A Framework for Modular Syntax and Proofs in Agda

Manuel Brea Carreras

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2023

Abstract

Proof assistants have the potential to greatly aid researchers in programming language theory, but their use remains hard and time-consuming. One aspect of this is the lack of elegant component reuse, forcing researchers to copy and paste definitions and proofs, patching as necessary in an ad-hoc manner. This causes a proliferation of duplicated code, which is a practical issue for both researchers and writers of teaching materials, as it makes proof developments hard to maintain. Some approaches to writing modular datatypes, predicates and proofs about them have been presented, but most are specialized for the Coq proof assistant. I present an improvement over the previous-best published approach to modular proofs in the Agda proof assistant in order to bridge the gap between Coq and Agda in this regard. I demonstrate the usefulness of this approach by implementing a modular version of the PCF calculus—which supports first-class functions, natural numbers as a primitive and recursion—and its proof of type-safety as a case study, which was not possible with the approach I improve upon. Finally, I explore the limitations of the approach with an application to the meta-theory of gradually typed languages.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Manuel Brea Carreras*)

Acknowledgements

Thank you to my supervisor, Prof Philip Wadler, for the valuable feedback and guidance provided throughout this project.

Table of Contents

1 Introduction **1**

1.1 My Contribution 4

1.2 Expected Background 5

2 Background **6**

2.1 Monolithic Implementation and the Expression Problem 6

2.2 Data Types à la Carte 7

2.3 Modular Type-Safety Proofs in Agda 9

3 Proposed Methodology **13**

3.1 Modular Syntax 13

3.2 Modular Recursive Functions 14

3.3 Structuring the Project 17

3.4 Delayed Lifting 20

3.5 Modular Predicates 21

3.6 Type Preservation 23

4 Case Study: PCF **25**

4.1 Type Syntax 25

4.2 Expression Syntax 27

4.3 Context and Lookup Relation 27

4.4 Typing Relation 27

4.5 Values 28

4.6 Substitution 29

4.7 Reduction Relation 31

4.8 Preservation 32

4.9 Progress 35

5 Limitations: Cast Calculus **38**

6 Conclusions and Future Work **40**

Bibliography **41**

A Additional Listings **45**

A.1 Helper Script for Instantiating Language Variant 45
A.2 Helper Script for Checking Instantiated Variant 46
A.3 Modular PCF Listings 47
A.3.1 Expression Syntax 47
A.3.2 Context 48
A.3.3 Lookup Relation 48
A.3.4 Typing 49
A.3.5 Substitution 51
A.3.6 Value 57
A.3.7 Reduction Relation 58
A.3.8 Preservation 60
A.3.9 Progress 62

Chapter 1

Introduction

Research in programming language theory usually involves long proofs about a language definition. The difficulty in these proofs often stems from their length and tedium, which makes it hard to keep track of necessary changes to a proof whenever the corresponding language definition changes (Aydemir et al., 2005).

Proof assistants are well-equipped to help with these proofs. First, they enforce consistency between definitions and proofs about them. Second, they provide a very high degree of trust: if a proof assistant accepts a proof, the researcher can be confident that the statement is true. Lastly, they enable new avenues for automatic verification of a compiler implementation: it is possible to prove correctness of programs in cases where verification via testing would be computationally expensive (Cockx et al., 2022).

The POPLMark (Aydemir et al., 2005) and POPLMark Reloaded (Abel et al., 2019) challenges provide benchmarks to evaluate and compare the usability of proof assistants, with the ultimate goal of driving adoption. One aspect they consider is *component reuse*, that is, the ability to reuse parts of definitions and proofs. Unfortunately, this is an unsolved problem (Forster and Stark, 2020). Reuse of proofs is commonly done through copy-pasting code and patching parts of it as needed. Besides being inelegant, developments written in this way are hard-to-maintain due to the large amount of code duplication.

This is not just a theoretical problem: it is a common pattern in both academic research and teaching materials for proof developments about the theory of programming languages to be based on a well-known *base language* with an interesting component added to it. The focus is on the new component, but the whole language must be formalized. Academic examples are the multitude of cast

and blame calculi in the theory of gradual typing (Siek and Chen, 2021; Siek and Taha, 2006, 2007; Siek, Thiemann, et al., 2021; Siek, Vitousek, et al., 2015), which often use a STLC (Simply Typed Lambda Calculus) (Pierce, 2002) as the base. In teaching, the problem is even more exacerbated: if an author wishes to iterate on the formalization of the base language in their textbook, they will need to also update all subsequent chapters that add constructs to that base language. Chapter *More* of PLFA, for example, (Wadler et al., 2022) introduces a variety of additional language features to PCF (Programming Computable Functions), which they use as their base language. In Types and Programming Languages (Pierce, 2002), while not using a proof assistant, the problem is clear: many chapters consider some additional interesting construct added to the STLC. In fact, the authors developed the specialized preprocessor TinkerType (Levin and Pierce, 2003) to generate the multiple definitions and keep them consistent; however while the authors speculate on ways this tool can be integrated with proof assistants, they do not attempt to do this.

To explore possible solutions to this problem in proof assistants, we must first consider the simpler case in traditional programming languages. In this context, Wadler (1998) named it the *expression problem.* In his own words:

> "The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."

The proposed solutions to the problem in proof assistants find their roots in Data Types à la Carte (DTC) (Swierstra, 2008), which uses Haskell. In DTC, a monolithic datatype is broken down into smaller *features*, each of them represented as a functor. Swierstra (2008) gives the example of integer and addition features, represented as:

```
data Val e = Val Int
data Add e = Add e e
```

Features are combined via the functor co-product **:+:**, and the expression type is instantiated as the fixpoint of the feature functors:

```
data Fix f = In (f (Fix f))
type Expr = Fix (Val :+: Add)
```

With some additional machinery for defining usable constructors and an interface to define recursive functions over the modular datatype, this approach elegantly solves the expression problem in Haskell. Unfortunately, a naive translation of this technique into a proof assistant will be rejected: the **Fix** datatype will fail the strict positivity check, and the suggested approach of defining recursive functions over the modular datatype will fail the termination check.

These issues are significant. Proof assistants generally work by taking advantage of the Curry-Howard correspondence (Pierce, 2002). Informally, this highlights a correspondence between logic systems and programming languages: every logical statement corresponds to a type in the programming language, and every proof of the statement corresponds to a procedure that constructs an element of the type. Thus, in order to ensure the logic of a proof assistant is consistent (i.e. free of contradictions), we must make sure it is impossible to construct an element of the empty type. Non-terminating functions and non-strictly-positive datatypes can be used to construct elements of the empty type, so they are forbidden. However, since it is undecidable to determine whether a program is terminating, these checks are generally very conservative and reject many terminating programs.

Exploration on how to solve the expression problem in proof assistants has been limited, and since the first attempt over ten years ago, we mainly have four approaches to consider; all of them are adaptations of DTC. First, Schwaab and Siek (2013) suggest appeasing the strict positivity checks by replacing the general formulation of functors with the more restrictive class of *polynomial functors*. This works, but at the cost of limiting the expressiveness of the approach. On top of this, they do not manage to solve the issue with the termination checker.

Delaware et al. (2013) and Keuchel and Schrijvers (2013) respectively use Church encodings and techniques from Datatype-Generic Programming to define appropriate fixpoint types, folds and algebras over them (as an alternative to recursive functions). This resolves both the strict-positivity and termination errors while being more expressive than the approach of Schwaab and Siek (2013). However, the resulting definitions are lengthy and hard to understand for non-experts (Forster and Stark, 2020).

Finally, Forster and Stark (2020) present a new observation. Rather than instantiating the fixpoint of functors dynamically (i.e. at runtime), we can do it statically (i.e. before compile-time). In the context of the previous Haskell example, this would be done by writing:

```
data Expr = InVal (Val Expr) | InAdd (Add Expr)
```

Translated to a proof assistant, this would pass the strict positivity check because the compiler is able to statically check that both the functors **Val** and **Add** are strictly-positive, whereas it knows nothing about an arbitrary functor argument f. Forster and Stark (2020) then provide an external tool—an extension to Autosubst 2 (Kaiser et al., 2017)—to generate such definitions with less boilerplate, and use MetaCoq (Sozeau et al., 2020) commands to aid in the definition of modular functions and proofs. The resulting approach is very expressive, and results in modular code closer to the monolithic version than previous alternatives.

Out of these four approaches, only Schwaab and Siek (2013) targets Agda, with the remaining three being written for Coq. This fact is not insignificant for an Agda programmer: Delaware et al. (2013) and Keuchel and Schrijvers (2013) each provide more than 1000 lines of supporting code that would need to be translated to Agda, with no guarantee that it would be possible due to the slight theoretical differences in the two languages. The approach from Forster and Stark (2020) suffers worse from this, as it relies on an external tool that is specialized for Coq, as well as in Coq meta-programs.

1.1 My Contribution

I provide a new approach that bridges the gap between Coq and Agda in modular proof techniques. My approach improves upon Schwaab and Siek (2013) in two ways. First, it solves the lack of expressiveness by replacing the dynamic fixpoint of polynomial functors with statically instantiated modular datatypes, similarly to Forster and Stark (2020). The side effect of this change is that the resulting definitions are more idiomatic and easy to read. Second, I fix the issue with termination using an external Agda preprocessing tool built from scratch for this project. This tool inlines the contents of one module into another, which when used in tandem with consistent use of module parameters lets Agda recognize the mutual induction present in our modular functions and prove termination. As an addendum, I provide some build scripts that help with the use of this tool.

In order to demonstrate the usefulness of my approach, I use it to formalize a modular version of PCF (Wadler et al., 2022) and its proof of type-safety. PCF is a simple functional language that can be broken down into four modular features:

lambda abstractions, variables, fixpoint construction and natural numbers (acting as a base type for the language). Lambda abstractions enable first-class functions, and the fixpoint construction enables recursion. A subset of these features—lambda abstractions, variables and naturals—constitutes the STLC, which is also a very common *base language* in the literature and teaching materials. Nonetheless, PCF still is a non-trivial language with variable binding, substitution and support for recursion. The widespread use and interesting features make it an appropriate case study for my approach.

