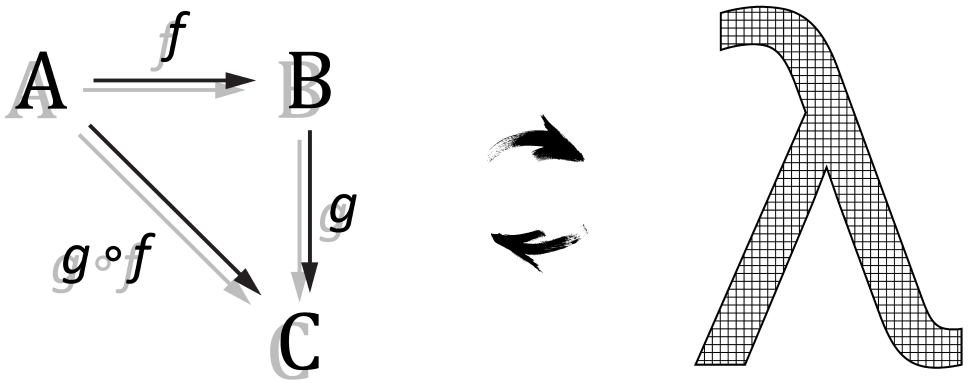

On Initial Categories with Families

Formalization of unityped and simply typed CwFs in Agda

Master's thesis in Computer Science – algorithms, languages and logic

KONSTANTINOS BRILAKIS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Master's thesis 2018

On Initial Categories with Families

Formalization of unityped and simply typed CwFs in Agda

KONSTANTINOS BRILAKIS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Supervisor: Peter Dybjer
Co-supervisor: Simon Huber
Examiner: Andreas Abel

Master's Thesis 2018
Department of Computer Science and Engineering
Programming Logic Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: basic category theory diagram on the left, next to an artistic Greek lambda letter.

Typeset in LaTeX
Font: Linux Libertine
Gothenburg, Sweden 2018

On Initial Categories with Families
Formalization of unityped and simply typed CwFs in Agda
KONSTANTINOS BRILAKIS
Department of Computer Science and Engineering
Programming Logic Group
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis explores a categorical model of type theory, namely, categories with families who provide a model for a basic framework of dependent type theory. The notion of a category with families is formalized as a generalized algebraic theory with extra structure with the purpose of modelling different lambda calculi. The work revolves around implementing in the Agda proof assistant categories with families at three levels: *(i)* untyped, *(ii)* simply typed, and *(iii)* dependently typed calculi. The formalization primarily consists of constructing initial objects in the category of categories with families and isomorphisms between them. The work investigates each notion from its core and proceeds by adding extra structure. Complete formalizations of untyped and simply typed categories with families are presented along with an incomplete picture for dependent types.

Keywords: Agda, lambda-calculus, categories-with-families.

Acknowledgements

First and foremost, an expression of appreciation and gratitude to my supervisors Peter and Simon for their guidance in this rough theoretical environment in which the project was founded on. Hopefully, my abuse of terminology throughout the work was not so terrible. It was a great experience overall and we are undeniably satisfied with the outcome. I also thank Jan Mas for proof reading a draft of the report.

Konstantinos Brilakis, Gothenburg, February 2018

Contents

Listings **xi**

1 Introduction **1**

2 Background **3**
2.1 Related work . . . 3
2.2 Agda . . . 3
2.3 Categories with Families . . . 3
2.3.1 CwFs Definition . . . 3
2.3.2 Generalized Algebraic Theory of CwFs . . . 4

3 Unityped CwFs **9**
3.1 Ucwf Notions . . . 9
3.1.1 Pure Ucwfs . . . 9
3.1.2 λ-Ucwfs . . . 12
3.1.3 λ-$\beta\eta$-Ucwfs . . . 13
3.2 Term Models of Ucwfs . . . 13
3.2.1 Ucwf with Explicit Substitutions . . . 13
3.2.2 Ucwf with Implicit Substitutions . . . 15
3.2.3 Isomorphism of Ucwfs . . . 20
3.3 Term Models of λ-Ucwfs . . . 25
3.3.1 λ-Ucwf with Explicit Substitutions . . . 25
3.3.2 λ-Ucwf with Implicit Substitutions . . . 26
3.3.3 Isomorphism of λ-Ucwfs . . . 27
3.4 Term Models of λ-$\beta\eta$-Ucwfs . . . 29

4 Simply Typed CwFs **31**
4.1 Scwf Notions . . . 32
4.1.1 Pure Scwfs . . . 32
4.1.2 λ-Scwfs . . . 34
4.1.3 λ-$\beta\eta$-Scwfs . . . 34
4.1.4 Extrinsic Scwfs . . . 35
4.2 Term Models of Scwfs . . . 36
4.2.1 Scwf with Explicit Substitutions . . . 36
4.2.2 Scwf with Implicit Substitutions . . . 37
4.2.3 Isomorphism of Scwfs . . . 41
4.3 Term Models of λ-Scwfs . . . 44
4.3.1 λ-Scwf with Explicit Substitutions . . . 44
4.3.2 λ-Scwf with Implicit Substitutions . . . 45

4.3.3 Isomorphism of λ-Scwfs 46
4.4 Term Models of λ-$\beta\eta$-Scwfs 47
4.5 Extrinsic Scwfs 48
4.5.1 Extrinsic and Intrinsic Scwfs 54

5 ΠU-CwFs **57**
5.1 ΠU-Cwf with Explicit Substitutions 57
5.2 ΠU-Cwf with Implicit Substitutions 61

6 Conclusion **67**

A Agda Code **I**

Listings

A.1 Equalities for ucwf with explicit substitutions. I
A.2 Lifting and extending a substitution distributes over composition. . . II
A.3 Congruence rules for untyped beta-eta equality. IV
A.4 Equations for scwf with explicit substitutions. V
A.5 Substitution and composition commute to other Scwf object. VI
A.6 Projection is preserved from scwf morphisms. VII
A.7 Congruence rules for simply typed lambda calculus with beta and eta. IX
A.8 Inversion lemmas for ΠU cwf calculus explicit substitutions. X
A.9 Raw isomorphism of well-scoped terms with Π and U. XII

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 predicate logic, whereas type theory is not. 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 laws 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 that brought cartesian closed categories as a part of a three way isomorphism.

The theory can be thought of as a typed functional programming language since it extends the simply typed λ-calculus with dependent types. A noteworthy property 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 has 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 [2]. It is a concept similar to Cartmell's categories with attributes [2] (they are in fact equivalent [3]), 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, i.e., the assignment of terms to free variables in a context. A cwf is a model of dependent type theory [4] and cwfs can be formalized as a generalized algebraic theory providing clear syntax [2]. This presentation is an algebraic formulation of type theory and some may consider it more canonical [5].

