exp' f' {con} {x'} {y} idGen
smallest⇒exp' f' {con} {x} {y} (ig-↓clo {y' = y'} y⊑y' idGen)
  = record
      { sub = ⊑e-proof.sub rec
      ; sub⊆f = ⊑e-proof.sub⊆f rec
      ; preablesub = ⊑e-proof.preablesub rec
      ; postablesub = ⊑e-proof.postablesub rec
      ; y⊑post = NbhSys.⊑-trans ℬ y⊑y' (⊑e-proof.y⊑post rec)
      ; pre⊑x = ⊑e-proof.pre⊑x rec
      }
  where rec = smallest⇒exp' f' {con} {x} {y'} idGen
smallest⇒exp' f' {cff p} {x} (ig-↑dir {y = y} {y'}
  idGeny idGeny' conyy')
  with (smallest⇒exp' f' {cff p} {x} {y} idGeny)
  | smallest⇒exp' f' {cff p} {x} {y'} idGeny'
... | record { sub = sub
             ; sub⊆f = sub⊆f'
             ; preablesub = preable
             ; postablesub = postable
             ; y⊑post = y⊑post
             ; pre⊑x = pre⊑x
             }
    | record { sub = sub'
             ; sub⊆f = sub'⊆f'
             ; preablesub = preable'
             ; postablesub = postable'
             ; y⊑post = y⊑post'
             ; pre⊑x = pre'⊑x
             }
  = record
      { sub = sub ∪ sub'
      ; sub⊆f = ∪⊆f
      ; preablesub = preable∪
      ; postablesub = postable∪
      ; y⊑post = NbhSys.⊑-trans ℬ
                   (⊑-⊔-lemma₃ ℬ _ conpost y⊑post y⊑post')
```

```
                (postLemma₃ postable postable' _ _ )
        ; pre⊑x = NbhSys.⊑-trans 𝒜 (preLemma₃ preable preable' _ _)
                (NbhSys.⊑-⊔ 𝒜 pre⊑x pre'⊑x conpre)
        }
  where preable∪ = preUnionLemma preable preable' pre⊑x pre'⊑x
        conpre = NbhSys.Con-⊔ 𝒜 pre⊑x pre'⊑x
        ∪⊆f = ∪-lemma₁ sub⊆f' sub'⊆f'
        postable∪ = p (∪-lemma₁ sub⊆f' sub'⊆f') preable∪
        conpost = NbhSys.Con-⊔ ℬ
                  (postLemma₁ {f = sub} {postable∪ = postable∪})
                  (postLemma₂ {f' = sub'} {postable∪ = postable∪})

appmapClosureCon : ∀ {f conf x y x' y'} →
                   AppmapClosure f conf x y →
                   AppmapClosure f conf x' y' →
                   NbhSys.Con 𝒜 x x' →
                   NbhSys.Con ℬ y y'
appmapClosureCon {f} {cff p} {x} {y} {x'} {y'}
  apcloxy apclox'y' conxx'
  with (smallest⇒exp' f {x = x} {y} apcloxy)
  | smallest⇒exp' f {x = x'} {y'} apclox'y'
... | record { sub = sub
             ; sub⊆f = sub⊆f
             ; preablesub = preable
             ; postablesub = postable
             ; y⊑post = y⊑post
             ; pre⊑x = pre⊑x
             }
    | record { sub = sub'
             ; sub⊆f = sub'⊆f
             ; preablesub = preable'
             ; postablesub = postable'
             ; y⊑post = y'⊑post'
             ; pre⊑x = pre'⊑x'
             }
  = NbhSys.Con-⊔ ℬ {z = post (sub ∪ sub') postable∪} y⊑post∪ y'⊑post∪
  where x⊔x' = [ 𝒜 ] x ⊔ x' [ conxx' ]
        presub⊑x⊔x' = ⊑-⊔-lemma₄ 𝒜 pre⊑x conxx'
        presub'⊑x⊔x' = ⊑-⊔-lemma₅ 𝒜 pre'⊑x' conxx'
        preable∪ = preUnionLemma preable preable' presub⊑x⊔x'
                   presub'⊑x⊔x'
        postable∪ = p (∪-lemma₁ sub⊆f sub'⊆f) preable∪
        y⊑post∪ = NbhSys.⊑-trans ℬ y⊑post
                  (postLemma₁ {f = sub} {postable∪ = postable∪})
        y'⊑post∪ = NbhSys.⊑-trans ℬ y'⊑post'
                   (postLemma₂ {f' = sub'} {postable∪ = postable∪})
```

```
SmallestAppmap : (f : NbhFinFun 𝒜 ℬ) → ConFinFun f → Appmap 𝒜 ℬ
Appmap._↦_ (SmallestAppmap f conf)     = AppmapClosure f conf
Appmap.↦-mono (SmallestAppmap f _)     = ig-mono
Appmap.↦-bottom (SmallestAppmap f _)   = ig-bot
Appmap.↦-↓closed (SmallestAppmap f _)  = ig-↓clo
Appmap.↦-↑directed (SmallestAppmap f _) = ig-↑dir
Appmap.↦-con (SmallestAppmap f _)      = appmapClosureCon

smallest⇒exp : (f f' : NbhFinFun 𝒜 ℬ) →
               (conf : ConFinFun f) →
               (conf' : ConFinFun f') →
               f ⋐ SmallestAppmap f' conf' →
               F f conf ⊑e F f' conf'
smallest⇒exp f f' conf conf' (⋐-intro p)
  = ⊑e-intro2 conf conf' (λ xy∈f →
    smallest⇒exp' f' {conf'} (p xy∈f))

exp⇒smallest' : (f f' : NbhFinFun 𝒜 ℬ) → ∀ {conf conf'} →
                F f conf ⊑e F f' conf' →
                ∀ {x y} → (x , y) ∈ f →
                [ SmallestAppmap f' conf' ] x ↦ y
exp⇒smallest' f f' (⊑e-intro2 _ con p) xy∈f with (p xy∈f)
exp⇒smallest' f f' (⊑e-intro2 _ con p) xy∈f
  | record { sub = f''
           ; sub⊆f = sub⊆f
           ; preablesub = preablef''
           ; postablesub = postablef''
           ; y⊑post = y⊑post
           ; pre⊑x = pre⊑x
           }
  = Appmap.↦-↓closed γ' y⊑post γx↦post
  where γ' = SmallestAppmap f' con
        γpref''↦postf'' = pre↦post f'' preablef'' postablef'' γ'
                          (⋐-lemma f'' f' sub⊆f γ'
                          (⋐-intro ig-inset))
        γx↦post = Appmap.↦-mono γ' pre⊑x γpref''↦postf''

exp⇒smallest : (f f' : NbhFinFun 𝒜 ℬ) →
               ∀ {conf conf'} →
               F f conf ⊑e F f' conf' →
               f ⋐ SmallestAppmap f' conf'
exp⇒smallest f f' f⊑f'
  = ⋐-intro (exp⇒smallest' f f' f⊑f')
```

B.0.51 Scwf/DomainScwf/ArrowStructure/NbhSys/Instance

```
{-# OPTIONS --safe #-}
```

```
module Scwf.DomainScwf.ArrowStructure.NbhSys.Instance where

open import Base.Core
open import Base.FinFun
open import NbhSys.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.AxiomProofs
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Consistency
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Transitivity

ArrNbhSys : (𝒜 ℬ : Ty) → NbhSys
NbhSys.Nbh (ArrNbhSys 𝒜 ℬ)     = ArrNbh 𝒜 ℬ
NbhSys._⊑_ (ArrNbhSys 𝒜 ℬ)     = _⊑ₑ_ 𝒜 ℬ
NbhSys.Con (ArrNbhSys 𝒜 ℬ)     = ArrCon 𝒜 ℬ
NbhSys._⊔_[_] (ArrNbhSys 𝒜 ℬ) = _⊔ₑ_[_] 𝒜 ℬ
NbhSys.⊥ (ArrNbhSys 𝒜 ℬ)       = ⊥ₑ
NbhSys.Con-⊔ (ArrNbhSys 𝒜 ℬ)  = Con-⊔ₑ 𝒜 ℬ
NbhSys.⊑-refl (ArrNbhSys 𝒜 ℬ)  = ⊑ₑ-refl 𝒜 ℬ
NbhSys.⊑-trans (ArrNbhSys 𝒜 ℬ) = ⊑ₑ-trans 𝒜 ℬ
NbhSys.⊑-⊥ (ArrNbhSys 𝒜 ℬ)     = ⊑ₑ-⊥ₑ 𝒜 ℬ
NbhSys.⊑-⊔ (ArrNbhSys 𝒜 ℬ)     = ⊑ₑ-⊔ₑ 𝒜 ℬ
NbhSys.⊑-⊔-fst (ArrNbhSys 𝒜 ℬ) = ⊑ₑ-⊔ₑ-fst 𝒜 ℬ
NbhSys.⊑-⊔-snd (ArrNbhSys 𝒜 ℬ) = ⊑ₑ-⊔ₑ-snd 𝒜 ℬ
```

B.0.52 Scwf/DomainScwf/ArrowStructure/NbhSys/Post

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.Post
  (𝒜 ℬ : Ty) where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒜 ℬ

data Postable : NbhFinFun 𝒜 ℬ → Set
post : (f : NbhFinFun 𝒜 ℬ) → Postable f → NbhSys.Nbh ℬ

data Postable where
  post-nil : Postable ∅
  post-cons : ∀ {x y f} → (postablef : Postable f) →
              NbhSys.Con ℬ y (post f postablef) → Postable ((x , y) :: f)

post ∅ _ = NbhSys.⊥ ℬ
```

```
post ((x , y) :: f) (post-cons postablef conxpostf)
  = [ 𝓑 ] y ⊔ post f postablef [ conxpostf ]

boundedPostable' : ∀ {f postablef max} →
                   (∀ {x y} → (x , y) ∈ f → [ 𝓑 ] y ⊑ max) →
                   [ 𝓑 ] post f postablef ⊑ max
boundedPostable' {∅} _ = NbhSys.⊑-⊥ 𝓑
boundedPostable' {(x , y) :: f}
  {postablef = post-cons postablef conypostf} bound
  = NbhSys.⊑-⊔ 𝓑 (bound here) rec conypostf
  where rec = boundedPostable' {postablef = postablef}
                λ x'y'∈f → bound (there x'y'∈f)

boundedPostable : ∀ {f max} →
                  (∀ {x y} → (x , y) ∈ f → [ 𝓑 ] y ⊑ max) →
                  Postable f
boundedPostable {∅} _ = post-nil
boundedPostable {(x , y) :: f} bound
  = post-cons (boundedPostable {f} (λ xy∈f → bound (there xy∈f)))
    (NbhSys.Con-⊔ 𝓑 (bound here)
    (boundedPostable' {f} λ xy∈f → bound (there xy∈f)))

postableProofIrr : (postablef₁ postablef₂ : Postable f) →
                   [ 𝓑 ] (post f postablef₁) ⊑ (post f postablef₂)
postableProofIrr {∅} post-nil post-nil = NbhSys.⊑-refl 𝓑
postableProofIrr {(x , y) :: f} (post-cons postablef₁ conxpostf₁)
  (post-cons postablef₂ conxpostf₂)
  = ⊑-⊔-lemma₃ 𝓑 _ _ (NbhSys.⊑-refl 𝓑)
    (postableProofIrr postablef₁ postablef₂)

postLemma₁ : ∀ {f f' postablef postable∪} →
             [ 𝓑 ] post f postablef ⊑ post (f ∪ f') postable∪
postLemma₁ {postablef = post-nil} = NbhSys.⊑-⊥ 𝓑
postLemma₁ {f = _ :: f} {postablef = post-cons postablef conxpostf}
  {post-cons postablef∪f' conxpost∪}
  = ⊑-⊔-lemma₃ 𝓑 _ _ (NbhSys.⊑-refl 𝓑) rec
  where rec = postLemma₁ {f = f} {postablef = postablef}

postLemma₂ : ∀ {f f' postablef' postable∪} →
             [ 𝓑 ] post f' postablef' ⊑ post (f ∪ f') postable∪
postLemma₂ {f = _} {∅} = NbhSys.⊑-⊥ 𝓑
postLemma₂ {f = ∅} {_ :: _} {postablef'}
  = NbhSys.⊑-trans 𝓑 (NbhSys.⊑-refl 𝓑)
    (postableProofIrr postablef' _)
postLemma₂ {f = (x , y) :: f} {(x' , y') :: f'}
  {post-cons postablef'tail conxpostf'tail}
  {post-cons postable∪tail x'con∪tail}
  = ⊑-⊔-lemma₅ 𝓑 rec x'con∪tail
```

```
  where postablef' = post-cons postablef'tail conxpostf'tail
        rec = postLemma₂ {f = f} {f' = (x' , y') :: f'}
                {postablef' = postablef'}

postLemma₃ : (postablef : Postable f) → (postablef' : Postable f') →
             (postableU : Postable (f ∪ f')) →
             (conpost : NbhSys.Con 𝒷 (post f postablef) (post f' postablef')) →
             [ 𝒷 ] ([ 𝒷 ] (post f postablef) ⊔
             (post f' postablef') [ conpost ])
             ⊑ (post (f ∪ f') postableU)
postLemma₃ postablef postablef' postableU conpost
  = NbhSys.⊑-⊔ 𝒷 postf⊑postU postf'⊑postU conpost
  where postf⊑postU = postLemma₁ {postablef = postablef} {postableU}
        postf'⊑postU = postLemma₂ {postablef' = postablef'} {postableU}

singletonIsPostable : ∀ {x y} → Postable ((x , y) :: ∅)
singletonIsPostable = post-cons post-nil (con⊥₂ 𝒷)
```

B.0.53 Scwf/DomainScwf/ArrowStructure/NbhSys/Pre

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.Pre
  (𝒶 𝒷 : Ty) where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒶 𝒷

data Preable : NbhFinFun 𝒶 𝒷 → Set
pre : (f : NbhFinFun 𝒶 𝒷) → Preable f → NbhSys.Nbh 𝒶

data Preable where
  pre-nil : Preable ∅
  pre-cons : ∀ {x y f} → (preablef : Preable f) →
             NbhSys.Con 𝒶 x (pre f preablef) → Preable ((x , y) :: f)

pre ∅ _ = NbhSys.⊥ 𝒶
pre ((x , y) :: f) (pre-cons preablef conxpref)
  = [ 𝒶 ] x ⊔ pre f preablef [ conxpref ]

preableProofIrr : (preablef₁ preablef₂ : Preable f) →
                  [ 𝒶 ] (pre f preablef₁) ⊑ (pre f preablef₂)
preableProofIrr {∅} pre-nil pre-nil = NbhSys.⊑-refl 𝒶
preableProofIrr {(x , y) :: f} (pre-cons preablef₁ conxpref₁)
```

```
    (pre-cons preablef₂ conxpref₂)
  = ⊑-⊔-lemma₃ 𝒜 _ _ (NbhSys.⊑-refl 𝒜)
      (preableProofIrr preablef₁ preablef₂)

preLemma₁ : ∀ {f f' preablef preable∪} →
              [ 𝒜 ] pre f preablef ⊑ pre (f ∪ f') preable∪
preLemma₁ {preablef = pre-nil} = NbhSys.⊑-⊥ 𝒜
preLemma₁ {f = _ :: f} {preablef = pre-cons preablef conxpref}
  {pre-cons preablef∪f' conxpre∪}
  = ⊑-⊔-lemma₃ 𝒜 _ _ (NbhSys.⊑-refl 𝒜) rec
  where rec = preLemma₁ {f = f} {preablef = preablef}

preLemma₂ : ∀ {f f' preablef' preable∪} →
              [ 𝒜 ] pre f' preablef' ⊑ pre (f ∪ f') preable∪
preLemma₂ {f = _} {∅} = NbhSys.⊑-⊥ 𝒜
preLemma₂ {f = ∅} {_ :: _} {preablef'}
  = NbhSys.⊑-trans 𝒜 (NbhSys.⊑-refl 𝒜)
      (preableProofIrr preablef' _)
preLemma₂ {f = (x , y) :: f} {(x' , y') :: f'}
  {pre-cons preablef'tail conxpref'tail}
  {pre-cons preable∪tail x'con∪tail}
  = ⊑-⊔-lemma₅ 𝒜 rec x'con∪tail
  where preablef' = pre-cons preablef'tail conxpref'tail
        rec = preLemma₂ {f = f} {f' = (x' , y') :: f'}
                {preablef' = preablef'}

preLemma₃'' : (preablef : Preable f) → (preablef' : Preable f') →
                (preable∪ : Preable (f ∪ f')) →
                NbhSys.Con 𝒜 (pre f preablef) (pre f' preablef')
preLemma₃'' {f} {f'} preablef preablef' preable∪
  = NbhSys.Con-⊔ 𝒜 pref⊑pre∪ pref'⊑pre∪
  where pref⊑pre∪ = preLemma₁ {f = f} {preable∪ = preable∪}
        pref'⊑pre∪ = preLemma₂ {f' = f'} {preable∪ = preable∪}

preLemma₃' : ∀ x → (preablef : Preable f) → (preablef' : Preable f') →
               (con₁ : NbhSys.Con 𝒜 x (pre f preablef)) →
               (con₂ : NbhSys.Con 𝒜 (pre f preablef) (pre f' preablef')) →
               NbhSys.Con 𝒜 ([ 𝒜 ] x ⊔ pre f preablef [ con₁ ])
                 (pre f' preablef') →
               NbhSys.Con 𝒜 x ([ 𝒜 ] (pre f preablef) ⊔
                 (pre f' preablef') [ con₂ ])
preLemma₃' {f} {f'} x preablef preablef' con₁ con₂ con₃
  = NbhSys.Con-⊔ 𝒜 (NbhSys.⊑-trans 𝒜 (NbhSys.⊑-⊔-fst 𝒜 con₁)
    (NbhSys.⊑-⊔-fst 𝒜 con₃))
    (⊑-⊔-lemma₃ 𝒜 _ _ (NbhSys.⊑-⊔-snd 𝒜 _) (NbhSys.⊑-refl 𝒜))

preLemma₃ : (preablef : Preable f) → (preablef' : Preable f') →
              (preable∪ : Preable (f ∪ f')) →
```

```
                (conpre : NbhSys.Con 𝒜 (pre f preablef) (pre f' preablef')) →
                [ 𝒜 ] (pre (f ∪ f') preable∪) ⊑
                ([ 𝒜 ] (pre f preablef) ⊔ (pre f' preablef')) [ conpre ])
preLemma₃ {∅} {f'} pre-nil _ _ _
  = ⊑-⊔-lemma₅ 𝒜 (preableProofIrr {f = f'} _ _) _
preLemma₃ {(x , y) :: f} {f'} (pre-cons preablef conxpref) preablef'
  (pre-cons preable∪ conxpre∪) conpre₁
  = NbhSys.⊑-trans 𝒜 (⊑-⊔-lemma₃ 𝒜 _ conxpre⊔ (NbhSys.⊑-refl 𝒜)
    (preLemma₃ {f} {f'} _ _ preable∪ conpre₂))
    (⊔-ass₂ 𝒜 _ conpre₂ conxpre⊔ _ (NbhSys.⊑-refl 𝒜))
  where conpre₂ = preLemma₃'' preablef preablef' preable∪
        conxpre⊔ = preLemma₃' x preablef preablef' conxpref conpre₂ conpre₁

preUnionLemma' : ∀ {max} → (preablef : Preable f) →
                 (preablef' : Preable f') →
                 (preable∪ : Preable (f ∪ f')) →
                 [ 𝒜 ] (pre f preablef) ⊑ max →
                 [ 𝒜 ] (pre f' preablef') ⊑ max →
                 [ 𝒜 ] (pre (f ∪ f') preable∪) ⊑ max
preUnionLemma' {∅} {f'} preablef preablef' preable∪ pref⊑max pref'⊑max
  = NbhSys.⊑-trans 𝒜 (preableProofIrr preable∪ preablef') pref'⊑max
preUnionLemma' {(x , y) :: f} (pre-cons preablef conxpref) preablef'
  (pre-cons preable∪ conxpre∪) prexyf⊑max pref'⊑max
  = NbhSys.⊑-⊔ 𝒜 x⊑max rec conxpre∪
  where pref⊑max = NbhSys.⊑-trans 𝒜 (NbhSys.⊑-⊔-snd 𝒜 conxpref)
                   prexyf⊑max
        rec = preUnionLemma' preablef preablef' preable∪ pref⊑max
              pref'⊑max
        x⊑max = NbhSys.⊑-trans 𝒜 (NbhSys.⊑-⊔-fst 𝒜 conxpref)
                prexyf⊑max

preUnionLemma : ∀ {max} → (preablef : Preable f) →
                (preablef' : Preable f') →
                [ 𝒜 ] (pre f preablef) ⊑ max →
                [ 𝒜 ] (pre f' preablef') ⊑ max → Preable (f ∪ f')
preUnionLemma {∅} _ preablef' _ _ = preablef'
preUnionLemma {(x , y) :: f} (pre-cons preablef conxpref)
  preablef' pref⊑x pref'⊑x
  = pre-cons rec (NbhSys.Con-⊔ 𝒜 x⊑max pre∪⊑max)
  where pref⊑max = NbhSys.⊑-trans 𝒜 (NbhSys.⊑-⊔-snd 𝒜 conxpref)
                   pref⊑x
        rec = preUnionLemma preablef preablef' pref⊑max pref'⊑x
        x⊑max = NbhSys.⊑-trans 𝒜 (NbhSys.⊑-⊔-fst 𝒜 conxpref) pref⊑x
        pre∪⊑max = preUnionLemma' preablef preablef' rec pref⊑max
                   pref'⊑x

singletonIsPreable : ∀ {x y} → Preable ((x , y) :: ∅)
singletonIsPreable = pre-cons pre-nil (con⊥₂ 𝒜)
```

B.0.54 Scwf/DomainScwf/ArrowStructure/NbhSys/Relation

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.Relation
  (𝒜 ℬ : Ty) where

open import Base.FinFun
open import NbhSys.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ

record ⊑ₑ-proof (f : NbhFinFun 𝒜 ℬ) (isCon : ConFinFun f)
                (x : NbhSys.Nbh 𝒜) (y : NbhSys.Nbh ℬ) :
                Set where
  field
    sub : NbhFinFun 𝒜 ℬ
    sub⊆f : sub ⊆ f
    preablesub : Preable sub
    postablesub : Postable sub
    y⊑post : NbhSys._⊑_ ℬ y (post sub postablesub)
    pre⊑x : NbhSys._⊑_ 𝒜 (pre sub preablesub) x

data _⊑ₑ_ : ArrNbh → ArrNbh → Set where
  ⊑ₑ-intro₁ : ∀ {x} → ⊥ₑ ⊑ₑ x
  ⊑ₑ-intro₂ : ∀ {f f'} → (conf : ConFinFun f) → (conf' : ConFinFun f') →
              (∀ {x y} → (x , y) ∈ f → ⊑ₑ-proof f' conf' x y) →
              (F f conf) ⊑ₑ (F f' conf')
```