Finally, I explore the applicability of my approach to a version of the internal cast calculus (Siek and Taha, 2006; Siek, Vitousek, et al., 2015), which raises an old software engineering question in a new context: What are the limits of modular proof techniques, and do we need to design languages with proof modularity in mind?

1.2 Expected Background

Ideally, the reader would be familiar with functional programming, programming language theory and its formalization in proof assistants—preferably Agda—at the level of an introductory course in the topic: Programming Language Foundations in Agda (Wadler et al., 2022) is the text I use as reference.

While I have aimed to make the explanations as accessible as possible, the topics at hand often require lengthy explanations even when targeted at other researchers in the same field (Forster and Stark, 2020; Schwaab and Siek, 2013; Siek and Chen, 2021; Swierstra, 2008). Adding any further background would detract from the coverage of my own contributions.

Chapter 2

Background

The development of my approach is easier to understand with appropriate context on the expression problem, the methods that my approach improves upon and their limitations. With that in mind, this chapter contains an overview of the expression problem, how Swierstra (2008) aimed to solve it, the refinements made by Schwaab and Siek (2013) to make it work in Agda, and the limitations of the latter method.

To turn the presentation into a cohesive story, I will be using the same calculus throughout this chapter and my own development in Chapter 3. This calculus is a simple arithmetic expression language with natural numbers and booleans as two separate features; the full specification is shown in Figure 2.1. Note that while this language is not very practical, it serves as a concise example on which to demonstrate the concepts I will be discussing. Discussions of existing methods will only work with the expression syntax and recursive functions over it—this is enough to highlight the limitations—, but for my own method I will discuss modular predicates, relations and proofs. Lastly, unless otherwise stated, all remaining code will be in Agda.

2.1 Monolithic Implementation and the Expression Problem

The most straightforward way of implementing the syntax for our arithmetic language is to define a single monolithic datatype. Suppose we begin by implementing the syntax for natural numbers and addition:

```
data Expr : Set where
  nat : ℕ → Expr
  _∔_ : Expr → Expr → Expr
```

Consider now that we implement a simple recursive function over this datatype which counts the number of natural number literals in the expression:

```
count : Expr → ℕ
count (nat n) = 1
count (e₁ ∔ e₂) = (count e₁) + (count e₂)
```

If we now wish to go back and extend Expr with the syntax for booleans, our only option is to patch both Expr and count with new cases:

```
data Expr : Set where
  nat : ℕ → Expr
  _∔_ : Expr → Expr → Expr
  true : Expr
  false : Expr
  if_then_else_ : Expr → Expr → Expr → Expr
count : Expr → Expr
count (nat n) = 1
count (e₁ ∔ e₂) = count e₁ + count e₂
count true = 0
count false = 0
count (if e₁ then e₂ else e₃) = count e₁ + count e₂ + count e₃
```

This is the expression problem. Note that the same issue can be extended to proofs, since such proofs are just dependently typed functions over the datatype.

2.2 Data Types à la Carte

We can now attempt to rewrite the same expression syntax using the approach proposed by Swierstra (2008). Note that I will be using Haskell for this section. The original approach makes heavy use of Haskell type classes, so the Agda translation quickly becomes very verbose.

Rather than defining a single datatype as in the monolithic approach, we break down the Expr type into two feature functors, and instantiate the expression type as a fixpoint of their co-product:

```
data Fix f = In (f (Fix f))
data ExprN e = Nat Int | Sum e e
data ExprB e = ETrue | EFalse | If e e e

data (f :+: g) a = Inl (f a) | Inr (g a)
type Expr = Fix (ExprN :+: ExprB)
```

Swierstra (2008) provides two ways of defining recursive functions over this datatype: with a generic fold over functors—using the Haskell **Functor** type class—and by directly defining the recursive function. Both suffer from the same termination issue when translated to Agda; I will focus on the latter approach. We define the count function modularly using type classes:

```
class Count f where
  countMod :: Count g ⇒ f (Fix g) → Int
count :: Count f ⇒ Fix f → Int
count (In t) = countMod t
```

Finally, all that is left is to close the recursive knot by defining instances of Count:

```
instance Count ExprN where
  countMod (Nat n) = 1
  countMod (Add e1 e2) = count e1 + count e2
instance Incr ExprB where
  countMod ETrue = 0
  countMod EFalse = 0
  countMod (If e1 e2 e3) = count e1 + count e2 + count e3
```

Unfortunately, a naive translation of this approach is rejected by Agda. There are two main hurdles to overcome:

1. The translated Fix datatype does not satisfy the strict positivity requirements that are imposed on inductive datatypes in Agda to maintain soundness. This is because if f is an arbitrary functor, it is assumed that it may use its argument in a non-strictly-positive way.

2. Agda is not able to verify termination of their version of count. In general, the mutual recursion needed to define these modular functions is beyond the capability of the termination checker, as it is not known until runtime which instance of countMod will be called by count.

Nonetheless, DTC is the foundation for the approach presented in the next section.

2.3 Modular Type-Safety Proofs in Agda

The key insight from Schwaab and Siek (2013) is that it becomes possible for Agda to verify strict positivity by restricting the universe of functors Expr is abstracted over so that *by definition* the argument to each functor can only be used strictly positively. This is because Agda must always be able to verify strict positivity statically. Consider the universe of polynomial functors, defined in Agda as:

```
infixl 6 _⊕_
infixr 7 _⊗_
data Functor : Set₁ where
  Id : Functor
  Const : Set → Functor
  _⊕_  : Functor → Functor → Functor
  _⊗_ : Functor → Functor → Functor
```

Additionally, we have the interpretation function that gives the above datatype a meaning as a functor (i.e. a Set → Set mapping):

```
[_] : Functor → Set → Set
[ Id ] B = B
[ Const C ] B = C
[ F ⊕ G ] B = [ F ] B   [ G ] B
[ F ⊗ G ] B = [ F ] B × [ G ] B
```

Armed with this, Schwaab and Siek (2013) define a fixpoint type that is be accepted by Agda:

```
data Fix (F : Functor) : Set where
  inn : [ F ] (Fix F) → Fix F
```

The first issue with this approach is the restrictiveness of polynomial functors: it is impossible to encode first-class functions. This is not an issue for our simple arithmetic language, whose syntax is encoded as follows:

```
Exprℕ : Functor
Exprℕ = Const ℕ ⊕ (Id ⊗ Id)
ExprB : Functor
ExprB = Const ⊤ ⊕ Const ⊤ ⊕ (Id ⊗ Id ⊗ Id)
FExpr : Functor
FExpr = Exprℕ ⊕ ExprB
Expr : Set
Expr = Fix FExpr

pattern nat n = (inj₁ n)
pattern _∔_ n m = (inj₂ (n , m))
pattern true = (inj₁ (inj₁ tt))
pattern false = (inj₁ (inj₂ tt))
pattern if_then_else_ e₁ e₂ e₃ = (inj₂ (e₁ , e₂ , e₃))
```

The authors face an issue with the termination checker when implementing their proof of type preservation. However, the problem is inherent to the way they write modular recursive functions; our simple count function is enough to illustrate it:

```
count-ℕ : ∀ {F : Functor}
  → (count : Fix F → ℕ)
  → [ Exprℕ ] (Fix F)
  → ℕ
count-ℕ count (nat n) = 1
count-ℕ count (e₁ ∔ e₂) = (count e₁) + (count e₂)

count-B : ∀ {F : Functor}
  → (count : Fix F → ℕ)
  → [ ExprB ] (Fix F)
  → ℕ
count-B count true = 0
```

```
count-B count false = 0
count-B count (if e₁ then e₂ else e₃)
  = count e₁ + count e₂ + count e₃

count : Expr → ℕ
count (inn (inj₁ exprℕ)) = count-ℕ count exprℕ
count (inn (inj₂ exprB)) = count-B count exprB
```

Agda is unable to prove termination of count, as it does not recognize the mutual induction between count, count-+ and count-ℕ. In fact, this is exactly the second issue that prevented DTC from being naively adapted to proof assistants.

$$\begin{aligned}\text{Types } \tau,\delta &::= \mathbb{N} \mid \text{Bool} \\ \text{Expressions } e &::= \text{nat } n \mid e_1 + e_2 \mid \text{true} \mid \text{false} \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3\end{aligned}$$

Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\vdash \text{nat } n : \mathbb{N}}\textsc{Nat} \qquad \frac{\vdash e_1 : \mathbb{N} \quad \vdash e_2 : \mathbb{N}}{\vdash e_1 + e_2 : \mathbb{N}}\textsc{Sum}$$

$$\frac{}{\vdash \text{true} : \text{Bool}}\textsc{True} \qquad \frac{}{\vdash \text{false} : \text{Bool}}\textsc{False} \qquad \frac{\vdash e_1 : \text{Bool} \quad \vdash e_2 : \tau \quad \vdash e_3 : \tau}{\vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}\textsc{If}$$

Reduction $\boxed{e_1 \longrightarrow e_2}$

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\xi\text{-}+_1 \qquad \frac{e_2 \longrightarrow e_2'}{e_1 + e_2 \longrightarrow e_1 + e_2'}\xi\text{-}+_2$$

$$\frac{}{\text{nat } n_1 + \text{nat } n_2 \longrightarrow \text{nat } (n_1 +_{\mathbb{N}} n_2)}\beta\text{-}+$$

$$\frac{e_1 \longrightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3}\xi\text{-IF}$$

$$\frac{}{\text{if true then } e_2 \text{ else } e_3 \longrightarrow e_2}\beta\text{-IF-TRUE} \qquad \frac{}{\text{if false then } e_2 \text{ else } e_3 \longrightarrow e_3}\beta\text{-IF-FALSE}$$

Figure 2.1: Specification of the simple arithmetic language used throughout this chapter.

Chapter 3

Proposed Methodology

Recall that the two issues that prevent DTC (Swierstra, 2008) from being usable in Agda are: the fixpoint of signature functors does not pass the strict positivity check, and recursive functions defined modularly fail the termination check.

My proposed approach brings two key changes to solve these issues. First, I instantiate the modular datatype statically—as done by Swierstra (2008)—to appease the strict positivity checker without having to encode modular datatypes using a restricted class of functors. Second, I provide an external tool that automates the inlining of Agda modules at build time. Paired with consistent use of module parameters, this enables Agda to statically determine that modular functions use valid mutual induction, and thus pass the termination check.