There are in fact different versions of cwfs that model more basic theories such as the untyped and simply typed λ-calculus, called unityped cwfs (ucwfs) and simply typed cwfs (scwfs), respectively. These notions can be extended with extra structure like lambda abstractions and application to make them a model of the respective λ-

calculus.

The work in this thesis is concerned with a formalization of different cwfs in Agda. In more detail, we consider the category of cwfs and define what it means to be a object in this category. Subsequently, we construct two cwfs, one with explicit and one with implicit substitutions. The motivation of the project is the construction of various initial cwfs and isomorphisms between them. The presentation of each cwf is expressed as a generalized algebraic theory.

The report is therefore divided into three sections for each cwf we discuss. First, we implement ucwfs using scope safe terms. Second, we implement two different versions of scwfs, one based on raw syntax supplemented with an external typing relation and one with intrinsically typed terms. Finally, we implement some parts of the full cwfs and dependent types with Π types and a universe à la Russell.

Aim

The aim of this thesis is to formalize unityped, simply typed, and some parts of the full cwfs in the proof assistant Agda. For each case, we construct objects in the respective category and construct an isomorphism between them. 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. The second cwf is more traditional with substitutions implemented as meta-level operations (implicit).

The formalization steps for each cwf are roughly the following:

1. Define the notion of a cwf as a record.
2. Construct an initial cwf with explicit substitutions.
3. Construct a second initial cwf with implicit substitutions.
4. Demonstrate an isomorphism between 2 and 3.

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 directly implement 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 full cwfs, it is not clear how to formalize intrinsically typed terms, but relevant work on that front has been done by Chapman [6].

2
Background

2.1 Related work

This project has some significance for the foundations of type theory. In particular, 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., by Ahrens, Lumsdain, and Voevodsky [7]; that work is also formalized in the Coq theorem prover.

The definition of cwfs as a generalized algebraic theory from *Internal Type Theory* [2] is the basis of our 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, and Dybjer [4] extensively; there 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]. It uses scope safe lambda terms and nameless variables represented by 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 [9].

2.2 Agda

The proofs and definitions of this thesis are formalized in the Agda language [10]. Agda is a dependently-typed functional programming language based on intuitionistic type theory. It has inductive families and mixfix operators and unicode support. Agda allows users not only to implement, but express and prove properties about their programs. Proofs are written in a functional style similar to Haskell. Moreover, it has numerous libraries readily available such as the Agda standard library which is used throughout our implementation.

2.3 Categories with Families

2.3.1 CwFs Definition

Categories with families are models for a basic framework of dependent types. Their structure 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 define a category with contexts and substitutions as the

objects and morphisms and 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 called 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 $\mathrm{T} : C^{op} \to Fam$. Intuitively, $\mathrm{T}(\Gamma)$ is the set of terms indexed by their type. A functor $\mathrm{Ty}(\Gamma)$ that defines the set of types in context Γ, where $\mathrm{Ty} : C^{op} \to Set$. This describes how substitutions act on types. Moreover, for each $A \in \mathrm{Ty}(\Gamma)$, there is the set of terms: $\mathrm{Tm}(\Gamma, A)$ of type A. Another notation for the terms is $\Gamma \vdash A$. Thus, T is given by $\mathrm{T}(\Gamma) = (\Gamma \vdash A)_{(A\in \mathrm{Ty}(\Gamma))}$. As a result, for any morphism γ of C, $\mathrm{T}(\gamma)$ interprets substitution both for terms and types.
- 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., $A \in \mathrm{Ty}(\Gamma)$, 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 cwf morphism 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 strictly.

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 (GAT) 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 [4].

The idea of a generalized algebraic theory, introduced by Cartmell [11], 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 [11].

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

Rules for the base category

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

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

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

$$(\gamma \circ \delta) \circ \theta = \gamma \circ (\delta \circ \theta)$$
$$id \circ \gamma = \gamma$$
$$\gamma \circ id = \gamma$$

Rules for the functor

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

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

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

$$A\,[\gamma \circ \delta] = A\,[\gamma]\,[\delta]$$
$$A\,[id] = A$$
$$\alpha\,[\gamma \circ \delta] = \alpha\,[\gamma]\,[\delta]$$
$$\alpha\,[id] = \alpha$$

Rules for the terminal object

$$\Diamond : \mathrm{Ctx}$$

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

$$\langle\rangle \circ \gamma = \langle\rangle$$
$$id_\Diamond = \langle\rangle$$

Rules for context comprehension

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

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

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

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

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

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 [5]. 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 and a universe (ala Russell) if the following rules and laws are satisfied.

Rules for dependent function types

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

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

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

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

Rules for universe

$$\frac{\Gamma : \mathrm{Ctx}}{U : \mathrm{Ty}(\Gamma)}$$

$$\frac{\Gamma : \mathrm{Ctx} \quad A : \Gamma \vdash U}{A : \mathrm{Ty}(\Gamma)}$$

$$U\,[\gamma] = U$$

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

3

Unityped CwFs

The work starts by considering a cwf with only one type, a special case of the full cwfs that we call unityped cwfs (ucwfs). It is model of a theory of n-place functions. Cwfs model dependent type theories, but they can be modified by stripping type information away. This results in a model of n-place functions that involve substitution and compositions of functions.

This chapter begins with a formalization of the notions of ucwfs. Considering that we aim to construct different ucwfs, we need a common framework to talk about these objects and their properties. The following sections present the three main notions we are interested in, namely:

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

Then, we continue by constructing two ucwfs for each item in the above list. One with explicit substitutions that arises from translating the generalized algebraic theory of cwfs into combinators. And one with implicit substitutions and concrete variables. That amounts to a total of six different ucwfs. At each step, we build morphisms between them and prove that they are isomorphic. Moving from one ucwf to another is done by adding extra structure on the previous.

3.1 Ucwf Notions

In this section we present only the abstract notions of ucwfs we are interested in. A ucwf can be implemented in different ways, but the underlying structure must be clearly defined before one can build specific ucwfs.