B.0.55 Scwf/DomainScwf/ArrowStructure/NbhSys/Transitivity

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ArrowStructure.NbhSys.Transitivity
  (𝒜 ℬ : Ty) where

open import Base.FinFun
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ArrowStructure.NbhSys.ConFinFun 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Definition 𝒜 ℬ
```

```
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Relation 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Post 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.NbhSys.Pre 𝒜 ℬ
open import Scwf.DomainScwf.ArrowStructure.Variables 𝒜 ℬ

-- This can be derived from F f ⊑e F f', and makes proving
-- transitivity very simple.
record ⊑e-proof₂ (f f' : NbhFinFun 𝒜 ℬ) (preablef : Preable f)
                 (postablef : Postable f) : Set where
  field
    sub : NbhFinFun 𝒜 ℬ
    preablesub : Preable sub
    postablesub : Postable sub
    pf⊑post : [ ℬ ] (post f postablef) ⊑ (post sub postablesub)
    pre⊑pf : [ 𝒜 ] (pre sub preablesub) ⊑ (pre f preablef)
    sub⊆f' : sub ⊆ f'

shrinkExp' : ∀ {conf' conf''} →
             f ⊆ f' → (F f' conf') ⊑e (F f'' conf'') →
             ∀ {x y} → (x , y) ∈ f →
             ⊑e-proof f'' conf'' x y
shrinkExp' f⊆f' (⊑e-intro₂ _ _ p) xy∈f
  = p (f⊆f' xy∈f)

-- If f ⊆ f' and f' ⊑e f'', then we can adapt the ⊑e-proof
-- of f' and f'' to one for f and f''.
shrinkExp : ∀ {conf conf' conf''} →
            f ⊆ f' → (F f' conf') ⊑e (F f'' conf'') →
            (F f conf) ⊑e (F f'' conf'')
shrinkExp {f = f} {f'' = f''} f⊆f' f'⊑f''
  = ⊑e-intro₂ _ _ (shrinkExp' f⊆f' f'⊑f'')

Ω : (f f' : NbhFinFun 𝒜 ℬ) →
    ∀ {conf conf' preablef postablef} →
    (F f conf) ⊑e (F f' conf') →
    ⊑e-proof₂ f f' preablef postablef
Ω ∅ f' f⊑f'
  = record { sub = ∅
           ; preablesub = pre-nil
           ; postablesub = post-nil
           ; pf⊑post = NbhSys.⊑-refl ℬ
           ; pre⊑pf = NbhSys.⊑-refl 𝒜
           ; sub⊆f' = ∅-isSubset
           }
Ω ((x , y) :: f'') f' (⊑e-intro₂ _ _ p) with (p here)
Ω ((x , y) :: f'') f' {cff conf} {cff conf'}
  {pre-cons preablef'' conxpref''} {post-cons postablef'' conypostf''}
  (⊑e-intro₂ _ _ p)
```

```
| record { sub = sub
         ; sub⊆f = sub⊆f
         ; preablesub = preablesub
         ; postablesub = postablesub
         ; y⊑post = y⊑post
         ; pre⊑x = pre⊑x
         }
= record
    { sub = sub ∪ sub'
    ; preablesub = preable∪
    ; postablesub = postable∪
    ; pf⊑post = NbhSys.⊑-trans 𝓑 (⊑-⊔-lemma₃ 𝓑 conypostf''
                  conpostsubs y⊑post
                  (NbhSys.⊑-trans 𝓑 (postableProofIrr postablef'' _)
                  (⊑ₑ-proof₂.pf⊑post recur)))
                  (postLemma₃ postablesub postablesub' postable∪ conpostsubs)
    ; pre⊑pf = NbhSys.⊑-trans 𝓐
                  (preLemma₃ preablesub preablesub' preable∪
                  conpresubs) (⊑-⊔-lemma₃ 𝓐 conpresubs conxpref'' pre⊑x
                  (NbhSys.⊑-trans 𝓐 (⊑ₑ-proof₂.pre⊑pf recur)
                  (preableProofIrr _ preablef'')))
    ; sub⊆f' = ∪-lemma₁ sub⊆f (⊑ₑ-proof₂.sub⊆f' recur)
    }
where preablef' = pre-cons {y = y} preablef'' conxpref''
      postablef' = post-cons {x = x} postablef'' conypostf''
      conTail = subsetIsCon (cff conf) ⊆-lemma₃
      recur = Ω f'' f' {conTail} {_} {preablef''} {postablef''}
                (shrinkExp {conf = conTail} ⊆-lemma₃
                (⊑ₑ-intro₂ (cff conf) _ p))
      sub' = ⊑ₑ-proof₂.sub recur
      preablesub' = ⊑ₑ-proof₂.preablesub recur
      postablesub' = ⊑ₑ-proof₂.postablesub recur
      ∪⊆f = ∪-lemma₁ sub⊆f (⊑ₑ-proof₂.sub⊆f' recur)
      presub⊑pref' = ⊑-⊔-lemma₄ 𝓐 pre⊑x conxpref''
      presub'⊑pref' = ⊑-⊔-lemma₅ 𝓐 (NbhSys.⊑-trans 𝓐
                        (⊑ₑ-proof₂.pre⊑pf recur)
                        (preableProofIrr preablef'' _)) conxpref''
      preable∪ = preUnionLemma preablesub preablesub' presub⊑pref'
                   presub'⊑pref'
      postable∪ = conf' ∪⊆f preable∪
      conpostsubs = NbhSys.Con-⊔ 𝓑 (postLemma₁
                      {postablef = postablesub} {postable∪})
                      (postLemma₂ {postablef' = postablesub'}
                      {postable∪})
      conpresubs = NbhSys.Con-⊔ 𝓐
                     (preLemma₁ {preablef = preablesub}
```

```
                                {preableU}) (preLemma₂ {preablef' = preablesub'}
                                {preableU})

⊑ₑ-trans' : ∀ {conf conf' conf''} →
              (F f conf) ⊑ₑ (F f' conf') → (F f' conf') ⊑ₑ (F f'' conf'') →
              ∀ {x y} → (x , y) ∈ f → ⊑ₑ-proof f'' conf'' x y
⊑ₑ-trans' {f} {f'} {f''} {conf} {conf'} (⊑ₑ-intro₂ _ _ p₁)
  (⊑ₑ-intro₂ preablef' preablef'' p₂) xy∈f
  = record
      { sub = f''sub
      ; sub⊆f = ⊑ₑ-proof₂.sub⊆f' f''proof₂
      ; preablesub = ⊑ₑ-proof₂.preablesub f''proof₂
      ; postablesub = ⊑ₑ-proof₂.postablesub f''proof₂
      ; y⊑post = NbhSys.⊑-trans 𝒷 (⊑ₑ-proof.y⊑post f'proof)
                   (⊑ₑ-proof₂.pf⊑post f''proof₂)
      ; pre⊑x = NbhSys.⊑-trans 𝒶 (⊑ₑ-proof₂.pre⊑pf f''proof₂)
                  (⊑ₑ-proof.pre⊑x f'proof)
      }
  where f'proof = p₁ xy∈f
        f'sub = ⊑ₑ-proof.sub f'proof
        f'subcon = subsetIsCon conf' (⊑ₑ-proof.sub⊆f f'proof)
        f'subpreable = ⊑ₑ-proof.preablesub f'proof
        f'subpostable = ⊑ₑ-proof.postablesub f'proof
        f''proof₂ = Ω f'sub f'' {conf = f'subcon}
                      {preablef = f'subpreable} {f'subpostable}
                      (shrinkExp
                      (⊑ₑ-proof.sub⊆f f'proof)
                      (⊑ₑ-intro₂ preablef' preablef'' p₂))
        f''sub = ⊑ₑ-proof₂.sub f''proof₂

⊑ₑ-trans : ∀ {x y z} → x ⊑ₑ y → y ⊑ₑ z → x ⊑ₑ z
⊑ₑ-trans {x = ⊥ₑ} _ _ = ⊑ₑ-intro₁
⊑ₑ-trans {x = F f _} {⊥ₑ} {⊥ₑ} x⊑y ⊑ₑ-intro₁ = x⊑y
⊑ₑ-trans {x = F f _} {F f' _} {F f'' _} x⊑y y⊑z
  = ⊑ₑ-intro₂ _ _ (⊑ₑ-trans' x⊑y y⊑z)
```

B.0.56 Scwf/DomainScwf/Comprehension/Morphism/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition

module Scwf.DomainScwf.Comprehension.Morphism.AxiomProofs
  (γ : tAppmap Δ Γ) (t : tAppmap Δ [ 𝒶 ]) where
```

```
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.Comprehension.Morphism.Relation

⟨⟩↦-mono : ∀ {x y z} → ⊑ᵥ Δ x y → [⟨ γ , t ⟩] x ↦ z →
                [⟨ γ , t ⟩] y ↦ z
⟨⟩↦-mono {x = x} {y} {⟨⟨ z ,, z ⟩⟩} x⊑y (⟨⟩↦-intro γx↦z tx↦z) =
  ⟨⟩↦-intro (Appmap.↦-mono γ x⊑y γx↦z)
  (Appmap.↦-mono t x⊑y tx↦z)

⟨⟩↦-bottom : ∀ {x} → [⟨ γ , t ⟩] x ↦ ⊥ᵥ
⟨⟩↦-bottom {x} = ⟨⟩↦-intro (Appmap.↦-bottom γ)
                        (Appmap.↦-bottom t)

⟨⟩↦-↓closed : ∀ {x y z} → ⊑ᵥ (𝒜 :: Γ) y z →
                  [⟨ γ , t ⟩] x ↦ z → [⟨ γ , t ⟩] x ↦ y
⟨⟩↦-↓closed {x = x} {⟨⟨ y ,, y ⟩⟩} {⟨⟨ z ,, z ⟩⟩}
  (⊑ᵥ-cons _ y⊑z y⊑z) (⟨⟩↦-intro γx↦z tx↦z)
  = ⟨⟩↦-intro γx↦y tx↦y
    where γx↦y = Appmap.↦-↓closed γ y⊑z γx↦z
          tup-y⊑z = (⊑ᵥ-cons [ 𝒜 ] y⊑z ⊑ᵥ-nil)
          tx↦y = Appmap.↦-↓closed t tup-y⊑z tx↦z

⟨⟩↦-↑directed : ∀ {x y z} → [⟨ γ , t ⟩] x ↦ y →
                    [⟨ γ , t ⟩] x ↦ z →
                    (conyz : ValCon _ y z) →
                    [⟨ γ , t ⟩] x ↦ (y ⊔ᵥ z [ conyz ])
⟨⟩↦-↑directed {x = x} {⟨⟨ y ,, y ⟩⟩} {⟨⟨ z ,, z ⟩⟩}
  (⟨⟩↦-intro γx↦y tx↦y) (⟨⟩↦-intro γx↦z tx↦z)
  (con-tup conyz conyz)
  = ⟨⟩↦-intro γx↦y⊔z tx↦y⊔z
    where γx↦y⊔z = Appmap.↦-↑directed γ γx↦y γx↦z conyz
          tx↦y⊔z = Appmap.↦-↑directed t tx↦y tx↦z (toValCon conyz)

⟨⟩↦-con : ∀ {x y x' y'} → [⟨ γ , t ⟩] x ↦ y →
              [⟨ γ , t ⟩] x' ↦ y' → ValCon _ x x' →
              ValCon _ y y'
⟨⟩↦-con {y = ⟨⟨ y ,, y ⟩⟩} {y' = ⟨⟨ y' ,, y' ⟩⟩}
  (⟨⟩↦-intro γx↦y tx↦y) (⟨⟩↦-intro γx'↦y' tx'↦y') conxx'
  = con-tup conyy' conyy'
  where conyy' = fromValCon (Appmap.↦-con t tx↦y tx'↦y' conxx')
        conyy' = Appmap.↦-con γ γx↦y γx'↦y' conxx'
```

B.0.57 Scwf/DomainScwf/Comprehension/Morphism/Instance

```
{-# OPTIONS --safe #-}
```

```
module Scwf.DomainScwf.Comprehension.Morphism.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Comprehension.Morphism.AxiomProofs
open import Scwf.DomainScwf.Comprehension.Morphism.Relation

⟨_,_⟩ : tAppmap Δ Γ → tAppmap Δ [ 𝒜 ] → tAppmap Δ (𝒜 :: Γ)
Appmap._↦_ ⟨ γ , t ⟩          = ⟨⟩↦ γ t
Appmap.↦-mono ⟨ γ , t ⟩       = ⟨⟩↦-mono γ t
Appmap.↦-bottom ⟨ γ , t ⟩     = ⟨⟩↦-bottom γ t
Appmap.↦-↓closed ⟨ γ , t ⟩    = ⟨⟩↦-↓closed γ t
Appmap.↦-↑directed ⟨ γ , t ⟩  = ⟨⟩↦-↑directed γ t
Appmap.↦-con ⟨ γ , t ⟩        = ⟨⟩↦-con γ t
```

B.0.58 Scwf/DomainScwf/Comprehension/Morphism/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Comprehension.Morphism.Relation where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition

open import Agda.Builtin.Nat

data ⟨⟩↦ (γ : tAppmap Δ Γ) (t : tAppmap Δ [ 𝒜 ]) :
          Valuation Δ → Valuation (𝒜 :: Γ) → Set where
  ⟨⟩↦-intro : ∀ {x y} → [ γ ] x ↦ (ctTail y) →
                [ t ] x ↦ ⟨⟨ ctHead y ⟩⟩ → ⟨⟩↦ γ t x y

-- Some simplifying notation.
[⟨_,_⟩]_↦_ : (γ : tAppmap Δ Γ) → (t : tAppmap Δ [ 𝒜 ]) →
                Valuation Δ → Valuation (𝒜 :: Γ) → Set
[⟨ γ , t ⟩] x ↦ y = ⟨⟩↦ γ t x y
```

B.0.59 Scwf/DomainScwf/Comprehension/p/AxiomProofs

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Comprehension.p.AxiomProofs where

open import Base.Core
open import Base.Variables
```

```
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.AxiomProofs
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.Comprehension.p.Relation

p↦-mono : ∀ {x y z} → ⊑ᵥ (𝒜 :: Γ) x y → x p↦ z → y p↦ z
p↦-mono (⊑ᵥ-cons _ _ x⊑y) (p↦-intro z⊑x)
  = p↦-intro z⊑taily
  where z⊑taily = NbhSys.⊑-trans (ValNbhSys _) z⊑x x⊑y

p↦-bottom : {x : Valuation (𝒜 :: Γ)} → x p↦ ⊥ᵥ
p↦-bottom {x = x} = p↦-intro (NbhSys.⊑-⊥ (ValNbhSys _))

p↦-↓closed : {x : Valuation (𝒜 :: Γ)} → ∀ {y z} →
             ⊑ᵥ Γ y z → x p↦ z → x p↦ y
p↦-↓closed y⊑z (p↦-intro z⊑x)
  = p↦-intro (NbhSys.⊑-trans (ValNbhSys _) y⊑z z⊑x)

p↦-↑directed : {x : Valuation (𝒜 :: Γ)} → ∀ {y z} →
               x p↦ y → x p↦ z →
               (conyz : ValCon _ y z) →
               x p↦ (y ⊔ᵥ z [ conyz ])
p↦-↑directed (p↦-intro y⊑x) (p↦-intro z⊑x) conyz
  = p↦-intro y⊔z⊑tailx
  where y⊔z⊑tailx = NbhSys.⊑-⊔ (ValNbhSys _) y⊑x z⊑x conyz

p↦-con : {x : Valuation (𝒜 :: Γ)} → ∀ {y x' y'} →
         x p↦ y → x' p↦ y' →
         ValCon _ x x' → ValCon _ y y'
p↦-con (p↦-intro y⊑x) (p↦-intro y'⊑x') (con-tup _ conxx')
  = Con-⊔ᵥ y⊑x⊔x' y'⊑x⊔x'
  where y⊑x⊔x' = ⊑-⊔-lemma₄ (ValNbhSys _) y⊑x conxx'
        y'⊑x⊔x' = ⊑-⊔-lemma₅ (ValNbhSys _) y'⊑x' conxx'
```

B.0.60 Scwf/DomainScwf/Comprehension/p/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Comprehension.p.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Comprehension.p.AxiomProofs
```

```
open import Scwf.DomainScwf.Comprehension.p.Relation

p : (Γ : Ctx n) → (𝒜 : Ty) → tAppmap (𝒜 :: Γ) Γ
Appmap._↦_ (p Γ 𝒜)          = _p↦_
Appmap.↦-mono (p Γ 𝒜)       = p↦-mono
Appmap.↦-bottom (p Γ 𝒜)     = p↦-bottom
Appmap.↦-↓closed (p Γ 𝒜)    = p↦-↓closed
Appmap.↦-↑directed (p Γ 𝒜)  = p↦-↑directed
Appmap.↦-con (p Γ 𝒜)        = p↦-con
```

B.0.61 Scwf/DomainScwf/Comprehension/p/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Comprehension.p.Relation where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Relation

data _p↦_ : Valuation (𝒜 :: Γ) → Valuation Γ → Set where
  p↦-intro : {x : Valuation (𝒜 :: Γ)} → ∀ {y} →
             ⊑ᵥ Γ y (ctTail x) → x p↦ y
```

B.0.62 Scwf/DomainScwf/Comprehension/q/AxiomProofs

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Comprehension.q.AxiomProofs where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.Comprehension.q.Relation

q↦-mono : ∀ {x y} → {z : Valuation [ 𝒜 ]} →
          ⊑ᵥ (𝒜 :: Γ) x y → x q↦ z →
          y q↦ z
q↦-mono {𝒜} (⊑ᵥ-cons _ x⊑y _) (q↦-intro z⊑x)
  = q↦-intro (NbhSys.⊑-trans 𝒜 z⊑x x⊑y)

q↦-bottom : {x : Valuation (𝒜 :: Γ)} → x q↦ ⊥ᵥ
```

```
q↦-bottom {𝒜 = 𝒜} = q↦-intro (NbhSys.⊑-⊥ 𝒜)

q↦-↓closed : {x : Valuation (𝒜 :: Γ)} → ∀ {y z} →
             ⊑ᵥ [ 𝒜 ] y z → x q↦ z → x q↦ y
q↦-↓closed {𝒜 = 𝒜} (⊑ᵥ-cons _ y⊑z _) (q↦-intro z⊑x)
  = q↦-intro (NbhSys.⊑-trans 𝒜 y⊑z z⊑x)

q↦-↑directed : {x : Valuation (𝒜 :: Γ)} → ∀ {y z} →
               x q↦ y → x q↦ z → ∀ conyz →
               x q↦ (y ⊔ᵥ z [ conyz ])
q↦-↑directed {𝒜 = 𝒜} {x = ⟨⟨ x ,, x ⟩⟩} {⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩} {⟨⟨ z ,, ⟨⟨⟩⟩ ⟩⟩}
  (q↦-intro y⊑x) (q↦-intro z⊑x) (con-tup conyz con-nil)
  = q↦-intro y⊔z⊑x
  where y⊔z⊑x = NbhSys.⊑-⊔ 𝒜 y⊑x z⊑x conyz

q↦-con : {x : Valuation (𝒜 :: Γ)} → ∀ {y x' y'} →
         x q↦ y → x' q↦ y' →
         ValCon _ x x' → ValCon _ y y'
q↦-con {𝒜 = 𝒜} {y = ⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩} {y' = ⟨⟨ y' ,, ⟨⟨⟩⟩ ⟩⟩}
  (q↦-intro y⊑x) (q↦-intro y'⊑x') (con-tup conxx' conxx')
  = NbhSys.Con-⊔ (ValNbhSys _) {z = ⟨⟨ [ 𝒜 ] _ ⊔ _ [ conxx' ] ⟩⟩}
    y⊑x⊔x'ᵥ y'⊑x⊔x'ᵥ
  where y⊑x⊔x' = ⊑-⊔-lemma₄ 𝒜 y⊑x conxx'
        y⊑x⊔x'ᵥ = ⊑ᵥ-cons [ 𝒜 ] y⊑x⊔x' ⊑ᵥ-nil
        y'⊑x⊔x' = ⊑-⊔-lemma₅ 𝒜 y'⊑x' conxx'
        y'⊑x⊔x'ᵥ = ⊑ᵥ-cons [ 𝒜 ] y'⊑x⊔x' ⊑ᵥ-nil
```

B.0.63 Scwf/DomainScwf/Comprehension/q/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Comprehension.q.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Comprehension.q.AxiomProofs
open import Scwf.DomainScwf.Comprehension.q.Relation

q : (Γ : Ctx n) → (𝒜 : Ty) → tAppmap (𝒜 :: Γ) [ 𝒜 ]
Appmap._↦_ (q Γ 𝒜)         = _q↦_
Appmap.↦-mono (q Γ 𝒜)      = q↦-mono
Appmap.↦-bottom (q Γ 𝒜)    = q↦-bottom
Appmap.↦-↓closed (q Γ 𝒜)   = q↦-↓closed
Appmap.↦-↑directed (q Γ 𝒜) = q↦-↑directed
Appmap.↦-con (q Γ 𝒜)       = q↦-con
```