I first introduce my method using the syntax of the simple language specified in Figure 2.1 and the count function from Chapter 2. Next, I show how to structure a larger project and how to define modular predicates, relations and proofs with my method. For this, I implement modular versions of the typing and reduction relations shown in Figure 2.1, as well as a proof of type preservation. For brevity, imports will be omitted from code listings except when needed to highlight a particular point.

3.1 Modular Syntax

Recall the fixpoint type from Schwaab and Siek (2013):

```
data Fix (F : Functor) : Set where
  inn : [ F ] (Fix F) → Fix F
```

The issue is that, in order for Agda to allow this definition, it needs to first be able to verify that every element of the Functor datatype is strictly positive. The restriction to polynomial functors achieves this, but at great cost in expressiveness. Instead, with a little refactoring we can rely on Agda's positivity checker more effectively. The key idea is to define the signature functors ahead of the fixpoint type, and *hardcode* said functors into its definition. In other words, we would build the datatype as follows:

```
data Exprℕ (Expr : Set) : Set where
  nat : ℕ → Exprℕ Expr
  _∔_ : Expr → Expr → Exprℕ Expr
data ExprB (Expr : Set) : Set where
  true : ExprB Expr
  false : ExprB Expr
  if_then_else_ : Expr → Expr → Expr → ExprB Expr
data Expr : Set where
  ↑exprℕ : Exprℕ Expr → Expr
  ↑exprB : ExprB Expr → Expr
```

This way, Agda can verify that each signature functor (Exprℕ and ExprB) is strictly positive when building the polarity graph, and thus allow the definition of Expr to pass the strict positivity check.

3.2 Modular Recursive Functions

In order to understand the issue with termination in modularly-defined recursive functions, consider the count function defined over the Expr datatype from Section 3.1, counting the number of natural literals in an expression:

```
count-ℕ : (Expr → ℕ) → Exprℕ Expr → ℕ
count-ℕ count (nat _) = 1
count-ℕ count (e₁ ∔ e₂) = count e₁ + count e₂
count-B : (Expr → ℕ) → ExprB Expr → ℕ
count-B count true = 0
count-B count false = 0
count-B count (if e₁ then e₂ else e₃) = count e₁ + count e₂ + count e₃
```

```
count : Expr → ℕ
count (↑exprℕ e) = count-ℕ count e
count (↑exprB e) = count-B count e
```

The function count will fail the termination check because it passes itself as an argument to another function, and Agda fails to identify the mutual recursion. In this case, the obvious solution is to rewrite this making use of forward declarations:

```
count : Expr → ℕ
count-ℕ : Exprℕ Expr → ℕ
count-ℕ (nat _) = 1
count-ℕ (e₁ +̇ e₂) = count e₁ + count e₂
count-B : ExprB Expr → ℕ
count-B true = 0
count-B false = 0
count-B (if e₁ then e₂ else e₃) = count e₁ + count e₂ + count e₃
count (↑exprℕ e) = count-ℕ e
count (↑exprB e) = count-B e
```

However, once each modular feature is separated into its own file, this will no longer be possible; passing the top-level count function as an argument to the modular functions will be inevitable. Therefore, to enable these modular functions to work with the current termination checker in Agda, we must somehow automate the rewriting of the first form into the second. Since this is only an issue whenever each function is in a separate file, we focus on that case.

My proposed solution combines the use of parameterized modules with a new external tool that inlines the content of a module into another. Consider the following code, split among several files in the same directory:

```
-- In file CountTemplate.agda
module CountTemplate where
count : Expr → ℕ
---!!!(INLINE_MODULE CountN)
---!!!(INLINE_MODULE CountB)
count (↑exprℕ e) = count-ℕ e
count (↑exprB e) = count-B e
-- In file CountN.agda
```

```
module CountN (count : Expr → ℕ) where
count-ℕ : Exprℕ Expr → ℕ
count-ℕ (nat _) = 1
count-ℕ (e₁ +̇ e₂) = count e₁ + count e₂
-- In file CountB.agda
module CountB (count : Expr → ℕ) where
count-B : ExprB Expr → ℕ
count-B true = 0
count-B false = 0
count-B (if e₁ then e₂ else e₃) = count e₁ + count e₂ + count e₃
```

We then run the command

```
agda-inline CountTemplate.agda Count.agda Count
```

where the arguments, in order, are the input template file, output file, and name of the top-level module in the output file.

After running the command, the file Count.agda will be created with the following contents:

```
module Count where
count : Expr → ℕ
countLeaf : (Expr → ℕ) → LeafExpr Expr → ℕ
countLeaf base = 1
countNode : (Expr → ℕ) → NodeExpr Expr → ℕ
countNode (node t₁ t₂) = count t₁ + count t₂
count (↑exprLeaf t) = countLeaf t
count (↑exprNode t) = countNode t
```

While this solves the problem, it is time-consuming to repeatedly run the agda-inline command during development, and your preferred editor plugin will likely not work as intended in the *template* file. To avoid this, we can make the template file compile in the interim without changing the output of agda-inline with the following code:

```
module CountTemplate where
count : Expr → ℕ
---!!!(INLINE_MODULE CountN)
```

```
---!!!(INLINE_MODULE CountB)
---!!!(REMOVE_START)
open import CountN count
open import CountB count
{-# TERMINATING #-}
---!!!(REMOVE_END)
count (↑exprℕ e) = count-ℕ e
count (↑exprB e) = count-B e
```

Everything between the REMOVE_START and REMOVE_END commands will be removed from the file by agda-inline during processing. The termination error in this template file can optionally be silenced in this way by adding {-# TERMIATING #-} right before the REMOVE_END command.

3.3 Structuring the Project

The meta-programming tool covered in Section 3.2 relies on an appropriate division of the project into separate files. Thus, before proceeding to develop functions and proofs over modular datatypes, this division warrants a detailed explanation. Note that, while there may be alternative ways to structure a project, this structure proved useful with the case studies I worked with. See Figure 3.1 for an overview of the final file structure of our simple arithmetic language (excluding any files autogenerated at build time).

With this project structure, each modular datatype and function is defined in multiple files. First, there is one file for each feature in the src/Features directory that implements the datatype or function. This is used to define the modular implementations (e.g. count-ℕ and count-B). Second, we have one file for each language in the src/Languages directory, where the particular datatype or function is instantiated from the modular implementations provided by each feature.

Each modular datatype and function should have access to previously defined datatypes and functions. In our example, the count function needs access to the Expr datatype. This is done by importing the *complete* datatype that is needed; in our example, we would rewrite the modular expression syntax to follow this structure as follows:

```
-- In file src/Features/Naturals/Expr.agda
```

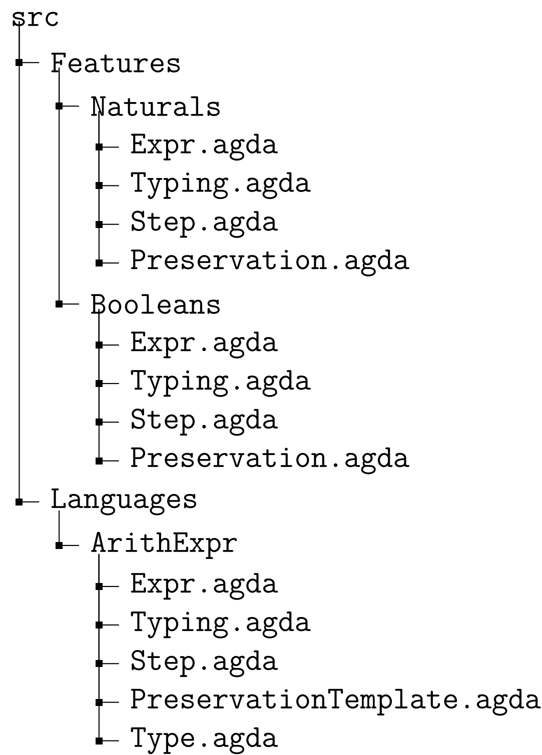

Figure 3.1: File tree of the development of our simple arithmetic language. Each directory inside `Features` contains all the modular code for a given feature. Conversely, directories inside `Languages` contain the *global* code for a given language formed from a set of features. Generally every file in the feature folders should have a matching file in the language folder.

```
module Features.Naturals.Expr (Expr : Set) where
data Exprℕ : Set where
  nat : ℕ → Exprℕ
  _+_ : Expr → Expr → Exprℕ

-- ... ExprB is done similarly, omitted here ...

-- In file src/Languages/ArithExpr/Expr.agda
module Languages.ArithExpr.Expr where
data Expr : Set
open import Features.Naturals.Expr Expr public
open import Features.Booleans.Expr Expr public
data Expr where
  ↑exprℕ : Exprℕ → Expr
```

```
↑exprB : ExprB → Expr
```

Then, the next modular datatype or function can import Languages.ArithExpr.Expr if needed:

```
-- In file Features/Naturals/Count.agda
open import Languages.ArithExpr.Expr
module Count (count : Expr → ℕ) where
count-ℕ : Exprℕ → ℕ
count-ℕ (nat _) = 1
count-ℕ (e₁ +̇ e₂) = count e₁ + count e₂
```

There are two issues with this so far:

1. Code within a modular feature should only be able to explicitly mention global types/functions it explicitly depends on, constructors/functions provided by that same feature, or from other features when there are explicit dependencies. However, this is not enforced by Agda; the author is responsible for ensuring this restriction is respected.

2. Directly importing modules defined under *Languages/ArithExpr* from modular feature code is not a good pattern, as whenever we may want to reuse a feature in a different language, we would need to rewrite all of these imports to point to the corresponding modules in the new language.

The first issue was already present before factoring out code into different files. Furthermore, it can be useful when it is not possible to fully express a proof or datatype related to a feature modularly. The author may want to *break out* of the modular structure for the problematic lines in the proof, but write everything else modularly.

The second issue has a simple solution. We can use an external script (presented in Appendix A.1) to generate stubs at the top level of the source directory automatically for each file in a given language folder. These stubs simply re-export the corresponding module from the language. An example generated stub is:

```
-- In file src/Expr.agda
open import Languages.ArithExpr.Expr public
```

We can now replace all imports of Languages.ArithExpr.Expr with imports of Expr, and whenever we wish to work with a different language we can simply run the script again to update the stubs. This approach extends effortlessly to the rest of the constructions we will present.

