Implementing the Model Theory of the λ-calculus

Konstantinos Brilakis

February 13, 2018

Contents

Chapter 1

Introduction

Martin-Löf type theory, also known as intuitionistic type theory, is a formal system which provides an alternative option to set theory for laying out the philosophical foundations of mathematics, specifically from the side of mathematical constructivism. An important distinction between set theory and type theory is that the former is built on-top of some deductive system like predicate logic, whereas type theory is not and is a deductive system per se. To elaborate further, this theory is based on intuitionistic logic and in particular, it internalizes the Brouwer–Heyting–Kolmogorov (BHK) interpretation of intuitionistic logic. In contrast to classical logic, an intuitionist would deny the validity of the law of excluded middle and double negation elimination, which implies that proofs by contradiction are not permissible. Moreover, the theory heavily relies on the Curry-Howard isomorphism (propositions-as-types principle), which in short, demonstrates a direct connection between computer programs and mathematical proofs [1]. This correspondence was also extended to category theory by Lambek [2] that brought cartesian closed categories as a part of a three way isomorphism.

The theory itself can be thought of as a typed functional programming language since it extends the simply typed λ-calculus with dependent types and a noteworthy property that follows is that all definable functions are total and always terminate. Martin-Löf type theory is quite expressive and notably suited for program construction because it allows the expression of both specifications and programs. Additionally, the verification of program correctness is also possible in the very same formalism [1].

Furthermore, by utilizing the language of category theory, one can talk about models of type theory. And considering the fact that categorical and type theoretical notions have been shown to be related by Lawvere, a plethora of such models have been constructed over the years. Hence, type theories have numerous categorical models worthy of attention. For example, the simply typed lambda calculus is the internal language of Cartesian closed categories. This thesis, on the other hand, is concerned with a different model; one that provides a fundamental framework for dependent type theories, namely, categories with families (cwfs) introduced by Dybjer [4]. It is a concept similar to Cartmell's categories with attributes [4] (they are in fact equivalent) [5], albeit cwfs are overtly closer to the syntax of dependent types, which is advantageous. It revolves around the idea of objects corresponding to contexts and morphisms to substitutions. A cwf is a model of dependent type theory [6] and cwfs can be formalized as a generalized algebraic theory providing some benefits like clear syntax [4]. This presentation is an algebraic formulation of type theory and some may consider it more canonical.

There are in fact different versions of cwfs that model more basic theories such as the untyped and simply typed λ-calculus, i.e., unityped cwfs (ucwfs) and simply typed cwfs (scwfs). Technically however, to acquire a model of the corresponding λ-calculus in both cases, the said cwfs are supplemented with some extra structure.

The work in this thesis concerns itself with a formalization in Agda of different cwf types and, in particular, the construction of cwf objects in the considered category of cwfs. The underlying motivation of the project is the construction of various cwf initial objects and demonstrating isomorphisms between them. The presentation of each cwf is expressed as a generalized algebraic theory. Thus, the first part is concerned with the construction of unityped cwf objects which will include the untyped λ-calculus. Second, simply typed cwf objects that model simply typed λ-calculus are constructed in two distinct versions: one based on raw syntax supplemented with an external typing relation and one with directly typed terms. Lastly, we formalize full cwf objects with Π types and a universe of small types ala Russel that are expressed through typing relations on raw syntax.

1.1 Aim

The aim of this thesis is to formalize unityped, simply typed, and some parts of the complete cwfs in the proof assistant Agda. In more detail, for each formalization we consider cwf objects in the category of cwfs and attempt to construct two initial objects in that category and show that they are isomorphic. Practically, this means that a calculus based on the generalized algebraic theory of cwfs has to be implemented, where all operators and laws are explicitly defined as part of the language. Simultaneously, the corresponding conventional λ-calculus with formalizations in the meta language is implemented. These two calculi are viewed as cwf objects and therefore morphisms between them can be defined with the purpose of showing that there exists an isomorphism.

The implementation steps for each cwf are roughly the following.

1. Define the abstract notion of a cwf object as a generalized algebraic theory.
2. Construct an initial cwf object using cwf combinators.
3. Construct the corresponding λ-calculus and prove that it is a cwf object.
4. Show the isomorphism between these two cwf objects.

Furthermore, there are a number of available options concerning the way one can formalize these concepts, particularly when types are added. For instance, for simple types, the immediate choice is whether to start with raw syntax and add typing relations or implemented directly typed terms. For this reason both versions are covered for the simply typed cwfs as this exercise is insightful in its own right. Alas, for dependent type theory and the complete cwfs, it is not clear how to formalize intrinsically typed terms, thus the construction there builds from raw syntax and typing rules.

Chapter 2

Background

2.1 Related work

This project has significance to the foundations of type theory and in particular, the fact that the use of a proof assistant to implement various related notions to cement the validity of certain results has attracted interest recently. Relevant work includes the study of categorical structures used to model type theories, e.g., [7] (Ahrens, Lumsdain, Voevodsky); the work in that paper is also formalized, but in the Coq theorem prover.

The definition of cwfs as a generalized algebraic theory from *Internal Type Theory* [4] is the basis of the formalizations. This formulation is modified accordingly to attain all cwf versions that are implemented in this project. The complete cwfs as a model of dependent type theory are discussed by Castellan, Clairambault, Dybjer [6] extensively where dependent type theory is constructed as the initial cwf with extra structure.

Furthermore, the pure untyped λ-calculus is formalized as presented in *Inductive Families* [8] that involve scope safe lambda terms and nameless variables using de Bruijn indices. Lastly, several supporting tools for the Agda implementation of the simply typed λ-calculus are based on a formalization of nbe by G. Allais [?].

2.2 Agda

The proofs and definitions of this thesis are formalized in the Agda language [9]. Agda is a dependently-typed functional programming language based on intuitionistic type theory. It has inductive families and enjoys mixfix operators and unicode support. Agda allows users to not only implement, but express and prove properties about their programs. Proofs are written in a functional style similar to Haskell and moreover, it has numerous libraries readily available.

2.3 Categories with Families

2.3.1 CwFs Definition

Categories with families are a model for a basic framework of dependent types where the syntax resembles closely the standard syntax of dependent types, while providing a clear categorical description [8]. As is the case with other categorical models of type theories, we want to view a category with contexts and substitutions as the objects and morphisms and, in particular, to view terms and types as families over contexts. Next, we describe the definition of CwFs as defined in *Internal Type Theory*.

Let **Fam** be the category of families of sets where objects are families of sets $(B(x))_{x \in A}$. A morphism in **Fam** from an object $(B(x))_{x \in A}$ to $(B'(x'))_{x' \in A'}$ is a pair of (i) a function $f : A \to A'$ and (ii) a family of functions $g(x) : B(x) \to B'(f(x))$ indexed by $x \in A$.

Definition 2.1. (**Category with families**) A cwf has four components.

- A base category C where objects are contexts and morphisms substitutions. The identity map is called $id : \Gamma \to \Gamma$ and composition of maps $\gamma : \Delta \to \Gamma$ and $\delta : \Theta \to \Delta$ is denoted as $\gamma \circ \delta : \Theta \to \Gamma$, where $\Gamma, \Delta, \Theta \in C$.

- A functor $T : C^{op} \to Fam$. One writes $T(\Gamma) = (\Gamma \vdash A)_{(A \in Type(\Gamma))}$, where Γ is a context (an object in C), to describe a family of types indexed by types in Γ. Additionally, let γ be a morphism of C, then $T(\gamma)$ interprets substitution both for terms and types. One writes $A\,[\gamma]$ to apply the first component of $T(\Gamma)$ to type A and $\alpha\,[\gamma]$ to apply the second to term α.

- A terminal object $\Diamond$ of C, the empty context. For each context Γ, there exists a unique morphism $!_\Gamma : \Gamma \to \Diamond$.

- Context comprehension; if we have a context Γ and a type over Γ, i.e., $A \in Ty(\Gamma)$, we can form a context Γ, A such that there exists a morphism $p : \Gamma, A \to \Gamma$ the projection morphism, a term $q \in \Gamma, A \vdash A\,[p]$. Moreover, p and q are the first and second projections respectively and so the universal property is that: for any object Δ in C, morphism $\gamma : \Delta \to \Gamma$ and term $\alpha \in \Delta \vdash A\,[\gamma]$, there is a unique morphism $\theta = \langle \gamma, \alpha \rangle : \Delta \to \Gamma, A$ such that $p \circ \theta = \gamma$ and $q\,[\theta] = \alpha$.

Furthermore, it is important to explain what a cwf morphism is. A cwf can be denoted by a pair (C, T) by a base category C and a functor T as defined above. Consequently, a morphism of cwf objects with domain (C, T) and codomain (C', T') is a pair (F, η) where $F : C \to C'$ is a functor and $\eta : T \to T'F$ is a natural transformation such that terminal object and context comprehension are preserved entirely.

$$(C, T) \xrightarrow{(F\eta)} (C', T')$$

This definition refers to a *strict* cwf morphism, in contrast to a weaker version called *pseudo* cwf morphism in which cwf-structure is preserved only up to isomorphism.

2.3.2 Generalized Algebraic Theory of CwFs

Categories with families can be formalized as a generalized algebraic theory and the GAT generated is a basic framework for dependent types. This a purely equational approach where sorts , judgements forms and operator symbols correspond to inference rules in a variable free formulation of a calculus with explicit substitutions for dependent type theory [6].

The idea of a generalized algebraic theory (GAT), introduced by Cartmell in [?], refers to the generalization of an algebraic theory of many-sorts in a way that does not restrict sorts to being constant types, i.e., to be interpreted as sets as they are in a regular algebraic theory. Hence, in a GAT, sorts can be variable types, thus they are interpreted as families of sets. The advantage of having variable types is that it suits theories of structures that are often found in category theory [?].

The sort and operator symbols are presented through their typing rules and indices are omitted for readability.

Rules for the base category

$$\frac{}{Ctx\ \ Sort} \qquad \frac{\Delta, \Gamma : Ctx}{\Delta \to \Gamma\ \ Sort}$$

$$\frac{\Theta, \Delta, \Gamma : Ctx \qquad \gamma : \Delta \to \Gamma \qquad \delta : \Theta \to \Delta}{\gamma \circ \delta : \Theta \to \Gamma}$$

$$\frac{\Gamma : Ctx}{id_\Gamma : \Gamma \to \Gamma}$$

$$\begin{aligned}(\gamma \circ \delta) \circ \theta &= \gamma \circ (\delta \circ \theta)\\ id \circ \gamma &= \gamma\\ \gamma \circ id &= \gamma\end{aligned}$$

Rules for the functor

$$\frac{\Gamma : Ctx}{Ty(\Gamma)\ \ sort} \qquad \frac{\Gamma : Ctx \qquad A : Ty(\Gamma)}{\Gamma \vdash A\ \ sort}$$

$$\frac{\Delta, \Gamma : Ctx \qquad A : Ty(\Gamma) \qquad \gamma : \Delta \to \Gamma}{A\,[\gamma] : Ty(\Delta)}$$

$$\frac{\Delta, \Gamma : Ctx \quad A : Ty(\Gamma) \quad \alpha : \Gamma \vdash A \quad \gamma : \Delta \to \Gamma}{\alpha\,[\gamma] : \Delta \vdash A\,[\gamma]}$$

$$\begin{aligned}A\,[\gamma \circ \delta] &= A\,[\gamma]\,[\delta]\\ A\,[id] &= A\\ \alpha\,[\gamma \circ \delta] &= \alpha\,[\gamma]\,[\delta]\\ \alpha\,[id] &= \alpha\end{aligned}$$

Rules for the terminal object

$$\Diamond : Ctx$$

$$\frac{\Gamma : Ctx}{\langle\rangle_\Gamma : \Gamma \to \Diamond}$$

$$\begin{aligned}\langle\rangle \circ \gamma &= \langle\rangle\\ id_\Diamond &= \Diamond\end{aligned}$$

Rules for context comprehension

$$\frac{\Gamma : Ctx \qquad A : Ty(\Gamma)}{\Gamma, A : Ctx}$$

$$\frac{\Delta, \Gamma : Ctx \quad A : Ty(\Gamma) \quad \gamma : \Delta \to \Gamma \quad \alpha : \Delta \vdash A\,[\gamma]}{\langle \gamma, \alpha \rangle : \Delta \to \Gamma, A}$$

$$\frac{\Gamma : Ctx \qquad A : Ty(\Gamma)}{p : \Gamma, A \to \Gamma}$$

$$\frac{\Gamma : Ctx \qquad A : Ty(\Gamma)}{q : \Gamma, A \vdash A\,[p]}$$

$$\begin{aligned} p \circ \langle \gamma, \alpha \rangle &= \gamma \\ q\,[\langle \gamma, \alpha \rangle] &= \alpha \\ \langle \delta, \alpha \rangle \circ \gamma &= \langle \delta \circ \gamma, \alpha\,[\gamma] \rangle \\ id_{\Gamma, A} &= \langle p_\Gamma, q_\Gamma \rangle \end{aligned}$$

This concludes the generalized algebraic theory of cwfs. Equality reasoning is handled in the metalanguage and hence there are no sort symbols for equality judgements or operators for equality rules.

As one can see, cwfs provide only the most minimal structure for interpreting dependent type theories and there is no additional structure that allows one to talk about type formers [?]. However, we can extend the definition if we consider the dependent function space and a universe. This is accomplished by adding introduction and elimination rules for our new type constructors and the appropriate equations they should satisfy. In other words, a cwf supports Π types if the following rules and laws are satisfied.

Rules for dependent function types

$$\frac{\Gamma : Ctx \quad A : Ty(\Gamma) \quad B : Ty(\Gamma, A)}{\Pi(A, B) : Ty(\Gamma)}$$

$$\frac{\Gamma : Ctx \quad A : Ty(\Gamma) \quad B : Ty(\Gamma, A) \quad t : \Gamma, A \vdash B}{\lambda(t) : \Gamma \vdash \Pi(A, B)}$$

$$\frac{\Gamma : Ctx \quad A : Ty(\Gamma) \quad B : Ty(\Gamma, A) \quad f : \Gamma \vdash \Pi(A, B) \quad t : \Gamma \vdash A}{app(f, a) : \Gamma \vdash B\,[\langle id_\Gamma, t \rangle]}$$

$$\begin{aligned} \Pi(A, B)\,[\gamma] &= \Pi(A\,[\gamma], B\,[\langle \gamma \circ p, q \rangle]) \\ \lambda(t)\,[\gamma] &= \lambda(t\,[\langle \gamma \circ p, q \rangle]) \\ app(f, t)\,[\gamma] &= app(f\,[\gamma], t\,[\gamma]) \\ app(\lambda(t), u) &= t\,[\langle id_\Gamma, u \rangle] \end{aligned}$$

Hence, in a similar fashion, one can add Σ types, identity types, or N_1 by expressing their typing rule in the GAT and enforcing the equations they must satisfy.

Chapter 3

Unityped CwFs

In the effort of building the model theory of the λ-calculus through this categorical model, we start by considering the untyped formulation of cwfs that we call unityped cwfs (ucwfs). Cwfs model dependent type theories, but they can be modified by stripping type information away. This results in a model of n-place functions with no type information.

This chapter starts with a formalization of the blueprint of a ucwf object. Since we construct different such objects, we need a common framework to talk about these objects and their properties. After, an implementation of an explicit ucwf object that fulfils the requirements of a ucwf by definition is presented. The resulting object should be initial in the category of ucwfs, albeit we do not concern ourselves with proving this result. Moreover, we implement concrete de Bruijn style variables with substitutions as vectors and the corresponding ucwf operations on the meta-level. In addition, it is shown that this formulation is also a ucwf object. Finally, we construct ucwf morphisms between these two objects and prove that they are inverses of each other.

The formalization is performed on three different levels that build upon the previous.

1. Pure ucwf.
2. λ-ucwf.
3. λ-$\beta\eta$-ucwf.

Pure ucwfs talk solely about variables, hence to model a λ-calculus one has to add extra structure, namely, lambda abstractions and applications on top of variables. The final step is to add beta and eta equalities as this how λ-calculus is usually presented.

3.1 Ucwfs and Scoped Variables

3.1.1 Pure Ucwfs

Based on the formulation of cwfs as a generalized algebraic theory presented in *Internal Type Theory* [4] and summarized in the preceding chapter, one can take the special case of having a single type (the unitype). Subsequently, one sees that the core ucwf is a model of n-place functions that includes operations regarding substitutions and composition of such functions. As in any GAT, it incorporates sort symbols, operator symbols, and equations between well-formed terms. In this sense, contexts and thus the objects of the base category become natural numbers. And so the terminal object is 0. In addition, since there is no type information, some constructs of the complete cwfs are no longer necessary. Terms are a type family indexed by just a natural number in the absence of types. The rules for this unityped GAT of cwfs are showcased through their formalization as a record in Agda.

```
record Ucwf {ℓ} : Set (suc ℓ) where
  field
    – Objects of the base category are natural numbers

    – Terms and substitutions
    Tm  : Nat → Set
    Sub : Nat → Nat → Set

    – two relations regarding equality of terms and substitutions
    _≈_  : ∀ {n} → Rel (Tm n) ℓ
    _≋_  : ∀ {m n} → Rel (Sub m n) ℓ

    IsEquivT : ∀ {n} → IsEquivalence (_≈_ {n})
    IsEquivS : ∀ {m n} → IsEquivalence (_≋_ {m} {n})

    – identity substitution
    id : ∀ {n} → Sub n n

    – composition
    _∘_ : ∀ {m n k} → Sub n k → Sub m n → Sub m k

    – explicit substitution
    _[_] : ∀ {m n} → Tm n → Sub m n → Tm m

    – empty substitution
    <> : ∀ {m} → Sub m 0

    – substitution extension
    <_,_>  : ∀ {m n} → Sub m n → Tm m → Sub m (suc n)

    – projection morphism
    p : ∀ {n} → Sub (suc n) n

    – last variable
    q : ∀ {n} → Tm (suc n)
```

In our considered formulation, contexts are assigned to the set of natural numbers beforehand. Therefore, Sub m n represents a length n substitution of terms with at most m free variables where m n : Nat. Naturally then, terms are indexed by the maximum number of free variables they may hold.

This description refers to a pure ucwf; so lambda abstractions and application are not present yet. It is a minimal theory for *n*-place functions. In fact, a pure ucwf can be viewed as a model of a calculus consisting solely of variables as constructs in the language; variables are built by applying the substitution operation on the projection morphism p, successively. For example, one can built the nth de Bruijn variable by performing this operation: q [p^n], where p^n is the composition of n projection substitutions, p ∘ … ∘ p. This is the weakening operation, but on variables it corresponds to applying the successor function on the index. This is because q represents the variable with de Bruijn index zero; p and q are also the first the second projections respectively. Further, the identity substitution id is a neutral element in composition and in substitution. It represents essentially a sequence of variable indices ranging from 0 to $n-1$. The theory also provides composition of substitutions and an explicit substitution operation. Lastly, a *snoc*-like operation for extending a substitution along with the empty substitution <>, which is also a left absorbing element under composition (a left zero as one would say in semi-group theory).

Moreover, the two equivalence relations in the record, one on Tm and one on Sub are required since there is no mention of equality in the GAT and it should thus be handled by the meta language. Further, we do not wish to restrict ourselves to propositional or some other notion of equality, hence setoids are employed.

Continuing, the ucwf axioms can be expressed in the record too using the relations as follows.

```
-- zero length id is the empty substitution
id₀ : id {0} ≋ <>

-- <> is a a left zero for composition
<>Lzero : ∀ {m n} (ts : Sub m n) → <> ∘ ts ≋ <>

-- extended identity is the projection with the last variable
idExt : ∀ {n} → id {suc n} ≋ < p , q >

-- category of substitutions
idL : ∀ {m n} (ts : Sub m n) → id ∘ ts ≋ ts
idR : ∀ {m n} (ts : Sub m n) → ts ∘ id ≋ ts
assoc : ∀ {m n k i} (ts : Sub n k) (us : Sub m n) (vs : Sub i m) →
  (ts ∘ us) ∘ vs ≋ ts ∘ (us ∘ vs)

-- substituting in id is neutral
subId : ∀ {n} (t : Tm n) → t [ id ] ≈ t

-- p is the first projection
pCons : ∀ {n k} (ts : Sub n k) t → p ∘ < ts , t > ≋ ts

-- q is the second projection
qCons : ∀ {m n} (ts : Sub n m) t → q [ < ts , t > ] ≈ t

-- substituting under a composition
subComp : ∀ {m n k} (ts : Sub m n) (us : Sub k m) t →
  t [ ts ∘ us ] ≈ t [ ts ] [ us ]

-- composing with an extended substitution
```

```
    compExt : ∀ {m n} (ts : Sub n m) (us : Sub m n) t →
      < ts , t > ∘ us ≋ < ts ∘ us , t [ us ] >

    – congruence rules for operators
    cong−<,> : ∀ {m n} {ts us : Sub m n} {t u} →
      t ≈ u →
      ts ≋ us →
      < ts , t > ≋ < us , u >

    cong−sub : ∀ {m n} {ts us : Sub m n} {t u} →
      t ≈ u →
      ts ≋ us →
      t [ ts ] ≈ u [ us ]

    cong−∘ : ∀ {m n k} {ts vs : Sub n k} {us zs : Sub m n} →
      ts ≋ vs →
      us ≋ zs →
      ts ∘ us ≋ vs ∘ zs
```

This completes the record of a pure ucwf; any instantiation of this record has to provide the two mains sorts, the corresponding relations in which one reasons about them, the constructors, and proofs of the laws. This record, essentially, describes what it means to be a ucwf object in the category of ucwfs.

Now, we can start constructing ucwf objects. Based on this description, one can create a ucwf combinator language that is trivially a ucwf by providing explicit constructs that match precisely the fields of the record. In more detail, we implement a ucwf object by *(i)* two mutually recursive data types that represent terms and substitutions and *(ii)* two inductive relations whose constructors are introduction rules for each ucwf axiom.

```
  data Tm : Nat → Set
  data Sub : Nat → Nat → Set

  data Tm where
    q  : ∀ {n} → Tm (suc n)
    _[_] : ∀ {m n} → Tm n → Sub m n → Tm m

  data Sub where
    id    : ∀ {m} → Sub m m
    _∘_   : ∀ {m n k} → Sub n k → Sub m n → Sub m k
    <>    : ∀ {m} → Sub m zero
    <_,_> : ∀ {m n} → Sub m n → Tm m → Sub m (suc n)
    p     : ∀ {n} → Sub (suc n) n

  data _≈_ : ∀ {n} → Tm n → Tm n → Set
  data _≋_ : ∀ {n m} → Sub n m → Sub n m → Set
```

We can easily instantiate this language as a ucwf since the fields match exactly the definitions here and the laws are explicit and thus there is no need to prove anything. The two equality relations encapsulate all laws of the Ucwf record and so they add nothing new apart from symmetry and transitivity. The equations are omitted but available in figure A.1 of the appendix.

The data type of Sub represents morphisms in the base category of the ucwf, it essential represents the Hom set. Objects are natural numbers and we have five explicit morphisms. The terminal object is zero. On the other hand, Tm is the second component of the ucwf, namely, the functor that describes a family of types indexed by a natural number.