B.0.64 Scwf/DomainScwf/Comprehension/q/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.Comprehension.q.Relation where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition

data _q↦_ : Valuation (𝒜 :: Γ) → Valuation [ 𝒜 ] → Set where
  q↦-intro : {x : Valuation (𝒜 :: Γ)} →
             {y : Valuation [ 𝒜 ]} →
             [ 𝒜 ] (ctHead y) ⊑ (ctHead x) → x q↦ y
```

B.0.65 Scwf/DomainScwf/ProductStructure/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core

module Scwf.DomainScwf.ProductStructure.AxiomProofs (𝒜 ℬ : Ty) where

open import Appmap.Equivalence
open import Base.Variables hiding (𝒜 ; ℬ)
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Instance
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ProductStructure.fst.Instance
open import Scwf.DomainScwf.ProductStructure.fst.Relation
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
  renaming (<_,_> to ⟨_,_⟩)
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance
open import Scwf.DomainScwf.ProductStructure.Pair.Instance
open import Scwf.DomainScwf.ProductStructure.Pair.Relation
open import Scwf.DomainScwf.ProductStructure.snd.Instance
open import Scwf.DomainScwf.ProductStructure.snd.Relation
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Instance

private
```

```
  variable
    t t' : tAppmap Γ [ 𝒜 ]
    u u' : tAppmap Γ [ ℬ ]
    v v' : tAppmap Γ [ 𝒜 × ℬ ]

fstAxiomLemma₁ : ∀ {x y} → [ fst < t , u > ] x ↦ y →
                           [ t ] x ↦ y
fstAxiomLemma₁ {t = t} (fst-intro₁ y⊑⊥)
  = Appmap.↦-↓closed t tup-y⊑⊥ (Appmap.↦-bottom t)
  where tup-y⊑⊥ = ⊑ᵥ-cons [ 𝒜 ] y⊑⊥ ⊑ᵥ-nil
fstAxiomLemma₁ (fst-intro₂ (<>↦-intro₂ tx↦y₁ _))
  = tx↦y₁

fstAxiomLemma₂ : ∀ {x y} → [ t ] x ↦ y →
                           [ fst < t , u > ] x ↦ y
fstAxiomLemma₂ {u = u} {y = ⟨ y₁ ,, ⟨⟩ ⟩} tx↦y₁
  = fst-intro₂ ⟨⟩x↦y₁⊥
  where ux↦⊥ = Appmap.↦-bottom u
        ⟨⟩x↦y₁⊥ = <>↦-intro₂ tx↦y₁ ux↦⊥

fstAxiom : fst < t , u > ≈ t
fstAxiom = ≈-intro (≼-intro fstAxiomLemma₁)
                   (≼-intro fstAxiomLemma₂)

sndAxiomLemma₁ : ∀ {x y} → [ snd < t , u > ] x ↦ y →
                           [ u ] x ↦ y
sndAxiomLemma₁ {u = u} (snd-intro₁ y⊑⊥)
  = Appmap.↦-↓closed u tup-y⊑⊥ (Appmap.↦-bottom u)
  where tup-y⊑⊥ = ⊑ᵥ-cons [ ℬ ] y⊑⊥ ⊑ᵥ-nil
sndAxiomLemma₁ (snd-intro₂ (<>↦-intro₂ _ ux↦y₂))
  = ux↦y₂

sndAxiomLemma₂ : ∀ {x y} → [ u ] x ↦ y →
                           [ snd < t , u > ] x ↦ y
sndAxiomLemma₂ {t = t} {y = ⟨ y₁ ,, ⟨⟩ ⟩} tx↦y₁
  = snd-intro₂ ⟨⟩x↦⊥y₁
  where tx↦⊥ = Appmap.↦-bottom t
        ⟨⟩x↦⊥y₁ = <>↦-intro₂ tx↦⊥ tx↦y₁

sndAxiom : snd < t , u > ≈ u
sndAxiom = ≈-intro (≼-intro sndAxiomLemma₁)
                   (≼-intro sndAxiomLemma₂)

pairSubLemma₁ : {γ : tAppmap Δ Γ} → ∀ {x y} →
                  [ < t , u > ∘ γ ] x ↦ y →
                  [ < t ∘ γ , u ∘ γ > ] x ↦ y
pairSubLemma₁ (∘↦-intro _ <>↦-intro₁)
  = <>↦-intro₁
```

```
pairSubLemma₁ (∘↦-intro tx↦z (<>↦-intro₂ tz↦y₁ uz↦y₂))
  = <>↦-intro₂ (∘↦-intro tx↦z tz↦y₁) (∘↦-intro tx↦z uz↦y₂)

pairSubLemma₂ : {γ : tAppmap Δ Γ} → ∀ {x y} →
                  [ < t ∘ γ , u ∘ γ > ] x ↦ y →
                  [ < t , u > ∘ γ ] x ↦ y
pairSubLemma₂ {γ = γ} <>↦-intro₁
  = ∘↦-intro (Appmap.↦-bottom γ) <>↦-intro₁
pairSubLemma₂ {t = t} {u = u} {γ}
  (<>↦-intro₂ (∘↦-intro γx↦z tz↦y₁) (∘↦-intro γx↦w uw↦y₂))
  = ∘↦-intro γx↦z⊔w zw↦y₁y₂
  where conzw = Appmap.↦-con γ γx↦z γx↦w valConRefl
        γx↦z⊔w = Appmap.↦-↑directed γ γx↦z γx↦w conzw
        z⊑z⊔w = NbhSys.⊑-⊔-fst (ValNbhSys _) conzw
        tz⊔w↦y₁ = Appmap.↦-mono t z⊑z⊔w tz↦y₁
        w⊑z⊔w = NbhSys.⊑-⊔-snd (ValNbhSys _) conzw
        uz⊔w↦y₂ = Appmap.↦-mono u w⊑z⊔w uw↦y₂
        zw↦y₁y₂ = <>↦-intro₂ tz⊔w↦y₁ uz⊔w↦y₂

pairSub : {γ : tAppmap Δ Γ} →
            (< t , u > ∘ γ) ≈ < (t ∘ γ) , (u ∘ γ) >
pairSub = ≈-intro (≲-intro pairSubLemma₁)
                  (≲-intro pairSubLemma₂)

fstCongLemma₁ : v ≈ v' → ∀ {x y} → [ fst v ] x ↦ y →
                  [ fst v' ] x ↦ y
fstCongLemma₁ _ (fst-intro₁ y⊑⊥)
  = fst-intro₁ y⊑⊥
fstCongLemma₁ (≈-intro (≲-intro p) _) (fst-intro₂ vx↦y₁y₂)
  = fst-intro₂ (p vx↦y₁y₂)

fstCong : v ≈ v' → fst v ≈ fst v'
fstCong v≈v'
  = ≈-intro (≲-intro (fstCongLemma₁ v≈v')) fst'≲fst
  where fst'≲fst = ≲-intro (fstCongLemma₁ (≈Symmetric v≈v'))

sndCongLemma₁ : v ≈ v' → ∀ {x y} → [ snd v ] x ↦ y →
                  [ snd v' ] x ↦ y
sndCongLemma₁ _ (snd-intro₁ y⊑⊥)
  = snd-intro₁ y⊑⊥
sndCongLemma₁ (≈-intro (≲-intro p) _) (snd-intro₂ vx↦y₁y₂)
  = snd-intro₂ (p vx↦y₁y₂)

sndCong : v ≈ v' → snd v ≈ snd v'
sndCong v≈v'
  = ≈-intro (≲-intro (sndCongLemma₁ v≈v')) snd'≲snd
  where snd'≲snd = ≲-intro (sndCongLemma₁ (≈Symmetric v≈v'))
```

```
pairCongLemma₁ : t ≈ t' → u ≈ u' →
                 {x : Valuation Γ} → ∀ {y} →
                 [ < t , u > ] x ↦ y →
                 [ < t' , u' > ] x ↦ y
pairCongLemma₁ _ _ <>↦-intro₁ = <>↦-intro₁
pairCongLemma₁ (≈-intro (≼-intro p₁) _)
  (≈-intro (≼-intro p₂) _) (<>↦-intro₂ tx↦y₁ ux↦y₂)
  = <>↦-intro₂ (p₁ tx↦y₁) (p₂ ux↦y₂)

pairCong : t ≈ t' → u ≈ u' → < t , u > ≈ < t' , u' >
pairCong t≈t' u≈u'
  = ≈-intro (≼-intro (pairCongLemma₁ t≈t' u≈u')) pair'≼pair
  where pair'≼pair = ≼-intro (pairCongLemma₁
                     (≈Symmetric t≈t') (≈Symmetric u≈u'))
```

B.0.66 Scwf/DomainScwf/ProductStructure/fst/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance

module Scwf.DomainScwf.ProductStructure.fst.AxiomProofs
  (t : tAppmap Γ [ 𝒜 × ℬ ]) where

open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Relation
open import Scwf.DomainScwf.ProductStructure.fst.Relation

fst↦-mono : ∀ {x y z} → ⊑ᵥ Γ x y → fst↦ t x z →
            fst↦ t y z
fst↦-mono {y = y} _ (fst-intro₁ z⊑⊥) =
  fst-intro₁ z⊑⊥
fst↦-mono {y = y} x⊑y (fst-intro₂ tx↦z₁z₂)
  = fst-intro₂ ty↦z₁z₂
    where ty↦z₁z₂ = Appmap.↦-mono t x⊑y tx↦z₁z₂

fst↦-bottom : ∀ {x} → fst↦ t x ⟨⟨ NbhSys.⊥ 𝒜 ⟩⟩
fst↦-bottom {x = x} = fst-intro₁ (NbhSys.⊑-refl 𝒜)
```

```
fst↦-↓closed : ∀ {x y z} → ⊑ᵥ [ 𝒜 ] y z → fst↦ t x z →
                           fst↦ t x y
fst↦-↓closed {x = x} {⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ y⊑z ⊑ᵥ-nil) (fst-intro₁ z⊑⊥)
  = fst-intro₁ (NbhSys.⊑-trans 𝒜 y⊑z z⊑⊥)
fst↦-↓closed {x = x} {⟨⟨ y ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ y⊑z₁ ⊑ᵥ-nil) (fst-intro₂ tx↦z₁z₂)
  = fst-intro₂ tx↦yz₂
  where yz₂⊑z₁z₂' = ⊑ₓ-intro₂ y⊑z₁ (NbhSys.⊑-refl ℬ)
        yz₂⊑z₁z₂ = ⊑ᵥ-cons [ 𝒜 × ℬ ] yz₂⊑z₁z₂' ⊑ᵥ-nil
        tx↦yz₂ = Appmap.↦-↓closed t yz₂⊑z₁z₂ tx↦z₁z₂

fst↦-↑directed : ∀ {x y z} → fst↦ t x y → fst↦ t x z →
                 (con : ValCon [ 𝒜 ] y z) → fst↦ t x (y ⊔ᵥ z [ con ])
fst↦-↑directed (fst-intro₁ y⊑⊥) (fst-intro₁ z⊑⊥)
  (con-tup conyz _)
  = fst-intro₁ y⊔z⊑⊥
  where y⊔z⊑⊥ = NbhSys.⊑-⊔ 𝒜 y⊑⊥ z⊑⊥ conyz
fst↦-↑directed (fst-intro₂ tx↦y₁y₂)
  (fst-intro₁ z⊑⊥) (con-tup cony₁z _)
  = fst-intro₂ tx↦y₁⊔zy₂
  where z⊑y₁ = NbhSys.⊑-trans 𝒜 z⊑⊥ (NbhSys.⊑-⊥ 𝒜)
        y₁⊔z⊑y = NbhSys.⊑-⊔ 𝒜 (NbhSys.⊑-refl 𝒜) z⊑y₁ cony₁z
        y₁⊔zy₂⊑y₁y₂' = ⊑ₓ-intro₂ y₁⊔z⊑y (NbhSys.⊑-refl ℬ)
        y₁⊔zy₂⊑y₁y₂ = ⊑ᵥ-cons [ 𝒜 × ℬ ] y₁⊔zy₂⊑y₁y₂' ⊑ᵥ-nil
        tx↦y₁⊔zy₂ = Appmap.↦-↓closed t y₁⊔zy₂⊑y₁y₂ tx↦y₁y₂
fst↦-↑directed {x = x} (fst-intro₁ y⊑⊥)
  (fst-intro₂ tx↦z₁z₂) (con-tup conyz₁ _)
  = fst-intro₂ tx↦y⊔z₁z₂
  where y⊑z₁ = NbhSys.⊑-trans 𝒜 y⊑⊥ (NbhSys.⊑-⊥ 𝒜)
        y⊔z₁⊑z₂ = NbhSys.⊑-⊔ 𝒜 y⊑z₁ (NbhSys.⊑-refl 𝒜) conyz₁
        y⊔z₁z₂⊑z₁z₂' = ⊑ₓ-intro₂ y⊔z₁⊑z₂ (NbhSys.⊑-refl ℬ)
        y⊔z₁z₂⊑z₁z₂ = ⊑ᵥ-cons [ 𝒜 × ℬ ] y⊔z₁z₂⊑z₁z₂' ⊑ᵥ-nil
        tx↦y⊔z₁z₂ = Appmap.↦-↓closed t y⊔z₁z₂⊑z₁z₂ tx↦z₁z₂
fst↦-↑directed {x = x} (fst-intro₂ tx↦y₁y₂)
  (fst-intro₂ tx↦z₁z₂) (con-tup _ _)
  with (Appmap.↦-con t tx↦y₁y₂ tx↦z₁z₂ valConRefl)
... | con-tup (con-pair cony₁z₁ cony₂z₂) _
  = fst-intro₂ tx↦⊔
  where tx↦⊔ = Appmap.↦-↑directed t tx↦y₁y₂ tx↦z₁z₂
                 (con-tup (con-pair _ cony₂z₂) con-nil)

fst↦-con : ∀ {x y x' y'} → fst↦ t x y → fst↦ t x' y' →
           ValCon Γ x x' → ValCon [ 𝒜 ] y y'
fst↦-con (fst-intro₁ y⊑⊥) (fst-intro₁ y'⊑⊥) _
  = toValCon (NbhSys.Con-⊔ 𝒜 y⊑⊥ y'⊑⊥)
fst↦-con (fst-intro₁ y⊑⊥) (fst-intro₂ _) _
```

```
    = toValCon (NbhSys.Con-⊔ 𝒜 y⊑y'₁ (NbhSys.⊑-refl 𝒜))
    where y⊑y'₁ = NbhSys.⊑-trans 𝒜 y⊑⊥ (NbhSys.⊑-⊥ 𝒜)
fst↦-con (fst-intro₂ _) (fst-intro₁ y'⊑⊥) _
    = toValCon (NbhSys.Con-⊔ 𝒜 (NbhSys.⊑-refl 𝒜) y'₁⊑y)
    where y'₁⊑y = NbhSys.⊑-trans 𝒜 y'⊑⊥ (NbhSys.⊑-⊥ 𝒜)
fst↦-con (fst-intro₂ tx↦y₁y₂)
    (fst-intro₂ tx'↦y'₁y'₂) con
    with (Appmap.↦-con t tx↦y₁y₂ tx'↦y'₁y'₂ con)
... | con-tup (con-pair cony₁y₂ _) _ = toValCon cony₁y₂
```

B.0.67 Scwf/DomainScwf/ProductStructure/fst/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.fst.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ProductStructure.fst.AxiomProofs
open import Scwf.DomainScwf.ProductStructure.fst.Relation
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance

fst : tAppmap Γ [ 𝒜 × ℬ ] → tAppmap Γ [ 𝒜 ]
Appmap._↦_ (fst t)          = fst↦ t
Appmap.↦-mono (fst t)       = fst↦-mono t
Appmap.↦-bottom (fst t)     = fst↦-bottom t
Appmap.↦-↓closed (fst t)    = fst↦-↓closed t
Appmap.↦-↑directed (fst t)  = fst↦-↑directed t
Appmap.↦-con (fst t)        = fst↦-con t
```

B.0.68 Scwf/DomainScwf/ProductStructure/fst/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.fst.Relation where

open import Appmap.Definition
open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance

data fst↦ (t : tAppmap Γ [ 𝒜 × ℬ ]) :
```

```
          Valuation Γ → Valuation [ 𝒜 ] → Set where
  fst-intro₁ : ∀ {x y} → [ 𝒜 ] y ⊑ NbhSys.⊥ 𝒜 → fst↦ t x ⟨⟨ y ⟩⟩
  fst-intro₂ : ∀ {x y₁ y₂} → [ t ] x ↦ ⟨⟨ < y₁ , y₂ > ⟩⟩ →
               fst↦ t x ⟨⟨ y₁ ⟩⟩
```

B.0.69 Scwf/DomainScwf/ProductStructure/NbhSys/AxiomProofs

```
{-# OPTIONS --safe #-}

open import NbhSys.Definition

module Scwf.DomainScwf.ProductStructure.NbhSys.AxiomProofs
  (D D' : NbhSys) where

open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition D D'
open import Scwf.DomainScwf.ProductStructure.NbhSys.Relation D D'

private
  variable
    x y z : ProdNbh

Con-⊔ₓ : x ⊑ₓ z → y ⊑ₓ z → ProdCon x y
Con-⊔ₓ {⊥ₓ} {⊥ₓ} _ _ = conₓ-⊥₂
Con-⊔ₓ {⊥ₓ} {< y₁ , y₂ >} _ _ = conₓ-⊥₂
Con-⊔ₓ {< x₁ , x₂ >} {y = ⊥ₓ} _ _ = conₓ-⊥₁
Con-⊔ₓ {< x₁ , x₂ >} {< z₁ , z₂ >} {< y₁ , y₂ >}
  (⊑ₓ-intro₂ x₁⊑z₁ x₂⊑z₂) (⊑ₓ-intro₂ y₁⊑z₁ y₂⊑z₂)
  = con-pair (NbhSys.Con-⊔ D x₁⊑z₁ y₁⊑z₁) (NbhSys.Con-⊔ D' x₂⊑z₂ y₂⊑z₂)

⊑ₓ-refl : x ⊑ₓ x
⊑ₓ-refl {x = ⊥ₓ} = ⊑ₓ-intro₁
⊑ₓ-refl {x = < x₁ , x₂ >}
  = ⊑ₓ-intro₂ (NbhSys.⊑-refl D) (NbhSys.⊑-refl D')

⊑ₓ-trans : x ⊑ₓ y → y ⊑ₓ z → x ⊑ₓ z
⊑ₓ-trans {z = z} ⊑ₓ-intro₁ ⊑ₓ-intro₁ = ⊑ₓ-intro₁
⊑ₓ-trans ⊑ₓ-intro₁ (⊑ₓ-intro₂ _ _) = ⊑ₓ-intro₁
⊑ₓ-trans (⊑ₓ-intro₂ x₁⊑y₁ x₂⊑y₂)
  (⊑ₓ-intro₂ y₁⊑z₁ y₂⊑z₂)
  = ⊑ₓ-intro₂ (NbhSys.⊑-trans D x₁⊑y₁ y₁⊑z₁)
    (NbhSys.⊑-trans D' x₂⊑y₂ y₂⊑z₂)

⊑ₓ-⊥ : ⊥ₓ ⊑ₓ x
⊑ₓ-⊥ {x = x} = ⊑ₓ-intro₁

⊑ₓ-⊔ : y ⊑ₓ x → z ⊑ₓ x → (con : ProdCon y z) → (y ⊔ₓ z [ con ]) ⊑ₓ x
⊑ₓ-⊔ {⊥ₓ} {x} {< z₁ , z₂ >} _ z⊑x _ = z⊑x
⊑ₓ-⊔ {⊥ₓ} {x} {⊥ₓ} _ _ - = ⊑ₓ-intro₁
```

```
⊑ₓ-⊔ {< y₁ , y₂ >} {x} {⊥ₓ} y⊑x _ - = y⊑x
⊑ₓ-⊔ {< y₁ , y₂ >} {x} {< z₁ , z₂ >}
  (⊑ₓ-intro₂ y₁⊑w₁ y₂⊑w₂)
  (⊑ₓ-intro₂ z₁⊑w₁ z₂⊑w₂) (con-pair cony₁z₁ cony₂z₂)
  = ⊑ₓ-intro₂ y₁⊔z₁⊑w₁ y₂⊔z₂⊑w₂
  where y₁⊔z₁⊑w₁ = NbhSys.⊑-⊔ D y₁⊑w₁ z₁⊑w₁ cony₁z₁
        y₂⊔z₂⊑w₂ = NbhSys.⊑-⊔ D' y₂⊑w₂ z₂⊑w₂ cony₂z₂