3.1.1 Pure Ucwfs

Based on the formulation of cwfs as a generalized algebraic theory presented in *Internal Type Theory* [2] and summarized in the preceding chapter, one can take the special case of a single type. Subsequently, one sees that the core ucwf is a model of n-place functions that includes operations of substitution and composition of such functions. Places refer to independent inputs; for example, a binary function is 2-place. As in any GAT, it incorporates sort symbols, operator symbols, and equations between well-formed terms. Contexts and thus the objects of the base category become natural numbers with the terminal object being zero. 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. The rules for a ucwf are formalized as a record in Agda. **Remark**: for simplicity, the universe level of the record is not polymorphic, so the record is in Set_1 and the relations are in level zero. So the relations are merely instances of $\forall\ \{A : \mathrm{Set}\} \to (a_1\ a_2 : A) \to \mathrm{Set}_0$.

```
record Ucwf : Set₁ where
  field
    – Objects of the base category are natural numbers

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

    – Equality relations of terms and substitutions
    _≈_  : ∀ {n} → Rel (Tm n) lzero
    _≋_  : ∀ {m n} → Rel (Sub m n) lzero

    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 or weakening substitution
    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. This notion allows

variable-free formulations of ucwfs. Variables can be built by applying the substitution operation on the projection substitution p, successively. For example, one can built the *n*th 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. Further, the identity substitution id is a neutral element in composition and in substitution. It represents a sequence of variable indices ranging from 0 to $n - 1$. The theory also provides composition of substitutions and a substitution operation. A *cons*-like operation for extending a substitution is also found (it is a *cons* but with the element on the right side). Lastly, we have an empty substitution <>, which is also a left absorbing element under composition.

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. Further, we do not wish to restrict ourselves to propositional or some other notion of equality.

Continuing, the ucwf axioms can be expressed in the same 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 with 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 with 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 main sorts, the corresponding relations in which one reasons about them, the constructors, and proofs of the laws. This record describes what it means to be a ucwf in the category of ucwfs.

3.1.2 λ-Ucwfs

Utilizing the pure ucwf record defined earlier, we can add extra structure for λ abstractions and function application. In order for a ucwf to support lambdas and application we have to add the constructors and laws that must be satisfied. A new Agda record captures this notion.

```
record Lambda–ucwf : Set₁
  field
    ucwf : Ucwf
  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′
```

Note the new rules for substituting a term of application and lambda. This occurs because substitution is defined by its conversion laws, so stability under substitution is necessary. Another addition is two congruence rules for the new operators.

3.1.3 λ-$\beta\eta$-Ucwfs

Finally, as a last extension of ucwf notions, we can add beta and eta rules as extra structure and hence create a λ-$\beta\eta$ ucwf. Now this is model of λ-calculus up to beta and eta and a more interesting notion. The extension is done simply by expressing the beta and eta laws as axioms.

```
record Lambda-βη-ucwf : Set₁
  field
    lambda-ucwf : Lambda-ucwf
  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
```

This is a significantly more interesting ucwf with extra structure than the previous one that contained only the term language of the λ-calculus. However for implementation purposes, it was useful to first construct our proofs at the λ-ucwf level and then add the beta-eta laws at the end, although this intermediate step can be skipped.

3.2 Term Models of Ucwfs

In this section we begin constructing ucwfs; first, without extra structure, that is, we provide implementations for theories of n-place functions in accordance to the record of a pure ucwf. These are to be instances of ucwf. Hence, we implement two term models, one with explicit substitutions that uses ucwf combinators and one with implicit substitutions (meta-level operations). After building these two calculi, we show that they are ucwfs and proceed to construct ucwf morphisms between them to finally show the isomorphism.

3.2.1 Ucwf with Explicit Substitutions

The first ucwf we consider is one using ucwf combinators. In more detail, we translate the Ucwf record to a straighforward implementation of a variable-free calculus of

explicit substitutions. We want this to be a ucwf trivially, so the definitions should match the fields of the record precisely. Thus, we implement a term model for a ucwf 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
```

This language can be easily instantiated 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 relations of equality encapsulate all laws of the Ucwf record and so they add nothing new apart from symmetry and transitivity as they need to be equivalence relations. The equations are omitted but available in listing A.1 of the appendix.

In this calculus, variables are constructed using q (variable with de Bruijn index zero) and the projection substitution p. The projection substitution represents a sequence of variable indices ranging from 1 to n and since substituting q drops the substitution and keeps the last variable, every variable can be represented using explicit substitution. For example, q [p] [p] is the variable with index two, since we substitute with p twice.

The two accompanying equalities on terms and substitutions are equivalence relations by construction (reflexivity is derived). Therefore, Tm and Sub are setoid families and we formalize them as such in Agda. This will allow us to use Agda's equational reasoning packages from the standard library later for proving propositions about them. 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 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.

This ucwf instance should be initial in the category of ucwfs. While the project was not concerned with proving this result, it might be easier to see how this is initial given its construction.

In order to provide an example of how one can use this calculus to prove something, we show that there exists a unique substitution from any natural number to zero. 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} {_})
```

3.2.2 Ucwf with Implicit Substitutions

As mentioned earlier, a pure ucwf is a model n-place function theories. This idea can be implemented using a data type with variables using de Bruijn indices, so we have nameless terms. These variables are also indexed by a natural number. The variable 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 or scope safe. 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 framework.

```
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 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. We show the type of tabulate from the standard library.

```
tabulate : ∀ {n a} {A : Set a} → (Fin n → A) → Vec A n
```

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 and the other operations above with the same names.

Subsequently, one can show that this theory of functions fulfils the ucwf axioms by proving them for these concrete definitions. This is the last piece of completing the construction of a ucwf. Next, we present a series of lemmas that prove the ucwf generalized algebraic theory's laws for our variables and substitutions. We consider propositional equality as the underlying relations for both terms and substitutions for now.

Lemma 3.1. id {1 + n} ≡ (p , q)

Proof. Reflexivity. Recall that p is a sequence of indices $\langle 1, \ldots, n \rangle$ 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 side 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. $[] \circ \rho \equiv []$

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. $p \circ (\sigma , t) \equiv \sigma$

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))
```

□

The expression $p \circ (\sigma , t)$ is transformed to mapping a lookup on the sequence of $\langle 1, \ldots, n \rangle$, which should then drop the first element since the lookup begins from the index 1.

Lemma 3.7. $(\sigma \circ \gamma) \circ \delta \equiv \sigma \circ (\gamma \circ \delta)$

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 {n = suc n} (x :: σ) = begin
  id {1 + n} ∘ (σ , x) ≡⟨⟩
  (p , q) ∘ (σ , x)    ≡⟨⟩
  p ∘ (σ , x) , x      ≡⟨ cong (_, x) (pCons σ x) ⟩
  σ , x                ∎
```

□

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
```

□

The congruence rule proofs are trivial and thus omitted. Propositional equality is obviously an equivalence relation thus fulfilling all ucwf requirements. As a result, we now have a second ucwf with variables and substitutions as meta operations.

3.2.3 Isomorphism of Ucwfs

This section presents a proof that the two ucwfs defined earlier are isomorphic. By viewing both of these constructs as objects in the category of ucwfs, one can define ucwf morphisms between them. Morphisms are functions that map terms and substitutions 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 ucwf with explicit substitutions should be initial in the category of ucwfs; therefore, by showing it is isomorphic to the other one shows that the latter is also initial. This would

First, we discuss the formal definition of a ucwf morphism. It consists of two maps between the terms of substitutions of the involved ucwfs. Also, it comes with a set of laws that ucwf structure should be preserved strictly. We formalize this notion with a record.