From this definition, we formalize these equalities as equivalence relations and instantiate them as setoids. This will allow us to use Agda's equational reasoning packages from the standard library that operate on any setoid. Here are the instances.

```
refl≈ : ∀ {n} {u : Tm n} → u ≈ u
refl≈ = trans≈ (sym≈ (subId _)) (subId _)

refl≋ : ∀ {n m} {h : Sub m n} → h ≋ h
refl≋ = trans≋ (sym≋ (idL _)) (idL _)

≈equiv : ∀ {n} → IsEquivalence (_≈_ {n})
≈equiv = record
  { refl = refl≈
  ; sym = sym≈
  ; trans = trans≈ }

TmSetoid : ∀ {n} → Setoid _ _
TmSetoid {n} = record
  { Carrier = Tm n
  ; _≈_ = _≈_
  ; isEquivalence = ≈equiv }

≋equiv : ∀ {n m} → IsEquivalence (_≋_ {n} {m})
≋equiv = record
  { refl = refl≋
  ; sym  = sym≋
  ; trans = trans≋ }

SubSetoid : ∀ {n m} → Setoid _ _
SubSetoid {n} {m} = record
  { Carrier = Sub m n
  ; _≈_ = _≋_
  ; isEquivalence = ≋equiv }
```

These data types form the most straightforward ucwf object one can construct. And by following the cwf's GAT formalization, the result is a kind of variable-free calculus of explicit substitutions with a set of conversion laws expressed equationally.

In order to provide an example of how one can use this calculus to prove something, we show that there exists a unique morphism from any object to zero, since it is a terminal object in the base category of the ucwf. In other words, any morphism with co-domain zero should be convertible to the empty morphism.

```
ter−arrow : ∀ {n} (ts : Sub n 0) → ts ≈ <>
ter−arrow ts = begin
  ts          ≈⟨ sym≈ (idL ts) ⟩
  id {0} ∘ ts ≈⟨ cong−∘ id₀ refl≈ ⟩
  <> ∘ ts     ≈⟨ <>Lzero ts ⟩
  <>          ∎
  where open EqR (SubSetoid {0} {_})
```

3.1.2 Well-scoped Variables as Ucwf object

As mentioned earlier, a pure ucwf is a model of a language consisting solely of variables. This idea is implemented using a data type with de Bruijn variables, so we have nameless terms. These variables are also indexed by a natural number n and the index is of type Fin n, that is, a number $i \in \mathbb{N}$ such that $0 \leq i < n$. We refer to these terms as well-scoped. Scope safe terms have some advantages when it comes to the implementation of some operations like substitution. For example, weakening a term extends its index, so the type contains valuable information using this set-up.

```
data Tm (n : Nat) : Set where
  var : (i : Fin n) → Tm n

q : ∀ {n} → Tm (1 + n)
q = var zero
```

Given these terms, we would like to construct a ucwf object in accordance to the record shown before. We can see that we have defined q which is a part of the ucwf theory. Moreover, we also need substitutions; they are implemented as vectors of specific length containing variables (using the standard library). Below, we present ucwf operators for this concrete implementation where they are non-explicit, but rather meta level operations.

- An empty substitution is the empty vector [].
- Extending a substitution is effectively the *cons* function for vectors, but the syntax is slightly changed to mirror ucwf style.
- Identity substitution, id; a sequence of variable indices from 0 to $n - 1$.
- Projection substitution, p; a sequence of variable indices from 1 to n.
- Composition of substitutions, $\sigma_1 \circ \sigma_2$; which means substituting all terms of σ_1 in σ_2.
- Substitution operation, _[_]; on variables, it corresponds to performing a lookup.

```
Sub : Nat → Nat → Set
Sub m n = Vec (Tm m) n
```

```
_,_ : ∀ {n m} → Sub m n → Tm m → Sub m (1 + n)
σ , t = t :: σ

id : ∀ {n} → Sub n n
id = tabulate var

p : ∀ {n} → Sub (1 + n) n
p = tabulate (var F.∘ suc)

_∘_ : ∀ {m n k} → Sub m n → Sub k m → Sub k n
σ₁ ∘ σ₂ = map (_[ σ₂ ]) σ₁

_[_] : ∀ {m n} → Tm n → Sub m n → Tm m
var i [ σ ] = lookup i σ
```

In defining these operations, we make heavy use of the standard library's functions on vectors. Also, composition of regular functions is distinguished by referring to it with a named import, F.

The type signatures hopefully start to resemble the fields of a ucwf; we notice that the main sorts and operators have been defined; Sub and Tm represent the morphisms and the functor.

Subsequently, one can show that this language of variables fulfils the ucwf axioms by proving them for these concrete definitions. This is the last piece of completing the construction of a ucwf object. Next, we present a series of lemmas that prove the ucwf generalized algebraic theory's laws for our variables and substitutions.

Lemma 3.1. id {1 + n} ≡ (p , q)

Proof. Reflexivity. Recall that p is a sequence of indices $\{1, \ldots, n\}$ and q is the zeroth index. And the way we have defined these vectors make the expression definitionally equal.

```
idExt : ∀ {n} → id {1 + n} ≡ (p , q)
idExt = refl
```

■

Lemma 3.2. t [id] ≡ t

Proof. Induction on t and a lookup property on id. This property reduces to verifying that looking up in id with some i, results in var i.

```
subId : ∀ {n} (t : Tm n) → t [ id ] ≡ t
subId (var i) = lookup−id i
```

■

Lemma 3.3. t [ρ ∘ σ] ≡ t [ρ] [σ]

Proof. Induction on t and mutual induction on ρ in the variable case.

```
subComp : ∀ {m n k} (t : Tm n) (ρ : Sub m n) (σ : Sub k m) →
  t [ ρ ∘ σ ] ≡ t [ ρ ] [ σ ]
```

```
subComp (var ()) [] σ
subComp (var zero) (x :: ρ) σ = refl
subComp (var (suc i)) (x :: ρ) σ = subComp (var i) ρ σ
```

■

Lemma 3.4. id {0} ≡ []

Proof. Reflexivity. A vector of length 0 is always the empty vector.

```
id₀ : id {0} ≡ []
id₀ = refl
```

■

Lemma 3.5. [] ∘ ρ ≡ []

Proof. Reflexivity. Composition was defined by mapping substitution on the left argument, thus mapping on the empty vector results in [].

```
[]Lzero : ∀ {m n} (ρ : Sub m n) → [] ∘ ρ ≡ []
[]Lzero _ = refl
```

■

Lemma 3.6. p ∘ (σ , t) ≡ σ

Proof. By properties relating p to lookup and map.

```
map−lookup−↑ : ∀ {n m} (ts : Sub m (1 + n)) →
      map (flip lookup ts) (tabulate suc) ≡ tail ts
map−lookup−↑ (t :: ts) = begin
   map (flip lookup (t :: ts)) (tabulate suc)
      ≡⟨ sym $ tabulate−∘ (flip lookup (t :: ts)) suc ⟩
   tabulate (flip lookup ts)
      ≡⟨ tabulate∘lookup ts ⟩
   ts
      ∎

p∘−lookup : ∀ {m n} (ts : Sub m (1 + n)) →
         p {n} ∘ ts ≡ map (flip lookup ts) (tabulate suc)
p∘−lookup ts = let pFin = tabulate suc in begin
   map (_[ ts ]) (tabulate (var F.∘ suc))
      ≡⟨ cong (map (_[ ts ])) (tabulate−∘ var suc) ⟩
   map (_[ ts ]) (map var pFin)
      ≡⟨⟩
   (map (_[ ts ]) F.∘ map (var)) pFin
      ≡⟨ sym $ map−∘ (_[ ts ]) (var) pFin ⟩
   map (λ i → (var i) [ ts ]) pFin
      ≡⟨⟩
   map (flip lookup ts) pFin
      ∎
```

```
pCons : ∀ {n k} (σ : Sub n k) t → p ∘ (σ , t) ≡ σ
pCons σ t = trans (p∘−lookup (σ , t)) (map−lookup−↑ (σ , t))
```

∎

This expression is transformed to mapping a lookup on the sequence of $\{1, \ldots, n\}$, which should then drop the first element since the lookup begins from the index 1.

Lemma 3.7. (σ ∘ γ) ∘ δ ≡ σ ∘ (γ ∘ δ)

Proof. This follows by a simple induction if we know that t [ρ ∘ σ] ≡ t [ρ] [σ].

```
assoc : ∀ {m n k j} (σ : Sub m n) (γ : Sub k m) (δ : Sub j k) →
  (σ ∘ γ) ∘ δ ≡ σ ∘ (γ ∘ δ)
assoc [] γ δ = refl
assoc (t :: σ) γ δ = sym $ begin
  (σ , t) ∘ (γ ∘ δ)          ≡⟨⟩
  σ ∘ (γ ∘ δ) , t [ γ ∘ δ ]   ≡⟨ cong (λ x → _ , x) (subComp t γ δ) ⟩
  σ ∘ (γ ∘ δ) , t [ γ ] [ δ ] ≡⟨ sym (cong (_, t [ γ ] [ δ ]) (assoc σ γ δ)) ⟩
  (σ ∘ γ) ∘ δ , t [ γ ] [ δ ] ∎
  where open P.≡−Reasoning
```

∎

Lemma 3.8. q [σ , t] ≡ t

Proof. Reflexivity. This reduces to looking up zero in the vector, thus, the first element is picked.

```
qCons : ∀ {m n} (σ : Sub m n) t → q [ σ , t ] ≡ t
qCons _ _ = refl
```

∎

Lemma 3.9. (σ , t) ∘ γ ≡ (σ ∘ γ) , t [γ]

Proof. Reflexivity; this is how composition is defined.

```
compExt : ∀ {m n} (σ : Sub n m) (γ : Sub m n) t →
  (σ , t) ∘ γ ≡ (σ ∘ γ) , t [ γ ]
compExt _ _ _ = refl
```

∎

Lemma 3.10. id ∘ σ ≡ σ

Proof. Induction on σ.

```
idL : ∀ {m n} (σ : Sub m n) → id ∘ σ ≡ σ
idL [] = refl
idL (x :: σ) = begin
  id ∘ (σ , x)       ≡⟨ ∘=∘₂ id (σ , x) ⟩
  p ∘₂ (σ , x) , x   ≡⟨ cong (_, x) (sym $ ∘=∘₂ p (σ , x)) ⟩
  p ∘ (σ , x) , x    ≡⟨ cong (_, x) (pCons σ x) ⟩
  σ , x              ∎
  where open P.≡−Reasoning
```

∎

Lemma 3.11. σ ∘ id ≡ σ

Proof. Induction on σ.

```
idR : ∀ {m n} (σ : Sub m n) → σ ∘ id ≡ σ
idR [] = refl
idR (t :: σ) rewrite subId t | idR σ = refl
```

■

Theorem 3.1 (Well-scoped variables with substitutions form a ucwf object). *With a base category where objects are elements of* Nat *and morphisms are defined by* Sub*, plus the functor* Tm*; these two components form a ucwf object.*

The congruence rule proofs are trivial and thus omitted and propositional equality is obviously an equivalence relation. As a result, we now have a second ucwf object and the next step is to show it is isomorphic to the explicit object.

3.1.3 Isomorphism of Ucwf objects

This section presents a proof that the ucwf combinator language and the de Bruijn style variables are isomorphic. By viewing both of these constructs as objects in the category of ucwfs, one can define ucwf morphisms between them. They are effectively functions that map terms from one data type to the other. Once the morphisms are defined, we can show the isomorphism by proving that the functions are inverses of each other, which is one method to demonstrate a bijection.

Earlier it was mentioned that the explicit ucwf object is an initial object in the category of ucwfs; therefore, by showing it is isomorphic to the ucwf of variables, it follows that the latter is also initial.

Several syntactic changes include the following: terms of the ucwf combinator language are renamed as Tm−cwf, while variable terms are called Tm−λ. Furthermore, to avoid name clashing, operation names on the scoped variables' side have a lambda as a suffix, e.g. p is now p−λ. Additionally, the arrows are symbolized by bracketed interpretation function style to provide cleaner proof syntax.

Definition 3.1. Ucwf morphisms

```
– Natural transformations between the functors
⟦_⟧   : ∀ {n} → Tm−λ n → Tm−cwf n
⟪_⟫   : ∀ {n} → Tm−cwf n → Tm−λ n

– Functors between the base categories
⟪_⟫′ : ∀ {m n} → Sub−cwf m n → Sub−λ m n
⟦_⟧′ : ∀ {m n} → Sub−λ m n → Sub−cwf m n

varCwf : ∀ {n} (i : Fin n) → Tm−cwf n
varCwf zero   = q
varCwf (suc i) = varCwf i [ p ]

⟦ var i ⟧ = varCwf i
⟦ [] ⟧′ = <>
⟦ t ∷ ts ⟧′ = < ⟦ ts ⟧′ , ⟦ t ⟧ >

⟪ q ⟫ = q−λ
⟪ t [ us ] ⟫ = ⟪ t ⟫ [ ⟪ us ⟫′ ]λ

⟪ id ⟫′ = id−λ
⟪ ts ∘ us ⟫′ = ⟪ ts ⟫′ ∘λ ⟪ us ⟫′
⟪ p ⟫′ = p−λ
⟪ <> ⟫′ = []
⟪ < ts , t > ⟫′ = ⟪ ts ⟫′ , ⟪ t ⟫
```

Variables on the Tm−cwf side are constructed by substituting q to compositions of the lifting substitution p. Intuitively, by composing lifting substitutions, one increases the variable indices; it starts with the sequence $\{1, \ldots, n\}$ and upon lifting once, it becomes $\{2, \ldots, n + 1\}$. And since q is the second projection, it returns the head of the substitution and thus q [p] is the variable with index 1. Moreover, the remaining explicit constructors are mapped to the meta-level operations described in section 3.1.2 on the scoped variable side. We also observe that arrows from the explicit object are mutually recursive which makes sense given that substitution is explicit.

There is substantial categorical context surrounding these morphisms that may not be obvious. Recall that a ucwf is represented by two components; first, a base category, in this case objects are shared by the two categories (Nat), while Sub−λ and Sub−cwf are Hom sets. The second component is a functor which is formalized as term data types. Therefore, ⟦_⟧′ and ⟪_⟫′ are functors that map substitution morphisms, while object maps resort to the identity morphism. One can look at these functors diagrammatically as follows.

$$\begin{tikzcd}
\text{Sub−λ m n} \arrow[r, "⟦\_⟧'"] \arrow[d, "dom"] & \text{Sub−cwf m n} \arrow[d, "dom"] \\
\text{m} \arrow[r, equal, "id_m"] & \text{m}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{Sub−cwf m n} \arrow[r, "⟪\_⟫'"] \arrow[d, "dom"] & \text{Sub−λ m n} \arrow[d, "dom"] \\
\text{m} \arrow[r, equal, "id_m"] & \text{m}
\end{tikzcd}$$

Equivalently, we have a similar diagram for the co-domain. Now, *dom* and *id* are not really defined in the implementation; there is no need. On the other hand, ⟪_⟫ and ⟦_⟧ are natural transformations between the functors Tm−λ and Tm−cwf. Therefore, in order to demonstrate an isomorphism of ucwf objects, we need an isomorphism of the base categories and a natural isomorphism between the functors. So these functions have to be strict inverses of each other.

It would be particularly nice if we had a formal notion of a cwf morphism and the whole category; however, it is not formalized in this thesis and, in fact, such a task presents itself with its own challenges.

Next, the proof part is presented. To begin with, two integral lemmas that assist the inverse proofs are shown. Firstly, we require a lemma that states when we interpret a call on the substitution function in the well-scoped world, it is convertible to interpreting the term and vector as an explicit substitution term. Secondly, an equivalent notion for composition, i.e., that interpreting a composition distributes to the other ucwf object.

Lemma 3.12. ⟦ t [σ]λ ⟧ ≈ ⟦ t ⟧ [⟦ σ ⟧′]

Proof. Induction on t and σ.

```
[]−comm : ∀ {m n} t (σ : Sub−λ m n) → ⟦ t [ σ ]λ ⟧ ≈ ⟦ t ⟧ [ ⟦ σ ⟧′ ]
[]−comm (var zero)  (x :: σ) = sym≈ (qCons ⟦ σ ⟧′ ⟦ x ⟧)
[]−comm (var (suc ι)) (x :: σ) = sym≈ $ begin
  ⟦ var ι ⟧ [ p ] [ < ⟦ σ ⟧′ , ⟦ x ⟧ > ]
    ≈⟨ sym≈ (subComp p < ⟦ σ ⟧′ , ⟦ x ⟧ > ⟦ var ι ⟧) ⟩
  ⟦ var ι ⟧ [ p ∘ < ⟦ σ ⟧′ , ⟦ x ⟧ > ]
    ≈⟨ (cong−sub refl≈ (pCons ⟦ σ ⟧′ ⟦ x ⟧)) ⟩
  ⟦ var ι ⟧ [ ⟦ σ ⟧′ ]
    ≈⟨ sym≈ ([]−comm (var ι) σ) ⟩
  ⟦ lookup ι σ ⟧
    ∎
  where open EqR (TmSetoid {_})
```

∎

Some basic equational reasoning using the ucwf axiomatization is required upon applying the inductive hypothesis.

Lemma 3.13. ⟦ σ ∘λ γ ⟧′ ≈ ⟦ σ ⟧′ ∘ ⟦ γ ⟧′

Proof. Induction on σ.

```
⟦⟧−∘−dist : ∀ {m n k} (σ : Sub−λ n k) (γ : Sub−λ m n) →
    ⟦ σ ∘λ γ ⟧′ ≈ ⟦ σ ⟧′ ∘ ⟦ γ ⟧′
⟦⟧−∘−dist [] γ = sym≋ (<>Lzero ⟦ γ ⟧′)
⟦⟧−∘−dist (t :: σ) γ = begin
  < ⟦ σ ∘λ γ ⟧′ , ⟦ t [ γ ]λ ⟧ >
    ≈⟨ cong−<,> refl≈ (⟦⟧−∘−dist σ γ) ⟩
  < ⟦ σ ⟧′ ∘ ⟦ γ ⟧′ , ⟦ t [ γ ]λ ⟧ >
    ≈⟨ cong−<,> ([]−comm t γ) refl≋ ⟩
  < ⟦ σ ⟧′ ∘ ⟦ γ ⟧′ , ⟦ t ⟧ [ ⟦ γ ⟧′ ] >
    ≈⟨ sym≋ (compExt ⟦ σ ⟧′ ⟦ γ ⟧′ ⟦ t ⟧) ⟩
  < ⟦ σ ⟧′ , ⟦ t ⟧ > ∘ ⟦ γ ⟧′
    ∎
  where open EqR (SubSetoid {_} {_})
```

∎

In some sense, these lemmas express the fact that composition and substitution are preserved when they are sent from the ucwf representing concrete variables with meta operations via the functor.

Subsequently, we continue with the inverse proofs; here are the signatures of the lemmas.

```
tm−λ⇒cwf : ∀ {n} (t : Tm−λ n) → t ≡ ⟪ ⟦ t ⟧ ⟫

tm−cwf⇒λ : ∀ {n} (t : Tm−cwf n) → t ≈ ⟦ ⟪ t ⟫ ⟧
```

```
sub−cwf⇒λ : ∀ {m n} (γ : Sub−cwf m n) → γ ≈ ⟦ ⟪ γ ⟫′ ⟧′

sub−λ⇒cwf : ∀ {m n} (ρ : Sub−λ m n) → ρ ≡ ⟪ ⟦ ρ ⟧′ ⟫′
```

The first two are on the term level, whereas the two other are on the substitutions level. All four proofs are by induction on the argument.

The first proof is a simple induction on the term which can only be a variable and then induction on the index of the variable. Additionally, we use a property that states a lookup in the projection substitution returns the successor, var (suc i).

On the cwf side, the q case is trivial; for the substitution case, we have two inductive hypotheses since the proofs are mutually recursive. Lastly, we use lemma 3.12, this allows us to avoid performing further pattern matching on the substitution and hence do extensive equational reasoning for each case.

Lemma 3.14. $t \equiv ⟪ ⟦ t ⟧ ⟫$ *and* $t \approx ⟦ ⟪ t ⟫ ⟧$.

Proof. Induction on *t*.

```
tm−λ⇒cwf (var zero) = refl
tm−λ⇒cwf (var (suc i))
  rewrite sym $ lookup−p i = cong (_[ p−λ ]λ) (tm−λ⇒cwf (var i))

tm−cwf⇒λ q = refl≈
tm−cwf⇒λ (t [ us ]) = sym≈ $ begin
  ⟦ ⟪ t ⟫ [ ⟪ us ⟫′ ]λ ⟧        ≈⟨ []−comm ⟪ t ⟫ ⟪ us ⟫′ ⟩
  ⟦ ⟪ t ⟫ ⟧ [ ⟦ ⟪ us ⟫′ ⟧′ ]  ≈⟨ sym≈ (cong−sub (tm−cwf⇒λ t) refl≈) ⟩
  t [ ⟦ ⟪ us ⟫′ ⟧′ ]          ≈⟨ sym≈ (cong−sub refl≈ (sub−cwf⇒λ us)) ⟩
  t [ us ]                    ∎
  where open EqR (TmSetoid {_})
```

∎

It has been therefore shown that there is isomorphism between functors Tm−l and Tm−cwf. Now, we have to show that the base categories are isomorphic too.

The inverse proof on the substitution level requires more work. The cases where we extend a substitution are proved by simply applying hypotheses to the operands. The composition case utilizes lemma 3.13 and the identity case is an induction on the length as well. The projection case is somewhat more difficult and is discussed momentarily. Next, we show the top level proofs.

Lemma 3.15. $\rho \equiv ⟪ ⟦ \rho ⟧′ ⟫′$ *and* $\gamma \approx ⟦ ⟪ \gamma ⟫′ ⟧′$.

Proof. Induction on ρ and γ.

```
sub−cwf⇒λ (id {zero}) = id₀
sub−cwf⇒λ (id {suc m}) = begin
  id {1 + m}        ≈⟨ idExt ⟩
  < p , q >         ≈⟨ cong−<,> refl≈ (sub−cwf⇒λ p) ⟩
  < ⟦ p−λ ⟧′ , q >  ∎
  where open EqR (SubSetoid {_} {_})
sub−cwf⇒λ (γ ∘ δ) = sym≈ $ begin
```

```
    ⟦ ⟪ γ ⟫′ ∘λ ⟪ δ ⟫′ ⟧′  ≈⟨ ⟦⟧−∘−dist ⟪ γ ⟫′ ⟪ δ ⟫′ ⟩
    ⟦ ⟪ γ ⟫′ ⟧′ ∘ ⟦ ⟪ δ ⟫′ ⟧′ ≈⟨ sym≈ (cong−∘ (sub−cwf⇒λ γ) refl≈) ⟩
    γ ∘ ⟦ ⟪ δ ⟫′ ⟧′         ≈⟨ sym≈ (cong−∘ refl≈ (sub−cwf⇒λ δ)) ⟩
    γ ∘ δ                   ∎
    where open EqR (SubSetoid {_} {_})
  sub−cwf⇒λ p = p−inverse
  sub−cwf⇒λ <> = refl≈
  sub−cwf⇒λ < γ , x > = cong−<,> (tm−cwf⇒λ x) (sub−cwf⇒λ γ)

  sub−λ⇒cwf [] = refl
  sub−λ⇒cwf (x :: ρ) = cong₂ _,_ (sub−λ⇒cwf ρ) (tm−λ⇒cwf x)
```

■

The projection case is proved in two steps; first, it is shown that p is convertible to its normalized form. By normalized form, we mean the actual sequence of successive weakenings on the zeroth variable q. This form is built by using *snoc* sequences of Fin elements and now we can iterate weakening to attain the form of p we need.

```
  data Fins : Nat → Nat → Set where
    <> : ∀ {m} → Fins m 0
    <_,_> : ∀ {m n} → Fins m n → Fin m → Fins m (suc n)

  varCwf : ∀ {n} (i : Fin n) → Tm−cwf n
  varCwf zero = q
  varCwf (suc i) = varCwf i [ p ]

  vars : ∀ {n m} (is : Fins m n) → Sub−cwf m n
  vars <> = <>
  vars < is , i > = < vars is , varCwf i >

  pNorm : ∀ n → Sub−cwf (suc n) n
  pNorm n = vars (pFins n)
```

Here, pFins refers to the sequence of Fins $\{1, \ldots, n\}$; it is the projection substitution for indices. Now we can show that p is convertible to this sequence. The translation function (ucwf morphism) constructs variables of Tm−l by invoking varCwf, thus this lemma is essential. It is an induction on the length, but we need significant dull reasoning with the sequences of Fins behind the scenes.