⊑ₓ-⊔-fst : (con : ProdCon x y) → x ⊑ₓ (x ⊔ₓ y [ con ])
⊑ₓ-⊔-fst {⊥ₓ} {_} _ = ⊑ₓ-intro₁
⊑ₓ-⊔-fst {< x₁ , y₁ >} {⊥ₓ} _ =
  ⊑ₓ-intro₂ (NbhSys.⊑-refl D) ((NbhSys.⊑-refl D'))
⊑ₓ-⊔-fst {< x₁ , y₁ >} {< x₂ , y₂ >} (con-pair conx₁x₂ cony₁y₂) =
  ⊑ₓ-intro₂ x₁⊑x₁⊔x₂ y₁⊑y₁⊔y₂
  where x₁⊑x₁⊔x₂ = NbhSys.⊑-⊔-fst D conx₁x₂
        y₁⊑y₁⊔y₂ = NbhSys.⊑-⊔-fst D' cony₁y₂

⊑ₓ-⊔-snd : (con : ProdCon x y) → y ⊑ₓ (x ⊔ₓ y [ con ])
⊑ₓ-⊔-snd {y = ⊥ₓ} _ = ⊑ₓ-intro₁
⊑ₓ-⊔-snd {⊥ₓ} {< x₂ , y₂ >} _ =
  ⊑ₓ-intro₂ (NbhSys.⊑-refl D) ((NbhSys.⊑-refl D'))
⊑ₓ-⊔-snd {< x₁ , y₁ >} {< x₂ , y₂ >} (con-pair conx₁x₂ cony₁y₂) =
  ⊑ₓ-intro₂ x₂⊑x₁⊔x₂ y₂⊑y₁⊔y₂
  where x₂⊑x₁⊔x₂ = NbhSys.⊑-⊔-snd D conx₁x₂
        y₂⊑y₁⊔y₂ = NbhSys.⊑-⊔-snd D' cony₁y₂
```

B.0.70 Scwf/DomainScwf/ProductStructure/NbhSys/Definition

```
{-# OPTIONS --safe #-}

open import NbhSys.Definition

module Scwf.DomainScwf.ProductStructure.NbhSys.Definition
  (D D' : NbhSys) where

data ProdNbh : Set where
  ⊥ₓ : ProdNbh
  <_,_> : NbhSys.Nbh D → NbhSys.Nbh D' → ProdNbh

data ProdCon : ProdNbh → ProdNbh → Set where
  conₓ-⊥₁ : ∀ {x} → ProdCon x ⊥ₓ
  conₓ-⊥₂ : ∀ {x} → ProdCon ⊥ₓ x
  con-pair : ∀ {x₁ x₂ x'₁ x'₂} → NbhSys.Con D x₁ x'₁ → NbhSys.Con D' x₂ x'₂ →
             ProdCon < x₁ , x₂ > < x'₁ , x'₂ >

_⊔ₓ_[_] : (x : ProdNbh) → (y : ProdNbh) → (con : ProdCon x y) →
  ProdNbh
```

```
⊥x ⊔x ⊥x [ _ ] = ⊥x
⊥x ⊔x < x1 , x2 > [ _ ] = < x1 , x2 >
< x1 , x2 > ⊔x ⊥x [ _ ] = < x1 , x2 >
< x1 , x2 > ⊔x < x'1 , x'2 > [ con-pair conx1x'1 conx2x'2 ]
  = < [ D ] x1 ⊔ x'1 [ conx1x'1 ] , [ D' ] x2 ⊔ x'2 [ conx2x'2 ] >
```

B.0.71 Scwf/DomainScwf/ProductStructure/NbhSys/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.NbhSys.Instance where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.AxiomProofs
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Relation

_×_ : NbhSys → NbhSys → NbhSys
NbhSys.Nbh (d1 × d2)     = ProdNbh d1 d2
NbhSys._⊑_ (d1 × d2)     = _⊑x_ d1 d2
NbhSys.Con (d1 × d2)     = ProdCon d1 d2
NbhSys._⊔_[_] (d1 × d2) = _⊔x_[_] d1 d2
NbhSys.⊥ (d1 × d2)       = ⊥x
NbhSys.Con-⊔ (d1 × d2)  = Con-⊔x d1 d2
NbhSys.⊑-refl (d1 × d2)  = ⊑x-refl d1 d2
NbhSys.⊑-trans (d1 × d2) = ⊑x-trans d1 d2
NbhSys.⊑-⊥ (d1 × d2)     = ⊑x-⊥ d1 d2
NbhSys.⊑-⊔ (d1 × d2)     = ⊑x-⊔ d1 d2
NbhSys.⊑-⊔-fst (d1 × d2) = ⊑x-⊔-fst d1 d2
NbhSys.⊑-⊔-snd (d1 × d2) = ⊑x-⊔-snd d1 d2
```

B.0.72 Scwf/DomainScwf/ProductStructure/NbhSys/Relation

```
{-# OPTIONS --safe #-}

open import NbhSys.Definition

module Scwf.DomainScwf.ProductStructure.NbhSys.Relation
  (D D' : NbhSys) where

open import Base.Core
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition D D'

data _⊑x_ : ProdNbh → ProdNbh → Set where
  ⊑x-intro1 : ∀ {x} → ⊥x ⊑x x
  ⊑x-intro2 : ∀ {x y x' y'} → [ D ] x ⊑ y →
                [ D' ] x' ⊑ y' →
                < x , x' > ⊑x < y , y' >
```

B.0.73 Scwf/DomainScwf/ProductStructure/Pair/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition

module Scwf.DomainScwf.ProductStructure.Pair.AxiomProofs
  (t : tAppmap Γ [ 𝒜 ])
  (u : tAppmap Γ [ ℬ ]) where

open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance
open import Scwf.DomainScwf.ProductStructure.NbhSys.Relation
open import Scwf.DomainScwf.ProductStructure.Pair.Relation

<>↦-mono : ∀ {x y z} → ⊑ᵥ Γ x y → <>↦ t u x z →
                  <>↦ t u y z
<>↦-mono {y = y} x⊑y <>↦-intro₁ = <>↦-intro₁
<>↦-mono {y = y} x⊑y (<>↦-intro₂ tx↦z₁ ux↦z₂)
  = <>↦-intro₂ ty↦z₁ uy↦z₂
  where ty↦z₁ = Appmap.↦-mono t x⊑y tx↦z₁
        uy↦z₂ = Appmap.↦-mono u x⊑y ux↦z₂

<>↦-bottom : ∀ {x} → <>↦ t u x ⟨⟨ NbhSys.⊥ (𝒜 × ℬ) ⟩⟩
<>↦-bottom = <>↦-intro₁

<>↦-↓closed : ∀ {x y z} → ⊑ᵥ [ 𝒜 × ℬ ] y z →
                    <>↦ t u x z → <>↦ t u x y
<>↦-↓closed {y = ⟨⟨ < y₁ , y₂ > ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ () ⊑ᵥ-nil) <>↦-intro₁
<>↦-↓closed {y = ⟨⟨ ⊥ₓ ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ _ ⊑ᵥ-nil) <>↦-intro₁
  = <>↦-intro₁
<>↦-↓closed {y = ⟨⟨ ⊥ₓ ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ _ ⊑ᵥ-nil) (<>↦-intro₂ _ _)
  = <>↦-intro₁
<>↦-↓closed {x = x} {⟨⟨ < y₁ , y₂ > ,, ⟨⟨⟩⟩ ⟩⟩}
  (⊑ᵥ-cons _ (⊑ₓ-intro₂ y₁⊑z₁ y₂⊑z₂) ⊑ᵥ-nil)
  (<>↦-intro₂ tx↦y₁ ux↦y₂)
  = <>↦-intro₂ ttx↦y₁ tux↦y₂
  where ty₁⊑z₁ = ⊑ᵥ-cons [ 𝒜 ] y₁⊑z₁ ⊑ᵥ-nil
        ttx↦y₁ = Appmap.↦-↓closed t ty₁⊑z₁ tx↦y₁
        ty₂⊑z₂ = ⊑ᵥ-cons [ ℬ ] y₂⊑z₂ ⊑ᵥ-nil
```

```
        tux↦y₂ = Appmap.↦-↓closed u ty₂⊑z₂ ux↦y₂

<>↦-↑directed : ∀ {x y z} → <>↦ t u x y → <>↦ t u x z →
                  (con : ValCon [ 𝒜 × ℬ ] y z) →
                  <>↦ t u x (y ⊔ᵥ z [ con ])
<>↦-↑directed <>↦-intro₁ <>↦-intro₁ (con-tup _ _)
  = <>↦-intro₁
<>↦-↑directed {x = x} <>↦-intro₁
  (<>↦-intro₂ tx↦z₁ ux↦z₂) (con-tup _ _)
  = <>↦-intro₂ tx↦z₁ ux↦z₂
<>↦-↑directed {x = x} (<>↦-intro₂ tx↦y₁ ux↦y₂)
  <>↦-intro₁ (con-tup _ _)
  = <>↦-intro₂ tx↦y₁ ux↦y₂
<>↦-↑directed {x = x} (<>↦-intro₂ tx↦y₁ ux↦y₂)
  (<>↦-intro₂ tx↦z₁ ux↦z₂)
  (con-tup (con-pair cony₁z₁ cony₂z₂) _)
  = <>↦-intro₂ tx↦y₁⊔z₁ ux↦y₂⊔z₂
  where tx↦y₁⊔z₁ = Appmap.↦-↑directed t tx↦y₁ tx↦z₁ (toValCon cony₁z₁)
        ux↦y₂⊔z₂ = Appmap.↦-↑directed u ux↦y₂ ux↦z₂ (toValCon cony₂z₂)

<>↦-con : ∀ {x y x' y'} → <>↦ t u x y → <>↦ t u x' y' →
            ValCon Γ x x' → ValCon [ 𝒜 × ℬ ] y y'
<>↦-con <>↦-intro₁ <>↦-intro₁ _
  = con-tup conₓ-⊥₁ con-nil
<>↦-con <>↦-intro₁ (<>↦-intro₂ _ _) _
  = con-tup conₓ-⊥₂ con-nil
<>↦-con (<>↦-intro₂ _ _) <>↦-intro₁ _
  = con-tup conₓ-⊥₁ con-nil
<>↦-con (<>↦-intro₂ tx↦y₁ ux↦y₂) (<>↦-intro₂ tx↦y₃ ux↦y₄) conxx'
  = con-tup cony₁y₂y₃y₄ con-nil
  where cony₁y₂ = fromValCon (Appmap.↦-con t tx↦y₁ tx↦y₃ conxx')
        cony₃y₄ = fromValCon (Appmap.↦-con u ux↦y₂ ux↦y₄ conxx')
        cony₁y₂y₃y₄ = con-pair cony₁y₂ cony₃y₄
```

B.0.74 Scwf/DomainScwf/ProductStructure/Pair/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Pair.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance
open import Scwf.DomainScwf.ProductStructure.Pair.AxiomProofs
open import Scwf.DomainScwf.ProductStructure.Pair.Relation
```

```
<_,_> : tAppmap Γ [ 𝒜 ] → tAppmap Γ [ ℬ ] → tAppmap Γ [ 𝒜 × ℬ ]
Appmap._↦_ < t , u >        = <>↦ t u
Appmap.↦-mono < t , u >     = <>↦-mono t u
Appmap.↦-bottom < t , u >   = <>↦-bottom t u
Appmap.↦-↓closed < t , u >  = <>↦-↓closed t u
Appmap.↦-↑directed < t , u > = <>↦-↑directed t u
Appmap.↦-con < t , u >      = <>↦-con t u
```

B.0.75 Scwf/DomainScwf/ProductStructure/Pair/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Pair.Relation where

open import Base.Core
open import Base.Variables
open import Appmap.Definition
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance

open import Agda.Builtin.Sigma

data <>↦ (t : tAppmap Γ [ 𝒜 ]) (u : tAppmap Γ [ ℬ ]) :
        Valuation Γ → Valuation [ 𝒜 × ℬ ] → Set where
  <>↦-intro₁ : ∀ {x} → <>↦ t u x ⟨⟨ ⊥ₓ ⟩⟩
  <>↦-intro₂ : ∀ {x y₁ y₂} → [ t ] x ↦ ⟨⟨ y₁ ⟩⟩ →
               [ u ] x ↦ ⟨⟨ y₂ ⟩⟩ →
               <>↦ t u x ⟨⟨ < y₁ , y₂ > ⟩⟩
```

B.0.76 Scwf/DomainScwf/ProductStructure/snd/AxiomProofs

```
{-# OPTIONS --safe #-}

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance

module Scwf.DomainScwf.ProductStructure.snd.AxiomProofs
  (t : tAppmap Γ [ 𝒜 × ℬ ]) where

open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Lemmata
```

```
open import Scwf.DomainScwf.Appmap.Valuation.Relation
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Relation
open import Scwf.DomainScwf.ProductStructure.snd.Relation

snd↦-mono : ∀ {x y z} → ⊑ᵥ Γ x y → snd↦ t x z →
                snd↦ t y z
snd↦-mono {y = y} _ (snd-intro₁ z⊑⊥)
  = snd-intro₁ z⊑⊥
snd↦-mono {y = y} x⊑y (snd-intro₂ tx↦z₁z₂)
  = snd-intro₂ ty↦z₁z₂
    where ty↦z₁z₂ = Appmap.↦-mono t x⊑y tx↦z₁z₂

snd↦-bottom : ∀ {x} → snd↦ t x ⟨⟨ NbhSys.⊥ 𝒷 ⟩⟩
snd↦-bottom {x = x} = snd-intro₁ (NbhSys.⊑-refl 𝒷)

snd↦-↓closed : ∀ {x y z} → ⊑ᵥ [ 𝒷 ] y z → snd↦ t x z →
                  snd↦ t x y
snd↦-↓closed {x = x} {⟨⟨ y ,, ⟨⟩ ⟩⟩}
  (⊑ᵥ-cons _ y⊑z ⊑ᵥ-nil) (snd-intro₁ z⊑⊥)
  = snd-intro₁ (NbhSys.⊑-trans 𝒷 y⊑z z⊑⊥)
snd↦-↓closed {x = x} {⟨⟨ y ,, ⟨⟩ ⟩⟩}
  (⊑ᵥ-cons _ y⊑z₂ ⊑ᵥ-nil) (snd-intro₂ tx↦z₁z₂)
  = snd-intro₂ tx↦z₁y
  where z₁y⊑z₁z₂' = ⊑ₓ-intro₂ (NbhSys.⊑-refl 𝒶) y⊑z₂
        z₁y⊑z₁z₂ = ⊑ᵥ-cons [ 𝒶 × 𝒷 ] z₁y⊑z₁z₂' ⊑ᵥ-nil
        tx↦z₁y = Appmap.↦-↓closed t z₁y⊑z₁z₂ tx↦z₁z₂

snd↦-↑directed : ∀ {x y z} → snd↦ t x y → snd↦ t x z →
                    (con : ValCon [ 𝒷 ] y z) →
                    snd↦ t x (y ⊔ᵥ z [ con ])
snd↦-↑directed {x = x} (snd-intro₁ y⊑⊥)
  (snd-intro₁ z⊑⊥) (con-tup conyz _)
  = snd-intro₁ (NbhSys.⊑-⊔ 𝒷 y⊑⊥ z⊑⊥ conyz)
snd↦-↑directed (snd-intro₂ tx↦y₁y₂) (snd-intro₁ z⊑⊥)
  (con-tup cony₂z _)
  = snd-intro₂ tx↦y₁y₂⊔z
  where z⊑y₂ = NbhSys.⊑-trans 𝒷 z⊑⊥ (NbhSys.⊑-⊥ 𝒷)
        y₂⊔z⊑y₂ = NbhSys.⊑-⊔ 𝒷 (NbhSys.⊑-refl 𝒷) z⊑y₂ cony₂z
        y₁y₂⊔z⊑y₁y₂' = ⊑ₓ-intro₂ (NbhSys.⊑-refl 𝒶) y₂⊔z⊑y₂
        y₁y₂⊔z⊑y₁y₂ = ⊑ᵥ-cons [ 𝒶 × 𝒷 ] y₁y₂⊔z⊑y₁y₂' ⊑ᵥ-nil
        tx↦y₁y₂⊔z = Appmap.↦-↓closed t y₁y₂⊔z⊑y₁y₂ tx↦y₁y₂
snd↦-↑directed (snd-intro₁ y⊑⊥) (snd-intro₂ tx↦z₁z₂)
  (con-tup conyz₂ _)
  = snd-intro₂ tx↦z₁y⊔z₂
  where y⊑z₂ = NbhSys.⊑-trans 𝒷 y⊑⊥ (NbhSys.⊑-⊥ 𝒷)
        y⊔z₂⊑z₂ = NbhSys.⊑-⊔ 𝒷 y⊑z₂ (NbhSys.⊑-refl 𝒷) conyz₂
        z₁y⊔z₂⊑z₂z₂' = ⊑ₓ-intro₂ (NbhSys.⊑-refl 𝒶) y⊔z₂⊑z₂
```

```
            z₁y⊔z₂⊑z₂z₂ = ⊑ᵥ-cons [ 𝒜 × ℬ ] z₁y⊔z₂⊑z₂z₂' ⊑ᵥ-nil
            tx↦z₁y⊔z₂ = Appmap.↦-↓closed t z₁y⊔z₂⊑z₂z₂ tx↦z₁z₂
snd↦-↑directed {x = x}
  (snd-intro₂ tx↦y₁y₂) (snd-intro₂ tx↦z₁z₂)
  (con-tup cony₂z₂ _)
  with (Appmap.↦-con t tx↦y₁y₂ tx↦z₁z₂ valConRefl)
... | con-tup (con-pair cony₁z₁ _) _
  = snd-intro₂ tx↦⊔
  where tx↦⊔ = Appmap.↦-↑directed t tx↦y₁y₂ tx↦z₁z₂
                 (con-tup (con-pair cony₁z₁ cony₂z₂) con-nil)

snd↦-con : ∀ {x y x' y'} → snd↦ t x y → snd↦ t x' y' → ValCon Γ x x' →
           ValCon [ ℬ ] y y'
snd↦-con (snd-intro₁ y⊑⊥) (snd-intro₁ y'⊑⊥) _
  = toValCon (NbhSys.Con-⊔ ℬ y⊑⊥ y'⊑⊥)
snd↦-con (snd-intro₁ y⊑⊥) (snd-intro₂ _) _
  = toValCon (NbhSys.Con-⊔ ℬ y⊑y'₁ (NbhSys.⊑-refl ℬ))
  where y⊑y'₁ = NbhSys.⊑-trans ℬ y⊑⊥ (NbhSys.⊑-⊥ ℬ)
snd↦-con (snd-intro₂ _) (snd-intro₁ y'⊑⊥) _
  = toValCon (NbhSys.Con-⊔ ℬ (NbhSys.⊑-refl ℬ) y'₁⊑y)
  where y'₁⊑y = NbhSys.⊑-trans ℬ y'⊑⊥ (NbhSys.⊑-⊥ ℬ)
snd↦-con (snd-intro₂ tx↦y₁y₂)
  (snd-intro₂ tx'↦y'₁y'₂) con
  with (Appmap.↦-con t tx↦y₁y₂ tx'↦y'₁y'₂ con)
... | con-tup (con-pair _ cony'₁y'₂) _ = toValCon cony'₁y'₂
```

B.0.77 Scwf/DomainScwf/ProductStructure/snd/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.snd.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance
open import Scwf.DomainScwf.ProductStructure.snd.AxiomProofs
open import Scwf.DomainScwf.ProductStructure.snd.Relation

snd : tAppmap Γ [ 𝒜 × ℬ ] → tAppmap Γ [ ℬ ]
Appmap._↦_ (snd t)         = snd↦ t
Appmap.↦-mono (snd t)      = snd↦-mono t
Appmap.↦-bottom (snd t)    = snd↦-bottom t
Appmap.↦-↓closed (snd t)   = snd↦-↓closed t
Appmap.↦-↑directed (snd t) = snd↦-↑directed t
Appmap.↦-con (snd t)       = snd↦-con t
```

B.0.78 Scwf/DomainScwf/ProductStructure/snd/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.snd.Relation where

open import Base.Core
open import Base.Variables
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.NbhSys.Instance

data snd↦ (t : tAppmap Γ [ 𝒜 × ℬ ]) :
            Valuation Γ → Valuation [ ℬ ] → Set where
  snd-intro₁ : ∀ {x y} → [ ℬ ] y ⊑ NbhSys.⊥ ℬ → snd↦ t x ⟨⟨ y ⟩⟩
  snd-intro₂ : ∀ {x y₁ y₂} → [ t ] x ↦ ⟨⟨ < y₁ , y₂ > ⟩⟩ →
               snd↦ t x ⟨⟨ y₂ ⟩⟩
```

B.0.79 Scwf/DomainScwf/ProductStructure/Unit/NSub

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Unit.NSub where

open import Appmap.Equivalence
open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ProductStructure.Unit.Mapping.Instance
open import Scwf.DomainScwf.ProductStructure.Unit.Mapping.Relation
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Instance

private
  variable
    γ : tAppmap Δ Γ

ℕ₁-subLemma₁ : ∀ {x y} → [ 0₁ ∘ γ ] x ↦ y →
                         [ 0₁ ] x ↦ y
ℕ₁-subLemma₁ _ = 0₁↦∀

ℕ₁-subLemma₂ : ∀ {x y} → [ 0₁ ] x ↦ y →
                         [ 0₁ ∘ γ ] x ↦ y
ℕ₁-subLemma₂ {γ = γ} _
```

```
  = ∘↦-intro γx↦⊥ 0₁↦∀
  where γx↦⊥ = Appmap.↦-bottom γ

ℕ₁-sub : (0₁ ∘ γ) ≈ 0₁
ℕ₁-sub = ≈-intro (≼-intro ℕ₁-subLemma₁)
                 (≼-intro ℕ₁-subLemma₂)
```