```
record Ucwf−Morphism : Set₁ where
  field
    src : Ucwf
    trg : Ucwf
  open Ucwf src
  open Ucwf trg
– renamings are hidden
  field
    – maps from terms and substitutions
    ⟦_⟧  : ∀ {n} → TmS n → TmT n
    ⟦_⟧′ : ∀ {m n} → SubS m n → SubT m n

    – Ucwf structure is preserved
    id−preserved : ∀ {n} → ⟦ idS {n} ⟧′ ≋T idT

    q−preserved : ∀ {n} → ⟦ qS {n} ⟧ ≈T qT

    p−preserved : ∀ {n} → ⟦ pS {n} ⟧′ ≋T pT

    ∘−preserved : ∀ {m n k} (σ₁ : SubS k n) (σ₂ : SubS m k)
      → ⟦ σ₁ ∘S σ₂ ⟧′ ≋T ⟦ σ₁ ⟧′ ∘T ⟦ σ₂ ⟧′

    <>−preserved : ∀ {m} → ⟦ <>S {m} ⟧′ ≋T <>T

    <,>−preserved : ∀ {m n} (t : TmS m) (σ : SubS m n)
      → ⟦ < σ , t >S ⟧′ ≋T < ⟦ σ ⟧′ , ⟦ t ⟧ >T

    sub−preserved : ∀ {m n} (t : TmS n) (σ : SubS m n)
```

```
  → ⟦ t [ σ ]S ⟧ ≈T ⟦ t ⟧ [ ⟦ σ ⟧′ ]T
```

The fields have been renamed with suffixes to distinguish them. Source ucwf has an S and target ucwf a T.

Now we construct ucwf morphisms for our specific ucwfs. Several notational 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. A ucwf morphism is a map that preserves ucwf structure entirely, so every operation like composition, and substitution has to be preserved when using these maps.

```
– Bijections for terms
⟦_⟧  : ∀ {n} → Tm−λ n → Tm−cwf n
⟪_⟫  : ∀ {n} → Tm−cwf n → Tm−λ n

– Bijections for substitutions
⟪_⟫′ : ∀ {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 ⟧ >

– Ucwf morphism by definition
⟪ 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 projection substitution p. Intuitively, by composing projection substitutions, one increases the variable indices. It starts with the sequence $\langle 1, \ldots, n \rangle$ and upon lifting once, it becomes $\langle 2, \ldots, n+1 \rangle$. And since q picks the head of the substitution, q [p] is the variable with index 1. Moreover, the remaining explicit constructors are mapped to the meta-level operations described in section 3.2.2. We also observe that functions from the ucwf with explicit substitutions are mutually recursive.

It would be particularly nice if we had a formal notion of the whole category of ucwfs; however, it is not formalized in this thesis and, in fact, such a task has its own

challenges.

Next, the proof part is presented. To begin with, two integral properties are necessary. The ucwf morphism with Tm−λ as source needs to satisfy preservation of ucwf structure. The two following properties prove that substitution and composition are preserved.

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

Proof. Induction on t and σ.

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

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

□

More preservation properties are needed but they are proven simultaneously with the inverse lemmas after.

Subsequently, we continue with the inverse properties; they are shown as two pairs: terms and substitutions.

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

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

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

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

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). The substitution extension is just an application of the hypotheses.

Lemma 3.14. $t \equiv \langle\!\langle\ [\![\ t\ ]\!]\ \rangle\!\rangle$ *and* $\rho \equiv \langle\!\langle\ [\![\ \rho\ ]\!]'\ \rangle\!\rangle'$.

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))

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

□

The first inverse is thus completed.

On the other 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.

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 after. Next, we show the top level proofs.

Lemma 3.15. $t \approx [\![\ \langle\!\langle\ t\ \rangle\!\rangle\ ]\!]$ *and* $\gamma \approx [\![\ \langle\!\langle\ \gamma\ \rangle\!\rangle'\ ]\!]'$.

Proof. Induction on ρ and γ.

```
tm−cwf⇒λ q = refl≈
tm−cwf⇒λ (t [ us ]) = sym≈ $ begin
  ⟦ ⟪ t ⟫ [ ⟪ us ⟫′ ]λ ⟧      ≈⟨ []−preserv ⟪ 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 {_})

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)
```

□

The function p-inverse must show that $p \approx [\![p\text{-}\lambda]\!]'$, i.e., that projection is preserved. This 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 sequences of Fin elements so 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 $\langle 1, \ldots, n \rangle$; it is the projection substitution for indices. Now we can show that p is convertible to this sequence. The translation function constructs the equivalent of variables in Tm-cwf 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 {_} {_})
```

□

Because different versions of p were used, the second step of that proof was to show that all definitions used are the same this last bit is omitted as it involves tedious calculations with vectors.

Consequently, the functions are inverses of each other and hence we have an isomorphism of ucwfs.

3.3 Term Models of λ-Ucwfs

In this section, we add extra structure to our two ucwfs to allow the formation of lambda abstractions and applications. These are considered different objects in a different category altogether but the work done in the previous section is kept for the most part.

In fact, only proofs using induction have to be extended to account for new elements in the set of terms.

3.3.1 λ-Ucwf with Explicit Substitutions

The construction of a λ-ucwf with ucwf combinators is done by extending the Tm data type and adding introduction rules for the laws. The substitutions remain the same and we acquire a λ-ucwf by definition that should be initial in the category of λ-ucwfs.

```
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

data _≈_ where
  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 ])
...
```

Overall, this formulation bears resemblance to the $\lambda\sigma$ calculus (Abadi et al) [12]. 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 ↑. 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.

3.3.2 λ-Ucwf with Implicit Substitutions

Regarding the ucwf with traditional variables, we can add the new constructors which gives us the term language of the 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.

```
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
```

We proceed by defining the new substitution operation. Now we have to deal with lambda and apply. 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 to account for the binder. Further, weakening a term t is renaming the free variables in the Fin sequence $\langle 1, \ldots, n \rangle$.

Afterwards, we have to prove that this term model is a λ-ucwf. We keep the proofs for pure ucwfs and extend two lemmas.

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 ↑-dist is in listing A.2. As a result, we constructed another λ-ucwf with the usual lambda terms.

3.3.3 Isomorphism of λ-Ucwfs

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. A λ-ucwf morphism is a ucwf morphism where application and lambda structure are preserved as well. The functions we define are λ-ucwf morphism by definition.

Definition 3.1. λ-ucwf morphisms.

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

⟦ 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, the fact that substitution is preserved by the map needs additional cases lemma (3.12).