Lemma 3.16. p ≈ pNorm n

Proof. Induction on n.

```
    p≈pNorm : ∀ n → p ≈ pNorm n
    p≈pNorm zero  = ter−arrow (p {0})
    p≈pNorm (suc n) = begin
      p
        ≈⟨ surj−<,> p ⟩
      < p ∘ p , q [ p ] >
        ≈⟨ cong−<,> refl≈ (cong−∘ (p≈pNorm n) (refl≈)) ⟩
    < vars (mapFins (tab F.id) suc) ∘ p , q [ p ] >
      ≈⟨ cong−<, > refl≈ (var−lemma (mapFins (tab F.id) suc)) ⟩
    < vars (mapFins (mapFins (tab F.id) suc) suc) , q [ p ] >
      ≈⟨ cong−<, > refl≈ help ⟩
    < vars (mapFins (tab suc) suc) , q [ p ] >
      ∎
    where open EqR (SubSetoid {_} {_})
```
■

The remaining proof is still not entirely trivial, albeit quite mundane. To complete this proof, a different version of p was used along with some utility higher order maps to make the induction work. Thus, it was necessary to prove that all these alternate definitions are actually the same (this last bit is omitted).

By showing that these functors are mutually inverse, we have shown that the categories are isomorphic. It was known beforehand that the objects were in one-to-one correspondence with each other, so in this proof we actually verified that morphisms are too.

Theorem 3.2. *There exists an isomorphism between the previously constructed ucwf objects. Objects of both base categories are natural numbers, morphisms* Sub−λ *and* Sub−cwf*, and functors* Tm−λ *and* Tm−cwf.

3.2 λ-Ucwfs and λ-calculus

In this part, we demonstrate that upon extending the basic ucwf with λ abstractions and applications, it becomes a model of the untyped λ-calculus. We call this new construct a λ-ucwf and we extend the generalized algebraic theory with the appropriate constructors and equations. Simultaneously, we extend the de Bruijn variable language with these new elements and show that it forms a λ-ucwf.

The proofs of the previous section have to be extended to cover the cases where there was an induction on terms to account for the new elements in the set of terms.

3.2.1 λ-Ucwfs

By utilizing the pure ucwf record defined earlier, we add a constructor for λ abstractions and function applications. Naturally, the appropriate laws and congruence rules are included. Using another Agda record, we create this new notion.

```
  record Lambda−ucwf {ℓ} : Set (suc ℓ)
    field
      ucwf : Ucwf {ℓ}
    open Ucwf ucwf public
    field

      – λ abstraction and application
      lam : ∀ {n} → Tm (suc n) → Tm n
      app : ∀ {n} → Tm n → Tm n → Tm n

      – equations for substituting under app and lam
```

```
subApp : ∀ {n m} (ts : Sub m n) t u →
  app (t [ ts ]) (u [ ts ]) ≈ (app t u) [ ts ]

subLam : ∀ {n m} (ts : Sub m n) t →
  lam t [ ts ] ≈ lam (t [ < ts ∘ p , q > ])

cong−lam : ∀ {n} {t u : Tm (suc n)} →
  t ≈ u →
  lam t ≈ lam u

cong−app : ∀ {n} {t u t′ u′ : Tm n} →
  t ≈ t′ →
  u ≈ u′ →
  app t u ≈ app t′ u′
```

One notices the new rules about substituting a term of application and lambda abstraction. This occurs because substitution is a term per se and defined by its conversion laws, so stability under substitution is necessary.

The construction of an explicit λ-ucwf object is performed by extending the Tm data type and adding introduction rules for the above equations. The substitutions remain the same and we acquire an explicit λ-ucwf object by definition.

```
data Tm where
  q     : ∀ {n} → Tm (suc n)
  _[_] : ∀ {m n} → Tm n → Sub m n → Tm m
  lam  : ∀ {n} → Tm (suc n) → Tm n
  app  : ∀ {n} → Tm n → Tm n → Tm n
```

Overall, this formulation bares great resemblance to the $\lambda\sigma$ calculus (Abadi et al) [10]. Substitution is an explicit operation within the calculus and one uses these algebraic operators to perform the necessary manipulations. Almost all operators can be found in the σ system. For instance, the projection substitution p, is alternatively called shift and is symbolized by an upwards arrow $\uparrow$. However, the core system σ does not have every rule of a ucwf; for example, id is not a right identity in σ, but the similarity is interesting nonetheless.

3.2.2 Untyped λ-calculus as λ-Ucwf object

The extension of the language of variables with lambda abstractions and function application results in the complete untyped λ-calculus. Terms are indexed by the maximum number of free variables they may contain. So the Tm type is extended with the appropriate constructors. An infix dot notation is used for application and a slashed lambda for abstractions.

Definition 3.2. Untyped terms

```
data Tm (n : Nat) : Set where
  var  : (i : Fin n) → Tm n
```

```
_·_  : (t u : Tm n) → Tm n
ƛ    : (t : Tm (1 + n)) → Tm n
```

The intention is to repeat the procedure of the previous section for this language. The structure allows us to only present the cases that are defined or proven by induction on Tm. Hence, most properties from before remain the same, at least the ones related to substitutions.

We proceed by defining the new substitution operation. The concept of renaming is required for this task. Renamings (Ren) are merely sequences of elements of Fin type; they are substitutions for scoped variables. The renaming operation is used to weaken a term.

```
Ren : Nat → Nat → Set
Ren m n = Vec (Fin m) n

lift−ren : ∀ {m n} → Ren m n → Ren (1 + m) (1 + n)
lift−ren = λ ρ → zero ∷ map suc ρ

ren : ∀ {n m} → Tm n → Ren m n → Tm m
ren (var i) ρ = var (lookup i ρ)
ren (ƛ t)  ρ = ƛ (ren t (lift−ren ρ))
ren (t · u) ρ = (ren t ρ) · (ren u ρ)

weaken : ∀ {m} → Tm m → Tm (1 + m)
weaken = flip ren (tabulate suc)

↑_ : ∀ {m n} → Sub m n → Sub (1 + m) (1 + n)
↑_ = (_, q) F.∘ map weaken

_[_] : ∀ {m n} → Tm n → Sub m n → Tm m
var i   [ σ ] = lookup i σ
ƛ t     [ σ ] = ƛ (t [ ↑ σ ])
(t · u) [ σ ] = t [ σ ] · u [ σ ]
```

The operations of renaming and substitution have overt similarities and they are connected. The interesting case is ƛ, where we have to weaken and extend the substitution. And so weakening a term t is renaming the free variables in the Fin sequence $\{1, \ldots, n\}$.

Further, to prove that this untyped λ-calculus is a λ-ucwf, two proofs are modified.

Lemma 3.17. t [id] ≡ t

Proof. By induction on t.

```
subId : ∀ {n} (t : Tm n) → t [ id ] ≡ t
subId (var i) = lookup−id i
subId (t · u) = cong₂ _·_ (subId t) (subId u)
subId (ƛ t) = cong ƛ $ begin
  t [ ↑ id ]  ≡⟨ cong (t [_] F.∘ (_, q)) (sym p=p′) ⟩
  t [ p , q ] ≡⟨⟩
  t [ id ]    ≡⟨ subId t ⟩
```

```
    t          ∎
    where open P.≡−Reasoning
```

■

Lemma 3.18. t [ρ ∘ σ] ≡ t [ρ] [σ]

Proof. Induction on t and mutual induction on ρ in the variable case.

```
subComp : ∀ {m n k} (t : Tm n) (ρ : Sub m n) (σ : Sub k m) →
      t [ ρ ∘ σ ] ≡ t [ ρ ] [ σ ]
subComp (var ()) [] σ
subComp (var zero) (x ∷ ρ) σ = refl
subComp (var (suc i)) (x ∷ ρ) σ = subComp (var i) ρ σ
subComp (t · u) ρ σ = cong₂ _·_ (subComp t ρ σ) (subComp u ρ σ)
subComp (ƛ t) ρ σ = cong ƛ $ begin
    t [ ↑ (ρ ∘ σ) ]  ≡⟨ cong (t [_]) (↑−dist ρ σ) ⟩
    t [ ↑ ρ ∘ ↑ σ ]  ≡⟨ subComp t (↑ ρ) (↑ σ) ⟩
    t [ ↑ ρ ] [ ↑ σ ] ∎
    where open P.≡−Reasoning
```

■

The key property for the λ case is that ↑ distributes over composition; the proof of which is in figure A.3. As a result, untyped λ-calculus is a λ-ucwf object.

3.2.3 Isomorphism of λ-Ucwf objects

The extension for the isomorphism proof is quite simple. The new translation functions for the two new cases have a nice recursive structure that results in the proofs being simple applications of the hypotheses. Only cases of λ abstractions and application are presented here to avoid repetition. Starting with the extended morphisms.

Definition 3.3. λ-ucwf morphisms.

```
⟦ ƛ t ⟧   = lam ⟦ t ⟧
⟦ t · u ⟧ = app ⟦ t ⟧ ⟦ u ⟧

⟪ lam t ⟫    = ƛ ⟪ t ⟫
⟪ app t u ⟫  = ⟪ t ⟫ · ⟪ u ⟫
```

And the extensions of lemma 3.14.

```
tm−λ⇒cwf (ƛ t) = cong ƛ (tm−λ⇒cwf t)
tm−λ⇒cwf (t · u) = cong₂ (_·_) (tm−λ⇒cwf t) (tm−λ⇒cwf u)

tm−cwf⇒λ (lam t) = cong−lam (tm−cwf⇒λ t)
tm−cwf⇒λ (app t u) = cong−app (tm−cwf⇒λ t) (tm−cwf⇒λ u)
```

Finally, an important supplementary property (lemma 3.12) is extended as well.

```
[]−comm (t · u) σ = begin
```

```
    app ⟦ t [ σ ]λ ⟧ ⟦ u [ σ ]λ ⟧
      ≈⟨ cong−app ([]−comm t σ) refl≈ ⟩
    app (⟦ t ⟧ [ ⟦ σ ⟧′ ]) (⟦ u [ σ ]λ ⟧)
      ≈⟨ cong−app refl≈ ([]−comm u σ) ⟩
    app (⟦ t ⟧ [ ⟦ σ ⟧′ ]) (⟦ u ⟧ [ ⟦ σ ⟧′ ])
      ≈⟨ subApp ⟦ σ ⟧′ ⟦ t ⟧ ⟦ u ⟧ ⟩
    app ⟦ t ⟧ ⟦ u ⟧ [ ⟦ σ ⟧′ ]
      ∎
    where open EqR (TmSetoid {_})
  []−comm (ƛ t) σ = begin
    lam ⟦ t [ ↑ σ ]λ ⟧
      ≈⟨ cong−lam $ []−comm t (↑ σ) ⟩
    lam (⟦ t ⟧ [ < ⟦ map weaken−λ σ ⟧′ , q > ])
      ≈⟨ cong−lam $ cong−sub refl≈ help ⟩
    lam (⟦ t ⟧ [ < ⟦ σ ∘λ p−λ ⟧′ , q > ])
      ≈⟨ cong−lam $ cong−sub refl≈
         (cong−<, > refl≈ (⟦⟧−∘−dist_p σ p−λ)) ⟩
    lam (⟦ t ⟧ [ < ⟦ σ ⟧′ ∘ ⟦ p−λ ⟧′ , q > ])
      ≈⟨ cong−lam $ cong−sub refl≈
         (cong−<, > refl≈ (cong−∘ refl≈ (sym≋ $ p−λ≈⟦p⟧))) ⟩
    lam (⟦ t ⟧ [ < ⟦ σ ⟧′ ∘ p , q > ])
      ≈⟨ sym≈ (subLam ⟦ σ ⟧′ ⟦ t ⟧) ⟩
    lam ⟦ t ⟧ [ ⟦ σ ⟧′ ]
      ∎
    where open EqR (TmSetoid {_})
      help : < ⟦ map weaken−λ σ ⟧′ , q > ≈ < ⟦ σ ∘λ p−λ ⟧′ , q >
      help rewrite sym (mapWk−∘p σ) = refl≈
```

These modifications that build on the previous work for the pure ucwfs are sufficient to extend the isomorphism to these λ-ucwf objects.

3.2.4 λ-βη-Ucwfs

Finally, as a last extension to the language, we can add beta and eta rules as extra structure and hence create a λ-βη ucwf. Since this is how one usually presents lambda calculus, it is nicer to extend our results up to beta and eta. The extension for an abstract object just adds the two equalities as follows.

```
  record Lambda−βη−ucwf {ℓ} : Set (suc ℓ)
    field
      lambda−ucwf : Lambda−ucwf {ℓ}
    open Lambda−ucwf lambda−ucwf public
    field
      – beta and eta equalities
      β : ∀ {n} {t : Tm (suc n)} {u} → app (lam t) u ≈ t [ < id , u > ]
      η : ∀ {n} {t : Tm n} → lam (app (t [ p ]) q) ≈ t
```

For the explicit λ-βη ucwf object, we need only add these two equations in the equality relation for terms.

```
data _≈_ where
  ...
  β : ∀ {n} {t : Tm (suc n)} {u} → app (lam t) u ≈ t [ < id , u > ]
  η : ∀ {n} {t : Tm n} → lam (app (t [ p ]) q) ≈ t
```

Here, Tm refers to the actual data type and not the record field; it is part of the equality relation for the explicit ucwf object with extra structure. And so by adding these equalities to the relation, that *λ*-ucwf object becomes a *λ*-*βη*-ucwf object directly.

On the other hand, the traditional calculus requires significant changes. To begin with, we need a relation that describes beta-eta convertibility for lambda terms.

```
data _~βη_ {n} : (_ _ : Tm n) → Set where

  – variables with the same index are convertible
  varcong : ∀ i → var i ~βη var i

  – congruence for application
  apcong : ∀ {t u t′ u′} →
             t ~βη t′ →
             u ~βη u′ →
             t · u ~βη t′ · u′

  – the ξ rule
  ξ : ∀ {t u} → t ~βη u → ƛ t ~βη ƛ u

  – beta and eta equalities
  β : ∀ {t u} → ƛ t · u ~βη t [ id , u ]
  η : ∀ {t} → ƛ (weaken t · q) ~βη t

  sym~βη : ∀ {t₁ t₂} → t₁ ~βη t₂ → t₂ ~βη t₁
  trans~βη : ∀ {t₁ t₂ t₃} → t₁ ~βη t₂ → t₂ ~βη t₃ → t₁ ~βη t₃

data _≈βη_ {m} : ∀ {n} (_ _ : Sub m n) → Set where

  – empty substitutions are convertible
  ⋄ : ∀ {ρ ρ′ : Sub m 0} → ρ ≈βη ρ′

  – if terms and substitutions are convertible, so is cons
  ext : ∀ {n} {t u : Tm m} {ρ ρ′ : Sub m n} →
    t ~βη u → ρ ≈βη ρ′ →
    (ρ , t) ≈βη (ρ′ , u)
```

Naturally, every property has to be proven at the level of this relation instead of propositional equality. However, the preceding work contains all the proofs as elements of identity, hence, it is easy to lift a proof in identity to a proof of beta-eta convertible.

```
≡−to~βη : ∀ {n} {t u : Tm n} → t ≡ u → t ~βη u
≡−to~βη refl = refl~βη

≡−to−≈βη : ∀ {m n} {ρ σ : Sub m n} → ρ ≡ σ → ρ ≈βη σ
≡−to−≈βη refl = refl≈βη
```

This means that all the properties, ucwf axioms, and inverse lemmas can be proven by a call to the equivalent proof in identity. There is, however, some extra work in proving the congruence rules; these proofs can be found in figure A.4.

Chapter 4

Simply Typed CwFs

This chapter explores two different formulations of simply typed cwfs (scwfs) by constructing isomorphisms between scwf objects. Much like the unityped version, the core scwfs model a basic framework for variables, but with types this time that depend on some context. The generalized algebraic theory of cwfs with some adjustments yields this theory for simple types.

Generally, there are two main options of formalizing simply typed terms; one way is to implement directly typed terms that represent typing derivations of untyped terms that are no visible. An example signature in Agda would be this: data Tm (Γ : Ctx) : Ty → Set, where Ctx and Ty represent context and type implementations. The constructors for this set construct typed terms that are referred to as intrinsically typed terms.

On the other hand, one can build types starting with untyped raw terms by adding a typing relation that gives types to terms, data _⊢_∈_ (Γ : Ctx) : Raw → Ty → Set, where Raw is the set of untyped terms. Here, the typing derivations are shown. This approach can be called extrinsic for the obvious antithesis with the former version.

In contrast to the untyped version, there are now significant different options to explore when it comes to scwfs and simply typed λ-calculus.

Considering all this, we start by formalizing intrinsic scwfs and typed variables and adding extra structure as usual to obtain a typed λ-calculus and λ-scwfs. This follows an identical pattern and structure to the untyped work. Afterwards, we show how the extrinsic scwf version works to compare differences. Overall, three different isomorphisms between some forms of scwf objects will be shown in this chapter. Consider the following figure that depicts the landscape in which we operate in.

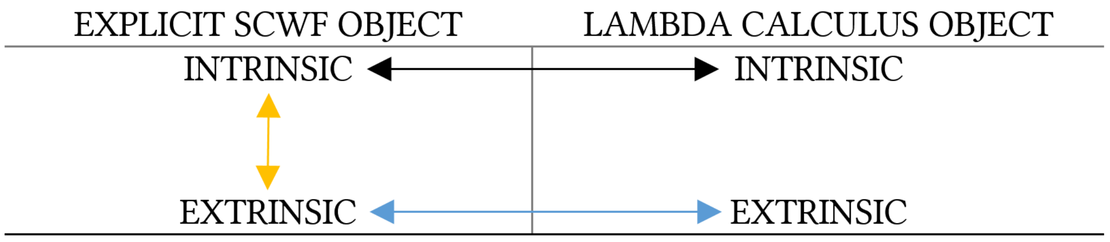

Figure 4.1: Scwf variations explored.

Figure ?? shows different possibilities one can explore. On the left column we have scwf objects that are implemented as calculi of explicit substitutions, whereas on the right we have traditional versions with substitutions formalized in the meta language.

First, the connection depicted by the black arrow in the diagram is investigated. This formalization is performed again on three levels, starting from pure and adding extra structure and concluding with beta and eta. Then, we move to extrinsic versions (blue arrow) and finally, we see an isomorphism between an intrinsic and extrinsic formulations.

4.1 Scwf and Variables

4.1.1 Pure Scwfs

Notwithstanding the addition of types and contexts, a scwf is quite similar to the untyped one with key differences being that substitutions are now indexed by 'real' contexts instead of natural numbers and terms are typed in some context. This formulation presents intrinsically typed scwfs, so terms are directly typed. As before, we start with pure scwfs and typed variables and extend to full lambda calculus later. Next, we show the notion of scwf as a record in Agda.

```
record Scwf {l} : Set (lsuc l) where
  field
    – types, contexts, terms, and substitutions
    Ty  : Set
    Ctx : Set
    Tm  : Ctx → Ty → Set
    Sub : Ctx → Ctx → Set

    – empty context (terminal object)
    ⋄    : Ctx

    – context extension
    _•_ : Ctx → Ty → Ctx

    – equality of terms and substitutions
    _≈_  : ∀ {Γ α} → Rel (Tm Γ α) l
    _≋_ : ∀ {Γ Δ} → Rel (Sub Γ Δ) l

    IsEquivT : ∀ {Γ α} → IsEquivalence (_≈_ {Γ} {α})
    IsEquivS : ∀ {Γ Δ} → IsEquivalence (_≋_ {Γ} {Δ})

    – last variable and explicit substitution
    q     : ∀ {Γ α}   → Tm (Γ • α) α
    _[_]  : ∀ {Γ Δ α} → Tm Γ α  → Sub Δ Γ → Tm Δ α

    – morphisms of the base category
    id    : ∀ {Γ}     → Sub Γ Γ
    _∘_   : ∀ {Γ Δ Θ} → Sub Γ Θ → Sub Δ Γ → Sub Δ Θ
    <>    : ∀ {Γ}     → Sub Γ ⋄
    <_,_> : ∀ {Γ Δ α} → Sub Γ Δ → Tm Γ α  → Sub Γ (Δ • α)
    p     : ∀ {Γ α}   → Sub (Γ • α) Γ
```

Two new sets, one for types and one for contexts have been added, along with the empty context ⋄ and context extension (add type). The empty context is the terminal object of the base category and now we explicitly present a context extension operator while in our untyped development it was the successor function.

Maintaining type information does not affect much the description. Contexts are going to be actual lists of types now, but this is a similar variable free calculus of explicit substitutions for typed functions. The field q refers to the last variable (or last assumption) and variables are constructed by weakening q. A morphism from a context Γ to Δ means that in environment Γ, we have a substitution where terms are typed in Δ.

We continue by adding typed cwf axioms in the record as fields.

```
– category of substitutions
idL   : ∀ {Γ Δ} (γ : Sub Δ Γ) → id ∘ γ ≋ γ
idR   : ∀ {Γ Δ} (γ : Sub Γ Δ) → γ ∘ id ≋ γ
assoc : ∀ {Γ Δ Θ Λ} (γ : Sub Δ Θ) (δ : Sub Γ Δ) (ζ : Sub Λ Γ) →
  (γ ∘ δ) ∘ ζ ≋ γ ∘ (δ ∘ ζ)

– rules for the functor
subId : ∀ {Γ α} (t : Tm Γ α) → t [ id ] ≈ t
subComp : ∀ {Γ Δ Θ α} (t : Tm Δ α) (γ : Sub Γ Δ) (δ : Sub Θ Γ) →
  t [ γ ∘ δ ] ≈ t [ γ ] [ δ ]

– rules for the terminal object
id0 : id {⋄} ≋ <>
<>Lzero : ∀ {Γ Δ} (γ : Sub Γ Δ) → <> ∘ γ ≋ <>

– rules for context comprehension
pCons   : ∀ {Δ Θ α} (t : Tm Δ α) (γ : Sub Δ Θ) → p ∘ < γ , t > ≋ γ
qCons   : ∀ {Γ Δ α} (t : Tm Γ α) (γ : Sub Γ Δ) → q [ < γ , t > ] ≈ t
idExt   : ∀ {Γ α} → id {Γ • α} ≋ < p , q >
compExt : ∀ {Γ Δ α} (t : Tm Δ α) (γ : Sub Δ Γ) (δ : Sub Γ Δ) →
  < γ , t > ∘ δ ≋ < γ ∘ δ , t [ δ ] >

– congruence closure
cong−sub : ∀ {Γ Δ α} {t t′ : Tm Γ α} {γ γ′ : Sub Δ Γ} →
  t ≈ t′ →
  γ ≋ γ′ →
  t [ γ ] ≈ t′ [ γ′ ]
cong−<,> : ∀ {Γ Δ α} {t t′ : Tm Γ α} {γ γ′ : Sub Γ Δ} →
  t ≈ t′ →
  γ ≋ γ′ →
  < γ , t > ≋ < γ′ , t′ >
cong−∘ : ∀ {Γ Δ Θ} {γ δ : Sub Δ Θ} {γ′ δ′ : Sub Γ Δ} →
  γ ≋ δ →
  γ′ ≋ δ′ →
  γ ∘ γ′ ≋ δ ∘ δ′
```

Having defined the abstract notion of scwf a generalized algebraic theory, we can start constructing scwf objects.