3.4 Delayed Lifting

Similarly to Swierstra (2008) and Schwaab and Siek (2013), we need to introduce a relation that *keeps track* of a way to inject a smaller datatype into a larger one.

```
variable
  ℓ : Level
record _<:_ (A B : Set ℓ) : Set ℓ where
  field
    inj : A → B
    retr : B → Maybe A
    retract-works : ∀ {x} → just x ≡ retr (inj x)
apply : {A B : Set ℓ} → {{A <: B}} → A → B
apply {{A<:B}} a = _<:_.inj A<:B a
retract : {A B : Set ℓ} → {{A <: B}} → B → Maybe A
retract {{A<:B}} b = _<:_.retr A<:B b

data LazyCoercion (B : Set ℓ) : Set (lsuc ℓ) where
  delay : {A : Set ℓ} → {{A <: B}} → A → LazyCoercion B
coerce : {B : Set ℓ} → LazyCoercion B → B
coerce (delay a) = apply a
```

The name LazyCoercion is used to emphasize that the coercion is delayed until the last moment possible. This avoids issues with unification in Agda (Schwaab and Siek, 2013).

Note that we are applying instance arguments to emulate the use of typeclasses in Haskell. This lets us avoid having to explicitly pass an instance of A <: B to apply, retract and delay. All that is left is to define the relevant instances for the modular expression syntax. For example, the instace for Exprℕ would be defined as follows

```
-- In file src/Languages/ArithExpr/Expr.agda
instance
  <:exprℕ : Exprℕ <: Expr
  _<:_.inj <:exprℕ = ↑exprℕ
  _<:_.retr <:exprℕ (↑exprℕ x) = just x
  _<:_.retr <:exprℕ _ = nothing
  _<:_.retract-works <:exprℕ = refl
```

Instances of _<:_ are mechanical and repetitive, so they will be omitted from listings except to highlight a particular point. Assume that such instances are defined for every modular datatype and relation unless otherwise noted.

3.5 Modular Predicates

Modular predicates, such as typing and reduction relations, are defined similarly to the modular expression syntax. Consider first the modular typing relation:

```
-- In file src/Features/Naturals/Typing.agda
module Features.Naturals.Typing (⊢_⦂_ : Expr → Type → Set) where
data ⊢ℕ_⦂_ : Exprℕ → Type → Set where
  ⊢nat : ∀ {n} → ⊢ℕ (nat n) ⦂ `ℕ
  ⊢sum : ∀ {e₁ e₂} → ⊢ e₁ ⦂ `ℕ → ⊢ e₂ ⦂ `ℕ
    → ⊢ℕ e₁ ∔ e₂ ⦂ `ℕ
-- In file src/Features/Booleans/Typing.agda
module Features.Booleans.Typing (⊢_⦂_ : Expr → Type → Set) where
data ⊢B_⦂_ : ExprB → Type → Set where
  ⊢true : ⊢B true ⦂ `Bool
  ⊢false : ⊢B false ⦂ `Bool
  ⊢if : ∀ {e₁ e₂ e₃ τ} → ⊢ e₁ ⦂ `Bool → ⊢ e₂ ⦂ τ → ⊢ e₃ ⦂ τ
    → ⊢B if e₁ then e₂ else e₃ ⦂ τ
-- In file src/Languages/ArithExpr/Typing.agda
module Languages.ArithExpr.Typing where
data ⊢_⦂_ : Expr → Type → Set
open import Features.Naturals.Typing ⊢_⦂_
open import Features.Booleans.Typing ⊢_⦂_
data ⊢_⦂_ where
```

```
↑⊢ℕ : ∀ {τ e} → ⊢ℕ e ⦂ τ → ⊢ apply e ⦂ τ
↑⊢B : ∀ {τ e} → ⊢B e ⦂ τ → ⊢ apply e ⦂ τ
```

Next, the modular reduction relation can be defined similarly:

```
-- In file src/Features/Naturals/Step.agda
module Features.Naturals.Step (_—→_ : Expr → Expr → Set₁) where
data _—→ℕ_ : Exprℕ → LazyCoercion Expr → Set₁ where
  ξ-+₁ : ∀ {e₁ e₁' e₂} → e₁ —→ e₁'
    → (e₁ ∔ e₂) —→ℕ delay (e₁' ∔ e₂)
  ξ-+₂ : ∀ {e₁ e₂ e₂'} → e₂ —→ e₂'
    → (e₁ ∔ e₂) —→ℕ delay (e₁ ∔ e₂')
  β-+ : ∀ {n m}
    → ((apply (nat n)) ∔ (apply (nat m))) —→ℕ delay (nat (n + m))
-- In file src/Features/Booleans/Step.agda
module Features.Booleans.Step (_—→_ : Expr → Expr → Set₁) where
data _—→B_ : ExprB → LazyCoercion Expr → Set₁ where
  ξ-if : ∀ {e₁ e₁' e₂ e₃} → e₁ —→ e₁'
    → (if e₁ then e₂ else e₃) —→B delay (if e₁' then e₂ else e₃)
  β-if-true : ∀ {e₂ e₃}
    → (if (apply true) then e₂ else e₃) —→B delay e₂
  β-if-false : ∀ {e₂ e₃}
    → (if (apply false) then e₂ else e₃) —→B delay e₃
-- In file src/Languages/ArithExpr/Step.agda
module Languages.ArithExpr.Step where
data _—→_ : Expr → Expr → Set₁
open import Features.Naturals.Step _—→_
open import Features.Booleans.Step _—→_
data _—→_ where
  ↑stepℕ : ∀ {e₁ e₂} → e₁ —→ℕ e₂ → apply e₁ —→ coerce e₂
  ↑stepB : ∀ {e₁ e₂} → e₁ —→B e₂ → apply e₁ —→ coerce e₂
```

In the case of the step relation, it is not possible to define an instance of **_<:_** due to a strict positivity error I was unable to resolve. Fortunately, this is not relevant for my approach, as in this case I can instead define a simpler relation that is sufficient for my purposes:

```
record _↑_ (A B : Set ℓ) : Set ℓ where
  field
    inj : A → B
lift : {A B : Set ℓ} → {{A ↑ B}} → A → B
lift {{A↑B}} a = _↑_.inj A↑B a
```

We then define instances of _↑_ as in the following example:

```
instance
  lift-stepℕ : ∀ {e₁ : Exprℕ} {e₂ : LazyCoercion Expr}
    → (e₁ —→ℕ e₂) ↑ (apply e₁ —→ coerce e₂)
  lift-stepℕ = record { inj = ↑stepℕ }
```

3.6 Type Preservation

The statements for type preservation for naturals and booleans mirror the global statement, but replacing the global typing relation and step relation in the arguments with the version for naturals and booleans respectively. The proofs are also very similar to what we would write in a monolithic version of this language:

```
-- In file Features/Naturals/Preservation.agda
module Features.Naturals.Preservation
  (preservation : ∀ {e e' : Expr} {τ : Type}
    → (e —→ e')
    → ⊢ e ⦂ τ
    → ⊢ e' ⦂ τ)
  where
preservationℕ : ∀ {e : Exprℕ} {e' : LazyCoercion Expr} {τ : Type}
  → (e —→ℕ e')
  → ⊢ℕ e ⦂ τ
  → ⊢ coerce e' ⦂ τ
preservationℕ (ξ-+₁ e₁→e₁') (⊢sum ⊢e₁ ⊢e₂) =
  apply (⊢sum (preservation e₁→e₁' ⊢e₁) ⊢e₂)
preservationℕ (ξ-+₂ e₂→e₂') (⊢sum ⊢e₁ ⊢e₂) =
  apply (⊢sum ⊢e₁ (preservation e₂→e₂' ⊢e₂))
preservationℕ β-+ (⊢sum ⊢e₁ ⊢e₂) = apply ⊢nat
```

```
-- In file Features/Booleans/Preservation.agda
module Features.Booleans.Preservation
  (preservation : ∀ {e e' : Expr} {τ : Type}
    → (e —→ e')
    → ⊢ e ⦂ τ
    → ⊢ e' ⦂ τ)
  where
preservationB : ∀ {e : ExprB} {e' : LazyCoercion Expr} {τ : Type}
  → (e —→B e')
  → ⊢B e ⦂ τ
  → ⊢ coerce e' ⦂ τ
preservationB (ξ-if cond→cond') (⊢if ⊢cond ⊢t ⊢f) =
  apply (⊢if (preservation cond→cond' ⊢cond) ⊢t ⊢f)
preservationB β-if-true (⊢if ⊢cond ⊢t ⊢f) = ⊢t
preservationB β-if-false (⊢if ⊢cond ⊢t ⊢f) = ⊢f
```

For the global type preservation proof, we need to make use of the external program presented in Section 3.2. We write the following template file:

```
-- In file Languages/ArithExpr/PreservationTemplate.agda
preservation : ∀ {e e' : Expr} {τ : Type}
  → (e —→ e')
  → ∅ ⊢ e ⦂ τ
  → ∅ ⊢ e' ⦂ τ

---!!!(INLINE_MODULE Preservationℕ)
---!!!(INLINE_MODULE PreservationB)
preservation (↑stepℕ st) (↑⊢ℕ wt) = preservationℕ st wt
preservation (↑stepB st) (↑⊢B wt) = preservationB st wt
```

To generate the full proof, we need to run the following command in the same directory as the template file:

```
agda-inline PreservationTemplate.agda Preservation.agda Preservation
```

The script presented in Appendix A.1 can also automatically process all template files in a given language directory.

Chapter 4

Case Study: PCF

The arithmetic language presented in Chapter 3 is very primitive, and lacks more interesting features such as variable binding and first-class functions. In order to demonstrate the applicability of the proposed methodology to a more practical language, this chapter presents a modular development and type-safety proof of PCF (Wadler et al., 2022). In total, the resulting formalization takes 1053 lines of code (741 excluding imports).

The syntax and semantics of PCF are summarized in Figure 4.1. The language decomposes naturally into four modular features: variables, lambda abstractions, naturals and fixpoint. PCF has the convenient property that all the typing and reduction rules mention only a single feature, so we need not worry about dependencies between features.