```
[]−preserv : ∀ {m n} t (σ : Sub−λ m n) → ⟦ t [ σ ]λ ⟧ ≈ ⟦ t ⟧ [ ⟦ σ ⟧′ ]
[]−preserv (t · u) σ = begin
  app ⟦ t [ σ ]λ ⟧ ⟦ u [ σ ]λ ⟧
    ≈⟨ cong−app ([]−preserv t σ) refl≈ ⟩
  app (⟦ t ⟧ [ ⟦ σ ⟧′ ]) (⟦ u [ σ ]λ ⟧)
    ≈⟨ cong−app refl≈ ([]−preserv u σ) ⟩
  app (⟦ t ⟧ [ ⟦ σ ⟧′ ]) (⟦ u ⟧ [ ⟦ σ ⟧′ ])
    ≈⟨ subApp ⟦ σ ⟧′ ⟦ t ⟧ ⟦ u ⟧ ⟩
  app ⟦ t ⟧ ⟦ u ⟧ [ ⟦ σ ⟧′ ]
    ∎
  where open EqR (TmSetoid {_})
[]−preserv (ƛ t) σ = begin
  lam ⟦ t [ ↑ σ ]λ ⟧
    ≈⟨ cong−lam $ []−preserv 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≈ (∘−preserv 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 are sufficient to extend the isomorphism to our λ-ucwfs.

3.4 Term Models of λ-$\beta\eta$-Ucwfs

Finally, we take the notion of λ-$\beta\eta$-ucwfs that models untyped λ-calculus with beta and eta and construct two such objects, morphisms, and isomorphism as before. This is an undoubtedly more interesting result.

Recall that the notion of λ-$\beta\eta$-ucwf is just a λ-Ucwf with beta and eta equalities. Therefore our variable-free calculus of explicit substitutions needs no changes apart from adding these beta laws in the equality relation of 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
```

And so by adding these equations to the relation, the λ-ucwf becomes a λ-$\beta\eta$-ucwf directly. Again, this object should be initial in the category of λ-$\beta\eta$-ucwfs.

On the other hand, the calculus with ordinary lambda terms requires significant changes. To begin with, we need a relation that describes beta-eta convertibility for lambda terms. Hitherto, we were using propositional equality but this is changed now. Next, we formalize the relation for beta-eta.

```
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)
```

Reflexivity is then derived.

```
refl~βη : ∀ {n} {t : Tm n} → t ~βη t
refl~βη = trans~βη (sym~βη (η _)) (η _)

refl≈βη : ∀ {m n} {ρ : Sub m n} → ρ ≈βη ρ
refl≈βη {ρ = []}  = ◇
refl≈βη {ρ = _ :: _} = ext refl~βη refl≈βη
```

Naturally, every property has to be proven at the level of this relation instead of propositional equality. However, given the preceding work that contains all require proofs for propositional equality, it is convenient to use them to lift a proof to this new relation.

```
≡–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 propositional equality proof. There is, however, some extra work in proving the congruence rules; these proofs can be found in listing A.3.

The morphisms for these objects are no different than before, but the underlying equalities have changed. By the same token, we can extend the isomorphism proof by using the propositional equality proofs for the λ-ucwfs.

4

Simply Typed CwFs

This chapter transitions to simply typed cwfs (scwfs). Scwf is a model of a theory of typed functions. Firstly, we formalize the notions of scwfs similarly to how we did for ucwfs. The three notions are again:

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

Following the same structure, we construct two instances for each notion. We have the variable free calculus of explicit substitutions that uses scwf combinators and a more traditional implementation with implicit substitutions. The former is always a scwf by construction. At each stage we build scwf morphisms and an isomorphism.

Generally, there are two main options of formalizing simply typed terms; one way is to implement typed terms that represent typing derivations of untyped terms. An example signature in Agda could 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. We refer to these as intrinsically typed terms.

On the other hand, one can start with untyped raw terms and add a typing relation that gives types to terms. For example, `data _⊢_∈_ (Γ : Ctx) : Raw → Ty → Set`, where `Raw` is the set of untyped terms (`Raw` could be the terms of chapter 3). This approach is called extrinsic for the obvious antithesis with the former version.

Considering all this, we start by formalizing intrinsic scwfs and adding extra structure as usual for lambdas and applications and concluding with beta and eta. This follows an identical pattern and structure to the untyped work. In this chapter there is extra content because we also formalize extrinsic scwfs based on the untyped terms of chapter 3. Overall, three different isomorphisms between three forms of scwfs will be shown in this chapter. Consider the following figure that depicts the landscape in which we operate in.

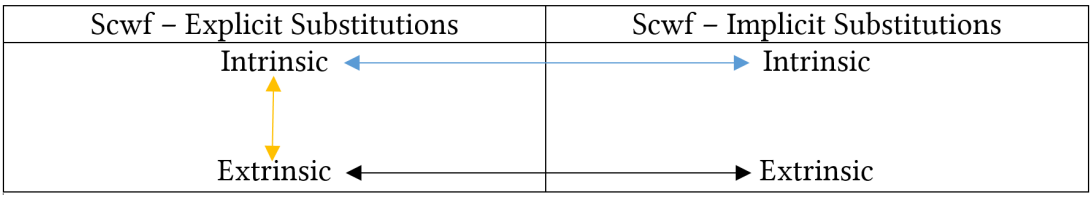

Figure 4.1: Scwf variations explored.

Figure 4.1 shows different possibilities one can explore. On the left column we have scwfs that are implemented as calculi of explicit substitutions, whereas on the right we have traditional versions with substitutions formalized as meta-level operations.

First, the connection depicted by the blue arrow in the diagram is investigated. This formalization is performed again on three levels, starting from pure and adding extra structure. Then, we move to extrinsic scwfs (black arrow) and finally we discuss extrinsic and intrinsic scwfs, both defined by calculi of explicit substitutions.

4.1 Scwf Notions

4.1.1 Pure Scwfs

The intrinsic way to build a scwf that models typed functions is based on the generalized algebraic theory of cwfs. The new features include a set of contexts and types. Moreover, substitutions are indexed by two contexts and terms are indexed by context and type. This notion is formalized as an Agda record.

```
record Scwf : Set₁ where
  field
    – types, contexts, terms, and substitutions
    Ty  : Set
    Ctx : Set
    Tm  : Ctx → Ty → Set
    Sub : Ctx → Ctx → Set

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

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

    – empty context (terminal object)
    ◇   : Ctx

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

    – 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 $\diamond$ and context extension that adds a 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. The field q refers to the last variable (or last assumption) and variables are constructed by weakening q. A substitution 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 scwf 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′
```

The equations carry type information, but other than that, there is no change in comparison to the ucwf equations.