A straightforward scwf object is implemented as a data type where all operators and laws are explicit in accordance to the record. First, define contexts as lists of types and start with a base type ♭.

```
data Ctxt (A : Set) : Set where
  ε  : Ctxt A
  _•_ : Ctxt A → A → Ctxt A

data Ty : Set where
  ♭  : Ty

Ctx : Set
Ctx = Ctxt Ty

data Tm : Ctx → Ty → Set
data Sub : Ctx → Ctx → Set
```

After, two mutually recursive data types representing terms and substitutions.

```
data Tm where
  q : ∀ {Γ α} → Tm (Γ • α) α
  _[_] : ∀ {Γ Δ α} → Tm Γ α  → Sub Δ Γ → Tm Δ α

data Sub where
  id    : ∀ {Γ} → Sub Γ Γ
  _∘_   : ∀ {Γ Δ Θ} → Sub Γ Θ → Sub Δ Γ → Sub Δ Θ
  <>    : ∀ {Γ} → Sub Γ ε
  <_,_> : ∀ {Γ Δ α} → Sub Γ Δ → Tm Γ α  → Sub Γ (Δ • α)
  p     : ∀ {Γ α} → Sub (Γ • α) Γ

data _≈_  : ∀ {Γ α} (t₁ t₂ : Tm Γ α) → Set
data _≋_ : ∀ {Γ Δ} (γ₁ γ₂ : Sub Γ Δ) → Set
```

These two relations include all the laws of the Scwf record defined earlier as introduction rules, the congruence rules, plus symmetry and transitivity to get an equivalence relation. The equations are omitted but available in figure A.2 of the appendix. The resulting object should be initial in the category of scwfs.

4.1.2 Typed Variables as Scwf object

Pure scwfs model typed variables and so the goal is to construct another scwf object, but with some notion of typed variable with non explicit substitution. Contexts are inductive lists of types accompanied with notions of inclusion and membership as inductive relations. In other words, a variable is a membership witness in the context it is typed in.

```
data _⊆_ {A : Set} : Ctxt A → Ctxt A → Set where
  base : ε ⊆ ε
  step : ∀ {Γ Δ : Ctxt A} {σ} (φ : Γ ⊆ Δ) → Γ ⊆ (Δ • σ)
  pop! : ∀ {Γ Δ : Ctxt A} {σ} (ψ : Γ ⊆ Δ) → (Γ • σ) ⊆ (Δ • σ)
```

Contexts are polymorphic and we define a set Ctx which instantiates a context with Ty. Moreover, the relation for membership that represents variables.

```
data _∈_ {A : Set} (α : A) : Ctxt A → Set where
  here  : {Γ : Ctxt A} → α ∈ Γ • α
  there : {Γ : Ctxt A} {α′ : A} → α ∈ Γ → α ∈ Γ • α′
```

Subsequently, we implement a data type for terms that consists only of variables.

```
data Tm (Γ : Ctx) : Ty → Set where
  var : ∀ {α} (v : α ∈ Γ) → Tm Γ α

weaken : ∀ {α} {Γ Δ : Ctx} (φ : Γ ⊆ Δ) (t : Tm Γ α) → Tm Δ α
weaken φ (var ∈Γ) = var (wk−ctx φ ∈Γ)
  where wk−ctx : ∀ {β : Ty} {Γ Δ} → Γ ⊆ Δ → β ∈ Γ → β ∈ Δ
    wk−ctx ⊆Δ v = sub−in ⊆Δ v

q : ∀ {Γ α} → Tm (Γ • α) α
q = var here
```

Where sub−in is a general proof that if some element is in a subset, then it is also in the superset. The cwf operator q is a proof that the type is at the top of the context.

Moving to well-typed substitutions, they are sequences of terms typed in some context; we use the standard library's product type to define them by recursion on the second context. We write Sub Δ Γ to note a substitution where terms are typed in Δ. Further, operators like the identity id and projection p substitutions are defined by recursion using weakening. Regarding the substitution operation, a lookup-like function based on a membership witness is needed.

```
Sub : (Δ Γ : Ctx) → Set
Sub Δ ε  = ⊤
Sub Δ (Γ • t) = Sub Δ Γ × Tm Δ t

– weakening a substitution applies weaken to each term
wk−sub : ({Δ} {Θ} Γ : Ctx) (φ : Δ ⊆ Θ) (ρ : Sub Δ Γ) → Sub Θ Γ
wk−sub ε  φ ρ  = tt
wk−sub (Γ • x) φ (ρ , t) = wk−sub Γ φ ρ , weaken φ t

– identity substitution
id : ∀ {Γ} → Sub Γ Γ
id {ε}  = tt
id {Γ • α} = wk−sub Γ ⊆−• id , q
```

```
– lookup for typed substitutions
tkVar : ∀ {Γ Δ α} (v : α ∈ Γ) (ρ : Sub Δ Γ) → Tm Δ α
tkVar here  (ρ , t) = t
tkVar (there v) (ρ , t) = tkVar v ρ

– substitution operation
_[_] : ∀ {Γ Δ α} → Tm Γ α → Sub Δ Γ → Tm Δ α
var v  [ ρ ] = tkVar v ρ

– projection substitution
p : ∀ {Γ α} → Sub (Γ • α) Γ
p {Γ} = wk–sub Γ ⊆–• id

– composition of substitutions
_∘_ : ∀ {Γ Δ Θ} → Sub Γ Θ → Sub Δ Γ → Sub Δ Θ
_∘_ {Θ = ε}  ρ  σ = tt
_∘_ {Θ = Θ • α} (ρ , t) σ = ρ ∘ σ , t [ σ ]
```

Where ⊆–• refers to a proof that Γ ⊆ (Γ • α), for any Γ.

These operations correspond to scwf operators presented in the Scwf record. We see that Sub represents morphisms between contexts and it is implemented as a product; this means that extending a substitution is done using the pair constructor for products _, _. Moreover, the empty morphism is tt, the single element of top (unit). This is because when the environment of a substitution is empty, we return the top set, ⊤. Regarding equality, propositional equality is considered for now.

The next step consists of proving the scwf laws using the operations just defined to finish the construction of a second intrinsic scwf object.

Lemma 4.1. id {Γ • α} ≡ (p , q)

Proof. Reflexivity.

```
idExt : ∀ {Γ α} → id {Γ • α} ≡ (p , q)
idExt = refl
```

■

Lemma 4.2. t [id] ≡ t

Proof. Lookup properties on id and weakened id.

```
tkVar–wk–id : ∀ {Γ Δ α} (v : α ∈ Γ) (φ : Γ ⊆ Δ) →
   tkVar v (wk–sub Γ φ id) ≡ var (sub–in φ v)
tkVar–wk–id {ε} () _
tkVar–wk–id {Γ • x} here φ = refl
tkVar–wk–id {Γ • x} (there v) φ =
   trans (cong (tkVar v) (▷–wk–2 Γ (step ⊆–refl) φ id))
      (trans (tkVar–wk–id v (⊆–trans (step ⊆–refl) φ))
         (cong var (sub–in–step φ v)))

tkVar–id : ∀ {Γ α} (v : α ∈ Γ) → tkVar v id ≡ var v
```

```
tkVar−id here = refl
tkVar−id {Γ = Γ • x} (there v) =
  trans (tkVar−wk−id v ⊆−•)
    (cong (var F.∘ there) (sub−in−refl v))

subId : ∀ {Γ α} (t : Tm Γ α) → t [ id ] ≡ t
subId (var v) = tkVar−id v
```

■

Lemma 4.3. t [γ ∘ δ] ≡ t [γ] [δ]

Proof. Induction on t and mutual induction on γ in the variable case.

```
subComp : ∀ {Γ Δ Θ α} (t : Tm Γ α) (γ : Sub Δ Γ) (δ : Sub Θ Δ) →
  t [ γ ∘ δ ] ≡ t [ γ ] [ δ ]
subComp (var here) γ δ = refl
subComp (var (there v)) (γ , u) δ = subComp (var v) γ δ
```

■

Lemma 4.4. id_0 : id {ε} ≡ tt

Proof. Reflexivity.

```
  id₀ : id {ε} ≡ tt
  id₀ = refl
```

■

Lemma 4.5. tt ∘ ρ ≡ tt

Proof. Reflexivity.

```
tt−lzero : ∀ {Γ Δ} (ρ : Sub Γ Δ) → tt ∘ ρ ≡ tt
tt−lzero _ = refl
```

■

Lemma 4.6. p ∘ (γ , t) ≡ γ

Proof. By lemma 4.10 and a general property: if a pair is composed with any weakened substitution, the last variable is forgotten.

```
pCons : ∀ {Δ Θ α} (t : Tm Δ α) (γ : Sub Δ Θ) → p ∘ (γ , t) ≡ γ
pCons {Θ = Θ} t = trans (∘−step Θ id _ t) F.∘ idL
```

■

Lemma 4.7. (γ ∘ δ) ∘ ζ ≡ γ ∘ (δ ∘ ζ)

Proof. Induction on γ.

```
assoc : ∀ {Γ Δ Θ Λ} (γ : Sub Δ Θ) (δ : Sub Γ Δ) (ζ : Sub Λ Γ) →
    (γ ∘ δ) ∘ ζ ≡ γ ∘ (δ ∘ ζ)
assoc {Θ = ε}  tt  δ ζ = refl
assoc {Θ = Θ • _} (γ , t) δ ζ =
  trans (cong ((γ ∘ δ) ∘ ζ ,_) (sym (subComp t δ ζ)))
    (cong (_, t [ δ ∘ ζ ]) (assoc γ δ ζ))
```

■

Lemma 4.8. q [ρ , t] ≡ t

Proof. Reflexivity.

```
qCons : ∀ {Γ Δ α} (t : Tm Γ α) (ρ : Sub Γ Δ) → q [ ρ , t ] ≡ t
qCons t ρ = refl
```

■

Lemma 4.9. (γ , t) ∘ δ ≡ (γ ∘ δ , (t [δ]))

Proof. Reflexivity.

```
compExt : ∀ {Γ Δ} {α : Ty} (t : Tm Δ α) (γ : Sub Δ Γ) (δ : Sub Γ Δ) →
  (γ , t) ∘ δ ≡ (γ ∘ δ , (t [ δ ]))
compExt = λ _ _ _ → refl
```

■

Lemma 4.10. id ∘ ρ ≡ ρ

Proof. Induction on ρ. Mutually proven with lemma 4.6

```
idL : ∀ {Γ Δ} (ρ : Sub Δ Γ) → id ∘ ρ ≡ ρ
idL {ε}  tt  = refl
idL {Γ • α} (ρ , t) = cong (_, t) (pCons t ρ)
```

■

Lemma 4.11. ρ ∘ id ≡ ρ

Proof. Induction on ρ.

```
idR : ∀ {Γ Δ} (ρ : Sub Γ Δ) → ρ ∘ id ≡ ρ
idR {Δ = ε}  tt  = refl
idR {Δ = Δ • x} (ρ , t) =
  trans (cong (_, t [ id ]) (idR ρ))
    (cong (ρ , _) (subId t))
```

■

Theorem 4.1 (Typed variables are a scwf object). *Typed variables as membership proofs with substitutions form a scwf object with the base category defined by objects in* Ctx *and morphisms in* Sub *and the functor* Tm.

4.1.3 Isomorphism of Scwf objects

Hitherto, we have defined two intrinsically typed scwf objects, the explicit and the traditional one with variables as membership proofs. Now we proceed to define scwf morphisms between these objects and prove that these objects are isomorphic.

Definition 4.1. Scwf morphisms

```
varCwf : ∀ {Γ α} (φ : α ∈ Γ) → Tm−cwf Γ α
varCwf here  = q
varCwf (there φ) = varCwf φ [ p ]
```

```
⟦_⟧   : ∀ {Γ α} → Tm−λ Γ α → Tm−cwf Γ α
⟪_⟫   : ∀ {Γ α} → Tm−cwf Γ α → Tm−λ Γ α
⟪_⟫′ : ∀ {Γ Δ} → Sub−cwf Δ Γ → Sub−λ Δ Γ
⟦_⟧′ : ∀ {Γ Δ} → Sub−λ Δ Γ → Sub−cwf Δ Γ

⟦ var v ⟧  = varCwf v

⟪ q ⟫ = q−λ
⟪ t [ ρ ] ⟫ = ⟪ t ⟫ [ ⟪ ρ ⟫′ ]λ

⟦_⟧′ {ε} _  = <>
⟦_⟧′ {Γ • α} (ρ , t) = < ⟦ ρ ⟧′ , ⟦ t ⟧ >

⟪ <> ⟫′      = tt
⟪ id ⟫′      = id−λ
⟪ p   ⟫′     = p−λ
⟪ γ ∘ γ′ ⟫′  = ⟪ γ ⟫′ ∘λ ⟪ γ′ ⟫′
⟪ < γ , t > ⟫′ = ⟪ γ ⟫′ , ⟪ t ⟫
```

As one can see, keeping type information does not affect these maps, the structure is identical to the unityped version. Agda's syntax highlighting also help in noticing how numerous inductive constructors in green are mapped to meta operations that are functions in blue.

The categorical context will be reiterated since it has significance. We are trying to show an isomorphism between scwf objects that are defined by a base category where objects are contexts and morphisms are substitutions. Objects of the base category, Ctx, are defined by the same set, while morphisms Sub−λ and Sub−cwf are naturally different. So ⟪_⟫′ and ⟦_⟧′ are functors that map morphisms (objects are mapped with identity). Additionally, the equivalent maps for terms, i.e., ⟪_⟫ ⟦_⟧ are natural transformations. In other words, an isomorphism of the base categories and a natural isomorphism between the functors would suffice to show isomorphism.

Moving to proofs now; the equivalent lemmas of 3.12 and 3.13 which are similarly vital are in figure B.3 of the appendix. These are only the signatures.

```
[]−comm : ∀ {Γ Δ α} (t : Tm−λ Γ α) (ρ : Sub−λ Δ Γ) →
      ⟦ t [ ρ ]λ ⟧ ≈ ⟦ t ⟧ [ ⟦ ρ ⟧′ ]

⟦⟧−∘−dist : ∀ {Γ Δ Θ} (ρ : Sub−λ Δ Θ) (σ : Sub−λ Γ Δ) →
      ⟦ ρ ∘λ σ ⟧′ ≋ ⟦ ρ ⟧′ ∘ ⟦ σ ⟧′
```

Next, we provide the top level inverse lemma signatures.

```
sub−cwf⇒λ : ∀ {Γ Δ} (γ : Sub−cwf Γ Δ) → ⟦ ⟪ γ ⟫′ ⟧′ ≋ γ

tm−λ⇒cwf : ∀ {Γ α} (t : Tm−λ Γ α) → ⟪ ⟦ t ⟧ ⟫ ≡ t

tm−cwf⇒λ : ∀ {Γ α} (t : Tm−cwf Γ α) → ⟦ ⟪ t ⟫ ⟧ ≈ t
```

```
sub−λ⇒cwf : ∀ {Γ Δ} (γ : Sub−λ Γ Δ) → ⟪ ⟦ γ ⟧′ ⟫′ ≡ γ
```

They are proven in similar fashion, the key difference is that supporting properties in this case are related to reasoning with contexts, inclusion, and membership; whereas in the untyped case, it was mostly vector reasoning. Moreover, it was occasionally necessary to provide some implicit argument to Agda to assist with some unsolved constraint, along with pattern matching on an appropriate implicit context for performing induction.

Lemma 4.12. ⟪ ⟦ t ⟧ ⟫ ≡ t *and* ⟦ ⟪ t ⟫ ⟧ ≈ t

Proof. Induction on t.

```
tm−λ⇒cwf (var here) = refl
tm−λ⇒cwf {α = α} (var (there v)) = begin
  ⟪ ⟦ var v ⟧ ⟫ [ p−λ ]λ
    ≡⟨ cong (_[ p−λ ]λ) (tm−λ⇒cwf (var v)) ⟩
  var v [ p−λ ]λ
    ≡⟨ sub−p (var v) ⟩
  weaken−λ ⊆−• (var v)
    ≡⟨⟩
  var (there (sub−in ⊆−refl v))
    ≡⟨ cong (var F.∘ there) (sub−in−refl v) ⟩
  var (there v)
    ∎
  where open P.≡−Reasoning

tm−cwf⇒λ q = refl≈
tm−cwf⇒λ (t [ γ ]) = begin
  ⟦ ⟪ t ⟫ [ ⟪ γ ⟫′ ]λ ⟧
    ≈⟨ []−comm ⟪ t ⟫ ⟪ γ ⟫′ ⟩
  ⟦ ⟪ t ⟫ ⟧ [ ⟦ ⟪ γ ⟫′ ⟧′ ]
    ≈⟨ cong−sub (tm−cwf⇒λ t) refl≈ ⟩
  t [ ⟦ ⟪ γ ⟫′ ⟧′ ]
    ≈⟨ cong−sub refl≈ (sub−cwf⇒λ γ) ⟩
  t [ γ ]
    ∎
  where open EqR (TmSetoid {_})
```

∎

This proves that structure is preserved and thus we have a natural isomorphism.

Next, the proof the the base categories are isomorphic.

Lemma 4.13. ⟦ ⟪ γ ⟫′ ⟧′ ≈ γ *and* ⟪ ⟦ γ ⟧′ ⟫′ ≡ γ.

Proof. Induction on γ.

```
sub−cwf⇒λ <> = refl≈
sub−cwf⇒λ (id {ε}) = sym≈ id₀
sub−cwf⇒λ (id {Γ • α}) = sym≈ $ begin
```

```
    id
      ≈⟨ idExt ⟩
    < p , q >
      ≈⟨ cong−<, > refl≈ (sym≈ (sub−cwf⇒λ p)) ⟩
    < ⟦ p−λ ⟧′ , q >
      ∎
    where open EqR (SubSetoid {_} {_})
  sub−cwf⇒λ p =p−inverse
  sub−cwf⇒λ (γ ∘ γ′) = begin
    ⟦ ⟪ γ ⟫′ ∘λ ⟪ γ′ ⟫′ ⟧′
      ≈⟨ ⟦⟧−∘−dist ⟪ γ ⟫′ ⟪ γ′ ⟫′ ⟩
    ⟦ ⟪ γ ⟫′ ⟧′ ∘ ⟦ ⟪ γ′ ⟫′ ⟧′
      ≈⟨ cong−∘ (sub−cwf⇒λ γ) refl≈ ⟩
    γ ∘ ⟦ ⟪ γ′ ⟫′ ⟧′
      ≈⟨ cong−∘ refl≈ (sub−cwf⇒λ γ′) ⟩
    γ ∘ γ′
      ∎
    where open EqR (SubSetoid {_} {_})
  sub−cwf⇒λ < γ , t > = cong−<, > (tm−cwf⇒λ t) (sub−cwf⇒λ γ)

  sub−λ⇒cwf {Δ = ε} tt = refl
  sub−λ⇒cwf {Δ = Δ • x} (γ , t) = cong₂ _,_ (sub−λ⇒cwf γ) (tm−λ⇒cwf t)
```
■

The projection case has a very similar structure to the one presented for unityped cwfs. There, we used sequences of Fins to construct the normalized projection morphism. Using this formulation here for simple types, we need sequences of witness proofs that a type is in a given context. The complete proof for the projection is in figure B.4.

Theorem 4.2. *There exists an isomorphism between the previously constructed scwf objects. Objects of both base categories are contexts defined by* Ctx*, morphisms* Sub−λ *and* Sub−cwf*, and functors* Tm−λ *and* Tm−cwf.

4.2 λ-Scwfs and λ-calculus

In this section, pure scwfs are extended with lambda abstractions and applications making it a model of simply typed λ-calculus. This scwf with extra structure is called λ-scwf.

4.2.1 λ-Scwfs

The extension of a scwf to model simply typed λ-calculus requires two new term constructors, a function type and the appropriate axioms.

```
  record Lambda−scwf {l} : Set (lsuc l) where
    field
      scwf : Scwf {l}
    open Scwf scwf public
    field
      − Function type
```

```
_'→_ : Ty → Ty → Ty

– λ abstractions and application
lam : ∀ {Γ α β} → Tm (Γ • α) β → Tm Γ (α '→ β)
app : ∀ {Γ α β} → Tm Γ (α '→ β) → Tm Γ α → Tm Γ β

– substituting under lam and app
subApp : ∀ {Γ Δ α β} (t : Tm Γ (α '→ β)) (u : Tm Γ α) (γ : Sub Δ Γ) →
  app (t [ γ ]) (u [ γ ]) ≈ (app t u) [ γ ]
subLam : ∀ {Γ Δ α β} (t : Tm (Γ • α) β) (γ : Sub Δ Γ) →
  lam t [ γ ] ≈ lam (t [ < γ ∘ p , q > ])

– congruence rules
cong−lam : ∀ {Γ α β} {t t′ : Tm (Γ • α) β} →
  t ≈ t′ →
  lam t ≈ lam t′
cong−app : ∀ {Γ α β} {t t′ : Tm Γ (α '→ β)} {u u′} →
  t ≈ t′ →
  u ≈ u′ →
  app t u ≈ app t′ u′
```

The initial λ-scwf explicit object also is expanded quite easily. Substitutions remain the same, while we add the two new elements to the set of terms Tm and a function type.

```
data Ty : Set where
  ♭ : Ty
  _⇒_ : (α β : Ty) → Ty

data Tm : Ctx → Ty → Set
data Sub : Ctx → Ctx → Set

data Tm where
  q : ∀ {Γ α} → Tm (Γ • α) α
  _[_] : ∀ {Γ Δ α} → Tm Γ α  → Sub Δ Γ → Tm Δ α
  lam : ∀ {Γ α β} → Tm (Γ • α) β → Tm Γ (α ⇒ β)
  app : ∀ {Γ α β} → Tm Γ (α ⇒ β) → Tm Γ α → Tm Γ β

data Sub where
  id    : ∀ {Γ} → Sub Γ Γ
  _∘_   : ∀ {Γ Δ Θ} → Sub Γ Θ → Sub Δ Γ → Sub Δ Θ
  <>    : ∀ {Γ} → Sub Γ ε
  <_,_> : ∀ {Γ Δ α} → Sub Γ Δ → Tm Γ α  → Sub Γ (Δ • α)
  p     : ∀ {Γ α} → Sub (Γ • α) Γ

data _≈_  : ∀ {Γ α} (t₁ t₂ : Tm Γ α) → Set
data _≋_ : ∀ {Γ Δ} (γ₁ γ₂ : Sub Γ Δ) → Set
```

The equations are in figure ?? they add the four laws shown in the Lambda−scwf for terms. It is intended for this object to be a λ-scwf by definition, ergo, these new laws are added as introduction rules in the equality relation.

Next, we proceed to implement simply typed λ-calculus and demonstrate that is forms a λ-scwf.

4.2.2 Simply Typed λ-calculus as Scwf object

Having typed variables as membership witnesses in contexts already; adding λ abstractions, applications, and a function type yields an intrinsic simply typed λ-calculus in the usual sense. These terms are directly (intrinsically) typed representing typing derivations of untyped terms. We show the new language and some important scwf operations on them.

```
data Ty : Set where
  ♭ : Ty
  _⇒_ : (α β : Ty) → Ty
```

Having a function type allows us to define lambda terms.

```
data Tm (Γ : Ctx) : Ty → Set where
  var : ∀ {α} (∈Γ : α ∈ Γ) → Tm Γ α
  _·_ : ∀ {α β} → Tm Γ (α ⇒ β) → Tm Γ α → Tm Γ β
  ƛ  : ∀ {α β} → Tm (Γ • α) β → Tm Γ (α ⇒ β)

weaken : ∀ {α} {Γ Δ : Ctx} (φ : Γ ⊆ Δ) (t : Tm Γ α) → Tm Δ α
weaken φ (var v) = var (sub−in φ v)
weaken φ (t · u)  = weaken φ t · weaken φ u
weaken φ (ƛ t)    = ƛ (weaken (pop! φ) t)
```

The substitution operation which depends on weakening now covers the new Term constructors. The λ case is interesting where the substitution ρ is weakened and extended with the last variable.

```
_[_] : ∀ {Γ Δ α} → Tm Γ α → Sub Δ Γ → Tm Δ α
var v  [ ρ ] = tkVar v ρ
ƛ t     [ ρ ] = ƛ (t [ wk−sub _ ⊆−• ρ , var here ])
(t · u) [ ρ ] = t [ ρ ] · u [ ρ ]
```

The other scwf operations remain the same as shown in the previous section.