B.0.80 Scwf/DomainScwf/ProductStructure/Unit/Mapping/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Unit.Mapping.Instance where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Definition
open import Scwf.DomainScwf.ProductStructure.Unit.Mapping.Relation
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Instance

0₁ : tAppmap Γ [ ℕ₁ ]
Appmap._↦_ 0₁          = _0₁↦_
Appmap.↦-mono 0₁       = λ _ _ → 0₁↦∀
Appmap.↦-bottom 0₁     = 0₁↦∀
Appmap.↦-↓closed 0₁    = λ _ _ → 0₁↦∀
Appmap.↦-↑directed 0₁  = λ _ _ _ → 0₁↦∀
Appmap.↦-con 0₁        = 0₁↦-con
```

B.0.81 Scwf/DomainScwf/ProductStructure/Unit/Mapping/Relation

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Unit.Mapping.Relation where

open import Base.Core
open import Base.Variables
open import Scwf.DomainScwf.Appmap.Valuation.Definition
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Instance

data _0₁↦_ {Γ : Ctx n} : Valuation Γ → Valuation [ ℕ₁ ] →
                         Set where
  0₁↦∀ : ∀ {x y} → x 0₁↦ y

0₁↦-con : ∀ {x y x' y'} → x 0₁↦ y → x' 0₁↦ y' →
          ValCon Γ x x' → ValCon _ y y'
0₁↦-con {y = ⟨ _ ,, ⟨⟩ ⟩} {y' = ⟨ _ ,, ⟨⟩ ⟩} _ _ _
  = con-tup allCon con-nil
```

B.0.82 Scwf/DomainScwf/ProductStructure/Unit/NbhSys/AxiomProofs

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Unit.NbhSys.AxiomProofs where

open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Definition

private
  variable
    x y z : UnitNbh

Con-⊔₁ : x ⊑₁ z → y ⊑₁ z → UnitCon x y
Con-⊔₁ _ _ = allCon

⊑₁-refl : x ⊑₁ x
⊑₁-refl {⊥₁} = ⊥₁-bot
⊑₁-refl {0₁} = 0₁-refl

⊑₁-trans : x ⊑₁ y → y ⊑₁ z → x ⊑₁ z
⊑₁-trans ⊥₁-bot _ = ⊥₁-bot
⊑₁-trans 0₁-refl 0₁-refl = 0₁-refl

⊑₁-⊥ : ⊥₁ ⊑₁ x
⊑₁-⊥ = ⊥₁-bot

⊑₁-⊔ : y ⊑₁ x → z ⊑₁ x → (con : UnitCon y z) → (y ⊔₁ z [ con ]) ⊑₁ x
⊑₁-⊔ ⊥₁-bot ⊥₁-bot _ = ⊥₁-bot
⊑₁-⊔ ⊥₁-bot 0₁-refl _ = 0₁-refl
⊑₁-⊔ 0₁-refl _ _ = 0₁-refl

⊑₁-⊔-fst : (con : UnitCon x y) → x ⊑₁ (x ⊔₁ y [ con ])
⊑₁-⊔-fst {⊥₁} _ = ⊥₁-bot
⊑₁-⊔-fst {0₁} _ = 0₁-refl

⊑₁-⊔-snd : (con : UnitCon x y) → y ⊑₁ (x ⊔₁ y [ con ])
⊑₁-⊔-snd {y = ⊥₁} _ = ⊥₁-bot
⊑₁-⊔-snd {x = ⊥₁} {y = 0₁} _ = 0₁-refl
⊑₁-⊔-snd {x = 0₁} {y = 0₁} _ = 0₁-refl
```

B.0.83 Scwf/DomainScwf/ProductStructure/Unit/NbhSys/Definition

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Definition where

data UnitNbh : Set where
  ⊥₁ : UnitNbh
  0₁ : UnitNbh
```

```
data _⊑₁_ : UnitNbh → UnitNbh → Set where
  ⊥₁-bot : ∀ {x} → ⊥₁ ⊑₁ x
  0₁-refl : 0₁ ⊑₁ 0₁

data UnitCon : UnitNbh → UnitNbh → Set where
  allCon : ∀ {x y} → UnitCon x y

_⊔₁_[_] : (x : UnitNbh) → (y : UnitNbh) → UnitCon x y → UnitNbh
⊥₁ ⊔₁ y [ _ ] = y
0₁ ⊔₁ y [ _ ] = 0₁
```

B.0.84 Scwf/DomainScwf/ProductStructure/Unit/NbhSys/Instance

```
{-# OPTIONS --safe #-}

module Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Instance where

open import NbhSys.Definition
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.AxiomProofs
open import Scwf.DomainScwf.ProductStructure.Unit.NbhSys.Definition

𝕟₁ : NbhSys
NbhSys.Nbh 𝕟₁     = UnitNbh
NbhSys._⊑_ 𝕟₁     = _⊑₁_
NbhSys.Con 𝕟₁     = UnitCon
NbhSys._⊔_[_] 𝕟₁ = _⊔₁_[_]
NbhSys.⊥ 𝕟₁       = ⊥₁
NbhSys.Con-⊔ 𝕟₁   = Con-⊔₁
NbhSys.⊑-refl 𝕟₁  = ⊑₁-refl
NbhSys.⊑-trans 𝕟₁ = ⊑₁-trans
NbhSys.⊑-⊥ 𝕟₁     = ⊑₁-⊥
NbhSys.⊑-⊔ 𝕟₁     = ⊑₁-⊔
NbhSys.⊑-⊔-fst 𝕟₁ = ⊑₁-⊔-fst
NbhSys.⊑-⊔-snd 𝕟₁ = ⊑₁-⊔-snd
```

B.0.85 Ucwf/LambdaBeta

```
{-# OPTIONS --safe #-}

module Ucwf.LambdaBeta where

open import Base.Variables
open import Ucwf.Plain

open import Agda.Builtin.Nat

record λβ-ucwf : Set₂ where
```

```
  field
    ucwf : Ucwf
  open Ucwf ucwf public
  field
    lam : Tm (suc m) → Tm m
    ap  : Tm m → Tm m → Tm m

    lamSub : (γ : Sub n m) → (t : Tm (suc m)) →
             (lam t [ γ ]) ≈ (lam (t [ ⟨ γ ∘ p , q ⟩ ]))
    apSub : (γ : Sub n m) → (t u : Tm m) →
            ap (t [ γ ]) (u [ γ ]) ≈ (ap t u [ γ ])

    β : {t : Tm m} → {u : Tm (suc m)} →
        ap (lam u) t ≈ (u [ ⟨ id , t ⟩ ])

    lamCong : ∀ {t t' : Tm (suc m)} → t ≈ t' →
              lam t ≈ lam t'
    apCong : {t t' : Tm m} → ∀ {u u'} →
             t ≈ t' → u ≈ u' →
             ap t u ≈ ap t' u'
```

B.0.86 Ucwf/Plain

```
{-# OPTIONS --safe #-}

module Ucwf.Plain where

open import Base.Core using (Rel ; IsEquivalence)
open import Base.Variables

open import Agda.Builtin.Nat

record Ucwf : Set₂ where
  field
    Tm : Nat → Set₁
    Sub : Nat → Nat → Set₁

    _≈_ : Rel (Tm n)
    _≅_ : Rel (Sub m n)

    isEquivT : IsEquivalence (_≈_ {n})
    isEquivS : IsEquivalence (_≅_ {m} {n})

    q : Tm (suc n)
    _[_] : Tm n → Sub m n → Tm m

    id : Sub n n
    _∘_ : Sub n o → Sub m n → Sub m o
```

```
⟨⟩ : Sub n 0
⟨_,_⟩ : Sub m n → Tm m → Sub m (suc n)
p : Sub (suc n) n

idL : (γ : Sub n m) → (id ∘ γ) ≋ γ
idR : (γ : Sub n m) → (γ ∘ id) ≋ γ
subAssoc : (γ : Sub m n) → (δ : Sub n o) →
           (θ : Sub o r) →
           ((θ ∘ δ) ∘ γ) ≋ (θ ∘ (δ ∘ γ))

idSub : (t : Tm n) → (t [ id ]) ≈ t
compSub : (t : Tm n) → (γ : Sub m n) →
          (δ : Sub o m) →
          (t [ (γ ∘ δ) ]) ≈ ((t [ γ ]) [ δ ])

id₀ : id ≋ ⟨⟩
<>-zero : (γ : Sub m n) → (⟨⟩ ∘ γ) ≋ ⟨⟩

pCons : (γ : Sub n m) → (t : Tm n) →
        (p ∘ ⟨ γ , t ⟩) ≋ γ
qCons : (γ : Sub n m) → (t : Tm n) →
        (q [ ⟨ γ , t ⟩ ]) ≈ t
idExt : (id {suc m}) ≋ ⟨ p , q ⟩
compExt : (t : Tm n) → (γ : Sub n m) → (δ : Sub m n) →
          (⟨ γ , t ⟩ ∘ δ) ≋ ⟨ γ ∘ δ , t [ δ ] ⟩

subCong : {t t' : Tm m} → {γ γ' : Sub n m} →
          t ≈ t' → γ ≋ γ' →
          (t [ γ ]) ≈ (t' [ γ' ])
<,>-cong : {t t' : Tm m} → {γ γ' : Sub m n} →
           t ≈ t' → γ ≋ γ' →
           ⟨ γ , t ⟩ ≋ ⟨ γ' , t' ⟩
∘-cong : {γ δ : Sub n o} → {γ' δ' : Sub m n} →
         γ ≋ δ →
         γ' ≋ δ' → (γ ∘ γ') ≋ (δ ∘ δ')
```

B.0.87 Ucwf/DomainUcwf/LambdaBetaInstance

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBetaInstance where

open import Ucwf.LambdaBeta
open import Ucwf.DomainUcwf.LambdaBeta.ap.Instance
open import Ucwf.DomainUcwf.LambdaBeta.apCong
open import Ucwf.DomainUcwf.LambdaBeta.apSub
open import Ucwf.DomainUcwf.LambdaBeta.beta
```

```
open import Ucwf.DomainUcwf.LambdaBeta.lam.Instance
open import Ucwf.DomainUcwf.LambdaBeta.lamCong
open import Ucwf.DomainUcwf.LambdaBeta.lamSub
open import Ucwf.DomainUcwf.PlainInstance

domλβUcwf : λβ-ucwf
λβ-ucwf.ucwf domλβUcwf     = domUcwf
λβ-ucwf.lam domλβUcwf      = lam
λβ-ucwf.ap domλβUcwf       = ap
λβ-ucwf.lamSub domλβUcwf = lamSub
λβ-ucwf.apSub domλβUcwf    = apSub
λβ-ucwf.β domλβUcwf        = β-equal
λβ-ucwf.lamCong domλβUcwf = lamCong
λβ-ucwf.apCong domλβUcwf = apCong
```

B.0.88 Ucwf/DomainUcwf/PlainInstance

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.PlainInstance where

open import Appmap.Equivalence
open import Base.Core
open import Base.FinFun
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Empty.Instance
open import Scwf.DomainScwf.Appmap.Identity.Instance
open import Scwf.DomainScwf.PlainAxiomProofs
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.p.Instance
open import Scwf.DomainScwf.Comprehension.q.Instance
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.UniType.Instance
open import Ucwf.Plain

domUcwf : Ucwf
Ucwf.Tm domUcwf n      = uAppmap n 1
Ucwf.Sub domUcwf m n   = uAppmap m n
Ucwf.id domUcwf {n}    = idMap (nToCtx n)
Ucwf._≈_ domUcwf       = _≈_
Ucwf._≅_ domUcwf       = _≈_
Ucwf.isEquivT domUcwf = ≈IsEquiv
Ucwf.isEquivS domUcwf = ≈IsEquiv
Ucwf.q domUcwf {n}     = q (nToCtx n) UniType
Ucwf._[_] domUcwf      = _∘_
Ucwf._∘_ domUcwf       = _∘_
Ucwf.⟨⟩ domUcwf        = emptyMap
```

```
Ucwf.⟨_,_⟩ domUcwf      = ⟨_,_⟩
Ucwf.p domUcwf {n}      = p (nToCtx n) UniType
Ucwf.idL domUcwf        = idL
Ucwf.idR domUcwf        = idR
Ucwf.subAssoc domUcwf = subAssoc
Ucwf.idSub domUcwf      = idSub
Ucwf.compSub domUcwf = compSub
Ucwf.id₀ domUcwf        = id₀
Ucwf.<>-zero domUcwf  = <>-zero
Ucwf.pCons domUcwf      = pCons
Ucwf.qCons domUcwf      = qCons
Ucwf.idExt domUcwf      = idExt
Ucwf.compExt domUcwf  = compExt
Ucwf.subCong domUcwf  = ∘-cong
Ucwf.<,>-cong domUcwf = <,>-cong
Ucwf.∘-cong domUcwf     = ∘-cong
```

B.0.89 Ucwf/DomainUcwf/Appmap/Definition

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.Appmap.Definition where

open import Scwf.DomainScwf.Appmap.Definition public
open import Ucwf.DomainUcwf.UniType.Instance

open import Agda.Builtin.Nat

-- Notation for approximable mappings between
-- neighborhoods of the universal type.
uAppmap : Nat → Nat → Set₁
uAppmap m n = tAppmap (nToCtx m) (nToCtx n)
```

B.0.90 Ucwf/DomainUcwf/Appmap/Valuation

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.Appmap.Valuation where

open import Scwf.DomainScwf.Appmap.Valuation.Definition public
open import Scwf.DomainScwf.Appmap.Valuation.Instance public
open import Scwf.DomainScwf.Appmap.Valuation.Relation public
open import Ucwf.DomainUcwf.UniType.Instance

open import Agda.Builtin.Nat

-- Notation for valuations of contexts in the ucwf.
uValuation : Nat → Set
uValuation n = Valuation (nToCtx n)
```

B.0.91 Ucwf/DomainUcwf/LambdaBeta/apCong

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.apCong where

open import Appmap.Equivalence
open import Base.Variables
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.ap.Instance
open import Ucwf.DomainUcwf.LambdaBeta.ap.Relation
open import Ucwf.DomainUcwf.UniType.Definition

private
  variable
    t t' u u' : uAppmap n 1

apCongLemma : t ≼ t' → u ≼ u' →
                    ∀ {x y} → [ ap t u ] x ↦ y →
                    [ ap t' u' ] x ↦ y
apCongLemma _ _ {y = ⟨⟨ ⊥ᵤ ,, ⟨⟨⟩⟩ ⟩⟩} _ = ap↦-intro₁
apCongLemma (≼-intro t≼t') (≼-intro u≼u') {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩}
  (ap↦-intro₂ tx↦g ux↦z zf⊑g)
  = ap↦-intro₂ (t≼t' tx↦g) (u≼u' ux↦z) zf⊑g

apCong : t ≈ t' → u ≈ u' → ap t u ≈ ap t' u'
apCong (≈-intro t≼t' t'≼t) (≈-intro u≼u' u'≼u)
  = ≈-intro (≼-intro (apCongLemma t≼t' u≼u'))
            (≼-intro (apCongLemma t'≼t u'≼u))
```

B.0.92 Ucwf/DomainUcwf/LambdaBeta/apSub

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.apSub where

open import Appmap.Equivalence
open import NbhSys.Definition
open import Base.Variables
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.ap.Instance
open import Ucwf.DomainUcwf.LambdaBeta.ap.Relation
open import Ucwf.DomainUcwf.UniType.Definition
```

```
open import Ucwf.DomainUcwf.UniType.Instance

private
  variable
    γ : uAppmap n m
    t u : uAppmap m 1

apSubLemma₁ : ∀ {x y} → [ ap (t ∘ γ) (u ∘ γ) ] x ↦ y →
                        [ ap t u ∘ γ ] x ↦ y
apSubLemma₁ {γ = γ} {y = ⟨⟨ ⊥ᵤ ,, _ ⟩⟩} ap↦-intro₁ =
  ∘↦-intro (Appmap.↦-bottom γ) ap↦-intro₁
apSubLemma₁ {γ = γ} {y = ⟨⟨ ⊥ᵤ ,, _ ⟩⟩}
  (ap↦-intro₂ (∘↦-intro γx↦y ty↦f) (∘↦-intro γx↦z uz↦x) xy⊑f)
  = ∘↦-intro (Appmap.↦-bottom γ) ap↦-intro₁
apSubLemma₁ {t = t} {γ = γ} {u = u} {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩}
  (ap↦-intro₂ (∘↦-intro γx↦y ty↦g) (∘↦-intro γx↦z uz↦x) xy⊑g)
  = ∘↦-intro γx↦y⊔z aptuyz↦y
  where y⊑y⊔z = NbhSys.⊑-⊔-fst (ValNbhSys _) valConAll
        tyz↦g = Appmap.↦-mono t y⊑y⊔z ty↦g
        z⊑y⊔z = NbhSys.⊑-⊔-snd (ValNbhSys _) valConAll
        uyz↦x = Appmap.↦-mono u z⊑y⊔z uz↦x
        aptuyz↦y = ap↦-intro₂ tyz↦g uyz↦x xy⊑g
        γx↦y⊔z = Appmap.↦-↑directed γ γx↦y γx↦z valConAll

apSubLemma₂ : ∀ {x y} → [ ap t u ∘ γ ] x ↦ y →
                        [ ap (t ∘ γ) (u ∘ γ) ] x ↦ y
apSubLemma₂ {y = ⟨⟨ ⊥ᵤ ,, ⟨⟨⟩⟩ ⟩⟩} _ = ap↦-intro₁
apSubLemma₂ {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩}
  (∘↦-intro γx↦z (ap↦-intro₂ tz↦g uz↦x xf⊑g))
  = ap↦-intro₂ t∘γx↦f u∘γx↦x xf⊑g
  where t∘γx↦f = ∘↦-intro γx↦z tz↦g
        u∘γx↦x = ∘↦-intro γx↦z uz↦x

apSub : (γ : uAppmap n m) → ∀ t u →
        (ap (t ∘ γ) (u ∘ γ)) ≈ ((ap t u) ∘ γ)
apSub γ t u = ≈-intro (≼-intro apSubLemma₁)
                      (≼-intro (apSubLemma₂))
```

B.0.93 Ucwf/DomainUcwf/LambdaBeta/beta

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.beta where

open import Appmap.Equivalence
open import Base.Core
open import Base.Variables
```

```
open import NbhSys.Definition
open import Scwf.DomainScwf.Appmap.Identity.Instance
open import Scwf.DomainScwf.Appmap.Identity.Relation
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Relation
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.ap.Instance
open import Ucwf.DomainUcwf.LambdaBeta.ap.Relation
open import Ucwf.DomainUcwf.LambdaBeta.lam.Instance
open import Ucwf.DomainUcwf.LambdaBeta.lam.Lemmata
open import Ucwf.DomainUcwf.LambdaBeta.lam.Relation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Instance
open import Ucwf.DomainUcwf.UniType.PrePost
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.SizedFinFun

open import Agda.Builtin.Nat

private
  variable
    t : tAppmap (nToCtx m) [ UniType ]
    u : tAppmap (nToCtx (suc m)) [ UniType ]

β-lemma₁ : ∀ {x y} →
             [ ap (lam u) t ] x ↦ y →
             [ u ∘ ⟨ idMap (nToCtx m) , t ⟩ ] x ↦ y
β-lemma₁ {m = m} {u = u} {t = t} {y = ⟨⟨ ⊥ᵤ ,, ⟨⟨⟩⟩ ⟩⟩} _
  = ∘↦-intro ⟨⟩⊥↦⊥ u⊥↦⊥
  where id⊥↦⊥ = Appmap.↦-bottom (idMap (nToCtx m))
        tx↦⊥ = Appmap.↦-bottom t
        ⟨⟩⊥↦⊥ = ⟨⟩↦-intro {y = ⟨⟨ _ ,, ⊥ᵥ ⟩⟩} id⊥↦⊥ tx↦⊥
        u⊥↦⊥ = Appmap.↦-bottom u
β-lemma₁ {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩}
  (ap↦-intro₂ _ _ (⊑ᵤ-intro₂ _ _ p))
  with (p _ _ here)
β-lemma₁ {m = m} {u = u} {t = t} {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩}
  (ap↦-intro₂ {x} lamux↦g tx↦x _)
  | record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = sub⊆f'
           }
  = ∘↦-intro (⟨⟩↦-intro {y = ⟨⟨ x ,, _ ⟩⟩} idx↦x tx↦x) uxx↦y
```

```
    where idx↦x = id↦-intro (NbhSys.⊑-refl (ValNbhSys _))
          y⊑post' = ⊑ᵥ-cons [ UniType ] y⊑ᵤpost ⊑ᵥ-nil
          prex⊑xx = ⊑ᵥ-cons (nToCtx (suc m)) pre⊑ᵤx
                    (NbhSys.⊑-refl (ValNbhSys _))
          uprex↦postx = ↓closed-lemma _ sub
                        (shrinklam sub⊆f' lamux↦g)
          uxx↦post = Appmap.↦-mono u prex⊑xx uprex↦postx
          uxx↦y = Appmap.↦-↓closed u y⊑post' uxx↦post

β-lemma₂' : ∀ x x' y' →
            [ u ] ⟨⟨ x' ,, x ⟩⟩ ↦ ⟨⟨ y' ⟩⟩ →
            ∀ {x y} → (x , y) ∈ₛ ((x' , y') :: ∅) →
            [ u ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
β-lemma₂' _ _ _ ux'x↦y' here = ux'x↦y'