For conciseness, definitions that are unchanged from the monolithic version or are otherwise not relevant to the discussion are omitted and listed in Appendix A. I will refer to those definitions as appropriate. I will also continue to omit import statements except when instantiating parameterized modules, and instances of _<:_ and _↑_.

4.1 Type Syntax

The modular type syntax for naturals and lambda abstractions is defined similarly to what we have seen before:

```
module Features.Lambda.Type (Type : Set) where
data Typeλ : Set where
  _⇒_ : Type → Type → Typeλ
```

```
module Features.Naturals.Type (Type : Set) where
data Typeℕ : Set where
  `ℕ : Typeℕ
```

However, we have not considered what to do when a feature does not introduce any new types. There are two options: either do not create a definition and skip the module when *assembling* a top-level definition, or create a datatype definition with no members. We use the latter in order to maintain a uniform structure for all modular features. Thus, the type syntax for variables would be defined as follows:

```
module Features.Variables.Type (Type : Set) where
data TypeV : Set where
  empty : ⊥ → TypeV
```

The type syntax for the fixpoint operator is also empty and defined similarly, so it is omitted here.

Finally, we define the type syntax for the PCF language by combining the modular definitions as we have done before:

```
module Languages.PCF.Type where
data Type : Set
open import Features.Naturals.Type Type public
open import Features.Variables.Type Type public
open import Features.Fixpoint.Type Type public
open import Features.Lambda.Type Type public
data Type where
  ↑typeℕ : Typeℕ → Type
  ↑typeV : TypeV → Type
  ↑typeλ : Typeλ → Type
  ↑typeF : TypeF → Type
```

Note that the instances of TypeX <: Type (where X stands for any appropriate suffix) are omitted here to avoid excessive repetition.

4.2 Expression Syntax

The modular expression syntax is also similar to the monolithic version from Wadler et al. (2022). The modular syntax for the lambda feature is listed below; the rest are defined similarly in Appendix A.3.1.

```
module Features.Lambda.Expr (Expr : Set) where
data Exprλ : Set where
  λ_⇒_ : Id → Expr → Exprλ
  _·_  : Expr → Expr → Exprλ
```

Combining the modular definitions into the expression syntax for PCF is also done in the same way as before; refer to the aforementioned appendix for the full listing.

4.3 Context and Lookup Relation

Both the Context datatype and the lookup relation _∋_⦂_ do not depend on any particular feature. Therefore, we can use a single shared definition for each, unchanged from the monolithic definition. They are defined in Appendix A.3.2 and A.3.3, respectively.

4.4 Typing Relation

The typing relation is defined modularly. Not much is changed from the modular typing relation we defined for the primitive arithmetic language, but it is worth highlighting that each of the modular typing relations is only indexed by the modular expression syntax datatype from that same feature, not the global expression syntax.

The modular typing relation for the lambda feature is listed below for illustration; the rest are defined similarly and can be found in Appendix A.3.4.

```
module Features.Lambda.Typing
  (_⊢_⦂_ : Context → Expr → Type → Set₁) where
infix 4 _⊢λ_⦂_
data _⊢λ_⦂_ : Context → Exprλ → Type → Set₁ where
```

```
⊢λ : ∀ {Γ τ δ x e}
  → (Γ ▷ x ⦂ τ) ⊢ e ⦂ δ
    -----------------------------------
  → Γ ⊢λ λ x ⇒ e ⦂ apply (τ ⇒ δ)
⊢ · : ∀ {Γ τ δ} {e₁ : Expr} {e₂ : Expr}
  → Γ ⊢ e₁ ⦂ apply (τ ⇒ δ)
  → Γ ⊢ e₂ ⦂ τ
    -----------------------
  → Γ ⊢λ e₁ · e₂ ⦂ δ
```

Remarkably, the typing rules are still very similar to the monolithic counterparts; the only additional boilerplate that has been added is the need to call apply wherever we mention concrete type syntax. Unfortunately, this does not seem to be easily avoidable: some typing rules require the relation to be indexed over arbitrary types, such as ⊢ · .

The global typing relation is defined as in the previous case study, and is listed in Appendix A.3.4.

4.5 Values

Values can also be defined modularly, as each feature may introduce its own. However, just as in the syntax for types, both the variables and fixpoint features provide an empty Value relation. Listed below is the definition of values for the lambda feature:

```
module Features.Lambda.Value (Value : Expr → Set) where
data Valueλ : Exprλ → Set where
  V-λ : ∀ {x e}
    → Valueλ (λ x ⇒ e)
```

Note that the modular value relations are also indexed over their respective modular syntax types rather than the global syntax type Expr.

The rest of the modular value datatype definitions and the global instantiation for PCF are listed in Appendix A.3.6.

4.6 Substitution

For PCF, it is enough to restrict ourselves to substitution by closed terms, which simplifies the development (Wadler et al., 2022). We can divide the work needed in two parts: the first is the substitution function _[_:=_] : Expr → Id → Expr → Expr, which is not dependently typed and acts on expression syntax. The second part is the proof that such a substitution function preserves types. Since we will repeat the type declarations for the latter and its related lemmas in every feature, it is worth defining the types in an auxiliary file for later reuse. As an example, we show how to define the type for the rename lemma:

```
module Shared.TypeSignaturesSub where
tp-rename : Set₁
tp-rename = ∀ {Γ Δ}
  → (∀ {x τ} → Γ ∋ x ⦂ τ → Δ ∋ x ⦂ τ)
  → (∀ {e τ} → Γ ⊢ e ⦂ τ → Δ ⊢ e ⦂ τ)
```

The types for the rest of the lemmas are defined similarly; the types themselves are unchanged from the monolithic version, and they are listed in Appendix A.3.5.

The lemmas ext, weaken, drop and swap do not need to be defined recursively, and thus can be defined just once. Since their definitions are the same as the monolithic version, they are listed in Appendix A.3.5.

With all this set up, we can now define the modular functions. We show the definition for the lambda feature as an example; refer to the aforementioned appendix for the rest.

```
module Features.Lambda.Substitution
  (_[_:=_] : Expr → Id → Expr → Expr)
  (rename : tp-rename)
  (subst : tp-subst _[_:=_])
  (ext : tp-ext)
  (weaken : tp-weaken)
  (drop : tp-drop)
  (swap : tp-swap)
  where
infix 9 _[_:=_]λ
_[_:=_]λ : Exprλ → Id → Expr → Expr
```

```
(λ y ⇒ e₁) [ x := e₂ ]λ with y ≟ x
... | yes _ = apply (λ y ⇒ e₁)
... | no  _ = apply (λ y ⇒ (e₁ [ x := e₂ ]))
(e₁ · e₁′) [ x := e₂ ]λ = apply ((e₁ [ x := e₂ ]) · (e₁′ [ x := e₂ ]))
renameλ : ∀ {Γ Δ}
  → (∀ {x τ} → Γ ∋ x ⦂ τ → Δ ∋ x ⦂ τ)
  → (∀ {e τ} → Γ ⊢λ e ⦂ τ → Δ ⊢λ e ⦂ τ)
renameλ ρ (⊢λ ⊢e) = ⊢λ (rename (ext ρ) ⊢e)
renameλ ρ (⊢· ⊢e₁ ⊢e₂) = ⊢· (rename ρ ⊢e₁) (rename ρ ⊢e₂)
substλ : ∀ {Γ x} {e₁ : Exprλ} {e₂ : Expr} {τ δ}
    → ∅ ⊢ e₂ ⦂ τ
    → (Γ ▹ x ⦂ τ) ⊢λ e₁ ⦂ δ
      ---------------------------------
    → Γ ⊢ apply (e₁ [ x := e₂ ]λ) ⦂ δ
substλ {x = y} ⊢e₂ (⊢λ {x = x} ⊢e) with x ≟ y
... | yes refl = apply (⊢λ (drop ⊢e))
... | no  x≢y  = apply (⊢λ (subst ⊢e₂ (swap x≢y ⊢e)))
substλ ⊢e₂ (⊢· x y) = apply (⊢· (subst ⊢e₂ x) (subst ⊢e₂ y))
```

We can now combine the modular definitions shown before:

```
module Languages.PCF.SubstitutionTemplate where
infix 9 _[_:=_]
_[_:=_] : Expr → Id → Expr → Expr
rename : tp-rename
subst : tp-subst _[_:=_]
-- ... ext, weaken, drop, swap definitions go here ...
---!!!(INLINE_MODULE Features.Naturals.Substitution)
---!!!(INLINE_MODULE Features.Variables.Substitution)
---!!!(INLINE_MODULE Features.Lambda.Substitution)
---!!!(INLINE_MODULE Features.Fixpoint.Substitution)
↑exprℕ a [ x := b ] = a [ x := b ]ℕ
↑exprV a [ x := b ] = a [ x := b ]V
↑exprλ a [ x := b ] = a [ x := b ]λ
↑exprF a [ x := b ] = a [ x := b ]F
rename ρ (↑⊢ℕ wt) = ↑⊢ℕ (renameℕ ρ wt)
```

```
rename ρ (↑⊢V wt) = ↑⊢V (renameV ρ wt)
rename ρ (↑⊢λ wt) = ↑⊢λ (renameλ ρ wt)
rename ρ (↑⊢F wt) = ↑⊢F (renameF ρ wt)
subst ⊢e₁ (↑⊢ℕ ⊢e₂) = substℕ ⊢e₁ ⊢e₂
subst ⊢e₁ (↑⊢V ⊢e₂) = substV ⊢e₁ ⊢e₂
subst ⊢e₁ (↑⊢λ ⊢e₂) = substλ ⊢e₁ ⊢e₂
subst ⊢e₁ (↑⊢F ⊢e₂) = substF ⊢e₁ ⊢e₂
```

4.7 Reduction Relation

The modular step relation is also similar to the monolithic version, although it is necessary to make use of the apply function whenever the left-hand side of a rule goes more than one level deep in the expression syntax tree. Note also that the right-hand side of the modular step relation is a LazyCoercion Expr; this is because the right-hand side could contain an expression provided by a different feature.