In order to demonstrate that this is a λ-scwf, we extend the proofs where there was an induction on terms, namely, substituting in id does not affect terms and associativity of substitution.

Lemma 4.14. t [id] ≡ t

Proof. By induction on t.

```
subId : ∀ {Γ α} (t : Tm Γ α) → t [ id ] ≡ t
subId (var v) = tkVar−id v
subId (t · u) = cong₂ _·_ (subId t) (subId u)
subId (ƛ t) = cong ƛ (subId t)
```

■

Lemma 4.15. t [γ ∘ δ] ≡ t [γ] [δ]

Proof. Induction on t and mutual induction on γ in the variable case.

```
↑_ : ∀ {Γ Δ α} → Sub Δ Γ → Sub (Δ • α) (Γ • α)
↑_ σ = wk−sub _ ⊆−• σ , q

subComp : ∀ {Γ Δ Θ α} (t : Tm Γ α) (γ : Sub Δ Γ) (δ : Sub Θ Δ) →
      t [ γ ∘ δ ] ≡ t [ γ ] [ δ ]
subComp (var here) γ δ = refl
subComp (var (there v)) (γ , _) δ = subComp (var v) γ δ
subComp (t · u) γ δ = cong₂ _·_ (subComp t γ δ) (subComp u γ δ)
subComp {Γ} {Δ} (ƛ t) γ δ = sym $ cong ƛ $ begin
   t [ ↑ γ ] [ ↑ δ ]
      ≡⟨ sym $ subComp t (↑ γ) (↑ δ) ⟩
   t [ proj₁ (↑ γ) ∘ (↑ δ) , q ]
      ≡⟨ cong (t [_] F.∘ (_, q)) $
         begin
            proj₁ (↑ γ) ∘ (↑ δ) ≡⟨ ∘−step Γ γ (proj₁ (↑ δ)) q ⟩
            γ ∘ proj₁ (↑ δ)     ≡⟨ wk−∘ Γ ⊆−• γ δ ⟩
            proj₁ (↑ (γ ∘ δ))  ∎ ⟩
   t [ ↑ (γ ∘ δ) ] ∎
```

■

We have therefore shown that this simply typed λ-calculus is a λ-scwf object.

4.2.3 Isomorphism of λ-Scwf objects

With the intention to extend the isomorphism proof to λ-scwf, the morphisms have to be extended to account for new elements. The change is only at the term level.

Definition 4.2. λ-scwf morphisms

```
⟦_⟧  : ∀ {Γ α} → Tm−λ Γ α → Tm−cwf Γ α
⟪_⟫  : ∀ {Γ α} → Tm−cwf Γ α → Tm−λ Γ α

⟦ var v ⟧ = varCwf v
⟦ t · u ⟧ = app ⟦ t ⟧ ⟦ u ⟧
⟦ ƛ t ⟧ = lam ⟦ t ⟧

⟪ q ⟫ = q−λ
⟪ t [ ρ ] ⟫ = ⟪ t ⟫ [ ⟪ ρ ⟫′ ]λ
⟪ lam t ⟫ = ƛ ⟪ t ⟫
⟪ app t u ⟫ = ⟪ t ⟫ · ⟪ u ⟫
```

The structure is obviously preserved when a λ abstraction or an application is mapped from one functor to another. Therefore, the extended proofs are easy as they are merely applications of the inductive hypotheses.

```
tm−λ⇒cwf (t · u) = cong₂ _·_ (tm−λ⇒cwf t) (tm−λ⇒cwf u)
tm−λ⇒cwf (ƛ t) = cong ƛ (tm−λ⇒cwf t)

tm−cwf⇒λ (lam t) = cong−lam (tm−cwf⇒λ t)
tm−cwf⇒λ (app t u) = cong−app (tm−cwf⇒λ t) (tm−cwf⇒λ u)
```

This concludes the isomorphism between the two λ-scwf objects.

4.2.4 λ-$\beta\eta$-Scwfs

The final extensions is to present simply typed λ-calculus as λ-scwf up to beta and eta. This construct builds on the two previous by adding only the beta and eta equalities as laws. We use another Agda record to denote this.

```
record Lambda−βη−scwf : Set₁ where
  field
    lambda−scwf : Lambda−scwf
  open Lambda−scwf lambda−scwf public
  field
    − beta and eta equalities
    β : ∀ {Γ α β} {t : Tm (Γ • α) β} {u} → app (lam t) u ≈ t [ < id , u > ]
    η : ∀ {Γ α β} {t : Tm Γ (α `→ β)} → lam (app (t [ p ]) q) ≈ t
```

For the explicit λ-$\beta\eta$ ucwf object, we need only add these two equations in the equality relation for terms.

```
data _≈_ where
  ...
  β : ∀ {Γ α β} {t : Tm (Γ • α) β} {u} → app (lam t) u ≈ t [ < id , u > ]
  η : ∀ {Γ α β} {t : Tm Γ (α ⇒ β)} → lam (app (t [ p ]) q) ≈ t
```

Here, Tm refers to the actual data type and not the record field; it is part of the equality relation for the explicit scwf object with extra structure. And so by adding these equalities to the relation, that λ-scwf object becomes a λ-$\beta\eta$-scwf object directly.

On the other hand, for the concrete implementation of simply typed λ-calculus, a relation that defines beta-eta convertibility is necessary. This axiomatization needs two data types for terms and substitutions.

```
data _~βη_ {Γ} : ∀ {α} (_ _ : Tm Γ α) → Set where

  − variable reflexivity
  varcong : ∀ {α} (v : α ∈ Γ) → var v ~βη var v

  − congruence for application
```

```
    apcong : ∀ {α β} {t t′ : Tm Γ (α ⇒ β)} {u u′} →
      t ~βη t′ →
      u ~βη u′ →
      (t · u) ~βη (t′ · u)

    – the ξ rule
    ξ : ∀ {α β} {t t′ : Tm (Γ • α) β} → t ~βη t′ → ƛ t ~βη ƛ t′

    – beta and eta equalities
    β : ∀ {α β} (t : Tm (Γ • α) β) u → ƛ t · u ~βη (t [ id , u ])
    η : ∀ {α β} {t : Tm Γ (α ⇒ β)} → ƛ ((t [ p ]) · q) ~βη t

    sym~βη : ∀ {α} {t₁ t₂ : Tm Γ α} → t₁ ~βη t₂ → t₂ ~βη t₁
    trans~βη : ∀ {α} {t₁ t₂ t₃ : Tm Γ α} →
      t₁ ~βη t₂ → t₂ ~βη t₃ → t₁ ~βη t₃

data _≈βη_ {Δ} : ∀ {Γ} (_ _ : Sub Δ Γ) → Set where

    – empty substitutions are convertible
    ⋄ : {γ γ′ : Sub Δ ε} → γ ≈βη γ′

    – if terms and substitutions are convertible, so is cons
    ext : ∀ {Γ α} {t u : Tm Δ α} {γ γ′ : Sub Δ Γ} →
      t ~βη u → γ ≈βη γ′ →
      (γ , t) ≈βη (γ′ , u)
```

Reflexivity is then derived like so.

```
refl~βη : ∀ {Γ α} {t : Tm Γ α} → t ~βη t
refl~βη {t = t} = trans~βη (sym~βη (β (var here) t)) (β (var here) t)

refl≈βη : ∀ {Γ Δ} {ρ : Sub Γ Δ} → ρ ≈βη ρ
refl≈βη {Δ = ε} = ⋄
refl≈βη {Δ = Δ • x} = ext refl~βη refl≈βη
```

Naturally, every property shown before has to be proven at the level of this relation instead of propositional equality. However, the preceding work contains all the proofs as elements of identity, hence, it is easy to lift a proof in identity to a proof of beta-eta convertible by pattern matching on the proof which gives us refl.

```
≡−to~βη : ∀ {Γ α} {t u : Tm Γ α} → t ≡ u → t ~βη u
≡−to~βη refl = refl~βη

≡−to−≈βη : ∀ {Γ Δ} {ρ σ : Sub Γ Δ} → ρ ≡ σ → ρ ≈βη σ
≡−to−≈βη refl = refl≈βη
```

By the same token, all properties and inverse lemmas based on this relation can be proven by a call to the equivalent proof in identity. Congruence rules can be found in figure B.5.

4.3 Extrinsic Scwfs

Diverging somewhat from the methodology followed hitherto, a version of scwfs based on raw well-scoped terms with external typing relations is also formalized. We use the two calculi of chapter 3 as the raw basis and add typing rules. This extension involves an implementation of explicit typing rules for the scwf object with explicit substitutions. While the lambda calculus with meta-level operations requires a number of proofs that verify the preservation of types. In other words, all typing rules of the explicit scwf object have to be proven. We are basing the result on the proofs at the raw level; we have implemented proofs of ucwf axioms and the isomorphism. Thus, we need only prove the typing rules for all cwf operators to extend the result to extrinsic scwfs.

The raw terms presented in chapter 3 (definition 3.2 are reused for this endeavor, so we proceed directly with the implementation of the first extrinsic scwf object. In this section, we consider λ-scwfs directly.

A quick recap of the raw grammar of terms and substitutions as Agda data types. These are the building blocks for the typing rules that follow.

```
data RTm : Nat → Set
data RSub : Nat → Nat → Set

data RTm where
  q     : ∀ {n} → RTm (suc n)
  _[_] : ∀ {m n} (t : RTm n) (ρ : RSub m n) → RTm m
  app  : ∀ {n} (t u : RTm n) → RTm n
  lam  : ∀ {n} (t : RTm (suc n)) → RTm n

data RSub where
  <>   : ∀ {m} → RSub m zero
  id   : ∀ {n} → RSub n n
  p    : ∀ {n} → RSub (suc n) n
  <_,_> : ∀ {m n} → RSub m n → RTm m → RSub m (suc n)
  _∘_  : ∀ {m n k} → RSub n k → RSub m n → RSub m k
```

The additions include typing relations that come in two groups; one for providing types to terms and one for providing environments to substitutions.

- `data _⊢_∈_ : ∀ {n} → Ctx n → RTm n → Ty → Set`
- `data _▹_⊢_ : ∀ {m n} → Ctx m → Ctx n → RSub n m → Set`

These two relations evidently interact, but there is a clear separation which is convenient. First, the typing rules for terms.

```
data _⊢_∈_ where
  q∈ : ∀ {n α} {Γ : Ctx n} → Γ • α ⊢ q ∈ α

  sub∈ : ∀ {m n} {Γ : Ctx m} {Δ : Ctx n} {α t ρ}
    → Γ ⊢ t ∈ α
    → Γ ▹ Δ ⊢ ρ
    → Δ ⊢ t [ ρ ] ∈ α

  app∈ : ∀ {n} {Γ : Ctx n} {α β t u}
    → Γ ⊢ t ∈ (α ⇒ β)
    → Γ ⊢ u ∈ α
    → Γ ⊢ app t u ∈ β

  lam∈ : ∀ {n} {Γ : Ctx n} {α β t}
    → Γ • α ⊢ t ∈ β
    → Γ ⊢ lam t ∈ (α ⇒ β)
```

One writes Γ ⊢ t ∈ α to say that a term t has type α in the context Γ. We see one typing rule associated with each term in the language. The last variable q is typed in an extended context, the standard typing rules for app and lam and finally, the fact that substituting a well-typed term preserves the type.

```
data _▷_⊢_ where
  ⊢<> : ∀ {m} {Δ : Ctx m} → ε ▷ Δ ⊢ <>

  ⊢id : ∀ {n} {Γ : Ctx n} → Γ ▷ Γ ⊢ id

  ⊢p : ∀ {n α} {Γ : Ctx n} → Γ ▷ Γ • α ⊢ p

  ⊢∘ : ∀ {m n k} {Γ : Ctx n} {Δ : Ctx m} {Θ : Ctx k}
    {ρ : RSub n k} {σ : RSub m n}
  → Θ ▷ Γ ⊢ ρ
  → Γ ▷ Δ ⊢ σ
  → Θ ▷ Δ ⊢ ρ ∘ σ

  ⊢<,> : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m}
    {t : RTm m} {ρ : RSub m n} {α}
    → Δ ⊢ t ∈ α
    → Γ ▷ Δ ⊢ ρ
    → Γ • α ▷ Δ ⊢ < ρ , t >
```

Regarding well-typed substitutions, one needs a typing rule for each operator. One writes Γ ▷ Δ ⊢ ρ to say that substitution ρ has environment Γ and that terms in ρ are typed Δ.

The relations in which one reasons about equality remain at the raw level, so we have untyped conversion; the rules can be found in figure A.2.

These raw terms and their typing rules represent a scwf object implemented the extrinsic way. This is in contrast to the formulation of the preceding two sections where we had directly typed terms. In the case of simply typed λ-calculus, the intrinsic way does not present any problems and having an implementation language with dependent types usually favors that method as opposed to this.

Afterwards, we can build a simply typed λ-calculus based on our untyped grammar with de Bruijn indices with the purpose of constructing a second scwf object. The raw lambda terms in Agda from chapter 3.

```
data Tm (n : Nat) : Set where
  var  : (i : Fin n) → Tm n
  _·_  : (t u : Tm n) → Tm n
  ƛ    : (t : Tm (1 + n)) → Tm n
```

Substitutions as vectors have the definitions of section 3.1.2. In short, a substitution Sub n m is a vector of length n that contains terms with at most m variables. For example, id contains variables with indices 0 to $n - 1$. The length of id is implicit.

First, we define contexts as vectors too; this allows us to perform a lookup with the variable index for the typing rule.

```
data Ty : Set where
  ♭ : Ty
  _⇒_ : Ty → Ty → Ty

Ctx : Nat → Set
Ctx = Vec.Vec Ty

_•_ : ∀ {n} (Γ : Ctx n) (α : Ty) → Ctx (suc n)
Γ • α = α :: Γ
```

Subsequently, a typing relation that provides types to terms is added.

```
data _⊢_∈_ {n} (Γ : Ctx n) : Tm n → Ty → Set where
  var : ∀ {i} →  Γ ⊢ var i ∈ lookup i Γ

  ƛ : ∀ {t α β}
    → Γ • α ⊢ t ∈ β
    → Γ ⊢ ƛ t ∈ α ⇒ β

  _·_ : ∀ {t u σ τ}
    → Γ ⊢ t ∈ σ ⇒ τ
    → Γ ⊢ u ∈ σ
    → Γ ⊢ t · u ∈ τ
```

Variables are de Bruijn style, thus a lookup is performed in the context; the other two rules are identical to the explicit scwf as expected. Naturally, there is no mention of substitution here as it is a meta level operation, so one has to prove that substitution is preserved after.

The relation for well-typed substitutions has two constructors, one describing the empty environment and one for inserting a well-typed term into a substitution. These are the tools one must use to prove the type preservation properties for the remaining scwf operators like composition and projection.

```
data _▷_⊢_ {n} : ∀ {m} → Ctx m → Ctx n → Sub n m → Set where
  [] : ∀ {Δ} → [] ▷ Δ ⊢ []
```

```
    ext : ∀ {m} {Γ : Ctx m} {Δ σ t ρ}
      → Δ ⊢ t ∈ σ
      → Γ ▹ Δ ⊢ ρ
      → Γ • σ ▹ Δ ⊢ ρ , t
```

Afterwards, the remaining typing rules have to proven for this calculus. What is left is to show that identity, projection, composition and substitution preserve types. Before we start presenting the aforementioned lemmas, we quickly mention some fundamental supporting properties that were necessary in the proofs.

```
weaken−preserv : ∀ {n Γ α β} {t : Tm n}
  → Γ ⊢ t ∈ β
  → Γ • α ⊢ weaken t ∈ β

map−weaken−preserv : ∀ {m n Γ Δ α} {ρ : Sub m n}
      → Γ ▹ Δ ⊢ ρ
      → Γ ▹ Δ • α ⊢ Vec.map weaken ρ

↑−preserv : ∀ {m n Γ Δ α} {ρ : Sub m n}
  → Γ ▹ Δ ⊢ ρ
  → Γ • α ▹ Δ • α ⊢ ↑ ρ

lookup−preserv : ∀ {m n} {Γ : Ctx m}
      {Δ : Ctx n} i {ρ}
  → Γ ▹ Δ ⊢ ρ
  → Δ ⊢ lookup i ρ ∈ lookup i Γ
```

The proofs of these properties are in appendix ??.

Lemma 4.16. Γ ▹ Γ ⊢ id

Proof. Induction on Γ.

```
id−preserv : ∀ {n} {Γ : Ctx n} → Γ ▹ Γ ⊢ id
id−preserv {Γ = []}  = []
id−preserv {Γ = _ :: _} = ↑−preserv id−preserv
```

∎

Lemma 4.17. Γ ▹ Γ • α ⊢ p

Proof. Induction on Γ.

```
p−preserv : ∀ {n α} {Γ : Ctx n} → Γ ▹ Γ • α ⊢ p
p−preserv {Γ = []}  = []
p−preserv {Γ = _ :: Γ} = map−weaken−preserv $ ↑−preserv id−preserv
```

∎

Lemma 4.18. Γ ⊢ t ∈ α → Γ ▹ Δ ⊢ ρ → Δ ⊢ t [ρ] ∈ α

Proof. Induction on the typing derivation of t .

```
subst−lemma : ∀ {m n} {Γ : Ctx m} {Δ : Ctx n} {α t ρ}
  → Γ ⊢ t ∈ α
  → Γ ▷ Δ ⊢ ρ
  → Δ ⊢ t [ ρ ] ∈ α
subst−lemma (var {i}) ⊢ρ = lookup−preserv i ⊢ρ
subst−lemma (ƛ t)   ⊢ρ = ƛ (subst−lemma t (↑−preserv ⊢ρ))
subst−lemma (t · u)  ⊢ρ = subst−lemma t ⊢ρ · subst−lemma u ⊢ρ
```

■

Lemma 4.19. $\Theta \triangleright \Gamma \vdash \rho \to \Gamma \triangleright \Delta \vdash \sigma \to \Theta \triangleright \Delta \vdash \rho \circ \sigma$

Proof. Induction on the typing derivation of ρ. Note the application of the previous lemma in the inductive case.

```
∘−preserv : ∀ {m n k} {Γ : Ctx n} {Δ : Ctx m} {Θ : Ctx k}
    {ρ : Sub n k} {σ : Sub m n}
  → Θ ▷ Γ ⊢ ρ
  → Γ ▷ Δ ⊢ σ
  → Θ ▷ Δ ⊢ ρ ∘ σ
∘−preserv [] _  = []
∘−preserv (ext x ⊢ρ) ⊢σ =
  ext (subst−lemma x ⊢σ)
    (∘−preserv ⊢ρ ⊢σ)
```

■

Consequently, we have shown that all typing rules of the explicit scwf object also hold for these lambda terms with de Bruijn variables. This means that the latter forms an extrinsic scwf object. Moreover, we simultaneously extended the isomorphism proof. Considering the fact that we have a proof at the raw level, it is enough to demonstrate the typing rules.

Finally, to officially present this as an extrinsic scwf object, the raw terms and substitutions have to be placed as pairs with their typing rule; for this reason, dependent sigma pairs are employed. Here are the pairs of composition, substitution, identity, and projection.

```
∘−ty : ∀ {m n k} {Γ : Ctx n} {Δ : Ctx m} {Θ : Ctx k}
  → Σ (Sub n k) (Θ ▷ Γ ⊢_)
  → Σ (Sub m n) (Γ ▷ Δ ⊢_)
  → Σ (Sub m k) (Θ ▷ Δ ⊢_)
∘−ty (ρ Σ., ⊢ρ) (σ Σ., ⊢σ) = ρ ∘ σ Σ., ∘−preserv ⊢ρ ⊢σ

sub−ty : ∀ {m n α} {Δ : Ctx m} {Γ : Ctx n}
    → Σ (Tm n) (Γ ⊢_∈ α)
    → Σ (Sub m n) (Γ ▷ Δ ⊢_)
    → Σ (Tm m) (Δ ⊢_∈ α)
sub−ty (t Σ., t∈) (ρ Σ., ⊢ρ) = t [ ρ ] Σ., subst−lemma t∈ ⊢ρ

id−ty : ∀ {n} {Γ : Ctx n} → Σ (Sub n n) (Γ ▷ Γ ⊢_)
id−ty = id′ Σ., id−preserv
```

```
p−ty : ∀ {n α} {Γ : Ctx n} → Σ (Sub (suc n) n) (Γ ▹ Γ • α ⊢_)
p−ty = p Σ., p−preserv
```

The other operators such as the empty substitution, extending a substitution and so on are provided by the typing relations defined earlier. The remaining pairs are in figure ?? of the appendix.

Conclusively, this demonstrates the extrinsic point of view of simple types for cwfs and lambda calculus. In this instance, the explicit constructors have corresponding explicit typing rules.

We reflect on this formulation against the intrinsic way previously shown by noting some vital differences. The most substantial difference is that the extrinsic formulation keeps untyped conversion and builds on-top of the work conducted for the raw grammar. This allows one to utilize useful results without the need to prove them again. On the other hand, performing the same proofs in an intrinsic formulation does not add significant burdens or challenges; thus, carrying type information around does not affect the reasoning, which is was quite pleasant. In fact, the top level proofs were identical to the untyped ones. One can also say that simply typed cwfs and lambda calculus are quite simple and either way is manageable insofar as we talk about scwfs with extra structure.

4.3.1 Extrinsic and Intrinsic Scwfs

This subsection summarizes another important theorem in the simply typed world, namely, that the intrinsic and extrinsic explicit scwf objects are isomorphic. Thus far, the focal point of the thesis was the connection between explicit cwf combinator languages and conventional lambda calculi with meta operations. Now, however, the connection between two scwf combinator objects is investigated. Fortunately, this is a much easier task since everything is explicitly defined beforehand in the language.

Specifically, we use the raw calculus of explicit substitutions with the typing relations shown in this section and the intrinsic calculus of explicit substitutions that have types built in.

First, one has to define the translation functions between these scwf objects. They consist of

- a map that takes a directly typed term and strips the type information away;
- a map that given a directly typed term returns the typing derivation using the stripped term;
- and a map that takes a raw term and its typing rule and produces a directly typed term.