β-lemma₂ : ∀ {x y} →
           [ u ∘ ⟨ idMap (nToCtx n) , t ⟩ ] x ↦ y →
           [ ap (lam u) t ] x ↦ y
β-lemma₂ {y = ⟨⟨ ⊥ᵤ ,, ⟨⟨⟩⟩ ⟩⟩} _ = ap↦-intro₁
β-lemma₂ {n = n} {u = u} {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩}
  (∘↦-intro (⟨⟩↦-intro (id↦-intro x'⊑x) tx↦x) uxx'↦y)
  = ap↦-intro₂ lamx↦xy tx↦x (NbhSys.⊑-refl UniType)
  where y = λᵤ f
        xx'⊑xx = ⊑ᵥ-cons (nToCtx (suc n))
                 (NbhSys.⊑-refl UniType) x'⊑x
        uxx↦y = Appmap.↦-mono u xx'⊑xx uxx'↦y
        lamx↦xy = lam↦-intro₂ (β-lemma₂' {u = u} _ _ _ uxx↦y)

β-equal : ap (lam u) t ≈ (u ∘ ⟨ idMap (nToCtx m) , t ⟩)
β-equal = ≈-intro (≼-intro β-lemma₁) (≼-intro β-lemma₂)
```

B.0.94 Ucwf/DomainUcwf/LambdaBeta/lamCong

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.lamCong where

open import Appmap.Equivalence
open import Base.Core
open import Base.Variables
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.lam.Instance
open import Ucwf.DomainUcwf.LambdaBeta.lam.Relation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Instance
```

```
open import Agda.Builtin.Nat

private
  variable
    t t' : uAppmap (suc n) 1

lamCongLemma : t ≼ t' → ∀ {x y} → [ lam t ] x ↦ y →
                      [ lam t' ] x ↦ y
lamCongLemma _ {y = ⟨⟨ ⊥u ,, ⟨⟨⟩⟩ ⟩⟩} _ = lam↦-intro1
lamCongLemma (≼-intro p1) {y = ⟨⟨ λu f ,, ⟨⟨⟩⟩ ⟩⟩}
  (lam↦-intro2 p2)
  = lam↦-intro2 λ xy∈f → p1 (p2 xy∈f)

lamCong : t ≈ t' → lam t ≈ lam t'
lamCong (≈-intro t≼t' t'≼t)
  = ≈-intro (≼-intro (lamCongLemma t≼t'))
            (≼-intro (lamCongLemma t'≼t))
```

B.0.95 Ucwf/DomainUcwf/LambdaBeta/lamSub

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.lamSub where

open import Appmap.Equivalence
open import Base.Variables
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Scwf.DomainScwf.Appmap.Composition.Instance
open import Scwf.DomainScwf.Appmap.Composition.Relation
open import Scwf.DomainScwf.Comprehension.Morphism.Instance
open import Scwf.DomainScwf.Comprehension.Morphism.Relation
open import Scwf.DomainScwf.Comprehension.p.Instance renaming (p to p')
open import Scwf.DomainScwf.Comprehension.p.Relation
open import Scwf.DomainScwf.Comprehension.q.Instance renaming (q to q')
open import Scwf.DomainScwf.Comprehension.q.Relation
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.lam.Instance
open import Ucwf.DomainUcwf.LambdaBeta.lam.Relation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Instance
open import Ucwf.DomainUcwf.UniType.SizedFinFun

open import Agda.Builtin.Nat

p : uAppmap (suc m) m
```

```
p {m} = p' (nToCtx m) UniType

q : uAppmap (suc m) 1
q {m} = q' (nToCtx m) UniType

private
  variable
    γ : uAppmap n m
    t : uAppmap (suc m) 1

  UT : NbhSys
  UT = UniType

lamSubLemma₁' : ∀ {x f} → [ lam t ∘ γ ] x ↦ ⟨⟨ λᵤ f ⟩⟩ →
                          ∀ {x y} → (x , y) ∈ₛ f →
                          [ t ∘ ⟨ γ ∘ p , q ⟩ ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
lamSubLemma₁' (∘↦-intro γx↦y (lam↦-intro₂ p)) xy∈f
  = ∘↦-intro γ∘pq↦ (p xy∈f)
  where q↦ = q↦-intro (NbhSys.⊑-refl UT)
        p↦x = p↦-intro (NbhSys.⊑-refl (ValNbhSys _))
        γ∘p↦ = ∘↦-intro p↦x γx↦y
        γ∘pq↦ = ⟨⟩↦-intro γ∘p↦ q↦

lamSubLemma₁ : ∀ {x y} → [ lam t ∘ γ ] x ↦ y →
                         [ lam (t ∘ ⟨ γ ∘ p , q ⟩) ] x ↦ y
lamSubLemma₁ {y = ⟨⟨ ⊥ᵤ ,, ⟨⟨⟩⟩ ⟩⟩} _ = lam↦-intro₁
lamSubLemma₁ {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩} (∘↦-intro γx↦y lamty↦f)
  = lam↦-intro₂ (lamSubLemma₁' lamx↦f)
  where lamx↦f = ∘↦-intro γx↦y lamty↦f

record P-Struct (γ : uAppmap n m)
                (t : uAppmap (suc m) 1)
                (x : uValuation n) (f : FinFunₛ) :
                Set where
  field
    y : uValuation m
    γx↦y : [ γ ] x ↦ y
    λty : ∀ {x y} → (x , y) ∈ₛ f → [ t ] ⟨⟨ x ,, y ⟩⟩ ↦ ⟨⟨ y ⟩⟩

getP-Struct' : ∀ x x y y z f →
                 [ t ∘ ⟨ γ ∘ p , q ⟩ ] x lam↦ ⟨⟨ λᵤ ((x , y) :: f) ⟩⟩ →
                 [ t ] ⟨⟨ x ,, y ⟩⟩ ↦ ⟨⟨ y ⟩⟩ →
                 (∀ {x' y'} → (x' , y') ∈ₛ f →
                 [ t ] ⟨⟨ x' ,, z ⟩⟩ ↦ ⟨⟨ y' ⟩⟩) →
                 ∀ {x' y'} → (x' , y') ∈ₛ ((x , y) :: f) →
                 [ t ] ⟨⟨ x' ,, y ⊔ᵥ z [ valConAll ] ⟩⟩ ↦ ⟨⟨ y' ⟩⟩
getP-Struct' {m} {t = t} x x y y z f _ txy↦y _ here
  = Appmap.↦-mono t xy⊑x⊔ txy↦y
```

```
  where y⊑⊔ = NbhSys.⊑-⊔-fst (ValNbhSys _) valConAll
        xy⊑x⊔ = ⊑ᵥ-cons (nToCtx (suc m))
                  (NbhSys.⊑-refl UT) y⊑⊔
getP-Struct' {m} {t = t} x x y y z f _ _ p
  (there x'y'∈f)
  = Appmap.↦-mono t x'r⊑x'⊔ (p x'y'∈f)
  where r⊑⊔ = NbhSys.⊑-⊔-snd (ValNbhSys _) valConAll
        x'r⊑x'⊔ = ⊑ᵥ-cons (nToCtx (suc m))
                    (NbhSys.⊑-refl UT) r⊑⊔

getP-Struct : ∀ x → (f : FinFunₛ) →
                [ t ∘ ⟨ γ ∘ p , q ⟩ ] x lam↦ ⟨⟨ λᵤ f ⟩⟩ →
                P-Struct γ t x f
getP-Struct {m} {γ = γ} x ∅ _
  = record { y = ⊥ᵥ
           ; γx↦y = Appmap.↦-bottom γ
           ; λty = xy∈∅-abs
           }
getP-Struct x ((x , y) :: f) (lam↦-intro₂ p)
  with (p here)
getP-Struct {m} {t = t} {γ = γ} x ((x , y) :: f)
  (lam↦-intro₂ p)
  | ∘↦-intro {y = ⟨⟨ z ,, z ⟩⟩}
    (⟨⟩↦-intro (∘↦-intro (p↦-intro y⊑x) γy↦z)
    (q↦-intro z⊑x)) tzz↦y
  = record { y = z ⊔ᵥ rec-y [ valConAll ]
           ; γx↦y = Appmap.↦-↑directed γ γx↦z rec-γx↦y valConAll
           ; λty = getP-Struct' x x y z rec-y f (lam↦-intro₂ p)
                     txz↦y rec-λty
           }
  where rec = getP-Struct {t = t} {γ = γ} x f
                (lam↦-intro₂ λ x'y'∈f → p (there x'y'∈f))
        rec-y = P-Struct.y rec
        rec-γx↦y = P-Struct.γx↦y rec
        rec-λty = P-Struct.λty rec
        γx↦z = Appmap.↦-mono γ y⊑x γy↦z
        zz⊑xz = ⊑ᵥ-cons (nToCtx (suc m)) z⊑x
                  (NbhSys.⊑-refl (ValNbhSys _))
        txz↦y = Appmap.↦-mono t zz⊑xz tzz↦y
        big⊔ = z ⊔ᵥ rec-y [ valConAll ]

lamSubLemma₂ : ∀ {x y} → [ lam (t ∘ ⟨ γ ∘ p , q ⟩) ] x ↦ y →
                 [ lam t ∘ γ ] x ↦ y
lamSubLemma₂ {m} {γ = γ} {y = ⟨⟨ ⊥ᵤ ,, ⟨⟨⟩⟩ ⟩⟩} _
  = ∘↦-intro γx↦⊥ lam⊥→⊥
  where γx↦⊥ = Appmap.↦-bottom γ
        lam⊥→⊥ = lam↦-intro₁
```

```
lamSubLemma₂ {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩} (lam↦-intro₂ p)
  with (getP-Struct _ _ (lam↦-intro₂ p))
lamSubLemma₂ {t = t} {γ = γ} {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩} _
  | record { y = y
           ; γx↦y = γx↦y
           ; λty = λty
           }
  = ∘↦-intro γx↦y (lam↦-intro₂ λty)

lamSub : (γ : uAppmap n m) → (t : uAppmap (suc m) 1) →
         (lam t ∘ γ) ≈ lam (t ∘ ⟨ (γ ∘ p) , q ⟩)
lamSub γ t = ≈-intro (≼-intro lamSubLemma₁)
                     (≼-intro lamSubLemma₂)
```

B.0.96 Ucwf/DomainUcwf/LambdaBeta/ap/AxiomProofs

```
{-# OPTIONS --safe --sized-types #-}

open import Ucwf.DomainUcwf.Appmap.Definition
open import Base.Variables

module Ucwf.DomainUcwf.LambdaBeta.ap.AxiomProofs
  {t u : uAppmap n 1} where

open import NbhSys.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.ap.Relation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Instance
open import Ucwf.DomainUcwf.UniType.PrePost
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.Transitivity
open import Ucwf.DomainUcwf.UniType.SizedFinFun

private
  UT : NbhSys
  UT = UniType

ap↦-mono : ∀ {x y z} → ⊑ᵥ (nToCtx n) x y →
           [ t , u ] x ap↦ z → [ t , u ] y ap↦ z
ap↦-mono _ ap↦-intro₁ = ap↦-intro₁
ap↦-mono x⊑y (ap↦-intro₂ tx↦f ux↦x xy⊑f)
  = ap↦-intro₂ ty↦f uy↦x xy⊑f
  where ty↦f = Appmap.↦-mono t x⊑y tx↦f
        uy↦x = Appmap.↦-mono u x⊑y ux↦x

ap↦-bottom : ∀ {x} → [ t , u ] x ap↦ ⟨⟨ ⊥ᵤ ⟩⟩
```

```
ap↦-bottom = ap↦-intro₁

ap↦-↓closed' : ∀ {f' x y f} →
                 [ UT ] (λᵤ f) ⊑ y →
                 [ UT ] λᵤ ((x , y) :: ∅) ⊑ λᵤ f' →
                 ∀ x' y' →
                 (x' , y') ∈ₛ ((x , λᵤ f) :: ∅) →
                 ⊑ᵤ-proof f' x' y'
ap↦-↓closed' {x = x} {y} f⊑y (⊑ᵤ-intro₂ _ _ p) _ _ here
  = record { sub = sub
           ; y⊑ᵤpost = NbhSys.⊑-trans UT f⊑y y⊑post
           ; pre⊑ᵤx = pre⊑x
           ; sub⊆f' = sub⊆f
           }
  where paxy = p x y here
        sub = ⊑ᵤ-proof.sub paxy
        pre⊑x = ⊑ᵤ-proof.pre⊑ᵤx paxy
        y⊑post = ⊑ᵤ-proof.y⊑ᵤpost paxy
        sub⊆f = ⊑ᵤ-proof.sub⊆f' paxy

ap↦-↓closed : ∀ {x y z} → ⊑ᵥ (nToCtx 1) y z →
                [ t , u ] x ap↦ z → [ t , u ] x ap↦ y
ap↦-↓closed {y = ⟨⟨ ⊥ᵤ ,, ⟨⟨⟩⟩ ⟩⟩} _ _ = ap↦-intro₁
ap↦-↓closed {y = ⟨⟨ λᵤ f ,, ⟨⟨⟩⟩ ⟩⟩} (⊑ᵥ-cons _ f⊑y ⊑ᵥ-nil)
  (ap↦-intro₂ {x = x'} {f = f'} tx↦f' ux↦x' x'y'⊑f')
  = ap↦-intro₂ tx↦f' ux↦x' x'f⊑f'
  where x'f⊑f' = ap↦-↓closed' f⊑y x'y'⊑f'
        x'f⊑f' = ⊑ᵤ-intro₂ ((x' , λᵤ f) :: ∅) f' x'f⊑f'

ap↦-↑directed' : ∀ {f f' x x' y y'} →
                   λᵤ ((x , y) :: ∅) ⊑ᵤ (λᵤ f) →
                   λᵤ ((x' , y') :: ∅) ⊑ᵤ (λᵤ f') → ∀ x'' y'' →
                   (x'' , y'') ∈ₛ
                   (((x ⊔ᵤ x' [ con-all ]) , (y ⊔ᵤ y' [ con-all ])) :: ∅) →
                   ⊑ᵤ-proof (f ∪ₛ f') x'' y''
ap↦-↑directed' {x = x} {x'} {y} {y'} (⊑ᵤ-intro₂ _ _ p₁)
  (⊑ᵤ-intro₂ _ _ p₂) x'' y'' here
  = record { sub = p₁sub ∪ₛ p₂sub
           ; y⊑ᵤpost = Ω-post {f = p₁sub} p₁y⊑post p₂y⊑post
           ; pre⊑ᵤx = Ω-pre {f = p₁sub} p₁pre⊑x p₂pre⊑x
           ; sub⊆f' = ∪ₛ-lemma₅ p₁sub⊆f p₂sub⊆f
           }
  where p₁xyh = p₁ x y here
        p₂x'y'h = p₂ x' y' here
        p₁sub = ⊑ᵤ-proof.sub p₁xyh
        p₂sub = ⊑ᵤ-proof.sub p₂x'y'h
        p₁y⊑post = ⊑ᵤ-proof.y⊑ᵤpost p₁xyh
```

```
    p₂y⊑post = ⊑ᵤ-proof.y⊑ᵤpost p₂x'y'h
    p₁pre⊑x = ⊑ᵤ-proof.pre⊑ᵤx p₁xyh
    p₂pre⊑x = ⊑ᵤ-proof.pre⊑ᵤx p₂x'y'h
    p₁sub⊆f = ⊑ᵤ-proof.sub⊆f' p₁xyh
    p₂sub⊆f = ⊑ᵤ-proof.sub⊆f' p₂x'y'h

ap↦-↑directed : ∀ {x y z} → [ t , u ] x ap↦ y →
                [ t , u ] x ap↦ z → (conyz : ValCon _ y z) →
                [ t , u ] x ap↦ (y ⊔ᵥ z [ conyz ])
ap↦-↑directed {y = ⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩} {⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩}
  x↦y _ (con-tup _ _) = x↦y
ap↦-↑directed {y = ⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩} {⟨⟨ λᵤ f' ,, ⟨⟩ ⟩⟩}
  _ x↦z (con-tup _ _) = x↦z
ap↦-↑directed {y = ⟨⟨ λᵤ f ,, ⟨⟩ ⟩⟩} {⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩}
  x↦y _ (con-tup _ _) = x↦y
ap↦-↑directed {y = ⟨⟨ λᵤ f ,, ⟨⟩ ⟩⟩} {⟨⟨ λᵤ f' ,, ⟨⟩ ⟩⟩}
  (ap↦-intro₂ {x} {_} {g} tx↦g ux↦x xf⊑g)
  (ap↦-intro₂ {x'} {_} {g'} tx↦g' ux↦x' x'f'⊑g')
  (con-tup _ _)
  = ap↦-intro₂ tx↦g∪g' ux↦x⊔x' big⊑
    where tx↦g∪g' = Appmap.↦-↑directed t tx↦g tx↦g'
                      (con-tup con-all con-nil)
          ux↦x⊔x' = Appmap.↦-↑directed u ux↦x ux↦x'
                      (con-tup con-all con-nil)
          f∪f' = λᵤ (f ∪ₛ f')
          big⊑ = ⊑ᵤ-intro₂ (([ UT ] x ⊔ x' [ con-all ] , f∪f') :: ∅)
                   (g ∪ₛ g') (ap↦-↑directed' xf⊑g x'f'⊑g')
```

B.0.97 Ucwf/DomainUcwf/LambdaBeta/ap/Instance

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.ap.Instance where

open import Base.Variables
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.ap.AxiomProofs
open import Ucwf.DomainUcwf.LambdaBeta.ap.Relation
open import Ucwf.DomainUcwf.UniType.Definition

ap : uAppmap n 1 → uAppmap n 1 → uAppmap n 1
Appmap._↦_ (ap t u)         = [_,_]_ap↦_ t u
Appmap.↦-mono (ap t u)      = ap↦-mono
Appmap.↦-bottom (ap t u)    = ap↦-bottom
Appmap.↦-↓closed (ap t u)   = ap↦-↓closed
Appmap.↦-↑directed (ap t u) = ap↦-↑directed
Appmap.↦-con (ap t u)       = λ _ _ _ → valConAll
```

B.0.98 Ucwf/DomainUcwf/LambdaBeta/ap/Relation

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.ap.Relation where

open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.SizedFinFun

open import Agda.Builtin.Nat

private
  variable
    n : Nat

data [_,_]_ap↦_ (t u : uAppmap n 1) (x : uValuation n) :
                  uValuation 1 → Set where
  ap↦-intro₁ : [ t , u ] x ap↦ ⟨⟨ ⊥ᵤ ⟩⟩
  ap↦-intro₂ : ∀ {x y f} →
               [ t ] x ↦ ⟨⟨ λᵤ f ⟩⟩ → [ u ] x ↦ ⟨⟨ x ⟩⟩ →
               (λᵤ ((x , y) :: ∅)) ⊑ᵤ (λᵤ f) →
               [ t , u ] x ap↦ ⟨⟨ y ⟩⟩
```

B.0.99 Ucwf/DomainUcwf/LambdaBeta/lam/AxiomProofs

```
{-# OPTIONS --safe --sized-types #-}

open import Base.Variables
open import Ucwf.DomainUcwf.Appmap.Definition

open import Agda.Builtin.Nat

module Ucwf.DomainUcwf.LambdaBeta.lam.AxiomProofs
  {t : uAppmap (suc n) 1} where

open import Base.Core
open import NbhSys.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.lam.Lemmata
open import Ucwf.DomainUcwf.LambdaBeta.lam.Relation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Instance
open import Ucwf.DomainUcwf.UniType.PrePost
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.SizedFinFun
```

```
lam↦-mono : ∀ {x y z} → ⊑ᵥ (nToCtx n) x y →
                [ t ] x lam↦ z → [ t ] y lam↦ z
lam↦-mono {y = y} x⊑y lam↦-intro₁ = lam↦-intro₁
lam↦-mono {x = x} {y = y} x⊑y (lam↦-intro₂ p)
  = lam↦-intro₂ λ xy∈f → Appmap.↦-mono t
    (⊑ᵥ-cons (nToCtx (suc n)) (NbhSys.⊑-refl UniType) x⊑y)
    (p xy∈f)

lam↦-bottom : ∀ {x} → [ t ] x lam↦ ⟨⟨ ⊥ᵤ ⟩⟩
lam↦-bottom = lam↦-intro₁

lam↦-↓closed' : ∀ {x f f'} → [ UniType ] λᵤ f ⊑ λᵤ f' →
                  [ t ] x lam↦ ⟨⟨ λᵤ f' ⟩⟩ →
                  ∀ {x y} → (x , y) ∈ₛ f →
                  [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
lam↦-↓closed' (⊑ᵤ-intro₂ _ _ p) _ xy∈f with (p _ _ xy∈f)
lam↦-↓closed' {x = x} _ tx↦f' xy∈f
  | record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = sub⊆f'
           }
  = Appmap.↦-↓closed t y⊑post' txx↦post
  where y⊑post' = ⊑ᵥ-cons (nToCtx 1) y⊑ᵤpost ⊑ᵥ-nil
        pre⊑post = ⊑ᵥ-cons (nToCtx (suc n)) pre⊑ᵤx
                   (NbhSys.⊑-refl (ValNbhSys _))
        tprex↦post = ↓closed-lemma x sub
                     (shrinklam sub⊆f' tx↦f')
        txx↦post = Appmap.↦-mono t pre⊑post tprex↦post

lam↦-↓closed : ∀ {x y z} → ⊑ᵥ (nToCtx 1) y z →
                 [ t ] x lam↦ z → [ t ] x lam↦ y
lam↦-↓closed {y = ⟨⟨ ⊥ᵤ ,, _ ⟩⟩} {⟨⟨ z ,, _ ⟩⟩}
  (⊑ᵥ-cons _ y⊑z ⊑ᵥ-nil) tx↦z = lam↦-intro₁
lam↦-↓closed {x = x} {⟨⟨ λᵤ f ,, _ ⟩⟩} {⟨⟨ λᵤ f' ,, _ ⟩⟩}
  (⊑ᵥ-cons _ f⊑f' ⊑ᵥ-nil) tx↦f'
  = lam↦-intro₂ (lam↦-↓closed' f⊑f' tx↦f')

lam↦-↑directed' : ∀ {x f f'} → [ t ] x lam↦ ⟨⟨ λᵤ f ⟩⟩ →
                    [ t ] x lam↦ ⟨⟨ λᵤ f' ⟩⟩ → ∀ {x y} →
                    (x , y) ∈ₛ (f ∪ₛ f') →
                    [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
lam↦-↑directed' {f = f} _ _ xy∈f∪f'
  with (∪ₛ-lemma₂ {f = f} xy∈f∪f')
lam↦-↑directed' (lam↦-intro₂ p) _ _
  | inl xy∈f = p xy∈f
lam↦-↑directed' _ (lam↦-intro₂ p) _
  | inr xy∈f' = p xy∈f'
```