4.1.2 λ-Scwfs

We add extra structure to the notion of scwf to accommodate lambda abstractions and function application. This yields a new notion of λ-scwfs that contains the term language of simply typed λ-calculus.

```
record Lambda−scwf : Set₁ where
  field
    scwf : Scwf
  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 equations expressing stability under substitution are also included, plus congruence closure.

4.1.3 λ-$\beta\eta$-Scwfs

The final extension is the notion of λ-$\beta\eta$-scwfs. This construct builds on the two previous by adding beta and eta equalities as laws. Again formalized as a record

```
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
```

This notion scwf with extra structure is a model of simply typed λ-calculus that is presented with beta and eta.

4.1.4 Extrinsic Scwfs

The notion of extrinsic scwf is another way of describing a scwf in general. Extrinsic formulations start with raw syntax and add external typing relations that provide types to terms. The developments of chapter 3 used untyped terms who can act as raw syntax for an extrinsic scwf. This notion can be formalized as a record too; the differences will becomes apparent.

```
record Scwf : Set₁ where
  field
    ucwf : Ucwf
  open Ucwf.Ucwf ucwf
  field
    – Types
    Ty : Set

    – Contexts
    Ctx : Nat → Set
    ε  : Ctx 0
    _•_ : ∀ {n} → Ctx n → Ty → Ctx (suc n)

    – typing relation - terms
    _⊢_∈_ : ∀ {n} (Γ : Ctx n) (t : RTm n) (α : Ty) → Set

    – typing relation - substitutions
    _▷_⊢_ : ∀ {m n} (Γ : Ctx n) (Δ : Ctx m) (ρ : RSub n m) → Set

    – sigma pairs of raw with typing rule

    – identitiy
    id–ty : ∀ {n} {Γ : Ctx n} → Σ (RSub n n) (Γ ▷ Γ ⊢_)

    – composition
    ∘–ty : ∀ {m n k} {Γ : Ctx n} {Δ : Ctx m} {Θ : Ctx k}
      → Σ (RSub n k) (Γ ▷ Θ ⊢_)
      → Σ (RSub m n) (Δ ▷ Γ ⊢_)
      → Σ (RSub m k) (Δ ▷ Θ ⊢_)

    – last variable
    q–ty  : ∀ {n} {Γ : Ctx n} {α} → Σ (RTm (suc n)) (Γ • α ⊢_∈ α)
```

```
  – projection substitution
  p–ty  : ∀ {n} {Γ : Ctx n} {α} → Σ (RSub (suc n) n) (Γ • α ▷ Γ ⊢_)

  – empty substitution
  <>–ty : ∀ {n} {Γ : Ctx n} → Σ (RSub n 0) (Γ ▷ ε ⊢_)

  – extend substitution
  <,>–ty : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} {α}
    → Σ (RSub m n) (Δ ▷ Γ ⊢_)
    → Σ (RTm m) (Δ ⊢_∈ α)
    → Σ (RSub m (suc n)) (Δ ▷ Γ • α ⊢_)

  – substitution operation
  sub–ty : ∀ {m n} {Γ : Ctx n} {Δ : Ctx m} {α}
    → Σ (RTm n) (Γ ⊢_∈ α)
    → Σ (RSub m n) (Δ ▷ Γ ⊢_)
    → Σ (RTm m) (Δ ⊢_∈ α)
```

Note how we use a ucwf to define a scwf. This means that equality and generally conversion is at the raw level. In this extrinsic formulation we want to capture scope safe terms, so contexts are also indexed by a natural number. We also see two sets of typing rules: one for terms and one for substitutions. Consequently, to formulate that we need the cwf operators to be well-typed we have to present a raw term or substitution with its respective typing rule. For this reason we use sigma pairs; this means that a well-typed term is the raw term paired up with its typing rule.

This is the only extrinsic notion we show in this section, if we want extra structure, we merely have to add the corresponding ucwf and add sigma pairs for lambda and application along with a function type.

4.2 Term Models of Scwfs

4.2.1 Scwf with Explicit Substitutions

Having defined the abstract notion of intrinsic scwf as a record, we move on to scwf construction. A straightforward scwf with explicit substitutions is implemented as a data type where all operators and laws are explicit in accordance to the record. First, assume some set for types and define contexts as lists of types.

```
postulate Ty : Set

data Ctx : Set where
  ε  : Ctx
  _•_ : Ctx→ Ty → Ctx

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 listing A.4 of the appendix. This is a scwf by definition and should be initial in the category of scwfs.

4.2.2 Scwf with Implicit Substitutions

A second intrinsic scwf is constructed with concrete variables. The formulation we chose uses contexts as inductive lists of types accompanied with relations of inclusion and membership. The variables are to be membership witnesses in the context they are typed in. Regarding equality, propositional equality is considered for now.

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

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

Contexts are polymorphic and we define a set Ctx which instantiates a context with Ty. The set of types is postulated. Moreover, we show 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
```

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

Moving to well-typed substitutions, they are defined sequences of terms typed in some context; we use the standard library's product type to implement 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 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

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

– identity substitution
id : ∀ {Γ} → Sub Γ Γ
id {ε}  = tt
id {Γ • α} = p , 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

– 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. Sub is implemented as a product; this means that extending a substitution is done using the pair constructor for products _,_. Moreover, the empty substitution is tt, the single element of top (unit). This is because when the environment of a substitution is empty, we return top ⊤, the singleton set.

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

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 = begin (
  tkVar v (wk−sub Γ φ p)
    ≡⟨ cong (tkVar v) (▷−wk−2 Γ ⊆−• φ id) ⟩
  tkVar v (wk−sub Γ (⊆−trans ⊆−• φ) id)
    ≡⟨ tkVar−wk−id v (⊆−trans ⊆−• φ) ⟩
  var (sub−in (⊆−trans ⊆−• φ) v)
    ≡⟨ cong var (sub−in−step φ v) ⟩
  var (sub−in φ (there 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. $\mathrm{id}_0 : \mathrm{id}\ \{\varepsilon\} \equiv \mathrm{tt}$

Proof. Reflexivity.

```
id₀ : id {ε} ≡ tt
id₀ = refl
```

□

Lemma 4.5. $\mathrm{tt} \circ \rho \equiv \mathrm{tt}$

Proof. Reflexivity.

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

□

Lemma 4.6. $\mathrm{p} \circ (\gamma , \mathrm{t}) \equiv \gamma$

Proof. Mutually with 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. $(\gamma \circ \delta) \circ \zeta \equiv \gamma \circ (\delta \circ \zeta)$

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. $\mathrm{q} [\rho , \mathrm{t}] \equiv \mathrm{t}$

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. $(\gamma , \mathrm{t}) \circ \delta \equiv (\gamma \circ \delta , (\mathrm{t} [\delta]))$

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))
```

□

Conclusively, we showed that these variables as membership proofs with typed substitutions form an intrinsic scwf.

4.2.3 Isomorphism of Scwfs

Hitherto, we have defined two intrinsically typed scwfs. The calculus of explicit substitutions and the traditional one with variables as membership proofs. Now we proceed to define scwf morphisms between these objects for the isomorphism.