Naturally, the equivalent translation functions between substitutions are necessary. The raw terms and substitutions have an R as a prefix, while Tm and Sub represent the intrinsically typed versions.

```
– Strips types from a typed term back to a raw term
strip : ∀ {n} {Γ : Ctx n} {α} → Tm Γ α → RTm n

– Strip for substitutions
```

```
strip▷ : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} → Sub Δ Γ → RSub n m

– Given a typed term, returns an element of the typing relation on the raw term
typing  : ∀ {n} {Γ : Ctx n} {α} (t : Tm Γ α) → Γ ⊢ strip t ∈ α

– Typing for substitutions
typing▷ : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m}
  (ρ : Sub Δ Γ) → Δ ▷ Γ ⊢ strip▷ ρ
```

Further, one sees that using the extrinsic formulation, one needs two maps to talk about raw terms and their typing derivations. And these two combined form the dependent pairs, as shown earlier, but for this explicit language in this case. Lastly, we need to join a raw term and its typing rule to produce a directly typed term.

```
– Raw term and typing rule give a directly typed term
join  : ∀ {n} {Γ : Ctx n} {α} → Σ (RTm n) (Γ ⊢_∈ α) → Tm Γ α

– Join for substitutions
join▷ : ∀ {m n} {Γ : Ctx m} {Δ : Ctx n}
  → Σ (RSub m n) (Δ ▷ Γ ⊢_) → Sub Δ Γ
```

These maps are very straightforward to define and they have a nice recursive structure. We show only the term maps.

```
strip q = q
strip (t [ ρ ]) = strip t [ strip▷ ρ ]
strip (app t u) = app (strip t) (strip u)
strip (lam t) = lam (strip t)

typing q = q∈
typing (t [ ρ ]) = sub∈ (typing t) (typing▷ ρ)
typing (app t u) = app∈ (typing t) (typing u)
typing (lam t) = lam∈ (typing t)

join (q , q∈) = q
join (t [ ρ ] , sub∈ t∈ ⊢ρ) = join (t , t∈) [ join▷ (ρ , ⊢ρ) ]
join (app t u , app∈ t∈ u∈) = app (join (t , t∈)) (join (u , u∈))
join (lam t , lam∈ t∈) = lam (join (t , t∈))
```

Subsequently, it is just as facile to prove that these are inverses of each other and hence show that these scwf objects are isomorphic. We need the following lemmas.

```
joinstrip▷ : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} (ρ : Sub Δ Γ) →
      join▷ (strip▷ ρ , typing▷ ρ) ≋ ρ

joinstrip : ∀ {n α} {Γ : Ctx n} (t : Tm Γ α) →
   join (strip t , typing t) ≈ t
```

```
stripjoin▷ : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} (ρ : RSub n m) (⊢ρ : Δ ▷ Γ ⊢ ρ) →
      strip▷ (join▷ (ρ , ⊢ρ)) ≋′ ρ

stripjoin : ∀ {n α} {Γ : Ctx n} (t : RTm n) (t∈ : Γ ⊢ t ∈ α) →
    strip (join (t , t∈)) ≈′ t
```

Equality is defined by the explicit relations that contain cwf laws as introduction rules. Here the equalities ≈′ and ≋′ refer to the raw level relations for terms and substitutions while the other non-primed to the typed equations.

Lemma 4.20. join▷ (strip▷ ρ , typing▷ ρ) ≋ ρ *and* join (strip t , typing t) ≈ t

Proof. Induction on ρ and t and it follows trivially. ■

Lemma 4.21. strip▷ (join▷ (ρ , ⊢ρ)) ≋′ ρ *and* strip (join (t , t∈)) ≈′ t

Proof. Induction on ρ and t and their typing rules and it follows trivially. ■

Conclusively, all these different formulations we have seen for simple types, i.e., intrinsic scwf calculus with explicit substitutions, intrinsic scwf as simply typed λ-calculus with meta-level substitutions and the equivalent extrinsic versions are all isomorphic.

As a summarizing note of the whole chapter, we describe the content and implications. First, an implementation of intrinsically typed scwf objects, one with explicit constructors and one traditional lambda calculus were formalized. A proof of the latter being a scwf object was implemented and subsequently, morphisms between these two objects were defined and proven to be strict inverses.

Further, an alternative way of implementing simple types was explored, namely, keeping raw terms and adding typing relations on-top to provide types to terms and substitutions. This kind of description was called extrinsic, since the types are not directly built-in, that is, raw terms remain the basis of the implementation. Following this structure, it was necessary to define explicitly typing relations for the straightforward scwf object, i.e., a typing rule for each cwf combinator. The final point was to show that types are preserved for all cwf operators in order to complete the proof that the extrinsic simply typed lambda calculus is a scwf object and, in extension, isomorphic to the one with the combinators.

Lastly, a comparison of explicit scwf formulations closed this chapter. It described the same connection, but this time between the directly typed (intrinsic) and extrinsic scwf combinator objects. These are solely a direct implementation of the generalized algebraic theory as a calculus. Hence, all sides of figure 4.1 were covered.

Chapter 5

ΠU-CwFs

In this chapter, a version of the complete categories with families is formalized. This means that we will be dealing with a dependently typed calculus which is what Cwfs model to begin with. The formulation almost has be extrinsically typed since in a dependently typed setting it is not clear how to formalize directly typed terms that depend on terms. There are numerous obstacles that are quite challenging to handle and would have impeded progress substantially.

Our language will consider a type former, namely Π and a universe of small types à la Russel. This yields what will be referred to as a ΠU-Cwf. An extrinsic ΠU-Cwf utilizes a raw grammar with the appropriate typing rules and the subsequent sections construct two such objects: one with explicit substitutions and cwf combinators and a typical lambda calculus. For the latter, we employ scope safe terms again.

5.1 Combinator ΠU-Cwf Object

To remain consistent with the order of concepts presented, we start with the formalization of the object with explicit cwf combinators in accordance to the generalized algebraic theory. As usual, it consists of a calculus of explicit substitutions, but this time with Π types and U as extra structure.

Before any typing rules are added, we will examine the new raw grammar since there are significant additions. An important new feature is that terms and types are collapsed under one set because types may now contain terms. It is effectively an extension of the unityped λ-ucwf. Therefore, the updated raw syntax for the explicit object will look as follows.

```
data Tm  : Nat → Set
data Sub : Nat → Nat → Set

data Tm where
  q     : ∀ {n} → Tm (1 + n)
  _[_] : ∀ {m n} → Tm n → Sub m n → Tm m
  lam   : ∀ {n} → Tm (1 + n) → Tm n
  app   : ∀ {n} → Tm n → Tm n → Tm n

  – Π types and universe
  Π     : ∀ {n} → Tm n → Tm (1 + n) → Tm n
```

```
    U    : ∀ {n} → Tm n

  data Sub where
    id   : ∀ {n} → Sub n n
    _∘_  : ∀ {m n k} → Sub n k → Sub m n → Sub m k
    p    : ∀ {n} → Sub (1 + n) n
    <>   : ∀ {m} → Sub m 0
    <_,_> : ∀ {m n} → Sub m n → Tm m → Sub m (1 + n)
```

There are two new constructors to create elements of Tm, i.e., Π and U. Substitutions are the same as in every preceding chapter.

Naturally, we need to define equalities for these two data types. Two mutually recursive relations are defined just like in figure A.2. We show only the new equations that express stability under substitution and a congruence on Π.

```
  data _≈_ : ∀ {n} (t t′ : Tm n) → Set
  data _≋_ : ∀ {m n} (γ γ′ : Sub m n) → Set

  data _≈_ where
    ...
    subΠ     : ∀ {n m} (γ : Sub m n) A B →
      (Π A B) [ γ ] ≈ Π (A [ γ ]) (B [ < γ ∘ p , q > ])
    subU     : ∀ {n m} {γ : Sub m n} → U [ γ ] ≈ U
    cong−Π  : ∀ {n} {A A′ : Tm n} {B B′} →
      A ≈ A′ →
      B ≈ B′ →
      Π A B ≈ Π A′ B′
```

As we can see, conversion is at the untyped level between raw terms. An alternative formulation is to define explicit equality judgements along with the typing judgements. That means that equality between raw terms would be expressed with type information present.

Subsequently, we proceed to define the type system. First, contexts which just contain terms.

```
  data Ctx where
    ⋄  : Ctx 0
    _•_ : ∀ {n} → Ctx n → Tm n → Ctx (1 + n)
```

Next, we show the typing rules. In this theory there are four primary judgements one makes. These are defined as mutually recursive data types in Agda where all rules are constructors. Moreover, since this is the calculus of explicit substitutions, each operator has its typing rule defined a priori. In the simply typed formulations, there were two typing relations: one for giving types to terms and one for giving environments to substitutions. In dependent type theories, we need to talk about contexts being well-formed as well as types themselves. However, since terms and types are under one set, both rules include solely contexts and terms, but there are conceptual differences.

- data _⊢ : ∀ {n} (Γ : Ctx n) → Set ; well-formed contexts.
- data _⊢_ : ∀ {n} (Γ : Ctx n) (A : Tm n) → Set ; well-formed types in a context.
- data _⊢_∈_ : ∀ {n} (Γ : Ctx n) (t : Tm n) (A : Tm n) → Set ; well-formed terms of some type.
- data _▷_⊢_ : ∀ {m n} (Δ : Ctx m) (Γ : Ctx n) (γ : Sub m n) → Set ; well-formed substitutions in two contexts.

First, we give the typing rules for contexts.

```
data _⊢ where
  c−emp : ⋄ ⊢

  c−ext : ∀ {n} {Γ : Ctx n} {A}
    → Γ ⊢
    → Γ ⊢ A
    → Γ • A ⊢
```

Rules for well-formed types with the usual rules for a universe à la Russel.

```
data _⊢_ where
  ty−U : ∀ {n} {Γ : Ctx n}
    → Γ ⊢
    → Γ ⊢ U

  ty−∈U : ∀ {n} {Γ : Ctx n} {A}
    → Γ ⊢ A ∈ U
    → Γ ⊢ A

  ty−sub : ∀ {m n} {Δ : Ctx m} {Γ : Ctx n} {A γ}
    → Δ ⊢ A
    → Γ ▷ Δ ⊢ γ
    → Γ ⊢ A [ γ ]

  ty−Π−F : ∀ {n} {Γ : Ctx n} {A B}
    → Γ ⊢ A
    → Γ • A ⊢ B
    → Γ ⊢ Π A B
```

Now there is a substitution for types, albeit, it is the same operation since they are in the same set.

Rules for well-formed terms.

```
data _⊢_∈_ where
  tm−q : ∀ {n} {Γ : Ctx n} {A}
    → Γ ⊢ A
    → Γ • A ⊢ q ∈ A [ p ]
  tm−sub : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m}
    {A t γ}
    → Δ ⊢ t ∈ A
    → Γ ▷ Δ ⊢ γ
    → Γ ⊢ t [ γ ] ∈ A [ γ ]

  tm−app : ∀ {n} {Γ : Ctx n} {f t A B}
    → Γ ⊢ A
    → Γ • A ⊢ B
    → Γ ⊢ f ∈ Π A B
    → Γ ⊢ t ∈ A
    → Γ ⊢ app f t ∈ B [ < id , t > ]

  tm−Π−I : ∀ {n} {Γ : Ctx n} {A B t}
    → Γ ⊢ A
    → Γ • A ⊢ B
    → Γ • A ⊢ t ∈ B
    → Γ ⊢ lam t ∈ Π A B

  tm−conv : ∀ {n} {Γ : Ctx n} {t A A′}
    → Γ ⊢ A′
    → Γ ⊢ t ∈ A
    → A′ ≈ A
    → Γ ⊢ t ∈ A′
```

In the case of q, we see that the type has to be lifted, which is another example how having scope safe terms prevent us from forgetting such things. In the last rule, tm−conv, we see the use of the equality of between raw terms, so as previously stated, there are no equality judgements in this calculus.

Lastly, we give the rules for well-formed substitutions.

```
data _▷_⊢_ where
  ⊢id : ∀ {n} {Γ : Ctx n}
    → Γ ⊢
    → Γ ▷ Γ ⊢ id

  ⊢∘ : ∀ {m n k} {Γ : Ctx n} {Δ : Ctx m}
    {Θ : Ctx k} {γ₁ γ₂}
    → Γ ▷ Θ ⊢ γ₁
    → Δ ▷ Γ ⊢ γ₂
    → Δ ▷ Θ ⊢ γ₁ ∘ γ₂

  ⊢p : ∀ {n} {Γ : Ctx n} {A}
    → Γ ⊢ A
    → Γ • A ▷ Γ ⊢ p

  ⊢<> : ∀ {n} {Γ : Ctx n}
```

```
    → Γ ⊢
    → Γ ▷ ⋄ ⊢ <>

  ⊢<,> : ∀ {m n} {Γ : Ctx n}
    {Δ : Ctx m} {A t γ}
    → Γ ▷ Δ ⊢ γ
    → Δ ⊢ A
    → Γ ⊢ t ∈ A [ γ ]
    → Γ ▷ Δ • A ⊢ < γ , t >
```

These rules are quite similar to the ones for simply typed scwfs, but with more assumptions. For instance, the rule for id requires the context to be well-formed. This notion did not exist in the simply typed world. By the same token, the rule for p needs the type to be well-formed, but not the context explicitly since this can be derived. Specifically, we use the minimum number of assumptions in order to derive some fundamental laws for this calculus.

When a term t has type A in context Γ, this fact only makes sense if Γ is a well-formed context and A is a correct type in Γ. Hence, the following properties hold for this calculus.

- ∀ {n} {Γ : Ctx n} {A} → Γ ⊢ A → Γ ⊢
- ∀ {n} {Γ : Ctx n} {A t} → Γ ⊢ t ∈ A → Γ ⊢
- ∀ {n} {Γ : Ctx n} {A t} → Γ ⊢ t ∈ A → Γ ⊢ A
- ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} {γ} → Δ ▷ Γ ⊢ γ → Γ ⊢ × Δ ⊢

These are proven mutually by induction on the derivation and by using only rules of the calculus (proofs in figure C.1).

Overall, this description defines a ΠU-cwf object by construction using explicit cwf combinators. It is definitely not clear-cut to say that this is the initial cwf object with extra structure, albeit it should be a good candidate. How to demonstrate that fact in type theory is also not clear. – any important points on this? I'm not sure.

5.2 λΠU-calculus

In this section we construct a dependently typed λ-calculus with substitution formalized in the meta-language. The goal is to construct another ΠU-cwf object starting with raw terms and building a type system on-top. This is essentially an extension of the unityped grammar that should now also accommodate Π types and a universe. The terms are well-scoped, that is, they are indexed by the maximum number of free variables they may contain. Substitutions are vectors as shown in chapter 3. Practically, we have to redefine operations that act differently depending on the given term. The raw substitutions, for example, have the exact same formulation as chapter in 3. On the other hand, the substitution operation and others that are defined by recursion on terms need to handle Π and U, but they are easy to define given the current framework.

Firstly, we show the new language with terms and types under one set.

```
data Tm (n : Nat) : Set where
  var : (i : Fin n) → Tm n
  ƛ   : Tm (suc n) → Tm n
  _·_ : Tm n → Tm n → Tm n
  Π   : Tm n → Tm (suc n) → Tm n
  U   : Tm n

q : ∀ {n} → Tm (suc n)
q = var zero
```

Afterwards, we need to define renaming and then substitution. Substitutions as vectors use the definitions from section 3.1.1.

```
– Renamings (substitutions for variables)
Ren : Nat → Nat → Set
Ren m n = Vec (Fin m) n

– Substitutions
Sub : Nat → Nat → Set
Sub m n = Vec (Tm m) n

– Lifting a renaming
↑−ren : ∀ {m n} → Ren m n → Ren (suc m) (suc n)
↑−ren ρ = zero :: map suc ρ

– Renaming operation
ren : ∀ {m n} → Tm n → Ren m n → Tm m
ren (var i) ρ = var (lookup i ρ)
ren (ƛ t)  ρ = ƛ (ren t (↑−ren ρ))
ren (t · u) ρ = (ren t ρ) · (ren u ρ)
ren (Π A B) ρ = Π (ren A ρ) (ren B (↑−ren ρ))
ren U      _ = U

– Weakening a term
wk : ∀ {n} → Tm n → Tm (suc n)
wk t = ren t (tabulate suc)

– Weakening a substitution
wk−sub : ∀ {m n} → Sub m n → Sub (suc m) n
wk−sub = map wk

– Lifting and extending a substitution
↑_ : ∀ {m n} (ρ : Sub m n) → Sub (suc m) (suc n)
↑ ρ = wk−sub ρ , q

– Substitution
_[_] : ∀ {m n} (t : Tm n) (ρ : Sub m n) → Tm m
```

```
var i     [ ρ ] = lookup i ρ
(t · u) [ ρ ] = (t [ ρ ]) · (u [ ρ ])
ƛ t       [ ρ ] = ƛ (t [ ↑ ρ ])
Π A B  [ ρ ] = Π (A [ ρ ]) (B [ ↑ ρ ])
U         [ _ ] = U
```

As we can see, in the case of U, no computation takes place, whereas for Π there are binders and therefore the substitution has to be extended and lifted.

Next, we quickly present two cwf laws that should hold for this extended grammar. An advantage is that properties proven for the unityped cwfs can be reused here.

1. Substituting in id returns the same term.

2. Associativity of substitution.

Lemma 5.1. t [id] ≡ t

Proof. Induction on t.

```
subId : ∀ {n} (t : Tm n) → t [ id ] ≡ t
subId (var i) = lookup−id i
subId (ƛ t) = cong ƛ (trans (cong (t [_]) lift−idExt) (subId t))
subId (t · u) = cong₂ _·_ (subId t) (subId u)
subId (Π A B) = cong₂ Π (subId A) (trans (cong (B [_]) lift−idExt) (subId B))
subId U = refl
```

■

Lemma 5.2. t [ρ ∘ σ] ≡ t [ρ] [σ]

Proof. Induction on t.

```
subComp : ∀ {m n k} t (ρ : Sub m n) (σ : Sub k m)
      → t [ ρ ∘ σ ] ≡ t [ ρ ] [ σ ]
subComp U _ _ = refl
subComp (var zero) (x ∷ ρ) σ = refl
subComp (var (suc i)) (x ∷ ρ) σ = subComp (var i) ρ σ
subComp (ƛ t)  ρ σ =
   cong ƛ (trans (cong (t [_]) (↑−dist ρ σ)) (subComp t (↑ ρ) (↑ σ)))
subComp (t · u) ρ σ = cong₂ _·_ (subComp t ρ σ) (subComp u ρ σ)
subComp (Π A B) ρ σ = begin
   Π (A [ ρ ∘ σ ])  (B [ ↑ (ρ ∘ σ) ])
      ≡⟨ cong (λ x → Π x _) (subComp A ρ σ) ⟩
   Π (A [ ρ ] [ σ ]) (B [ ↑ (ρ ∘ σ) ])
      ≡⟨ cong (λ x → Π _ x) (cong (B [_]) (↑−dist ρ σ)) ⟩
   Π (A [ ρ ] [ σ ]) (B [ ↑ ρ ∘ ↑ σ ])
      ≡⟨ cong (λ x → Π _ x) (subComp B (↑ ρ) (↑ σ)) ⟩
   Π (A [ ρ ] [ σ ]) (B [ ↑ ρ ] [ ↑ σ ])
      ∎
```

■

These are two cwf laws and we showed how these are extended to cover Π and U.

Considering the fact that we are working with an extrinsically typed cwf, we need an isomorphism between the cwf combinator calculus, first at the raw level. Thus it is required to show this language satisfies the untyped axioms and then proceed to extend the untyped morphisms and inverse proofs. The work conducted thus far has demonstrated a large portion of this isomorphism. The lemmas that are proven by induction on Tm need two extra cases for type former and universe. They are merely tedious calculations, so the extended isomorphism at the raw level is in appendix C.2.

Subsequently, we add a type system for this calculus. The difference between the combinator one and this is that rules about meta-level operations are not constructors in the rules themselves, but they have to be proven afterwards. We start by defining contexts.

```
data Ctx where
  ⋄  : Ctx 0
  _•_ : ∀ {n} → Ctx n → Tm n → Ctx (1 + n)

lookup−ct : ∀ {n} (i : Fin n) (Γ : Ctx n) → Tm n
lookup−ct zero  (Γ • A) = wk A
lookup−ct (suc i) (Γ • _) = wk $ lookup−ct i Γ
```

Contexts are the same set, but since our language uses de Bruijn indices, a lookup function is necessary. Note how the term is weakened in the lookup.

Continuing, we have the same four judgements expressing that contexts, types, terms, and substitutions are well-formed.

```
data _⊢    : ∀ {n} (Γ : Ctx n) → Set

data _⊢_  : ∀ {n} (Γ : Ctx n) (A : Tm n) → Set

data _⊢_∈_ : ∀ {n} (Γ : Ctx n) (t : Tm n) (A : Tm n) → Set

data _▹_⊢_ : ∀ {m n} (Δ : Ctx m) (Γ : Ctx n) (γ : Sub m n) → Set

data _⊢ where
  c−emp : ⋄ ⊢

  c−ext : ∀ {n} {Γ : Ctx n} {A}
    → Γ ⊢
    → Γ ⊢ A
    → Γ • A ⊢

data _⊢_ where
  ty−U : ∀ {n} {Γ : Ctx n}
    → Γ ⊢
    → Γ ⊢ U

  ty−∈U : ∀ {n} {Γ : Ctx n} {A}
```

```
    → Γ ⊢
    → Γ ⊢ A ∈ U
    → Γ ⊢ A

  ty−Π−F : ∀ {n} {Γ : Ctx n} {A B}
          → Γ ⊢ A
          → Γ • A ⊢ B
          → Γ ⊢ Π A B

data _⊢_∈_ where
  tm−var : ∀ {n} {i : Fin n} {Γ : Ctx n}
          → Γ ⊢
          → Γ ⊢ var i ∈ lookup−ct i Γ

  tm−app : ∀ {n} {Γ : Ctx n} {f t A B}
          → Γ ⊢ A
          → Γ • A ⊢ B
          → Γ ⊢ f ∈ Π A B
          → Γ ⊢ t ∈ A
          → Γ ⊢ f · t ∈ B [ id , t ]

  tm−Π−I : ∀ {n} {Γ : Ctx n} {A B t}
          → Γ ⊢ A
          → Γ • A ⊢ B
          → Γ • A ⊢ t ∈ B
          → Γ ⊢ ƛ t ∈ Π A B

data _▷_⊢_ where
  ⊢<> : ∀ {n} {Γ : Ctx n}
    → Γ ⊢
    → Γ ▷ ◇ ⊢ []

  ⊢<,> : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} {A t γ}
    → Γ ▷ Δ ⊢ γ
    → Δ ⊢ A
    → Γ ⊢ t ∈ A [ γ ]
    → Γ ▷ Δ • A ⊢ (γ , t)
```

Here I would sketch the typing rule proofs...

Bibliography

[1] B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's Type Theory: An Introduction.* New York, NY, USA: Clarendon Press, 1990.

[2] J. Lambek and P. J. Scott, *Introduction to Higher Order Categorical Logic.* New York, NY, USA: Cambridge University Press, 1986.

[3] P. Clairambault, "From categories with families to locally cartesian closed categories," tech. rep., ENS Lyon, 2006.

[4] P. Dybjer, "Internal type theory," in *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers* (S. Berardi and M. Coppo, eds.), vol. 1158 of *Lecture Notes in Computer Science*, pp. 120–134, Springer, 1995.

[5] M. Hofmann, *Syntax and semantics of dependent types*, pp. 13–54. London: Springer London, 1997.

[6] S. Castellan, P. Clairambault, and P. Dybjer, "Undecidability of equality in the free locally cartesian closed category," in *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, pp. 138–152, 2015.

[7] B. Ahrens, P. L. Lumsdaine, and V. Voevodsky, "Categorical Structures for Type Theory in Univalent Foundations," in *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)* (V. Goranko and M. Dam, eds.), vol. 82 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 8:1–8:16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[8] P. Dybjer, "Inductive families," *Formal Aspects of Computing*, vol. 6, pp. 440–465, Jul 1994.

[9] U. Norell, *Towards a practical programming language based on dependent type theory.* Doktorsavhandlingar vid Chalmers tekniska högskola. Ny serie, no: 2677. Technical report D - Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, no: 33, Institutionen för data- och informationsteknik, Datavetenskap, Programmeringslogik (Chalmers), Chalmers tekniska högskola,, 2007. 166.

[10] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy, "Explicit substitutions," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, (New York, NY, USA), pp. 31–46, ACM, 1990.