The modular step relation for lambda is listed below; the rest are defined similarly and can be found in Appendix A.3.7.

```
module Features.Lambda.Step (_—→_ : Expr → Expr → Set₁) where
data _—→λ_ : Exprλ → LazyCoercion Expr → Set₁ where
 ξ-·₁ : ∀ {e₁ e₁′ e₂}
   → e₁ —→ e₁′
     -------------------------------
   → (e₁ · e₂) —→λ delay (e₁′ · e₂)
 ξ-·₂ : ∀ {e₁ e₂ e₂′}
   → e₂ —→ e₂′
     -------------------------------
   → (e₁ · e₂) —→λ delay (e₁ · e₂′)
 β-λ : ∀ {e₁ e₂ : Expr} {x : Id}
   → Value e₂
     ---------------------------------------------------
   → (apply (λ x ⇒ e₁) · e₂) —→λ delay (e₁ [ x := e₂ ])
```

The global step relation is defined just like the previous example; refer to Appendix A.3.7 for the full listing.

4.8 Preservation

Just as in substitution, the type of the preservation proof will be needed for every feature, so it makes sense to factor it out into an auxiliary file:

```
tp-preservation : Set₁
tp-preservation = ∀ {e e′ : Expr} {τ : Type}
  → (e —→ e′)
  → ∅ ⊢ e ⦂ τ
  → ∅ ⊢ e′ ⦂ τ
```

However, attempting to prove the modular preservation statements will highlight a new issue. Consider the preservation proof for the lambda feature; the first two cases are straightforward, but the third one is problematic:

```
module Features.Lambda.Preservation (preservation : tp-preservation) where
preservationλ : ∀ {e : Exprλ} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→λ e′
  → ∅ ⊢λ e ⦂ τ
    --------------------
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationλ (ξ-·₁ e₁—→e₁′) (⊢· ⊢e₁ ⊢e₂) =
  apply (⊢· (preservation e₁—→e₁′ ⊢e₁) ⊢e₂)
preservationλ (ξ-·₂ e₂—→e₂′) (⊢· ⊢e₁ ⊢e₂) =
  apply (⊢· ⊢e₁ (preservation e₂—→e₂′ ⊢e₂))
preservationλ (β-λ v-e₂) (⊢· ⊢f ⊢e₂) = ?
```

Note that the rule β-λ implies that ⊢f is an instance of type ∅ ⊢ apply (λ x ⇒ e_1) ⦂ apply (τ ⇒ δ). While we know that ⊢f must be equal to apply (⊢λ ⊢e) for appropriate ⊢e, we cannot match on that pattern directly without *peeking into* constructors for the global relation _⊢_⦂_, which would break our abstraction.

A solution is to require a canonical forms lemma, which I define as follows:

```
module Shared.CanonicalForms
  (SubType : Set) {{<:type : SubType <: Type}}
  (SubExpr : Set) {{<:expr : SubExpr <: Expr}}
  (SubValue : SubExpr → Set)
```

```
    {{<:val : ∀ {e : SubExpr} → (SubValue e) <: Value (apply e)}}
    (_⊢Sub_⨾_ : Context → SubExpr → Type → Set₁)
    {{<:val : ∀ {Γ e τ} → (Γ ⊢Sub e ⨾ τ) <: (Γ ⊢ (apply e) ⨾ τ)}}
    where

  tp-canonical-forms : Set₁
  tp-canonical-forms = ∀
    {Γ : Context} {τ : SubType} {e : Expr}
    → (val-e : Value e)
    → (⊢e : Γ ⊢ e ⨾ apply τ)
    → Σ[ s ∈ SubExpr ]
      Σ[ ⊢s ∈ Γ ⊢Sub s ⨾ apply τ ]
      Σ[ val-s ∈ SubValue s ]
      Σ[ e≡s ∈ e ≡ apply s ]
      (subst (λ {x → Γ ⊢ x ⨾ apply τ}) e≡s ⊢e ≡ apply ⊢s
        × subst Value e≡s val-e ≡ apply val-s)
```

Informally, the lemma states that given a well-typed expression whose type is provided by a given feature *A*, if that expression is also a value then

1. the expression syntax is provided by feature *A*,
2. the typing derivation is provided by feature *A* and
3. the value is provided by feature *A*.

We now prove such a lemma for the lambda feature. Another such lemma is needed for the fixpoint feature, but the proof is no different than the former and so it is omitted.

```
  module Languages.PCF.CanonicalForms where
  canonical-forms-λ : tp-canonical-forms Typeλ Exprλ Valueλ _⊢λ_⨾_
  canonical-forms-λ (↑val-λ {e = e} V-e) (↑⊢λ ⊢e) =
    e , ⊢e , V-e , refl , refl , refl
  canonical-forms-λ (↑val-ℕ ()) (↑⊢ℕ (⊢caseℕ _ _ _))
  canonical-forms-λ (↑val-V (empty ()))
  canonical-forms-λ (↑val-F (empty ()))
```

Note that this lemma is proved globally: I have not found a way to make this proof modular. Nonetheless, using this we can now complete the preservation proof for the lambda feature:

```
preservationλ (β-λ v-e₂) (⊢ · ⊢f ⊢e₂) with canonical-forms-λ (apply V-λ) ⊢f
... | _ , ⊢λ ⊢e₁ , _ , refl , refl , refl = subst ⊢e₂ ⊢e₁
```

A preservation proof for the naturals follows similarly now:

```
module Features.Naturals.Preservation (preservation : tp-preservation) where
preservationℕ : ∀ {e : Exprℕ} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→ℕ e′
  → ∅ ⊢ℕ e ⦂ τ
    ---------
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationℕ (ξ-suc st) (⊢suc wt) = apply (⊢suc (preservation st wt))
preservationℕ (ξ-caseℕ e₁—→e₁′) (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃) =
  apply (⊢caseℕ (preservation e₁—→e₁′ ⊢e₁) ⊢e₂ ⊢e₃)
preservationℕ β-zero (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃) = ⊢e₂
preservationℕ (β-suc v-v) (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃)
  with canonical-forms-ℕ (apply (v-suc v-v)) ⊢e₁
... | _ , ⊢suc ⊢v , _ , refl , refl , refl = subst ⊢v ⊢e₃
```

Finally, the proofs for fixpoint and variables do not require the canonical forms lemma:

```
module Features.Fixpoint.Preservation (preservation : tp-preservation) where
preservationF : ∀ {e : ExprF} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→F e′
  → ∅ ⊢F e ⦂ τ
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationF β-μ (⊢μ ⊢e) = subst (apply (⊢μ ⊢e)) ⊢e
```

```
module Features.Variables.Preservation (preservation : tp-preservation) where
preservationV : ∀ {e : ExprV} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→V e′
  → ∅ ⊢V e ⦂ τ
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationV (empty ())
```

The only thing left to do is to combine the modular statements into the preservation theorem for PCF. This is done just as in our previous case study, and we leave it for Appendix A.3.8.

4.9 Progress

The techniques discussed so far allow us to prove progress without further complications. First, we need an auxiliary datatype:

```
module Shared.Prog where
data Prog (e : Expr) : Set₁ where
  step : ∀ {e′} → (e ⟶ e′) → Prog e
  done : Value e → Prog e
```

The type for progress is also used in every feature, so we factor it out to an auxiliary module:

```
tp-progress : Set₁
tp-progress = ∀ {e τ}
  → ∅ ⊢ e ⦂ τ
  → Prog e
```

Then, the canonical form lemma makes the modular progress proofs straightforward:

```
module Features.Lambda.Progress
  (progress : tp-progress)
 where
progressλ : {e : Exprλ} {τ : Type}
  → ∅ ⊢λ e ⦂ τ
  → Prog (apply e)
progressλ (⊢λ _) = done (apply V-λ)
progressλ (⊢ · ⊢f ⊢v) with progress ⊢f
... | step f⟶f′ = step (lift (ξ- · ₁ f⟶f′))
... | done val-f with canonical-forms-λ val-f ⊢f | progress ⊢v
...   | _ | step v⟶v′ = step (lift (ξ- · ₂ v⟶v′))
...   | (λ x ⇒ t) , _ , _ , refl , _ , _ | done val-v = step (lift (β-λ val-v))
```

```
module Features.Naturals.Progress
  (progress : tp-progress)
  where
progressℕ : {e : Exprℕ} {τ : Type}
  → ∅ ⊢ℕ e ⦂ τ
  → Prog (apply e)
progressℕ ⊢zero   = done (apply (v-zero))
progressℕ (⊢suc ⊢e) with progress ⊢e
... | done v-e   = done (apply (v-suc v-e))
... | step e—→e′ = step (lift (ξ-suc e—→e′))
progressℕ (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃) with progress ⊢e₁
... | step e₁—→e₁′ = step (lift (ξ-caseℕ e₁—→e₁′))
... | done v-e₁ with canonical-forms-ℕ v-e₁ ⊢e₁
...   | `zero  , _ , _ , refl , refl , refl = step (lift (β-zero))
...   | `suc _ , _ , v-suc v , refl , refl , refl = step (lift (β-suc v))

module Features.Variables.Progress (progress : tp-progress) where
progressV : {e : ExprV} {τ : Type}
  → ∅ ⊢V e ⦂ τ
  → Prog (apply e)
progressV (⊢` ())

module Features.Fixpoint.Progress
  (progress : tp-progress)
 where
progressF : {e : ExprF} {τ : Type}
  → ∅ ⊢F e ⦂ τ
  → Prog (apply e)
progressF (⊢μ ⊢e) = step (lift β-μ)
```

Finally, we combine the modular statements into the progress theorem for PCF using INLINE_MODULE tool commands as before. The listing can be found in Appendix A.3.9.

$$
\begin{aligned}
\text{Types } \tau, \delta &::= \mathbb{N} \mid \tau \to \delta \\
\text{Expressions } e &::= x \mid \lambda x \Rightarrow e \mid e_1 \cdot e_2 \mid \text{zero} \mid \text{suc } e \\
&\quad \mid \text{case } e_1 \; [\text{zero} \Rightarrow e_2 \mid \text{suc } x \Rightarrow e_3] \mid \mu x \Rightarrow e \\
\text{Context } \Gamma &::= \emptyset \mid \Gamma, x : \tau \\
\text{Values } v &::= \lambda x \Rightarrow e \mid \text{zero} \mid \text{suc } v
\end{aligned}
$$