```
lam↦-↑directed : ∀ {x y z} → [ t ] x lam↦ y →
                 [ t ] x lam↦ z → (conyz : ValCon _ y z) →
                 [ t ] x lam↦ (y ⊔ᵤ z [ conyz ])
lam↦-↑directed {y = ⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩} {⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩} _ txlam↦z
  (con-tup _ _)
  = txlam↦z
lam↦-↑directed {y = ⟨⟨ λᵤ f ,, ⟨⟩ ⟩⟩} {⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩} txlam↦y _
  (con-tup _ _)
  = txlam↦y
lam↦-↑directed {y = ⟨⟨ ⊥ᵤ ,, ⟨⟩ ⟩⟩} {⟨⟨ λᵤ f' ,, ⟨⟩ ⟩⟩} _ txlam↦z
  (con-tup _ _)
  = txlam↦z
lam↦-↑directed {x = x} {⟨⟨ λᵤ f ,, ⟨⟩ ⟩⟩} {⟨⟨ λᵤ f' ,, ⟨⟩ ⟩⟩}
  txlam↦f txlam↦f' (con-tup _ _)
  = lam↦-intro₂ txx↦y
  where txx↦y = lam↦-↑directed' txlam↦f txlam↦f'
```

B.0.100 Ucwf/DomainUcwf/LambdaBeta/lam/Instance

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.lam.Instance where

open import Base.Variables
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.lam.AxiomProofs
open import Ucwf.DomainUcwf.LambdaBeta.lam.Relation

open import Agda.Builtin.Nat

lam : uAppmap (suc n) 1 → uAppmap n 1
Appmap._↦_ (lam t)         = [_]_lam↦_ t
Appmap.↦-mono (lam t)      = lam↦-mono
Appmap.↦-bottom (lam t)    = lam↦-bottom
Appmap.↦-↓closed (lam t)   = lam↦-↓closed
Appmap.↦-↑directed (lam t) = lam↦-↑directed
Appmap.↦-con (lam t)       = λ _ _ _ → valConAll
```

B.0.101 Ucwf/DomainUcwf/LambdaBeta/lam/Lemmata

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.lam.Lemmata where

open import Base.Variables
```

```
open import NbhSys.Definition
open import NbhSys.Lemmata
open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.LambdaBeta.lam.Relation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Instance
open import Ucwf.DomainUcwf.UniType.PrePost
open import Ucwf.DomainUcwf.UniType.SizedFinFun

open import Agda.Builtin.Nat

private
  variable
    t : uAppmap (suc n) 1

pre-biggest : ∀ f x y → (x , y) ∈ₛ f →
                [ UniType ] x ⊑ pre f
pre-biggest ((x , y) :: f) x y here
  = NbhSys.⊑-⊔-fst UniType con-all
pre-biggest ((x' , y') :: f') x y (there xy∈f')
  = ⊑-⊔-lemma₅ UniType (pre-biggest f' x y xy∈f') con-all

shrinklam : ∀ {x f f'} → f ⊆ₛ f' →
              [ t ] x lam↦ ⟨⟨ λᵤ f' ⟩⟩ → [ t ] x lam↦ ⟨⟨ λᵤ f ⟩⟩
shrinklam {f = f} f⊆f' (lam↦-intro₂ p)
  = lam↦-intro₂ (λ xy∈f → p (f⊆f' xy∈f))

↓closed-lemma' : ∀ x f → [ t ] x lam↦ ⟨⟨ λᵤ f ⟩⟩ →
                         ∀ x y → (x , y) ∈ₛ f →
                         [ t ] ⟨⟨ pre f ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩
↓closed-lemma' {n} {t = t} x (x :: f') (lam↦-intro₂ p)
  x' y' x'y'∈f
  = Appmap.↦-mono t ax⊑pfx (p x'y'∈f)
  where a⊑pf = pre-biggest (x :: f') x' y' x'y'∈f
        ax⊑pfx = ⊑ᵥ-cons (nToCtx (suc n)) a⊑pf
                         (NbhSys.⊑-refl (ValNbhSys _))

↓closed-lemma : ∀ x f →
                        [ t ] x lam↦ ⟨⟨ λᵤ f ⟩⟩ →
                        [ t ] ⟨⟨ pre f ,, x ⟩⟩ ↦ ⟨⟨ post f ⟩⟩
↓closed-lemma {t = t} x ∅ _ = Appmap.↦-bottom t
↓closed-lemma {n} {t = t} x ((x , y) :: f') lamtx↦f
  = Appmap.↦-↑directed t tpref'↦y tfx↦pf' (con-tup con-all con-nil)
  where f' = (x , y) :: f'
        tpref'↦y = ↓closed-lemma' x f' lamtx↦f x y here
        pf'⊑pf = NbhSys.⊑-⊔-snd UniType con-all
        pf'x⊑pfx = ⊑ᵥ-cons (nToCtx (suc n)) pf'⊑pf
```

```
                (NbhSys.⊑-refl (ValNbhSys _))
  tpf'x↦pf' = ↓closed-lemma x f'
                (shrinklam (λ y∈f' → there y∈f')
                lamtx↦f)
  tfx↦pf' = Appmap.↦-mono t pf'x⊑pfx tpf'x↦pf'
```

B.0.102 Ucwf/DomainUcwf/LambdaBeta/lam/Relation

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.LambdaBeta.lam.Relation where

open import Ucwf.DomainUcwf.Appmap.Definition
open import Ucwf.DomainUcwf.Appmap.Valuation
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.SizedFinFun

open import Agda.Builtin.Nat

private
  variable
    n : Nat

data [_]_lam↦_ (t : uAppmap (suc n) 1) :
                 uValuation n → uValuation 1 → Set where
  lam↦-intro₁ : ∀ {x} → [ t ] x lam↦ ⟨⟨ ⊥ᵤ ⟩⟩
  lam↦-intro₂ : ∀ {x f} →
                 (∀ {x y} → (x , y) ∈ₛ f →
                 [ t ] ⟨⟨ x ,, x ⟩⟩ ↦ ⟨⟨ y ⟩⟩) →
                 [ t ] x lam↦ ⟨⟨ λᵤ f ⟩⟩
```

B.0.103 Ucwf/DomainUcwf/UniType/AxiomProofs

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.AxiomProofs where

open import Base.Core hiding (_,_)
open import NbhSys.Definition
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Lemmata
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.SizedFinFun

private
  variable
    x y z : UniNbh
```

```
⊑ᵤ-refl' : ∀ f x y → (x , y) ∈ₛ f → ⊑ᵤ-proof f x y
⊑ᵤ-refl' ((x' , y') :: f') x y (there xy∈f)
  = lift⊑ᵤ-proof f' ((x' , y') :: f') x y
      (λ z∈f → there z∈f) (⊑ᵤ-refl' f' x y xy∈f)
⊑ᵤ-refl' (_ :: f') ⊥ᵤ ⊥ᵤ here
  = record { sub = ∅
           ; y⊑ᵤpost = ⊑ᵤ-intro₁
           ; pre⊑ᵤx = ⊑ᵤ-intro₁
           ; sub⊆f' = ∅-isSubsetₛ
           }
⊑ᵤ-refl' (_ :: f') ⊥ᵤ (λᵤ f) here
  = record { sub = (⊥ᵤ , λᵤ f) :: ∅
           ; y⊑ᵤpost = ⊑ᵤ-intro₂ f f (⊑ᵤ-refl' f)
           ; pre⊑ᵤx = ⊑ᵤ-intro₁
           ; sub⊆f' = ⊆ₛ-lemma₄ here ∅-isSubsetₛ
           }
⊑ᵤ-refl' (_ :: f') (λᵤ f) ⊥ᵤ here
  = record { sub = (λᵤ f , ⊥ᵤ) :: ∅
           ; y⊑ᵤpost = ⊑ᵤ-intro₁
           ; pre⊑ᵤx = ⊑ᵤ-intro₂ f f (⊑ᵤ-refl' f)
           ; sub⊆f' = ⊆ₛ-lemma₄ here ∅-isSubsetₛ
           }
⊑ᵤ-refl' (_ :: f') (λᵤ f) (λᵤ f'') here
  = record { sub = (λᵤ f , λᵤ f'') :: ∅
           ; y⊑ᵤpost = ⊑ᵤ-intro₂ f'' f'' (⊑ᵤ-refl' f'')
           ; pre⊑ᵤx = ⊑ᵤ-intro₂ f f (⊑ᵤ-refl' f)
           ; sub⊆f' = ⊆ₛ-lemma₄ here ∅-isSubsetₛ
           }

⊑ᵤ-refl : ∀ {x} → x ⊑ᵤ x
⊑ᵤ-refl {⊥ᵤ} = ⊑ᵤ-intro₁
⊑ᵤ-refl {λᵤ f} = ⊑ᵤ-intro₂ f f (⊑ᵤ-refl' f)

⊑ᵤ-⊔ᵤ' : ∀ {f f' f''} → λᵤ f' ⊑ᵤ λᵤ f → λᵤ f'' ⊑ᵤ λᵤ f →
          ∀ x y → (x , y) ∈ₛ (f' ∪ₛ f'') →
          ⊑ᵤ-proof f x y
⊑ᵤ-⊔ᵤ' {f' = f'} _ _ x y xy∈∪
  with (∪ₛ-lemma₂ {f = f'} xy∈∪)
⊑ᵤ-⊔ᵤ' (⊑ᵤ-intro₂ _ _ p) _ x y xy∈∪ | inl xy∈f'
  = record { sub = f'sub
           ; y⊑ᵤpost = f'y⊑ᵤpost
           ; pre⊑ᵤx = f'pre⊑ᵤx
           ; sub⊆f' = f'sub⊆f
           }
  where f'proof = p x y xy∈f'
        f'sub = ⊑ᵤ-proof.sub f'proof
```

```
            f'y⊑ᵤpost = ⊑ᵤ-proof.y⊑ᵤpost f'proof
            f'pre⊑ᵤx = ⊑ᵤ-proof.pre⊑ᵤx f'proof
            f'sub⊆f = ⊑ᵤ-proof.sub⊆f' f'proof
⊑ᵤ-⊔ᵤ' _ (⊑ᵤ-intro₂ _ _ p) x y xy∈∪ | inr xy∈f'
  = record { sub = f'sub
           ; y⊑ᵤpost = f'y⊑ᵤpost
           ; pre⊑ᵤx = f'pre⊑ᵤx
           ; sub⊆f' = f'sub⊆f
           }
  where f'proof = p x y xy∈f'
        f'sub = ⊑ᵤ-proof.sub f'proof
        f'y⊑ᵤpost = ⊑ᵤ-proof.y⊑ᵤpost f'proof
        f'pre⊑ᵤx = ⊑ᵤ-proof.pre⊑ᵤx f'proof
        f'sub⊆f = ⊑ᵤ-proof.sub⊆f' f'proof

⊑ᵤ-⊔ᵤ : y ⊑ᵤ x → z ⊑ᵤ x → UniCon y z → (y ⊔ᵤ z [ con-all ]) ⊑ᵤ x
⊑ᵤ-⊔ᵤ {⊥ᵤ} {x} {⊥ᵤ} _ _ _ = ⊑ᵤ-intro₁
⊑ᵤ-⊔ᵤ {λᵤ f} {x} {⊥ᵤ} y⊑x _ _ = y⊑x
⊑ᵤ-⊔ᵤ {⊥ᵤ} {x} {λᵤ f} _ z⊑x _ = z⊑x
⊑ᵤ-⊔ᵤ {λᵤ f'} {λᵤ f} {λᵤ f''} y⊑x z⊑x _
  = ⊑ᵤ-intro₂ (f' ∪ₛ f'') f (⊑ᵤ-⊔ᵤ' y⊑x z⊑x)

⊑ᵤ-⊔ᵤ-helper₁ : ∀ {x} → x ⊑ᵤ (x ⊔ᵤ ⊥ᵤ [ con-all ])
⊑ᵤ-⊔ᵤ-helper₁ {x} rewrite (⊔ᵤ-⊥ᵤ-rightid x)
  = ⊑ᵤ-refl

⊑ᵤ-⊔ᵤ-helper₂ : ∀ {x} → (x ⊔ᵤ ⊥ᵤ [ con-all ]) ⊑ᵤ x
⊑ᵤ-⊔ᵤ-helper₂ {x} rewrite (⊔ᵤ-⊥ᵤ-rightid x)
  = ⊑ᵤ-refl

⊑ᵤ-⊔ᵤ-fst : ∀ {x y} → UniCon x y → x ⊑ᵤ (x ⊔ᵤ y [ con-all ])
⊑ᵤ-⊔ᵤ-fst {⊥ᵤ} _ = ⊑ᵤ-intro₁
⊑ᵤ-⊔ᵤ-fst {λᵤ f} {⊥ᵤ} _ = ⊑ᵤ-refl
⊑ᵤ-⊔ᵤ-fst {λᵤ f} {λᵤ f'} _
  = ⊑ᵤ-intro₂ f (f ∪ₛ f') λ x y xy∈f →
    record { sub = (x , y) :: ∅
           ; y⊑ᵤpost = ⊑ᵤ-⊔ᵤ-helper₁
           ; pre⊑ᵤx = ⊑ᵤ-⊔ᵤ-helper₂
           ; sub⊆f' = ⊆ₛ-lemma₄ (∪ₛ-lemma₃ xy∈f)
                      ∅-isSubsetₛ
           }

⊑ᵤ-⊔ᵤ-snd : ∀ {x y} → UniCon x y → y ⊑ᵤ (x ⊔ᵤ y [ con-all ])
⊑ᵤ-⊔ᵤ-snd {y = ⊥ᵤ} _ = ⊑ᵤ-intro₁
⊑ᵤ-⊔ᵤ-snd {⊥ᵤ} {λᵤ f} _ = ⊑ᵤ-refl
⊑ᵤ-⊔ᵤ-snd {λᵤ f} {λᵤ f'} _
  = ⊑ᵤ-intro₂ f' (f ∪ₛ f') λ x y xy∈f' →
    record { sub = (x , y) :: ∅
```

```
  ; y⊑ᵤpost = ⊑ᵤ-⊔ᵤ-helper₁
  ; pre⊑ᵤx = ⊑ᵤ-⊔ᵤ-helper₂
  ; sub⊆f' = ⊆ₛ-lemma₄ (∪ₛ-lemma₄ xy∈f')
               ∅-isSubsetₛ
  }
```

B.0.104 Ucwf/DomainUcwf/UniType/Definition

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.Definition where

open import Agda.Builtin.Size

-- We give pairs, finite functions, neighborhoods of the
-- universal type, and related concepts sizes.
-- This lets Agda see that the recursion used in the
-- transitivity proof is well-founded.
data ×ₛ : {i : Size} → Set
data FinFunₛ : {i : Size} → Set
data UniNbh : {i : Size} → Set

data ×ₛ where
  _,_ : ∀ {i} → (x y : UniNbh {i}) → ×ₛ {i}

data FinFunₛ where
  ∅ : ∀ {i} → FinFunₛ {i}
  _::_ : ∀ {i} → ×ₛ {i} → FinFunₛ {i} → FinFunₛ {i}

data UniNbh where
  ⊥ᵤ : ∀ {i} → UniNbh {i}
  -- Note that λᵤ increases the size!
  λᵤ : ∀ {i} → FinFunₛ {i} → UniNbh {↑ i}

_∪ₛ_ : ∀ {i} → FinFunₛ {i} → FinFunₛ {i} → FinFunₛ {i}
∅ ∪ₛ f' = f'
(x :: f) ∪ₛ f' = x :: (f ∪ₛ f')

data UniCon : UniNbh → UniNbh → Set where
  con-all : ∀ {x y} → UniCon x y

_⊔ᵤ_[_] : ∀ {i} → (x y : UniNbh {i}) →
          UniCon x y → UniNbh {i}
⊥ᵤ ⊔ᵤ ⊥ᵤ [ _ ] = ⊥ᵤ
⊥ᵤ ⊔ᵤ (λᵤ f) [ _ ] = λᵤ f
(λᵤ f) ⊔ᵤ ⊥ᵤ [ _ ] = λᵤ f
(λᵤ f) ⊔ᵤ (λᵤ f') [ _ ] = λᵤ (f ∪ₛ f')
```

B.0.105 Ucwf/DomainUcwf/UniType/Instance

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.Instance where

open import Base.Core
open import NbhSys.Definition
open import Ucwf.DomainUcwf.UniType.AxiomProofs
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.Transitivity

open import Agda.Builtin.Nat

UniType : NbhSys
NbhSys.Nbh UniType     = UniNbh
NbhSys._⊑_ UniType     = _⊑ᵤ_
NbhSys.Con UniType     = UniCon
NbhSys._⊔_[_] UniType = _⊔ᵤ_[_]
NbhSys.⊥ UniType       = ⊥ᵤ
NbhSys.⊑-refl UniType  = ⊑ᵤ-refl
NbhSys.⊑-trans UniType = ⊑ᵤ-trans
NbhSys.⊑-⊥ UniType     = ⊑ᵤ-intro₁
NbhSys.⊑-⊔ UniType     = ⊑ᵤ-⊔ᵤ
NbhSys.⊑-⊔-fst UniType = ⊑ᵤ-⊔ᵤ-fst
NbhSys.⊑-⊔-snd UniType = ⊑ᵤ-⊔ᵤ-snd
NbhSys.Con-⊔ UniType  = λ _ _ → con-all