Appendix A

Untyped

A.1 Ucwf Equations

The relations that provide the equational theory for term and substitution equality for the explicit ucwf and λ-ucwf objects.

```
data _≈_ where
  subId : ∀ {n} (t : Tm n) → t [ id ] ≈ t
  qCons : ∀ {m n} (ts : Sub n m) t → q [ < ts , t > ] ≈ t
  subComp : ∀ {m n k} (ts : Sub k n) (us : Sub m k) t →
    t [ ts ∘ us ] ≈ t [ ts ] [ us ]
  cong−sub : ∀ {m n} {t u : Tm n} {ts us : Sub m n} →
    t ≈ u → ts ≋ us → t [ ts ] ≈ u [ us ]
  sym≈ : ∀ {n} {u u′ : Tm n} → u ≈ u′ → u′ ≈ u
  trans≈ : ∀ {m} {t u v : Tm m} → t ≈ u → u ≈ v → t ≈ v

data _≋_ where
  id₀ : id {0} ≋ <>
  <>Lzero : ∀ {m n} (ts : Sub m n) → <> ∘ ts ≋ <>
  idExt : ∀ {n} → id {suc n} ≋ < p , q >
  idL : ∀ {m n} (ts : Sub m n) → id ∘ ts ≋ ts
  idR : ∀ {m n} (ts : Sub m n) → ts ∘ id ≋ ts
  assoc : ∀ {m n k p} (ts : Sub n k) (us : Sub m n) (vs : Sub p m) →
    (ts ∘ us) ∘ vs ≋ ts ∘ (us ∘ vs)
  pCons : ∀ {m n} (us : Sub m n) u → p ∘ < us , u > ≋ us
  compExt : ∀ {m n k} (ts : Sub n k) (us : Sub m n) t →
    < ts , t > ∘ us  ≋ < ts ∘ us , t [ us ] >
  cong−<,> : ∀ {m n} {ts us : Sub m n} {t u} →
    t ≈ u → ts ≋ us → < ts , t > ≋ < us , u >
  cong−∘ : ∀ {m n k} {ts vs : Sub n k} {us zs : Sub m n} →
    ts ≋ vs → us ≋ zs → ts ∘ us ≋ vs ∘ zs
  sym≋ : ∀ {m n} {h : Sub m n} {t : Sub m n} → h ≋ t → t ≋ h
  trans≋ : ∀ {m n} {h t v : Sub m n} → h ≋ t → t ≋ v → h ≋ v
```

Figure A.1: Explicit ucwf object equality relations.

```
data _≈_ where
  subId : ∀ {n} (t : Tm n) → t [ id ] ≈ t
  qCons : ∀ {m n} (ts : Sub n m) t → q [ < ts , t > ] ≈ t
  subComp : ∀ {m n k} (ts : Sub k n) (us : Sub m k) t →
    t [ ts ∘ us ] ≈ t [ ts ] [ us ]
  subApp : ∀ {n m} (ts : Sub m n) t u →
    app (t [ ts ]) (u [ ts ]) ≈ app t u [ ts ]
  subLam : ∀ {n m} (ts : Sub m n) t →
    lam t [ ts ] ≈ lam (t [ ⇑ ts ])
  cong−app : ∀ {n} {t u t′ u′ : Tm n} →
    t ≈ t′ → u ≈ u′ → app t u ≈ app t′ u′
  cong−sub : ∀ {m n} {t u : Tm n} {ts us : Sub m n} →
    t ≈ u → ts ≋ us → t [ ts ] ≈ u [ us ]
  cong−lam : ∀ {n} {t u : Tm (1 + n)} →
    t ≈ u → lam t ≈ lam u
  sym≈ : ∀ {n} {u u′ : Tm n} → u ≈ u′ → u′ ≈ u
  trans≈ : ∀ {m} {t u v : Tm m} → t ≈ u → u ≈ v → t ≈ v

data _≋_ where
  id₀ : id {0} ≋ <>
  <>Lzero : ∀ {m n} (ts : Sub m n) → <> ∘ ts ≋ <>
  idExt : ∀ {n} → id {suc n} ≋ < p , q >
  idL : ∀ {m n} (ts : Sub m n) → id ∘ ts ≋ ts
  idR : ∀ {m n} (ts : Sub m n) → ts ∘ id ≋ ts
  assoc : ∀ {m n k p} (ts : Sub n k) (us : Sub m n) (vs : Sub p m) →
    (ts ∘ us) ∘ vs ≋ ts ∘ (us ∘ vs)
  pCons : ∀ {m n} (us : Sub m n) u → p ∘ < us , u > ≋ us
  compExt : ∀ {m n k} (ts : Sub n k) (us : Sub m n) t →
    < ts , t > ∘ us  ≋ < ts ∘ us , t [ us ] >
  cong−<, > : ∀ {m n} {ts us : Sub m n} {t u} →
    t ≈ u → ts ≋ us → < ts , t > ≋ < us , u >
  cong−∘ : ∀ {m n k} {ts vs : Sub n k} {us zs : Sub m n} →
    ts ≋ vs → us ≋ zs → ts ∘ us ≋ vs ∘ zs
  sym≋ : ∀ {m n} {h g : Sub m n} → h ≋ g → g ≋ h
  trans≋ : ∀ {m n} {h t v : Sub m n} → h ≋ t → t ≋ v → h ≋ v
```

Figure A.2: Explicit λ-ucwf object equality relations.

A.2 Properties

```
↑−dist : ∀ {m n k} (ts : Sub m n) (us : Sub k m) → ↑ (ts ∘ us) ≡ (↑ ts) ∘ (↑ us)
↑−dist ts us = begin
  ↑ (ts ∘ us)
    ≡⟨⟩
  map (λ t → ren t pR) (map (_[ us ]) ts) , q
    ≡⟨ cong (_, q) (sym (map−∘ _ _ ts)) ⟩
  map (λ t → ren (t [ us ]) pR) ts , q
```

```
    ≡⟨ cong (_, q) (map−cong (sym F.∘ ∘r−asso us pR) ts) ⟩
  map (_[ us ∘r pR ]) ts  , q
    ≡⟨ cong (_, q) (map−cong (λ x → cong (x [_]) (∘pR−↑ us)) ts) ⟩
  map (_[ pR r∘ (↑ us) ]) ts , q
    ≡⟨ cong (_, q) (map−cong (r∘−asso _ _) ts) ⟩
  map (_[ ↑ us ] F.∘ flip ren pR) ts , q
    ≡⟨ cong (_, q) (map−∘ _ _ ts) ⟩
  map (_[ ↑ us ]) (map (flip ren pR) ts) , q
    ≡⟨⟩
  map (_[ ↑ us ]) (↑ ts)
    ≡⟨⟩
  (↑ ts) ∘ (↑ us)
    ∎
```

Figure A.3: Lifting and extending a substitution distributes over composition.

```
cong−ext : ∀ {m n} {t t′ : Tm n} {ρ ρ′ : Sub n m} →
  t ~βη t′ → ρ ≈βη ρ′ →
  (ρ , t) ≈βη (ρ′ , t′)
cong−ext t~t′ ⋄  = ext t~t′ ⋄
cong−ext t~t′ (ext x ρ≈ρ′) = ext t~t′ (cong−ext x ρ≈ρ′)

lookup−sub : ∀ {m n} {ρ ρ′ : Sub m n} (i : Fin n) →
    ρ ≈βη ρ′ → lookup i ρ ~βη lookup i ρ′
lookup−sub ()      ⋄
lookup−sub zero  (ext t~u _)  = t~u
lookup−sub (suc i) (ext _  ρ≈ρ′) = lookup−sub i ρ≈ρ′

η−help : ∀ {n m} (t : Tm n) (ρ : Sub m n) → weaken (t [ ρ ]) ≡ (weaken t) [ ↑ ρ ]
η−help t ρ = sym $ begin
  weaken t [ ↑ ρ ]
    ≡⟨⟩
  weaken t [ weaken−subst ρ , q ]
    ≡⟨ cong (λ x → weaken t [ x , q ]) (sym (mapWk−∘p ρ)) ⟩
  weaken t [ ρ ∘ p , q ]
    ≡⟨ cong (_[ ρ ∘ p , q ]) (wk−[p] t) ⟩
  t [ p ] [ ρ ∘ p , q ]
    ≡⟨ sym (subComp t p (ρ ∘ p , q)) ⟩
  t [ p ∘ (ρ ∘ p , q) ]
    ≡⟨ cong (t [_]) (pCons (ρ ∘ p) q) ⟩
  t [ ρ ∘ p ]
    ≡⟨ subComp t ρ p ⟩
  (t [ ρ ]) [ p ]
    ≡⟨ sym (wk−[p] (t [ ρ ])) ⟩
  weaken (t [ ρ ])
    ∎ where open P.≡−Reasoning

β−help : ∀ {m n} (t : Tm (suc n)) u (ρ : Sub m n) →
```

```
               t [ ↑ ρ ] [ id , u [ ρ ] ] ≡ t [ id , u ] [ ρ ]
β−help t u ρ = begin
   t [ ↑ ρ ] [ id , u [ ρ ] ]
      ≡⟨ sym (subComp t (↑ ρ) (id , u [ ρ ])) ⟩
   t [ ↑ ρ ∘ (id , u [ ρ ]) ]
      ≡⟨⟩
   t [ (weaken−subst ρ , q) ∘ (id , u [ ρ ]) ]
      ≡⟨ cong (λ x → t [ (x , q) ∘ (id , u [ ρ ]) ]) (sym (mapWk−∘p _)) ⟩
   t [ (ρ ∘ p , q) ∘ (id , u [ ρ ]) ]
      ≡⟨⟩
   t [ (ρ ∘ p) ∘ (id , u [ ρ ]) , u [ ρ ] ]
      ≡⟨ cong (λ x → t [ x , u [ ρ ] ]) (assoc ρ p _) ⟩
   t [ ρ ∘ (p ∘ (id , u [ ρ ])) , u [ ρ ] ]
      ≡⟨ cong (λ x → t [ ρ ∘ x , u [ ρ ] ]) (pCons _ _) ⟩
   t [ ρ ∘ id , u [ ρ ] ]
      ≡⟨ cong (λ x → t [ x , u [ ρ ] ]) (idR ρ) ⟩
   t [ ρ , u [ ρ ] ]
      ≡⟨ cong (λ x → t [ x , u [ ρ ] ]) (sym (idL ρ)) ⟩
   t [ id ∘ ρ , u [ ρ ] ]
      ≡⟨⟩
   t [ (id , u) ∘ ρ ]
      ≡⟨ subComp t (id , u) ρ ⟩
   t [ id , u ] [ ρ ]
      ∎ where open P.≡−Reasoning

congSub−t : ∀ {m n} {t t′ : Tm n} {ρ : Sub m n}
              → t ~βη t′ → t [ ρ ] ~βη t′ [ ρ ]
congSub−t (varcong i)  = refl~βη
congSub−t (apcong t~t′ t~t″)
   = apcong (congSub−t t~t′) (congSub−t t~t″)
congSub−t (ξ t~t′)  = ξ (congSub−t t~t′)
congSub−t {ρ = ρ} (β t u)
   rewrite sym $ β−help t u ρ = β (t [ ↑ ρ ]) (u [ ρ ])
congSub−t {ρ = ρ} (η a)
   rewrite cong (ƛ F.∘ (_· q)) (sym (η−help a ρ)) = η (a [ ρ ])
congSub−t (sym~βη t~t′) = sym~βη (congSub−t t~t′)
congSub−t (trans~βη t~t′ t~t″)
   = trans~βη (congSub−t t~t′) (congSub−t t~t″)

cong−∘≈₁ : ∀ {m n k} {σ σ′ : Sub m n} {γ : Sub k m}
              → σ ≈βη σ′ → σ ∘ γ ≈βη σ′ ∘ γ
cong−∘≈₁ {σ = []} {[]} ◇ = refl≈βη
cong−∘≈₁ {γ = γ} (ext t~u σ≈σ′)
   = cong−ext (congSub−t {ρ = γ} t~u) (cong−∘≈₁ σ≈σ′)

↑ρ−pr : ∀ {m n} {γ δ : Sub m n} → γ ≈βη δ → ↑ γ ≈βη ↑ δ
↑ρ−pr {γ = γ} {δ} γ≈δ
   rewrite  sym (mapWk−∘p γ)
```

```
        | sym (mapWk−∘p δ) = cong−ext refl∼βη (cong−∘≈₁ γ≈δ)

congSub−s : ∀ {m n} {t : Tm n} {ρ ρ′ : Sub m n}
          → ρ ≈βη ρ′ → t [ ρ ] ∼βη t [ ρ′ ]
congSub−s {ρ = []} {[]} ◇ = refl∼βη
congSub−s {t = var zero} (ext x ρ≈ρ′) = x
congSub−s {t = var (suc i)} (ext x ρ≈ρ′)
  = congSub−s {t = var i} ρ≈ρ′
congSub−s {t = a · b} (ext x ρ≈ρ′) =
  apcong (congSub−s {t = a} (ext x ρ≈ρ′))
    (congSub−s {t = b} (ext x ρ≈ρ′))
congSub−s {t = ƛ b} (ext x ρ≈ρ′)
  = ξ (congSub−s {t = b} (↑ρ−pr (ext x ρ≈ρ′)))

cong−[] : ∀ {m n} {t t′ : Tm n} {ρ ρ′ : Sub m n} →
          t ∼βη t′ → ρ ≈βη ρ′ →
          t [ ρ ] ∼βη t′ [ ρ′ ]
cong−[] {t′ = t′} t∼t′ ρ≈ρ′ =
  trans∼βη (congSub−t t∼t′) (congSub−s {t = t′} ρ≈ρ′)

cong−∘≈ : ∀ {m n k} {ρ σ : Sub m n} {ρ′ σ′ : Sub k m} →
          ρ ≈βη σ → ρ′ ≈βη σ′ →
          ρ ∘ ρ′ ≈βη σ ∘ σ′
cong−∘≈ ◇ ρ′∼σ′ = ◇
cong−∘≈ (ext t ρ≈σ) ◇ = ext (cong−[] t ◇) (cong−∘≈ ρ≈σ ◇)
cong−∘≈ (ext t ρ≈σ) (ext u ρ′≈σ′) =
  ext (cong−[] t (ext u ρ′≈σ′)) (cong−∘≈ ρ≈σ (ext u ρ′≈σ′))
```

Figure A.4: Congruence rules for untyped beta-eta equality.

Appendix B

Simply Typed

B.1 Scwf Equations

The relations that provide the equational theory for term and substitution equality for the explicit scwf and λ-scwf objects.

```
data _≈_ where
  subId : ∀ {Γ α} (t : Tm Γ α) → t [ id ] ≈ t
  qCons : ∀ {Γ Δ α} (t : Tm Γ α) (γ : Sub Γ Δ) → q [ < γ , t > ] ≈ t
  subComp : ∀ {Γ Δ Θ α} (t : Tm Δ α) (γ : Sub Γ Δ) (δ : Sub Θ Γ) →
    t [ γ ∘ δ ] ≈ t [ γ ] [ δ ]
  cong−sub : ∀ {Γ Δ α} {t t′ : Tm Γ α} {γ γ′ : Sub Δ Γ} →
    t ≈ t′ → γ ≋ γ′ → t [ γ ] ≈ t′ [ γ′ ]
  sym≈ : ∀ {Γ α} {t t′ : Tm Γ α} → t ≈ t′ → t′ ≈ t
  trans≈ : ∀ {Γ α} {t t′ t″ : Tm Γ α} →
    t ≈ t′ → t′ ≈ t″ → t ≈ t″

data _≋_ where
  id₀ : id {ε} ≋ <>
  <>Lzero : ∀ {Γ Δ} (γ : Sub Γ Δ) → <> ∘ γ ≋ <>
  idExt : ∀ {Γ α} → id {Γ • α} ≋ < p , q >
  idL : ∀ {Γ Δ} (γ : Sub Δ Γ) → id ∘ γ ≋ γ
  idR : ∀ {Γ Δ} (γ : Sub Γ Δ) →  γ ∘ id ≋ γ
  assoc : ∀ {Γ Δ Θ Λ} (γ : Sub Δ Θ) (δ : Sub Γ Δ) (ζ : Sub Λ Γ) →
    (γ ∘ δ) ∘ ζ ≋ γ ∘ (δ ∘ ζ)
  pCons : ∀ {Δ Θ α} (t : Tm Δ α) (γ : Sub Δ Θ) → p ∘ < γ , t > ≋ γ
  compExt : ∀ {Γ Δ Θ α} (t : Tm Δ α) (γ : Sub Δ Θ) (δ : Sub Γ Δ) →
    < γ , t > ∘ δ ≋ < γ ∘ δ , t [ δ ] >
  cong−<,> : ∀ {Γ Δ α} {t t′ : Tm Γ α} {γ γ′ : Sub Γ Δ} →
    t ≈ t′ → γ ≋ γ′ → < γ , t > ≋ < γ′ , t′ >
  cong−∘ : ∀ {Γ Δ Θ} {γ δ : Sub Δ Θ} {γ′ δ′ : Sub Γ Δ} →
    γ ≋ δ → γ′ ≋ δ′ → γ ∘ γ′ ≋ δ ∘ δ′
  sym≋ : ∀ {Γ Δ} {γ γ′ : Sub Γ Δ} → γ ≋ γ′ → γ′ ≋ γ
  trans≋ : ∀ {Γ Δ} {γ γ′ γ″ : Sub Γ Δ} →
    γ ≋ γ′ → γ′ ≋ γ″ → γ ≋ γ″
```

Figure B.1: Explicit scwf object equality relations.

```
data _≈_ where
  subId : ∀ {Γ α} (t : Tm Γ α) → t [ id ] ≈ t
  qCons : ∀ {Γ Δ α} (t : Tm Γ α) (γ : Sub Γ Δ) → q [ < γ , t > ] ≈ t
  subComp : ∀ {Γ Δ Θ α} (t : Tm Δ α) (γ : Sub Γ Δ) (δ : Sub Θ Γ) →
    t [ γ ∘ δ ] ≈ t [ γ ] [ δ ]
  subApp : ∀ {Γ Δ α β} (t : Tm Γ (α ⇒ β)) (u : Tm Γ α) (γ : Sub Δ Γ) →
    app (t [ γ ]) (u [ γ ]) ≈ app t u [ γ ]
  subLam : ∀ {Γ Δ α β} (t : Tm (Γ • α) β) (γ : Sub Δ Γ) →
    lam t [ γ ] ≈ lam (t [ < γ ∘ p , q > ])
  β : ∀ {Γ α β} {t : Tm (Γ • α) β} {u} → app (lam t) u ≈ t [ < id , u > ]
  η : ∀ {Γ α β} {t : Tm Γ (α ⇒ β)} → lam (app (t [ p ]) q) ≈ t
  cong−sub : ∀ {Γ Δ α} {t t′ : Tm Γ α} {γ γ′ : Sub Δ Γ} →
    t ≈ t′ → γ ≋ γ′ → t [ γ ] ≈ t′ [ γ′ ]
  cong−app : ∀ {Γ α β} {t t′ : Tm Γ (α ⇒ β)} {u u′ : Tm Γ α} →
    t ≈ t′ → u ≈ u′ → app t u ≈ app t′ u′
  cong−lam : ∀ {Γ α β} {t t′ : Tm (Γ • α) β} →
    t ≈ t′ → lam t ≈ lam t′
  sym≈ : ∀ {Γ α} {t t′ : Tm Γ α} → t ≈ t′ → t′ ≈ t
  trans≈ : ∀ {Γ α} {t t′ t″ : Tm Γ α} →
    t ≈ t′ →  t′ ≈ t″ → t ≈ t″

data _≋_ where
  id₀ : id {ε} ≋ <>
  <>Lzero : ∀ {Γ Δ} (γ : Sub Γ Δ) → <> ∘ γ ≋ <>
  idExt : ∀ {Γ α} → id {Γ • α} ≋ < p , q >
  idL : ∀ {Γ Δ} (γ : Sub Δ Γ) → id ∘ γ ≋ γ
  idR : ∀ {Γ Δ} (γ : Sub Γ Δ) →  γ ∘ id ≋ γ
  assoc : ∀ {Γ Δ Θ Λ} (γ : Sub Δ Θ) (δ : Sub Γ Δ) (ζ : Sub Λ Γ) →
    (γ ∘ δ) ∘ ζ ≋ γ ∘ (δ ∘ ζ)
  pCons : ∀ {Δ Θ α} (t : Tm Δ α) (γ : Sub Δ Θ) → p ∘ < γ , t > ≋ γ
  compExt : ∀ {Γ Δ Θ α} (t : Tm Δ α) (γ : Sub Δ Θ) (δ : Sub Γ Δ) →
    < γ , t > ∘ δ ≋ < γ ∘ δ , t [ δ ] >
  cong−<,> : ∀ {Γ Δ α} {t t′ : Tm Γ α} {γ γ′ : Sub Γ Δ} →
    t ≈ t′ →  γ ≋ γ′ → < γ , t > ≋ < γ′ , t′ >
  cong−∘ : ∀ {Γ Δ Θ} {γ δ : Sub Δ Θ} {γ′ δ′ : Sub Γ Δ} →
    γ ≋ δ → γ′ ≋ δ′ → γ ∘ γ′ ≋ δ ∘ δ′
  sym≋ : ∀ {Γ Δ} {γ γ′ : Sub Γ Δ} → γ ≋ γ′ → γ′ ≋ γ
  trans≋ : ∀ {Γ Δ} {γ γ′ γ″ : Sub Γ Δ} →
    γ ≋ γ′ → γ′ ≋ γ″ → γ ≋ γ″
```