Lookup $\boxed{\Gamma \ni x : \tau}$

$$
\frac{}{\Gamma, x : \tau \ni x : \tau}\,\text{Z} \qquad \frac{x \neq y \quad \Gamma \ni x : \tau}{\Gamma, y : \delta \ni x : \tau}\,\text{S}
$$

Typing $\boxed{\Gamma \vdash e : \tau}$

$$
\frac{\Gamma \ni x : \tau}{\Gamma \vdash x : \tau}\,\textsc{Var} \qquad \frac{\Gamma, x : \tau \vdash e : \delta}{\Gamma \vdash \lambda x \Rightarrow e : \tau \to \delta}\,\textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau \to \delta \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \cdot e_2 : \delta}\,\textsc{App}
$$

$$
\frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_2 : \tau \quad \Gamma, x : \mathbb{N} \vdash e_3 : \tau}{\Gamma \vdash \text{case } e_1 \; [\text{zero} \Rightarrow e_2 \mid \text{suc } x \Rightarrow e_3] : \tau}\,\textsc{Case} \qquad \frac{}{\Gamma \vdash \text{zero} : \mathbb{N}}\,\textsc{Zero}
$$

$$
\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \text{suc } e : \mathbb{N}}\,\textsc{Suc} \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mu x \Rightarrow e : \tau}\,\textsc{Fix}
$$

Reduction $\boxed{e_1 \longrightarrow e_2}$

$$
\frac{e_1 \longrightarrow e_1'}{e_1 \cdot e_2 \longrightarrow e_1' \cdot e_2}\,\xi_{\cdot 1} \qquad \frac{e_2 \longrightarrow e_2'}{e_1 \cdot e_2 \longrightarrow e_1 \cdot e_2'}\,\xi_{\cdot 2} \qquad \frac{}{(\lambda x \Rightarrow e) \cdot v \longrightarrow e[\![x := v]\!]}\,\beta_\lambda
$$

$$
\frac{e_1 \longrightarrow e_1'}{\text{case } e_1 \; [\text{zero} \Rightarrow e_2 \mid \text{suc } x \Rightarrow e_3] \longrightarrow \text{case } e_1' \; [\text{zero} \Rightarrow e_2 \mid \text{suc } x \Rightarrow e_3]}\,\xi_{\text{case}}
$$

$$
\frac{e \longrightarrow e'}{\text{suc } e \longrightarrow \text{suc } e'}\,\xi_{\text{suc}} \qquad \frac{}{\text{case zero } [\text{zero} \Rightarrow e_2 \mid \text{suc } x \Rightarrow e_3] \longrightarrow e_2}\,\beta_{\text{zero}}
$$

$$
\frac{}{\text{case suc } v \; [\text{zero} \Rightarrow e_2 \mid \text{suc } x \Rightarrow e_3] \longrightarrow e_3[\![x := v]\!]}\,\beta_{\text{suc}} \quad \frac{}{\mu x \Rightarrow e \longrightarrow e[\![x := \mu x \Rightarrow e]\!]}\,\beta_\mu
$$

Figure 4.1: Specification of the PCF language, as presented in Programming Language Foundations in Agda (Wadler et al., 2022).

Chapter 5

Limitations: Cast Calculus

After successfully implementing a modular version of PCF, it is natural to ask how far this approach can be taken. To answer this, the latter part of this project was spent attempting to implement a modular version of the internal Cast Calculus from Siek and Taha (2006); in particular the variation used by Siek, Vitousek, et al. (2015) which includes blame tracking. This chapter briefly summarizes the findings.

The main issue we encounter lies with the following case for values

$$\text{Values } v \ ::= \dots \mid \langle \tau_1 \to \delta_1 \leadsto_\ell \tau_2 \to \delta_2 \rangle\, v$$

where $\langle \tau \leadsto_\ell \delta \rangle\, e$ stands for a cast from type τ to type δ of expression e. Informally, this means that casts of a value are also values if both the source and target types are function types. What this implies is that when you have a function application, a proof that the expression in the left-hand side (function position) has a function type and is a value is not sufficient to prove that it is in fact a lambda abstraction, as it may also be a cast. This is a problem for two reasons:

1. My statement of the canonical forms lemma from Section 4.8 no longer holds, and thus an alternative must be found for the proofs that need them.

2. The proof of progress for the lambda abstraction feature will need to consider the case of a function application where the left-hand side is a value with function type. This could be either a lambda abstraction (thus necessitating an application of the lambda β-reduction rule) or a cast (producing the cast application rule). In this way, the casts feature *pollutes* the lambda feature. Since the casts feature already explicitly depends on the lambda feature (i.e.

needs to directly mention some of its syntax and relations), this causes a circular dependency.

The second issue appears to be the most severe one. I see two possible solutions. An option is to include casts as part of the lambda feature; this sidesteps both issues, and is the option I have currently implemented in approximately 1300 lines of code. This option, however, would not scale well to larger languages with more non-trivial features (e.g. product types) as it essentially brings together all interdependent features into a monolithic block, defeating the purpose of a modular approach.

The second solution is to reformulate the cast calculus in such a way that casts do not *pollute* other features. As per the suggestion of P. Wadler, this could be achieved via an adaptation of the cast application rule (or *wrap* rule) (Siek, Vitousek, et al., 2015). However, this is non-trivial to achieve and out of scope for this project: P. Wadler reports that making this adaptation in a development of The Gradual Guarantee (Siek, Vitousek, et al., 2015) took several months.

These issues raise an interesting question. In conventional software engineering, designing programs with low coupling between components is widely accepted to be a good idea (Gamma et al., 1995), and the assumption has been implicitly made that this is also possible for the mechanization of programming language meta-theory. However, it may be worthwhile to explore in future research whether high coupling between features is a necessary evil for designing more complex programming languages, in which case the trade-off of designing for modularization in research applications could be too great.

Chapter 6

Conclusions and Future Work

My proposed approach presents a way to break apart definitions and proofs of programming languages into modular features in Agda. Compared to previous similar work in Agda (Schwaab and Siek, 2013), this approach supports more languages, does not suffer from the termination error which caused the previous method to be rejected by Agda and is closer in style to how monolithic proofs would be written. I have also demonstrated how to mechanize a modular type-safety proof for PCF as a case study, something which was not possible with the previous approach. Finally, I have demonstrated that the approach is not without its limitations, using an example from the meta-theory of gradual typing that cannot be elegantly modularized.

Regarding possible future improvements to this approach, explicit polarity annotations (Poiret et al., 2023) are currently being worked on for Agda. If merged, this would allow the dynamic fixpoint of functors as defined by Swierstra (2008) to be accepted by Agda. Furthermore, adding an `fmap` primitive has been discussed, which would solve the termination error that causes Agda to reject the approach from Swierstra (2008). Both of these improvements would improve the flexibility of my approach, and the latter may be enough to remove the meta-programming from the workflow.

In the meantime, it should also be possible to use Agda's built-in meta-programming tools to substitute the external program, which would simplify the workflow (removing a dependency on an external executable) and improve compatibility with existing editor tooling.

Bibliography

Abel, Andreas, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark (2019). "POPLMark reloaded: Mechanizing proofs by logical relations". In: *Journal of Functional Programming* 29. Publisher: Cambridge University Press, e19. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796819000170. (Visited on 04/23/2023).

Aydemir, Brian E., Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic (2005). "Mechanized Metatheory for the Masses: The PoplMark Challenge". In: *Theorem Proving in Higher Order Logics*. Ed. by Joe Hurd and Tom Melham. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 50–65. ISBN: 978-3-540-31820-0. DOI: 10.1007/11541868_4.

Cockx, Jesper, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell (Sept. 6, 2022). "Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs". In: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Haskell 2022. New York, NY, USA: Association for Computing Machinery, pp. 108–122. ISBN: 978-1-4503-9438-3. DOI: 10.1145/3546189.3549920. URL: https://doi.org/10.1145/3546189.3549920 (visited on 11/17/2022).

Delaware, Benjamin, Bruno C. d. S. Oliveira, and Tom Schrijvers (Jan. 23, 2013). "Meta-theory à la carte". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '13. New York, NY, USA: Association for Computing Machinery, pp. 207–218. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429094. URL: https://dl.acm.org/doi/10.1145/2429069.2429094 (visited on 04/10/2023).

Forster, Yannick and Kathrin Stark (Jan. 22, 2020). "Coq à la carte: a practical approach to modular syntax with binders". In: *Proceedings of the 9th ACM*

SIGPLAN International Conference on Certified Programs and Proofs. CPP 2020. New York, NY, USA: Association for Computing Machinery, pp. 186–200. ISBN: 978-1-4503-7097-4. DOI: 10.1145/3372885.3373817. URL: https://dl.acm.org/doi/10.1145/3372885.3373817 (visited on 04/10/2023).

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (Mar. 14, 1995). *Design patterns : elements of reusable object-oriented software*. 1st edition. Reading, Mass: Addison-Wesley. 416 pp. ISBN: 978-0-201-63361-0.

Kaiser, Jonas, Steven Schäfer, and Kathrin Stark (Sept. 8, 2017). "Autosubst 2: Towards Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions". In: *Proceedings of the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTP '17. New York, NY, USA: Association for Computing Machinery, pp. 10–14. ISBN: 978-1-4503-5374-8. DOI: 10.1145/3130261.3130263. URL: https://dl.acm.org/doi/10.1145/3130261.3130263 (visited on 08/12/2023).

Keuchel, Steven and Tom Schrijvers (Sept. 28, 2013). "Generic datatypes à la carte". In: *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*. WGP '13. New York, NY, USA: Association for Computing Machinery, pp. 13–24. ISBN: 978-1-4503-2389-5. DOI: 10.1145/2502488.2502491. URL: https://doi.org/10.1145/2502488.2502491 (visited on 04/10/2023).

Levin, Michael Y. and Benjamin C. Pierce (Mar. 2003). "TinkerType: a language for playing with formal systems". In: *Journal of Functional Programming* 13.2. Publisher: Cambridge University Press, pp. 295–316. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796802004550. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/tinkertype-a-language-for-playing-with-formal-systems/48B7A773D0B8FDB7D1A3E644F8BEA9CA (visited on 08/16/2023).