A scwf morphism is a map that preserves structure for all scwf operators strictly. We will not present a formalization of a scwf morphism, but the structure preservation properties are as they were shown for ucwf morphisms but with type information. Moreover, in our implementation, both scwfs used the same sets of contexts and types for brevity. Technically, it would have been more appropriate to define different sets for contexts and types and then have maps between them as well, but these sets would be isomorphic by definition anyway.

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 equivalent lemmas of 3.12 and 3.13 which are similarly vital are in listing A.5 of the appendix. Maps preserving structure for substitution and composition.

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

∘−preserv : ∀ {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. $\langle\!\langle\ [\![\ t\ ]\!]\ \rangle\!\rangle \equiv t$ *and* $\langle\!\langle\ [\![\ \gamma\ ]\!]'\ \rangle\!\rangle' \equiv \gamma$

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

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

□

Lemma 4.13. $[\![\ \langle\!\langle\ t\ \rangle\!\rangle\ ]\!] \approx t$ *and* $[\![\ \langle\!\langle\ \gamma\ \rangle\!\rangle'\ ]\!]' \approx\!\!\approx \gamma$.

Proof. Induction on γ.

```
tm−cwf⇒λ q = refl≈
tm−cwf⇒λ (t [ γ ]) = begin
  [[ ⟪ t ⟫ [ ⟪ γ ⟫' ]λ ]]
    ≈⟨ []−preserv ⟪ 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 {_})
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)
```

□

The projection (p–inverse) case has a very similar structure to the one presented for unityped cwfs. There, we used sequences of Fins to construct the normalized p. Using this idea here for simple types, we need sequences of witness proofs that a type is in a given context. The complete proof is in listing A.6.

4.3 Term Models of λ-Scwfs

4.3.1 λ-Scwf with Explicit Substitutions

To obtain a λ-scwf for our variable free formulation with scwf combinators, we add a function type, two constructors, and the laws for substitution stability. Sub is not altered.

```
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 _≈_ : ∀ {Γ α} (t₁ t₂ : Tm Γ α) → Set where
  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 > ])
  ...
```

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.

4.3.2 λ-Scwf with Implicit Substitutions

Having typed variables as membership witnesses in contexts already, we add the same constructors here as well. This gives us the term language of simply typed lambda calculus.

```
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} (v : 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∈ 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 Tm constructors. The λ case is interesting where the substitution ρ is weakened and extended because a lambda abstraction has a binder.

```
_[_] : ∀ {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: (i) substituting in id does not affect terms and (ii) 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 side 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) ] ∎
```

□

All properties from our scwf with implicit substitutions are kept.

4.3.3 Isomorphism of λ-Scwfs

With the intention to extend the isomorphism to λ-scwf, the morphisms have to be extended to account for new elements. Fortunately, they preserve lambda and application structure by definition.

```
⟦_⟧  : ∀ {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 scwf to the other. Therefore, the inverse proofs follow directly from 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-scwfs.

4.4 Term Models of λ-$\beta\eta$-Scwfs

Both λ-$\beta\eta$-scwfs will be presented here as the extensions are minor and be explained succinctly. Regarding the λ-$\beta\eta$ scwf with explicit substitutions, the two equations are added as laws in the equality relation for terms like so.

```
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
```

As a result the λ-scwf becomes a λ-$\beta\eta$-scwf directly.

On the other hand, for the implicit substitutions more work is needed. First, a relation that defines beta-eta convertibility is necessary. This axiomatization simply typed λ-calculus up to beta-eta is formalized. We also need a relation for 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 propositional equality. Hence, it is easy to lift those proofs to 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 of propositional equality. Congruence rules can be found in listing A.7.

This presentation is identical to the ucwfs with beta and eta. There are no new morphisms, we merely have to express all properties using the new relations.

4.5 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 with explicit substitutions. While the scwf with meta-level operations requires a number of proofs that verify the preservation of types for scwf operations. In other words, all typing rules of the explicit scwf have to be proven. We are basing the result on proofs at the raw level of chapter 3.

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

The raw grammar of terms and substitutions is recapitulated.

```
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 $\Gamma \vdash t \in \alpha$ 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 $\Gamma \triangleright \Delta \vdash \rho$ 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 listing A.1.

These raw terms and their typing rules represent a scwf implemented the extrinsic way. We could pair these using sigma pairs according to the extrinsic notion of scwf.

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 with extra structure. Now we return to variables as indices and substitutions as vectors. We start by recalling the grammar for raw lambda 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
```

Substitutions as vectors have the definitions of section 3.2.2. In short, a substitution Sub n m is a vector of length n that contains terms with at most m variables. For example, id is a Sub n n and contains variables with indices 0 to $n - 1$.

First, we define contexts as vectors too; this allows us to perform a lookup using 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 preserves types 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 typing rules are next.

Lemma 4.16. $\Gamma \triangleright \Gamma \vdash \mathrm{id}$

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. $\Gamma \triangleright \Gamma \bullet \alpha \vdash p$

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. $\Gamma \vdash t \in \alpha \to \Gamma \triangleright \Delta \vdash \rho \to \Delta \vdash t\ [\ \rho\ ] \in \alpha$

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 a λ-scwf hold for these lambda terms with de Bruijn variables. Thus it forms an extrinsic λ-scwf. This also means that the two λ-scwfs are isomorphic since we have a proof already at the raw level.

Finally, to officially present this as an extrinsic λ-scwf, 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, so they equivalent pairs can be formed as shown.

Conclusively, this demonstrates the extrinsic point of view of scwfs. We reflect on this formulation in comparison to 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 was pleasant. In fact, the top level proofs were identical to the untyped ones. For simply typed scwfs, both methods are quite manageable; whereas for dependent types, the same cannot be said.

4.5.1 Extrinsic and Intrinsic Scwfs

This subsection summarizes another important theorem in the simply typed world, namely, that the intrinsic and extrinsic scwf with explicit substitutions 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. 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 the previous section and the intrinsic calculus of explicit substitutions that have types built-in.

First, one has to define the translation functions between these scwf. 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 and preserve structure by definition. We show only maps for term.

```
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 easy to prove that these are inverses of each other. 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 scwf laws as introduction rules. Here the equalities $\approx'$ and $\approx\!\!\approx'$ refer to the raw level relations for terms and substitutions while the other non-primed refer to the intrinsically 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, one with explicit

constructors and one with implicit was formalized. A proof of the latter being a scwf was implemented and subsequently, scwf morphisms between these two objects were defined and proven to be strict inverses.

Secondly, 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. Following this structure, it was necessary to define explicit typing relations for the scwf, i.e., a typing rule for each scwf combinator. The final point was to show that types are preserved for all scwf operators.

Lastly, a comparison of scwfs with explicit substitutions closed this chapter. It described the same connection, but this time between intrinsic and extrinsic scwfs.

5

ΠU-CwFs

In this chapter, a version of the full 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 will be extrinsically typed, since in a dependently typed setting it is much easier.

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 with de Bruijn indices. The notion is not formalized, so no Agda record that describes the components of a ΠU-Cwf; instead we start constructing two objects directly.

5.1 ΠU-Cwf with Explicit Substitutions

To remain consistent with the order of concepts presented, we start with the formalization of the object with explicit cwf combinators. 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 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. 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.

The rules for terms having a type

```
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, it only makes sense if Γ is a well-formed context and A is a correct type in Γ. Hence, the following admissible 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 listing A.8).