-- In a ucwf contexts are simply natural numbers.
-- As we want to use approximable mappings as initially
-- defined for scwfs, we define a function that "translates"
-- natural numbers to scwf-contexts.
nToCtx : ∀ (n) → Ctx n
nToCtx zero = []
nToCtx (suc n) = UniType :: (nToCtx n)
```

B.0.106 Ucwf/DomainUcwf/UniType/Lemmata

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.Lemmata where

open import Base.Core hiding (_,_)
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.PrePost
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.SizedFinFun
```

```
open import Agda.Builtin.Equality
open import Agda.Builtin.Size

lift⊑ᵤ-proof : ∀ {i j} → (f f' : FinFunₛ {i}) →
                  (x y : UniNbh {j}) → f ⊆ₛ f' →
                  ⊑ᵤ-proof f x y → ⊑ᵤ-proof f' x y
lift⊑ᵤ-proof f f' x y f⊆f'
  record { sub = sub
         ; y⊑ᵤpost = y⊑ᵤpost
         ; pre⊑ᵤx = pre⊑ᵤx
         ; sub⊆f' = sub⊆f'
         }
  = record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = ⊆ₛ-trans sub⊆f' f⊆f'
           }

shrink⊑ᵤ : ∀ {i j} → {f f' : FinFunₛ {i}} →
           {f'' : FinFunₛ {j}} → λᵤ f' ⊑ᵤ λᵤ f'' → f ⊆ₛ f' →
           λᵤ f ⊑ᵤ λᵤ f''
shrink⊑ᵤ {f = f} {f'} {f''} (⊑ᵤ-intro₂ _ _ p) f⊆f'
  = ⊑ᵤ-intro₂ f f'' (λ x y xy∈f → p x y (f⊆f' xy∈f))

∅-⊥ᵤ : ∀ {f} → (λᵤ ∅) ⊑ᵤ (λᵤ f)
∅-⊥ᵤ {f} = ⊑ᵤ-intro₂ ∅ f (λ x y → xy∈∅-abs)

⊑ᵤ-⊔ᵤ-lemma₁ : ∀ {i j} → (x : UniNbh {i}) →
                  (y z : UniNbh {j}) → x ⊑ᵤ y →
                  x ⊑ᵤ (y ⊔ᵤ z [ con-all ])
⊑ᵤ-⊔ᵤ-lemma₁ _ y z ⊑ᵤ-intro₁ = ⊑ᵤ-intro₁
⊑ᵤ-⊔ᵤ-lemma₁ (λᵤ f) (λᵤ f') ⊥ᵤ (⊑ᵤ-intro₂ _ _ p)
  = ⊑ᵤ-intro₂ f f' p
⊑ᵤ-⊔ᵤ-lemma₁ (λᵤ f) (λᵤ f') (λᵤ f'') (⊑ᵤ-intro₂ _ _ p) =
  ⊑ᵤ-intro₂ f (f' ∪ₛ f'') (λ x' y' x'y'∈f →
    lift⊑ᵤ-proof f' (f' ∪ₛ f'') x' y' ∪ₛ-lemma₃ (p x' y' x'y'∈f))

⊑ᵤ-⊔ᵤ-lemma₂ : ∀ {i j} → (x : UniNbh {i}) →
                  (y z : UniNbh {j}) → x ⊑ᵤ z →
                  x ⊑ᵤ (y ⊔ᵤ z [ con-all ])
⊑ᵤ-⊔ᵤ-lemma₂ _ y z ⊑ᵤ-intro₁ = ⊑ᵤ-intro₁
⊑ᵤ-⊔ᵤ-lemma₂ (λᵤ f) ⊥ᵤ (λᵤ f'') (⊑ᵤ-intro₂ _ _ p)
  = ⊑ᵤ-intro₂ f f'' p
⊑ᵤ-⊔ᵤ-lemma₂ (λᵤ f) (λᵤ f') (λᵤ f'') (⊑ᵤ-intro₂ _ _ p)
  = ⊑ᵤ-intro₂ f (f' ∪ₛ f'') (λ x' y' x'y'∈f →
      lift⊑ᵤ-proof f'' (f' ∪ₛ f'') x' y' ∪ₛ-lemma₄ (p x' y' x'y'∈f))
```

```
⊑ᵤ-⊔ᵤ-lemma₃' : ∀ {i j} → {f f' : FinFunₛ {i}} →
                  {f'' f''' : FinFunₛ {j}} → (λᵤ f) ⊑ᵤ (λᵤ f'') →
                  (λᵤ f') ⊑ᵤ (λᵤ f''') →
                  ∀ x y → (x , y) ∈ₛ (f ∪ₛ f') →
                  ⊑ᵤ-proof (f'' ∪ₛ f''') x y
⊑ᵤ-⊔ᵤ-lemma₃' {f = f} {f'} _ _ x y xy∈∪
  with (∪ₛ-lemma₂ {f = f} xy∈∪)
⊑ᵤ-⊔ᵤ-lemma₃' (⊑ᵤ-intro₂ _ _ p) _ x y _
  | inl xy∈f with (p x y xy∈f)
⊑ᵤ-⊔ᵤ-lemma₃' {f'' = f''} {f'''} _ _ _ _ _ | _
  | record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = sub⊆f'
           }
  = record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = λ x'y'∈sub → ∪ₛ-lemma₃ (sub⊆f' x'y'∈sub)
           }
⊑ᵤ-⊔ᵤ-lemma₃' _ (⊑ᵤ-intro₂ _ _ p) x y _
  | inr xy∈f' with (p x y xy∈f')
⊑ᵤ-⊔ᵤ-lemma₃' {f'' = f''} {f'''} _ _ _ _ _ | _
  | record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = sub⊆f'
           }
  = record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = λ x'y'∈sub → ∪ₛ-lemma₄ (sub⊆f' x'y'∈sub)
           }

⊑ᵤ-⊔ᵤ-lemma₃ : ∀ {i j} → (x y : UniNbh {i}) →
                 (z w : UniNbh {j}) → x ⊑ᵤ z → y ⊑ᵤ w →
                 (x ⊔ᵤ y [ con-all ]) ⊑ᵤ (z ⊔ᵤ w [ con-all ])
⊑ᵤ-⊔ᵤ-lemma₃ ⊥ᵤ ⊥ᵤ _ _ _ _ = ⊑ᵤ-intro₁
⊑ᵤ-⊔ᵤ-lemma₃ (λᵤ f) _ z w x⊑z ⊑ᵤ-intro₁
  = ⊑ᵤ-⊔ᵤ-lemma₁ (λᵤ f ⊔ᵤ ⊥ᵤ [ con-all ]) z w x⊑z
⊑ᵤ-⊔ᵤ-lemma₃ ⊥ᵤ (λᵤ f') z (λᵤ f'''') _ y⊑w
  = ⊑ᵤ-⊔ᵤ-lemma₂ (⊥ᵤ ⊔ᵤ λᵤ f' [ con-all ]) z (λᵤ f'''') y⊑w
⊑ᵤ-⊔ᵤ-lemma₃ _ _ _ _ (⊑ᵤ-intro₂ f f' p₁) (⊑ᵤ-intro₂ f'' f''' p₂)
  = ⊑ᵤ-intro₂ (f ∪ₛ f'') (f' ∪ₛ f''') f∪f''⊑f'∪f'''
  where f∪f''⊑f'∪f''' = ⊑ᵤ-⊔ᵤ-lemma₃' (⊑ᵤ-intro₂ f f' p₁)
                                      (⊑ᵤ-intro₂ f'' f''' p₂)
```

```
⊔ᵤ-⊥ᵤ-leftid : ∀ {i} → (x : UniNbh {i}) →
                 ⊥ᵤ ⊔ᵤ x [ con-all ] ≡ x
⊔ᵤ-⊥ᵤ-leftid ⊥ᵤ = refl
⊔ᵤ-⊥ᵤ-leftid (λᵤ f) = refl

⊔ᵤ-⊥ᵤ-rightid : ∀ {i} → (x : UniNbh {i}) →
                  x ⊔ᵤ ⊥ᵤ [ con-all ] ≡ x
⊔ᵤ-⊥ᵤ-rightid ⊥ᵤ = refl
⊔ᵤ-⊥ᵤ-rightid (λᵤ f) = refl

⊔ᵤ-ass' : {i : Size} → {x y : ×ₛ {i}} → {f f' : FinFunₛ {i}} →
          x ≡ y → λᵤ f ≡ λᵤ f' →
          λᵤ (x :: f) ≡ λᵤ (y :: f')
⊔ᵤ-ass' refl refl = refl

⊔ᵤ-ass : {i : Size} → (x y z : UniNbh {i}) →
         (x ⊔ᵤ y [ con-all ]) ⊔ᵤ z [ con-all ] ≡
         x ⊔ᵤ (y ⊔ᵤ z [ con-all ]) [ con-all ]
⊔ᵤ-ass ⊥ᵤ ⊥ᵤ z rewrite (⊔ᵤ-⊥ᵤ-leftid (⊥ᵤ ⊔ᵤ z [ con-all ])) = refl
⊔ᵤ-ass ⊥ᵤ (λᵤ f') z rewrite (⊔ᵤ-⊥ᵤ-leftid (λᵤ f' ⊔ᵤ z [ con-all ]))
  = refl
⊔ᵤ-ass (λᵤ f) ⊥ᵤ ⊥ᵤ = refl
⊔ᵤ-ass (λᵤ f) ⊥ᵤ (λᵤ f') = refl
⊔ᵤ-ass (λᵤ f) (λᵤ f') ⊥ᵤ = refl
⊔ᵤ-ass (λᵤ ∅) (λᵤ f') (λᵤ f'') = refl
⊔ᵤ-ass (λᵤ ((x₁ , x₂) :: g)) (λᵤ f') (λᵤ f'')
  = ⊔ᵤ-ass' refl (⊔ᵤ-ass (λᵤ g) (λᵤ f') (λᵤ f''))

⊔ᵤ-cong : {i : Size} → {x y z w : UniNbh {i}} → x ≡ z →
          y ≡ w → (x ⊔ᵤ y [ con-all ]) ≡ (z ⊔ᵤ w [ con-all ])
⊔ᵤ-cong refl refl = refl

post-≡ : {i : Size} → (f f' : FinFunₛ {i}) →
         post (f ∪ₛ f') ≡ (post f ⊔ᵤ post f' [ con-all ])
post-≡ ∅ f' rewrite (⊔ᵤ-⊥ᵤ-leftid (post f')) = refl
post-≡ ((x₁ , x₂) :: g) f'
  rewrite (⊔ᵤ-ass x₂ (post g) (post f'))
  = ⊔ᵤ-cong refl (post-≡ g f')

pre-≡ : {i : Size} → (f f' : FinFunₛ {i}) →
        pre (f ∪ₛ f') ≡ (pre f ⊔ᵤ pre f' [ con-all ])
pre-≡ ∅ f' rewrite (⊔ᵤ-⊥ᵤ-leftid (pre f')) = refl
pre-≡ ((x₁ , x₂) :: g) f'
  rewrite (⊔ᵤ-ass x₁ (pre g) (pre f'))
  = ⊔ᵤ-cong refl (pre-≡ g f')
```

B.0.107 Ucwf/DomainUcwf/UniType/PrePost

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.PrePost where

open import Ucwf.DomainUcwf.UniType.Definition

pre : ∀ {i} → FinFun_s {i} → UniNbh {i}
pre ∅ = ⊥_u
pre ((x , y) :: f) = x ⊔_u pre f [ con-all ]

post : ∀ {i} → FinFun_s {i} → UniNbh {i}
post ∅ = ⊥_u
post ((x , y) :: f) = y ⊔_u post f [ con-all ]
```

B.0.108 Ucwf/DomainUcwf/UniType/Relation

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.Relation where

open import Base.Core hiding (_,_)
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.PrePost
open import Ucwf.DomainUcwf.UniType.SizedFinFun

open import Agda.Builtin.Size

private
  variable i j : Size

record ⊑_u-proof {i j : Size} (f : FinFun_s {j})
                 (x y : UniNbh {i}) : Set

data _⊑_u_ : UniNbh {i} → UniNbh {j} → Set

record ⊑_u-proof f x y where
  inductive
  field
    sub : FinFun_s
    y⊑_upost : y ⊑_u (post sub)
    pre⊑_ux : (pre sub) ⊑_u x
    sub⊆f' : sub ⊆_s f

data _⊑_u_ where
  ⊑_u-intro_1 : ∀ {i j} → {x : UniNbh {j}} →
                (⊥_u {i}) ⊑_u x
  ⊑_u-intro_2 : ∀ {i j} → (f : FinFun_s {i}) →
                (f' : FinFun_s {j}) →
                ((x y : UniNbh {i}) → (x , y) ∈_s f →
                ⊑_u-proof {i} {j} f' x y) →
                _⊑_u_ {↑ i} {↑ j} (λ_u f) (λ_u f')
```

B.0.109 Ucwf/DomainUcwf/UniType/SizedFinFun

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.SizedFinFun where

open import Base.Core hiding (_,_)
open import Ucwf.DomainUcwf.UniType.Definition

open import Agda.Builtin.Size
open import Agda.Primitive

private
  variable
    f f' f'' f''' : FinFun_s

data _∈_s_ : {i : Size} → ×_s {i} → FinFun_s {i} → Set where
  here : {i : Size} → {x : ×_s {i}} → {f : FinFun_s {i}} →
         x ∈_s (x :: f)
  there : {i : Size} → {x x' : ×_s {i}} → {f : FinFun_s {i}} →
          x ∈_s f → x ∈_s (x' :: f)

_⊆_s : {i : Size} → FinFun_s {i} → FinFun_s {i} → Set
_⊆_s {i} f f' = ∀ {x} → _∈_s {i} x f → _∈_s {i} x f'

⊆_s-refl : {i : Size} → {f : FinFun_s {i}} → f ⊆_s f
⊆_s-refl x∈f = x∈f

⊆_s-trans : {i : Size} → {f f' f'' : FinFun_s {i}} → f ⊆_s f' →
            f' ⊆_s f'' → f ⊆_s f''
⊆_s-trans f⊆f' f'⊆f'' x∈f = f'⊆f'' (f⊆f' x∈f)

⊆_s-lemma_3 : {i : Size} → ∀ {x} → {f : FinFun_s {i}} →
              f ⊆_s (x :: f)
⊆_s-lemma_3 y∈f = there y∈f

⊆_s-lemma_4 : ∀ {x} → x ∈_s f → f' ⊆_s f → (x :: f') ⊆_s f
⊆_s-lemma_4 x∈f _ here = x∈f
⊆_s-lemma_4 x∈f f'⊆f (there y∈f) = f'⊆f y∈f

∅-isSubset_s : {i : Size} → {f : FinFun_s {i}} → ∅ ⊆_s f
∅-isSubset_s ()

∪_s-lemma_1 : {i : Size} → {f f' f'' : FinFun_s {i}} →
              f ⊆_s f'' → f' ⊆_s f'' → (f ∪_s f') ⊆_s f''
∪_s-lemma_1 {f = ∅} _ f'⊆f'' y∈f∪f' = f'⊆f'' y∈f∪f'
∪_s-lemma_1 {f = x :: _} f⊆f'' _ here = f⊆f'' here
∪_s-lemma_1 {f = x :: f'''} f⊆f'' f'⊆f'' (there y∈f∪f')
  = ∪_s-lemma_1 (⊆_s-trans ⊆_s-lemma_3 f⊆f'') f'⊆f'' y∈f∪f'
```

```
∪ₛ-lemma₂ : {i : Size} → {f f' : FinFunₛ {i}} → ∀ {x} →
              x ∈ₛ (f ∪ₛ f') → (x ∈ₛ f) ∨ (x ∈ₛ f')
∪ₛ-lemma₂ {f = ∅} here = inr here
∪ₛ-lemma₂ {f = ∅} (there x∈xs) = inr (there x∈xs)
∪ₛ-lemma₂ {f = x :: _} here = inl here
∪ₛ-lemma₂ {f = x :: f'} (there y∈∪)
  with (∪ₛ-lemma₂ y∈∪)
∪ₛ-lemma₂ (there y∈∪) | inl y∈f'' = inl (there y∈f'')
∪ₛ-lemma₂ (there y∈∪) | inr y∈f' = inr y∈f'

∪ₛ-lemma₃ : {i : Size} → {f f' : FinFunₛ {i}} → ∀ {x} →
              x ∈ₛ f → x ∈ₛ (f ∪ₛ f')
∪ₛ-lemma₃ {f = x :: f'} here = here
∪ₛ-lemma₃ {f = x :: f'} (there y∈f'') = ⊆ₛ-lemma₃ y∈f'∪f'
  where y∈f'∪f' = ∪ₛ-lemma₃ y∈f''

∪ₛ-lemma₄ : {i : Size} → {f f' : FinFunₛ {i}} → ∀ {x} →
              x ∈ₛ f' → x ∈ₛ (f ∪ₛ f')
∪ₛ-lemma₄ {f = ∅} x∈f' = x∈f'
∪ₛ-lemma₄ {f = x :: f'} y∈f' = ⊆ₛ-lemma₃ y∈f'∪f'
  where y∈f'∪f' = ∪ₛ-lemma₄ y∈f'

∪ₛ-lemma₅ : f ⊆ₛ f' → f' ⊆ₛ f''' → (f ∪ₛ f') ⊆ₛ (f'' ∪ₛ f''')
∪ₛ-lemma₅ _ _ x∈f∪f' with (∪ₛ-lemma₂ x∈f∪f')
∪ₛ-lemma₅ f⊆f'' _ _ | inl x∈f = ∪ₛ-lemma₃ (f⊆f'' x∈f)
∪ₛ-lemma₅ _ f'⊆f''' _ | inr x∈f' = ∪ₛ-lemma₄ (f'⊆f''' x∈f')

-- From a proof that a pair of neighborhoods is
-- in the empty set, anything.
xy∈∅-abs : {p : Set} → ∀ {x y} → (x , y) ∈ₛ ∅ → p
xy∈∅-abs ()
```

B.0.110 Ucwf/DomainUcwf/UniType/Transitivity

```
{-# OPTIONS --safe --sized-types #-}

module Ucwf.DomainUcwf.UniType.Transitivity where

open import Base.Core
open import Ucwf.DomainUcwf.UniType.AxiomProofs
open import Ucwf.DomainUcwf.UniType.Definition
open import Ucwf.DomainUcwf.UniType.Lemmata
open import Ucwf.DomainUcwf.UniType.Relation
open import Ucwf.DomainUcwf.UniType.PrePost
open import Ucwf.DomainUcwf.UniType.SizedFinFun

open import Agda.Builtin.Size
```

```
private
  variable
    i j : Size

record ⊑ᵤ-proof₂ (f : FinFunₛ {i}) (f' : FinFunₛ {j}) :
                  Set where
  field
    sub : FinFunₛ
    pf⊑ᵤpost : (post f) ⊑ᵤ (post sub)
    pre⊑ᵤpf : (pre sub) ⊑ᵤ (pre f)
    sub⊆f' : sub ⊆ₛ f'

Ω-post : ∀ {i j} → {x y : UniNbh {i}} →
         {f f' : FinFunₛ {j}} → x ⊑ᵤ post f →
         y ⊑ᵤ post f' → (x ⊔ᵤ y [ con-all ]) ⊑ᵤ post (f ∪ₛ f')
Ω-post {x = x} {y} {f} {f'} x⊑postf y⊑postf' rewrite (post-≡ f f')
  = ⊑ᵤ-⊔ᵤ-lemma₃ x y (post f) (post f') x⊑postf y⊑postf'

Ω-pre : ∀ {i j} → {x y : UniNbh {i}} →
        {f f' : FinFunₛ {j}} → pre f ⊑ᵤ x →
        pre f' ⊑ᵤ y → pre (f ∪ₛ f') ⊑ᵤ (x ⊔ᵤ y [ con-all ])
Ω-pre {x = x} {y} {f} {f'} pref⊑x pref'⊑y rewrite (pre-≡ f f')
  = ⊑ᵤ-⊔ᵤ-lemma₃ (pre f) (pre f') x y pref⊑x pref'⊑y

Ω : ∀ {i j} → (f : FinFunₛ {i}) → (f' : FinFunₛ {j}) →
    (λᵤ f) ⊑ᵤ (λᵤ f') → ⊑ᵤ-proof₂ {i} {j} f f'
Ω ∅ f' _ =
  record { sub = ∅
         ; pf⊑ᵤpost = ⊑ᵤ-intro₁
         ; pre⊑ᵤpf = ⊑ᵤ-intro₁
         ; sub⊆f' = ∅-isSubsetₛ
         }
Ω ((x₁ , x₂) :: f'') f' (⊑ᵤ-intro₂ _ _ p)
  with (p x₁ x₂ here)
Ω ((x₁ , x₂) :: f'') f' (⊑ᵤ-intro₂ _ _ p)
  | record { sub = sub
           ; y⊑ᵤpost = y⊑ᵤpost
           ; pre⊑ᵤx = pre⊑ᵤx
           ; sub⊆f' = sub⊆f'
           }
  = record { sub = sub ∪ₛ sub'
           ; pf⊑ᵤpost = Ω-post {f = sub} y⊑ᵤpost pf⊑ᵤpost'
           ; pre⊑ᵤpf = Ω-pre {f = sub} pre⊑ᵤx pre⊑ᵤpf'
           ; sub⊆f' = ∪ₛ-lemma₁ sub⊆f' sub'⊆f'
           }
  where recur = Ω f'' f' (⊑ᵤ-intro₂ f'' f'
                (λ a b ab∈f'' → p a b (there ab∈f'')))
```

```
            sub' = ⊑ᵤ-proof₂.sub recur
            pf⊑ᵤpost' = ⊑ᵤ-proof₂.pf⊑ᵤpost recur
            pre⊑ᵤpf' = ⊑ᵤ-proof₂.pre⊑ᵤpf recur
            sub'⊆f' = ⊑ᵤ-proof₂.sub⊆f' recur

⊑ᵤ-trans : ∀ {i x} → {y : UniNbh {i}} → ∀ {z} →
             x ⊑ᵤ y → y ⊑ᵤ z → x ⊑ᵤ z

⊑ᵤ-trans' : ∀ {i} → ∀ f → (f' : FinFunₛ {i}) → ∀ f'' →
             (λᵤ f) ⊑ᵤ (λᵤ f') → (λᵤ f') ⊑ᵤ (λᵤ f'') →
             ∀ x' y' → (x' , y') ∈ₛ f → ⊑ᵤ-proof f'' x' y'

⊑ᵤ-trans {x = ⊥ᵤ} _ _ = ⊑ᵤ-intro₁
⊑ᵤ-trans {x = λᵤ ∅} {⊥ᵤ} {⊥ᵤ} ()
⊑ᵤ-trans {x = λᵤ ∅} {λᵤ f'} {⊥ᵤ} (⊑ᵤ-intro₂ _ _ _) ()
⊑ᵤ-trans {x = λᵤ ∅} {λᵤ f'} {λᵤ f''} _ _ = ∅-⊥ᵤ
⊑ᵤ-trans {x = λᵤ (x :: g)} {λᵤ f'} {λᵤ f''} x⊑y y⊑z
  = ⊑ᵤ-intro₂ (x :: g) f'' (⊑ᵤ-trans' (x :: g) f' f'' x⊑y y⊑z)

⊑ᵤ-trans' f f' f'' (⊑ᵤ-intro₂ _ _ p) f'⊑f'' x y xy∈f
  with (p x y xy∈f)
⊑ᵤ-trans' f f' f'' f⊑f' f'⊑f'' x y xy∈f
  | record { sub = sub' ; sub⊆f' = sub⊆f' }
  with (Ω sub' f'' (shrink⊑ᵤ {f' = f'} f'⊑f'' sub⊆f'))
⊑ᵤ-trans' f f' f'' (⊑ᵤ-intro₂ _ _ p) f'⊑f'' x y xy∈f
  | record { sub = sub'
           ; pre⊑ᵤx = pre'⊑ᵤx
           ; y⊑ᵤpost = y⊑ᵤpost'
           ; sub⊆f' = sub'⊆f'
           }
  | record { sub = sub''
           ; pf⊑ᵤpost = pf⊑ᵤpost''
           ; pre⊑ᵤpf = pre''⊑ᵤpf
           ; sub⊆f' = sub''⊆f''
           }
  = record { sub = sub''
           ; y⊑ᵤpost = ⊑ᵤ-trans y⊑ᵤpost' pf⊑ᵤpost''
           ; pre⊑ᵤx = ⊑ᵤ-trans pre''⊑ᵤpf pre'⊑ᵤx
           ; sub⊆f' = sub''⊆f''
           }
```