Figure B.2: Explicit λ-scwf object equality relations

B.2 Properties

```
[]−comm {ε} (var ()) tt
```

```
[]−comm (var here) (ρ , t) = sym≈ (qCons ⟦ t ⟧ ⟦ ρ ⟧′)
[]−comm (var (there ∈Γ)) (ρ , t) = begin
   ⟦ tkVar ∈Γ ρ ⟧
      ≈⟨ []−comm (var ∈Γ) ρ ⟩
   ⟦ var ∈Γ ⟧ [ ⟦ ρ ⟧′ ]
      ≈⟨ cong−sub refl≈ (sym≋ (pCons ⟦ t ⟧ ⟦ ρ ⟧′)) ⟩
   ⟦ var ∈Γ ⟧ [ p ∘ < ⟦ ρ ⟧′ , ⟦ t ⟧ > ]
      ≈⟨ subComp ⟦ var ∈Γ ⟧ p < ⟦ ρ ⟧′ , ⟦ t ⟧ > ⟩
   ⟦ var ∈Γ ⟧ [ p ] [ < ⟦ ρ ⟧′ , ⟦ t ⟧ > ]
      ∎
   where open EqR (TmSetoid {_})

[]−comm (t · u) ρ = begin
   app ⟦ t [ ρ ]λ ⟧ ⟦ u [ ρ ]λ ⟧
      ≈⟨ cong−app ([]−comm t ρ) refl≈ ⟩
   app (⟦ t ⟧ [ ⟦ ρ ⟧′ ]) ⟦ u [ ρ ]λ ⟧
      ≈⟨ cong−app refl≈ ([]−comm u ρ) ⟩
   app (⟦ t ⟧ [ ⟦ ρ ⟧′ ]) (⟦ u ⟧ [ ⟦ ρ ⟧′ ])
      ≈⟨ subApp ⟦ t ⟧ ⟦ u ⟧ ⟦ ρ ⟧′ ⟩
   app ⟦ t ⟧ ⟦ u ⟧ [ ⟦ ρ ⟧′ ]
      ∎
   where open EqR (TmSetoid {_})

[]−comm {Γ} (ƛ {α = α} t) ρ = begin
   lam ⟦ t [ wk−sub Γ ⊆−• ρ , var here ]λ ⟧
      ≈⟨ cong−lam ([]−comm t (wk−sub Γ ⊆−• ρ , var here)) ⟩
   lam (⟦ t ⟧ [ < ⟦ wk−sub Γ ⊆−• ρ ⟧′ , q > ])
      ≈⟨ cong−lam (cong−sub refl≈ (help)) ⟩
   lam (⟦ t ⟧ [ < ⟦ ρ ∘λ p−λ ⟧′ , q > ])
      ≈⟨ cong−lam (cong−sub refl≈
         (cong−<, > refl≈ (⟦⟧−∘−dist_p ρ p−λ))) ⟩
   lam (⟦ t ⟧ [ < ⟦ ρ ⟧′ ∘ ⟦ p−λ ⟧′ , q > ])
      ≈⟨ cong−lam (cong−sub refl≈
         (cong−<, > refl≈ (cong−∘ refl≋ (sym≋ p~⟦p⟧)))) ⟩
   lam (⟦ t ⟧ [ < ⟦ ρ ⟧′ ∘ p , q > ])
      ≈⟨ sym≈ (subLam ⟦ t ⟧ ⟦ ρ ⟧′) ⟩
   lam ⟦ t ⟧ [ ⟦ ρ ⟧′ ]
      ∎
   where open EqR (TmSetoid {_})
      help : < ⟦ wk−sub Γ (⊆−• {a = α}) ρ ⟧′ , q >
         ≋ < ⟦ ρ ∘λ p−λ ⟧′ , q >
      help rewrite wk−sub−∘−p {Γ} {α = α} ρ = refl≋

⟦⟧−∘−dist {Θ = ε} tt σ = sym≋ (<>Lzero ⟦ σ ⟧′)
⟦⟧−∘−dist {Θ = Θ • x} (ρ , t) σ = begin
   < ⟦ ρ ∘λ σ ⟧′ , ⟦ t [ σ ]λ ⟧ >
      ≈⟨ cong−<, > refl≈ (⟦⟧−∘−dist ρ σ) ⟩
   < ⟦ ρ ⟧′ ∘ ⟦ σ ⟧′ , ⟦ t [ σ ]λ ⟧ >
```

```
      ≈⟨ cong–<,> ([]–comm t σ) refl≈ ⟩
    < ⟦ ρ ⟧′ ∘ ⟦ σ ⟧′ , ⟦ t ⟧ [ ⟦ σ ⟧′ ] >
      ≈⟨ sym≈ (compExt ⟦ t ⟧ ⟦ ρ ⟧′ ⟦ σ ⟧′) ⟩
    < ⟦ ρ ⟧′ , ⟦ t ⟧ > ∘ ⟦ σ ⟧′
      ∎
    where open EqR (SubSetoid {_} {_})
```

Figure B.3: Substitution and composition commute to other Scwf object.

```
vars : ∀ {Γ Δ} → Δ ▶ Γ → Sub–cwf Δ Γ
vars {ε}  tt           = <>
vars {Γ • x} (ρ , t) = < vars ρ , varCwf t >

▶–to–hom : ∀ {Δ Γ} (f : ∀ {α} → α ∈ Δ → Tm–cwf Δ α)
      → Δ ▶ Γ → Sub–cwf Δ Γ
▶–to–hom {Γ = ε}  _ tt  = <>
▶–to–hom {Γ = Γ • x} f (ρ , t) = < ▶–to–hom f ρ , f t >

map≈mapcwf : ∀ {Γ Δ} (ρ : Δ ▶ Γ) →
          ⟦ ▶–to–▷ var ρ ⟧′ ≈ ▶–to–hom varCwf ρ
map≈mapcwf {ε}  tt  = refl≈
map≈mapcwf {Γ • x} (ρ , _) = cong–<,> refl≈ (map≈mapcwf ρ)

pCwf : ∀ {Γ α} → Sub–cwf (Γ • α) Γ
pCwf = ▶–to–hom varCwf pV

vars≈hom : ∀ {Γ Δ} (ρ : Δ ▶ Γ) → vars ρ ≈ ▶–to–hom varCwf ρ
vars≈hom {ε}  tt          = refl≈
vars≈hom {Γ • x} (ρ , t) = cong–<,> refl≈ (vars≈hom ρ)

var–lemma : ∀ {Γ Δ α} (ρ : Δ ▶ Γ) →
            vars ρ ∘ (p {α = α}) ≈ vars (map–∈ there ρ)
var–lemma {ε} tt = <>Lzero p
var–lemma {Γ • x} (ρ , t) = begin
    < vars ρ , varCwf t > ∘ p
      ≈⟨ compExt (varCwf t) (vars ρ) p ⟩
    < vars ρ ∘ p , varCwf t [ p ] >
      ≈⟨ cong–<,> refl≈ (var–lemma ρ) ⟩
    <  vars (map–∈ there ρ) , varCwf t [ p ] >
      ∎
    where open EqR (SubSetoid {_} {_})

help : ∀ {Γ x α} →
      vars (map–∈ {α = x} (there) (map–∈ {α = α} (there) idV)) ≈
      vars (map–∈ (there) (▶–weaken Γ (step ⊆–refl) idV))
help {Γ} {x} {α} rewrite mapwk {Γ} {α = x} {α} idV = refl≈

p≈vars : ∀ {Γ α} → p {α = α} ≈ vars (pV {Γ} {α})
```

```
p≈vars {ε} = ter−arrow p
p≈vars {Γ • x} {α} = let (ρ , t) = (pV {Γ • x})
                       in  begin
  p
    ≈⟨ surj−<, > p ⟩
  < p ∘ p , q [ p ] >
    ≈⟨ cong−<, > refl≈ (cong−∘ p≈vars refl≈) ⟩
  < vars pV ∘ p , q [ p ] >
    ≈⟨ cong−<, > refl≈ (var−lemma pV) ⟩
  < vars (map−∈ there pV) , q [ p ] >
    ≈⟨ cong−<, > refl≈ help ⟩
  < vars ρ , q [ p ] >
    ∎
  where open EqR (SubSetoid {_} {_})

p−inverse {Γ} {α} =
  trans≈ p≈vars (trans≈ (vars≈hom _)
    (trans≈ (sym≈ (map≈mapcwf _)) g))
  where g : ⟦ ▶−to−▷ var (pV {Γ} {α}) ⟧′ ≈ ⟦ p−λ ⟧′
    g rewrite pIsVarP {Γ} {α} = refl≈
```

Figure B.4: Projection inverse for scwfs.

```
cong−ext : ∀ {Γ Δ α} {t t′ : Tm Γ α} {ρ ρ′ : Sub Γ Δ} →
          t ~βη t′ → ρ ≈βη ρ′ →
          (ρ , t) ≈βη (ρ′ , t′)
cong−ext t~t′ ⋄  = ext t~t′ ⋄
cong−ext t~t′ (ext x ρ≈ρ′) = ext t~t′ (cong−ext x ρ≈ρ′)

β−help : ∀ {Γ Δ α β} (t : Tm (Γ • α) β) u (γ : Sub Δ Γ) →
            t [ wk−sub _ ⊆−• γ , q ] [ id , u [ γ ] ] ≡ t [ id , u ] [ γ ]
β−help t u γ = begin
  t [ wk−sub _ ⊆−• γ , q ] [ id , u [ γ ] ]
    ≡⟨ cong (λ x → t [ x , q ] [ id , u [ γ ] ])
          (sym (wk−sub−∘−p γ)) ⟩
  t [ γ ∘ p , q ] [ id , u [ γ ] ]
    ≡⟨ sym $ subComp t (γ ∘ p , q) (id , (u [ γ ])) ⟩
  t [ (γ ∘ p , q) ∘ (id , u [ γ ]) ]
    ≡⟨⟩
  t [ (γ ∘ p) ∘ (id , u [ γ ]) , u [ γ ] ]
    ≡⟨ cong (λ x → t [ x , u [ γ ] ]) (assoc γ _ _) ⟩
  t [ γ ∘ (p ∘ (id , u [ γ ])) , u [ γ ] ]
    ≡⟨ cong (λ x → t [ γ ∘ x , u [ γ ] ])
          (pCons (u [ γ ]) id) ⟩
  t [ γ ∘ id , u [ γ ] ]
    ≡⟨ cong (λ x → t [ x , u [ γ ] ]) (idR γ) ⟩
  t [ γ , u [ γ ] ]
    ≡⟨ cong (λ x → t [ x , u [ γ ] ]) (sym (idL γ)) ⟩
    t [ id ∘ γ , u [ γ ] ]
      ≡⟨⟩
    t [ (id , u ) ∘ γ ]
      ≡⟨ subComp t (id , u) γ ⟩
    t [ id , u ] [ γ ]
      ∎
    where open P.≡−Reasoning

  η−help : ∀ {Γ Δ α β} (t : Tm Γ (α ⇒ β)) (γ : Sub Δ Γ) → (t [ γ ])
                [ p {α = α} ] ≡ t [ p ] [ wk−sub Γ ⊆−• γ , q ]
  η−help t γ = sym $ begin
    t [ p ] [ wk−sub _ ⊆−• γ , q ]
      ≡⟨ cong (λ x → t [ p ] [ x , q ]) (sym (wk−sub−∘−p γ)) ⟩
    t [ p ] [ γ ∘ p , q ]
      ≡⟨ sym (subComp t _ (γ ∘ p , q)) ⟩
    t [ p ∘ (γ ∘ p , q ) ]
      ≡⟨ cong (t [_]) (pCons q (γ ∘ p)) ⟩
    t [ γ ∘ p ]
      ≡⟨ subComp t γ p ⟩
    t [ γ ] [ p ]
      ∎
    where open P.≡−Reasoning

  congSub−t : ∀ {Γ Δ α} {t t′ : Tm Γ α} {ρ : Sub Δ Γ}
                → t ~βη t′ → t [ ρ ] ~βη t′ [ ρ ]
  congSub−t (varcong v) = refl~βη
  congSub−t (apcong t~t′ t~t″) =
    apcong (congSub−t t~t′) (congSub−t t~t″)
  congSub−t (ξ t~t′) = ξ (congSub−t t~t′)
  congSub−t {ρ = ρ} (β t u)
    rewrite sym $ β−help t u ρ =
      β (t [ wk−sub _ ⊆−• ρ , q ]) (u [ ρ ])
  congSub−t {Γ} {Δ} {α = F ⇒ G} {ρ = ρ} (η a)
    rewrite cong (λ x → ƛ (x · q)) (sym $ η−help a ρ) = η (a [ ρ ])
  congSub−t (sym~βη t~t′) = sym~βη (congSub−t t~t′)
  congSub−t (trans~βη t~t′ t~t″) =
    trans~βη (congSub−t t~t′) (congSub−t t~t″)

  cong−∘≈₁ : ∀ {Γ Δ Ξ} {σ σ′ : Sub Δ Γ} {γ : Sub Ξ Δ}
                → σ ≈βη σ′ → σ ∘ γ ≈βη σ′ ∘ γ
  cong−∘≈₁ {σ = tt} {tt} ⋄ = refl≈βη
  cong−∘≈₁ {γ = γ} (ext x σ≈σ′) =
    cong−ext (congSub−t {ρ = γ} x) (cong−∘≈₁ σ≈σ′)

  ↑−preserv : ∀ {Γ Δ α} {γ δ : Sub Γ Δ} → γ ≈βη δ
                → (wk−sub _ (⊆−• {a = α}) γ , q) ≈βη (wk−sub _ ⊆−• δ , q)
  ↑−preserv {α = α} {γ = γ} {δ} γ≈δ
    rewrite sym $ (wk−sub−∘−p {α = α} γ)
```

```
          | sym $ (wk−sub−∘−p {α = α} δ)
            = cong−ext refl∼βη (cong−∘≈₁ γ≈δ)

congSub−s : ∀ {Γ Δ α} {t : Tm Δ α} {ρ ρ′ : Sub Γ Δ}
          → ρ ≈βη ρ′ → t [ ρ ] ∼βη t [ ρ′ ]
congSub−s {ρ = tt}           ⋄  = refl∼βη
congSub−s {t = var here}  (ext x ρ≈ρ′) = x
congSub−s {t = var (there v)} (ext x ρ≈ρ′) = congSub−s {t = var v} ρ≈ρ′
congSub−s {t = a · b}        (ext x ρ≈ρ′) =
   apcong (congSub−s {t = a} (ext x ρ≈ρ′)) (congSub−s {t = b} (ext x ρ≈ρ′))
congSub−s {t = ƛ b}          (ext x ρ≈ρ′) =
   ξ (congSub−s {t = b} (↑−preserv (ext x ρ≈ρ′)))
```

Figure B.5: Congruence rules for simply typed lambda calculus with beta and eta.

Appendix C

Dependently Typed

C.1 Inversion Lemmas of Cwf Calculus

```
lemma−1 : ∀ {n} {Γ : Ctx n} {A} → Γ ⊢ A → Γ ⊢

lemma−2 : ∀ {n} {Γ : Ctx n} {A t} → Γ ⊢ t ∈ A → Γ ⊢ A

lemma−2B : ∀ {n} {Γ : Ctx n} {A t} → Γ ⊢ t ∈ A → Γ ⊢

lemma−3 : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} {γ} → Δ ▹ Γ ⊢ γ → Γ ⊢ × Δ ⊢

lemma−3 (⊢id Γ⊢) = Γ⊢ , Γ⊢
lemma−3 (⊢∘ ⊢γ₁ ⊢γ₂) = π₁ (lemma−3 ⊢γ₁) , π₂ (lemma−3 ⊢γ₂)
lemma−3 (⊢p ⊢A) = lemma−1 ⊢A , c−ext (lemma−1 ⊢A) ⊢A
lemma−3 (⊢<> Δ⊢) = c−emp , Δ⊢
lemma−3 (⊢<,> ⊢γ ⊢A _) = c−ext (lemma−1 ⊢A) ⊢A , π₂ (lemma−3 ⊢γ)

lemma−1 (ty−U Γ⊢) = Γ⊢
lemma−1 (ty−∈U A∈U) = lemma−2B A∈U
lemma−1 (ty−Π−F ⊢A _) = lemma−1 ⊢A
lemma−1 (ty−sub _ ⊢γ) = π₂ (lemma−3 ⊢γ)

lemma−2B (tm−q ⊢A) = c−ext (lemma−1 ⊢A) ⊢A
lemma−2B (tm−sub _ ⊢γ) = π₂ (lemma−3 ⊢γ)
lemma−2B (tm−app _ _ _ t∈A) = lemma−2B t∈A
lemma−2B (tm−conv _ t∈A _) = lemma−2B t∈A
lemma−2B (tm−Π−I _ _ t∈A) with lemma−2B t∈A
... | c−ext Γ⊢ _ = Γ⊢

lemma−2 (tm−q ⊢A) = ty−sub ⊢A (⊢p ⊢A)
lemma−2 (tm−sub t∈A ⊢γ) = ty−sub (lemma−2 t∈A) ⊢γ
lemma−2 (tm−Π−I ⊢A ⊢B _) = ty−Π−F ⊢A ⊢B
lemma−2 (tm−app ⊢A ⊢B _ t∈A) =
  let ⊢id = ⊢id (lemma−1 ⊢A)
  in ty−sub ⊢B (⊢<,> ⊢id ⊢A
      (tm−conv (ty−sub ⊢A ⊢id) t∈A (subId _)))
lemma−2 (tm−conv ⊢A′ _ _) = ⊢A′
```

Figure C.1: Inversion lemmas for ΠU cwf calculus.

C.2 Raw Isomorphism

The isomorphism of the raw $\lambda\Pi$U calculus and the cwf combinators. Only the proofs for the functor are shown since only those have changed.

```
– Natural transformations between the functors of the cwfs
⟦_⟧  : ∀ {n} → Tm−λ n → Tm−cwf n
⟪_⟫ : ∀ {n} → Tm−cwf n → Tm−λ n

– Functors between the base categories of the cwfs
⟦_⟧′ : ∀ {m n} → Sub−λ m n → Sub−cwf m n
⟪_⟫′ : ∀ {m n} → Sub−cwf m n → Sub−λ m n

– Variable representation in the variable free calculus
varCwf : ∀ {n} (i : Fin n) → Tm−cwf n
varCwf zero  = q
varCwf (suc i) = varCwf i [ p ]

⟦ var i ⟧ = varCwf i
⟦ ƛ t  ⟧ = lam ⟦ t ⟧
⟦ t · u ⟧ = app ⟦ t ⟧ ⟦ u ⟧
⟦ Π A B ⟧ = Π ⟦ A ⟧ ⟦ B ⟧
⟦ U   ⟧ = U

⟪ q        ⟫ = q−λ
⟪ t [ γ ] ⟫ = ⟪ t ⟫ [ ⟪ γ ⟫′ ]λ
⟪ lam t    ⟫ = ƛ ⟪ t ⟫
⟪ app t u ⟫ = ⟪ t ⟫ · ⟪ u ⟫
⟪ Π A B   ⟫ = Π ⟪ A ⟫ ⟪ B ⟫
⟪ U        ⟫ = U

⟦ []   ⟧′ = <>
⟦ t :: ρ ⟧′ = < ⟦ ρ ⟧′ , ⟦ t ⟧ >

⟪ id      ⟫′ = id−λ
⟪ γ ∘ γ′  ⟫′ = ⟪ γ ⟫′ ∘λ ⟪ γ′ ⟫′
⟪ p       ⟫′ = p−λ
⟪ <>      ⟫′ = []
⟪ < γ , t > ⟫′ = ⟪ γ ⟫′ , ⟪ t ⟫

sub−comm : ∀ {m n} (t : Tm−λ n) (σ : Sub−λ m n)
         → ⟦ t [ σ ]λ ⟧ ≈ ⟦ t ⟧ [ ⟦ σ ⟧′ ]
sub−comm (var zero)  (t :: σ) = sym≈ (qCons ⟦ t ⟧ ⟦ σ ⟧′)
```

```
sub−comm (var (suc i)) (t :: σ) = begin
   ⟦ lookup i σ ⟧                      ≈⟨ sub−comm (var i) σ ⟩
   ⟦ var i ⟧ [ ⟦ σ ⟧′ ]                ≈⟨ cong−sub refl≈ (pCons ⟦ t ⟧ ⟦ σ ⟧′) ⟩
   ⟦ var i ⟧ [ p ∘ < ⟦ σ ⟧′ , ⟦ t ⟧ > ] ≈⟨ subComp ⟦ var i ⟧ p < ⟦ σ ⟧′ , ⟦ t ⟧ > ⟩
   ⟦ var i ⟧ [ p ] [ < ⟦ σ ⟧′ , ⟦ t ⟧ > ] ∎
   where open EqR (TmSetoid {_})
sub−comm (t · u) σ =
   trans≈ (cong−app (sub−comm t σ) (sub−comm u σ))
      (subApp ⟦ σ ⟧′ ⟦ t ⟧ ⟦ u ⟧)
sub−comm U     _ = sym≈ subU
sub−comm (ƛ t)  σ =  begin
   lam ⟦ t [ ↑ σ ]λ ⟧
      ≈⟨ cong−lam $ sub−comm t (↑ σ) ⟩
   lam (⟦ t ⟧ [ < ⟦ wk−sub σ ⟧′ , q > ])
      ≈⟨ cong−lam $ cong−sub refl≈ help ⟩
   lam (⟦ t ⟧ [ < ⟦ σ ∘λ p−λ ⟧′ , q > ])
      ≈⟨ cong−lam $ cong−sub refl≈
         (cong−<, > refl≈ (⟦⟧−∘−dist σ p−λ)) ⟩
   lam (⟦ t ⟧ [ < ⟦ σ ⟧′ ∘ ⟦ p−λ ⟧′ , q > ])
      ≈⟨ cong−lam $ cong−sub refl≈
         (cong−<, > refl≈ (cong−∘ refl≈ (sym≈ p−inverse))) ⟩
   lam (⟦ t ⟧ [ < ⟦ σ ⟧′ ∘ p , q > ])
      ≈⟨ sym≈ (subLam ⟦ t ⟧ ⟦ σ ⟧′) ⟩
   lam ⟦ t ⟧ [ ⟦ σ ⟧′ ]
      ∎
   where open EqR (TmSetoid {_})
      help : < ⟦ wk−sub σ ⟧′ , q >
             ≈ < ⟦ σ ∘λ p−λ ⟧′ , q >
      help rewrite wkSub−∘−p σ = refl≈
sub−comm (Π A B) σ = begin
   Π ⟦ A [ σ ]λ ⟧ ⟦ B [ ↑ σ ]λ ⟧
      ≈⟨ cong−Π (sub−comm A σ) (sub−comm B (↑ σ)) ⟩
   Π (⟦ A ⟧ [ ⟦ σ ⟧′ ]) (⟦ B ⟧ [ < ⟦ wk−sub σ ⟧′ , q > ])
      ≈⟨ cong−Π refl≈ help ⟩
   Π (⟦ A ⟧ [ ⟦ σ ⟧′ ]) (⟦ B ⟧ [ < ⟦ σ ∘λ p−λ ⟧′ , q > ])
      ≈⟨ cong−Π refl≈ (cong−sub refl≈
         (cong−<, (⟦⟧−∘−dist σ p−λ))) ⟩
   Π (⟦ A ⟧ [ ⟦ σ ⟧′ ]) (⟦ B ⟧ [ < ⟦ σ ⟧′ ∘ ⟦ p−λ ⟧′ , q > ])
      ≈⟨ cong−Π refl≈ (cong−sub refl≈
         (cong−<, (cong−∘₂ (sym≈ p−inverse)))) ⟩
   Π (⟦ A ⟧ [ ⟦ σ ⟧′ ]) (⟦ B ⟧ [ < ⟦ σ ⟧′ ∘ p , q > ])
      ≈⟨ sym≈ (subΠ ⟦ σ ⟧′ ⟦ A ⟧ ⟦ B ⟧) ⟩
   Π ⟦ A ⟧ ⟦ B ⟧ [ ⟦ σ ⟧′ ]
      ∎
   where open EqR (TmSetoid {_})
      help : ⟦ B ⟧ [ < ⟦ wk−sub σ ⟧′ , q > ]
             ≈ ⟦ B ⟧ [ < ⟦ σ ∘λ p−λ ⟧′ , q > ]
      help rewrite wkSub−∘−p σ = refl≈
```

```
t−λ⇒cwf : ∀ {n} (t : Tm−λ n) → ⟪ ⟦ t ⟧ ⟫ ≡ t

t−cwf⇒λ : ∀ {n} (t : Tm−cwf n) → ⟦ ⟪ t ⟫ ⟧ ≈ t

t−λ⇒cwf (var zero) = refl
t−λ⇒cwf (var (suc i))
  rewrite sym (lookup−p i) = cong (_[ p−λ ]λ) (t−λ⇒cwf (var i))
t−λ⇒cwf (ƛ t) = cong ƛ (t−λ⇒cwf t)
t−λ⇒cwf (t · u) = cong₂ _·_ (t−λ⇒cwf t) (t−λ⇒cwf u)
t−λ⇒cwf (Π A B) = cong₂ Π (t−λ⇒cwf A) (t−λ⇒cwf B)
t−λ⇒cwf U = refl

t−cwf⇒λ q = refl≈
t−cwf⇒λ (t [ γ ]) =
  trans≈ (sub−comm ⟪ t ⟫ ⟪ γ ⟫′)
    (cong−sub (t−cwf⇒λ t) (s−cwf⇒λ γ))
t−cwf⇒λ (lam t) = cong−lam (t−cwf⇒λ t)
t−cwf⇒λ (app t u) = cong−app (t−cwf⇒λ t) (t−cwf⇒λ u)
t−cwf⇒λ (Π A B) = cong−Π (t−cwf⇒λ A) (t−cwf⇒λ B)
t−cwf⇒λ U = refl≈
```