Overall, this description defines a ΠU-Cwf by construction. It is definitely not clear-cut to say that this is the initial object in the category of ΠU-Cwfs, albeit it is a candidate.

5.2 ΠU-Cwf with Implicit Substitutions

In this section we construct a dependently typed λ-calculus with substitution as a meta function. The goal is to construct another ΠU-Cwf starting with raw terms and building a type system. 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 a few 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 3.2.1.

```
– 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 isomorphism of section 3.3.3 is extended with the proofs in appendix A.9.

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)
```

We now have a specific rule for variables, as opposed to the variable-free calculus and the rules for substitution of types and terms are missing. Moreover, the rules for composition, projection, and identity are also not present. These are meta-level operations so they have to be proven given this framework. Other than that, the remaining rules are the same.

The formalization of the proofs of typing rules is not presented as it not completed yet; it is a work in progress at the moment. Alas, we conclude this chapter with just the construction of the second object which should be a ΠU-Cwf. In more detail, to prove that this term model with implicit substitutions is a ΠU-Cwf, the typing rules for the following operations have to preserved: substitution (for term and type), identity, projection, and composition. Upon proving those, we also have an isomorphism between these extrinsic ΠU-Cwfs because the raw level isomorphism was extended to cover pi types and a universe.

6
Conclusion

In summary, this thesis was primarily concerned with formalizing various formulations of categories with families in the Agda proof assistant. The full categories with families provide a model for a basic framework of dependent types, but we considered the unityped and simply typed cwfs as starting points. The implementation consisted of first, defining different notions of cwfs and second, constructing two instances, two term models for each. One instance was built using cwf combinators and explicit substitutions. The intention with this first object was to make it a cwf by construction. The objects using this formulation should be initial in the respective category, although a proof of initiality was not an objective of this thesis. The second term model was implemented using implicit substitutions, that is, operations formalized as meta-level functions. This was a more traditional approach where we saw ordinary lambda calculi in their typical representation. Furthermore, cwf morphisms between these instances were defined and an isomorphism was constructed at each level. Thus, we constructed isomorphisms of initial objects, assuming initiality of the term model with explicit substitutions.

A number of different notions and isomorphisms were discussed. In total we have eight isomorphisms fully formalized.

- Three for ucwfs: ucwf, λ-ucwf, and λ-$\beta\eta$-ucwf.
- Three for intrinsic scwfs at the same three levels.
- One between extrinsic λ-$\beta\eta$-scwfs.
- One between an extrinsic and intrinsic λ-$\beta\eta$-scwf.

Ideally, we would have liked to have completed all typing rules for the ΠU-Cwf term model of implicit substitutions to obtain an isomorphism for dependent types too. Consequently, it now falls under future work. Moreover, a more formal approach to the categorical context surrounding cwfs would be an excellent addition to our developments. In particular, we presented no formalization of the underlying cwf categories and morphisms we discussed throughout the paper, so having these notions formalized would, in some sense, unify everything in this thesis. Our notions as records defined potential objects of a category formalization. Finally, a formalized proof of initiality for the term models with explicit substitutions would also improve the results presented in this thesis.

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] 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.

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

[4] 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.

[5] A. Abel, T. Coquand, and P. Dybjer, "On the algebraic foundation of proof assistants for intuitionistic type theory," in *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings*, pp. 3–13, 2008.

[6] J. Chapman, "Type theory should eat itself," *Electr. Notes Theor. Comput. Sci.*, vol. 228, pp. 21–36, 2009.

[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] G. Allais, "Formalizing nbe in agda." https://github.com/gallais/agda-nbe, 2012.

[10] U. Norell, *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[11] J. Cartmell, "Generalised algebraic theories and contextual categories," *Annals of Pure and Applied Logic*, vol. 32, no. Supplement C, pp. 209 – 243, 1986.

[12] 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.

A

Agda Code

This appendix contains listings of Agda code that are referenced in the report. It contains only important parts of the formalization and supplements the content presented in the main body.

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

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
```

Listing A.1: Equalities for ucwf with explicit substitutions.

```
↑−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)
    ∎
```

Listing A.2: 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′))
```

Listing A.3: Congruence rules for untyped beta-eta equality.

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

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″
```

Listing A.4: Equations for scwf with explicit substitutions.

```
[]−preserv {ε} (var ()) tt
[]−preserv (var here) (ρ , t) = sym≈ (qCons ⟦ t ⟧ ⟦ ρ ⟧′)
[]−preserv (var (there ∈Γ)) (ρ , t) = begin
  ⟦ tkVar ∈Γ ρ ⟧
    ≈⟨ []−preserv (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 {_})

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

[]−preserv {Γ} (ƛ {α = α} t) ρ = begin
```

```
    lam ⟦ t [ wk−sub Γ ⊆−• ρ , var here ]λ ⟧
      ≈⟨ cong−lam ([]−preserv 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−<,> ([]−preserv t σ) refl≋ ⟩
  < ⟦ ρ ⟧′ ∘ ⟦ σ ⟧′ , ⟦ t ⟧ [ ⟦ σ ⟧′ ] >
    ≈⟨ sym≋ (compExt ⟦ t ⟧ ⟦ ρ ⟧′ ⟦ σ ⟧′) ⟩
  < ⟦ ρ ⟧′ , ⟦ t ⟧ > ∘ ⟦ σ ⟧′
    ∎
  where open EqR (SubSetoid {_} {_})
```

Listing A.5: 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≋
```

Listing A.6: Projection is preserved from scwf morphisms.

```
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′)))
```

Listing A.7: Congruence rules for simply typed lambda calculus with beta and eta.

```
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 (⊢<> ΔΓ⊢) = c−emp , ΔΓ⊢
lemma−3 (⊢<, > ⊢γ ⊢A _) = c−ext (lemma−1 ⊢A) ⊢A , π₂ (lemma−3 ⊢γ)

lemma−1 (ty−U ΓΓ⊢) = ΓΓ⊢
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 ΓΓ⊢ _ = ΓΓ⊢

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′
```

Listing A.8: Inversion lemmas for Π*U* cwf calculus explicit substitutions.
An isomorphism between the raw languages of the two term models of ΠU-cwfs.
The proofs involving terms are shown since they have been solely extended.

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

⟦_⟧′ : ∀ {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≈
```

Listing A.9: Raw isomorphism of well-scoped terms with Π and U.