Pierce, Benjamin C. (Feb. 5, 2002). *Types and Programming Languages*. Cambridge, Mass: MIT Press. 648 pp. ISBN: 978-0-262-16209-8.

Poiret, Josselin, Lucas Escot, Joris Ceulemans, Malin Altenmüller, and Andreas Nuyts (June 15, 2023). "Read the Mode and Stay Positive". In: 29th International Conference on Types for Proofs and Programs, Location: Valencia, Spain. URL: https://lirias.kuleuven.be/4087236 (visited on 08/13/2023).

Schwaab, Christopher and Jeremy G. Siek (Jan. 22, 2013). "Modular type-safety proofs in Agda". In: *Proceedings of the 7th workshop on Programming languages meets program verification*. PLPV '13. New York, NY, USA: Association for

Computing Machinery, pp. 3–12. ISBN: 978-1-4503-1860-0. DOI: 10.1145/2428116.2428120. URL: https://dl.acm.org/doi/10.1145/2428116.2428120 (visited on 04/10/2023).

Siek, Jeremy G. and Tianyu Chen (2021). "Parameterized cast calculi and reusable meta-theory for gradually typed lambda calculi". In: *Journal of Functional Programming* 31. Publisher: Cambridge University Press, e30. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796821000241. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/parameterized-cast-calculi-and-reusable-metatheory-for-gradually-typed-lambda-calculi/7A6E772C5AEB832ED157E80C5D2085D4 (visited on 01/20/2023).

Siek, Jeremy G. and Walid Taha (2006). "Gradual Typing for Functional Languages". In: *Scheme and Functional Programming Workshop*, p. 12.

— (2007). "Gradual Typing for Objects". In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 2–27. ISBN: 978-3-540-73589-2. DOI: 10.1007/978-3-540-73589-2_2.

Siek, Jeremy G., Peter Thiemann, and Philip Wadler (2021). "Blame and coercion: Together again for the first time". In: *Journal of Functional Programming* 31. 0 citations (Crossref) [2022-11-01] Publisher: Cambridge University Press, e20. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796821000101. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/blame-and-coercion-together-again-for-the-first-time/657CEF36278FE66B039DEF1D06DF8178 (visited on 10/18/2022).

Siek, Jeremy G., Michael M. Vitousek, Matteo Cimini, and John Tang Boyland (2015). "Refined Criteria for Gradual Typing". In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 274–293. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.274. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5031 (visited on 11/01/2022).

Sozeau, Matthieu, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter

(June 1, 2020). "The MetaCoq Project". In: *Journal of Automated Reasoning* 64.5, pp. 947–999. ISSN: 1573-0670. DOI: 10.1007/s10817-019-09540-0. URL: https://doi.org/10.1007/s10817-019-09540-0 (visited on 08/12/2023).

Swierstra, Wouter (July 2008). "Data types à la carte". In: *Journal of Functional Programming* 18.4. Publisher: Cambridge University Press, pp. 423–436. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796808006758. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/data-types-a-la-carte/14416CB20C4637164EA9F77097909409 (visited on 03/26/2023).

Wadler, Philip (Nov. 12, 1998). *The Expression Problem*. E-mail. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (visited on 04/12/2023).

Wadler, Philip, Wen Kokke, and Jeremy G. Siek (Aug. 2022). *Programming Language Foundations in Agda*. URL: https://plfa.inf.ed.ac.uk/22.08/.

Appendix A

Additional Listings

A.1 Helper Script for Instantiating Language Variant

```
#!/bin/sh
lang_name="$1"

generate_stub() {
   feature_name="$1"
   echo "open import Languages.$lang_name.$feature_name public"
     ↪  >src/"$feature_name".agda
}

process_template() {
   name_base="$1"
   without_ext="$2"
   without_template="${without_ext%Template}"
   (cd "src/Languages/$lang_name" && agda-inline "$name_base"
     ↪  "$without_template.agda"
     ↪  "Languages.$lang_name.$without_template") || exit 1
}

echo "Instantiating language $lang_name..."
for file in src/Languages/"$lang_name"/*.agda; do
   name_base="$(basename "$file")"
   without_ext="${name_base%.*}"
```

```
    case "$name_base" in
        *Template.agda)
            process_template "$name_base" "$without_ext"
            generate_stub "$without_template"
            ;;
        *.agda)
            generate_stub "$without_ext"
            ;;
    esac
done
```

A.2 Helper Script for Checking Instantiated Variant

```
#!/bin/sh
variant_name="$1"
curr_dir="$(dirname "$0")"

cd "$curr_dir/src" || exit 1

variant_dir="Languages/$variant_name"

for file in "$variant_dir"/*.agda; do
    case "$file" in
        *Template.agda)
            echo "Skipping template $file"
            ;;
        *.agda)
            echo "Checking $file"
            output_stderr="$(agda "$file")"
            # If exit code is not 0, print stderr
            if [ $? -ne 0 ]; then
                printf "\033[31m%s\n%s\033[0m\n" "Failed on $file
                ↪  with output:" "$output_stderr"
                exit 1
```

```
        fi
        ;;
    esac
done
```

A.3 Modular PCF Listings

The listings in this section exclude all required imports for conciseness, just as in chapter 4. The complete source can be viewed in the attached code archive.

A.3.1 Expression Syntax

```
module Features.Lambda.Expr (Expr : Set) where
data Exprλ : Set where
  λ_⇒_ : Id → Expr → Exprλ
  _·_  : Expr → Expr → Exprλ

module Features.Variables.Expr (Expr : Set) where
data ExprV : Set where
  var : Id → ExprV

module Features.Fixpoint.Expr (Expr : Set) where
data ExprF : Set where
  μ_⇒_ : Id → Expr → ExprF

module Features.Naturals.Expr (Expr : Set) where
data Exprℕ : Set where
  `zero : Exprℕ
  `suc : Expr → Exprℕ
  `caseℕ_[zero⇒_|suc_⇒_] : Expr → Expr → Id → Expr → Exprℕ

module Languages.PCF.Expr where
data Expr : Set
open import Features.Naturals.Expr Expr public
open import Features.Variables.Expr Expr public
open import Features.Lambda.Expr Expr public
open import Features.Fixpoint.Expr Expr public
```

```
data Expr where
  ↑exprℕ : Exprℕ → Expr
  ↑exprV : ExprV → Expr
  ↑exprλ : Exprλ → Expr
  ↑exprF : ExprF → Expr
-- ... Omitted instances of ExprX <: Expr ...
```

A.3.2 Context

```
module Shared.Context where
infixl 5 _▷_⦂_
data Context : Set where
  ∅ : Context
  _▷_⦂_ : Context → Id → Type → Context
```

A.3.3 Lookup Relation

```
module Shared.Lookup where
infix 4 _∋_⦂_
data _∋_⦂_ : Context → Id → Type → Set where
  Z : ∀ {Γ : Context} {τ : Type} {x : Id}
    → (Γ ▷ x ⦂ τ) ∋ x ⦂ τ
  S : ∀ {Γ τ δ x y}
    → x ≢ y
    → Γ ∋ x ⦂ τ
    → (Γ ▷ y ⦂ δ) ∋ x ⦂ τ

S′ : ∀ {Γ x y A B}
   → {x≢y : False (x ≟ y)}
   → Γ ∋ x ⦂ A
     ------------------
   → (Γ ▷ y ⦂ B) ∋ x ⦂ A
S′ {x≢y = x≢y} x = S (toWitnessFalse x≢y) x
```

A.3.4 Typing

```
-- In file src/Features/Lambda/Typing.agda
module Features.Lambda.Typing
  (_⊢_⦂_ : Context → Expr → Type → Set₁) where
infix 4 _⊢λ_⦂_
data _⊢λ_⦂_ : Context → Exprλ → Type → Set₁ where
  ⊢λ : ∀ {Γ τ δ x e}
    → (Γ ▷ x ⦂ τ) ⊢ e ⦂ δ
      ------------------------------------
    → Γ ⊢λ λ x ⇒ e ⦂ apply (τ ⇒ δ)
  ⊢· : ∀ {Γ τ δ} {e₁ : Expr} {e₂ : Expr}
    → Γ ⊢ e₁ ⦂ apply (τ ⇒ δ)
    → Γ ⊢ e₂ ⦂ τ
      ------------------------
    → Γ ⊢λ e₁ · e₂ ⦂ δ

-- In file src/Features/Variables/Typing.agda
module Features.Variables.Typing
  (_⊢_⦂_ : Context → Expr → Type → Set₁) where
infix 4 _⊢V_⦂_
data _⊢V_⦂_ : Context → ExprV → Type → Set₁ where
  ⊢` : ∀ {Γ τ x}
    → Γ ∋ x ⦂ τ
      ------------------
    → Γ ⊢V var x ⦂ τ

-- In file src/Features/Fixpoint/Typing.agda
module Features.Fixpoint.Typing
  (_⊢_⦂_ : Context → Expr → Type → Set₁) where
infix 4 _⊢F_⦂_
data _⊢F_⦂_ : Context → ExprF → Type → Set₁ where
  ⊢μ : ∀ {Γ x e τ}
    → (Γ ▷ x ⦂ τ) ⊢ e ⦂ τ
      -------------------
    → Γ ⊢F μ x ⇒ e ⦂ τ
```

```
-- In file src/Features/Naturals/Typing.agda
module Features.Naturals.Typing
  (_⊢_⦂_ : Context → Expr → Type → Set₁) where
infix 4 _⊢ℕ_⦂_
data _⊢ℕ_⦂_ : Context → Exprℕ → Type → Set₁ where
  ⊢zero : ∀ {Γ}
      ------------------------------
    → Γ ⊢ℕ `zero ⦂ apply `ℕ
  ⊢suc : ∀ {Γ} {e₁ : Expr}
    → Γ ⊢ e₁ ⦂ apply `ℕ
      -------------------------------
    → Γ ⊢ℕ `suc e₁ ⦂ apply `ℕ
  ⊢caseℕ : ∀ {Γ τ x} {e₁ e₂ e₃ : Expr}
    → Γ ⊢ e₁ ⦂ apply `ℕ
    → Γ ⊢ e₂ ⦂ τ
    → (Γ ▹ x ⦂ (apply `ℕ)) ⊢ e₃ ⦂ τ
      -------------------------------
    → Γ ⊢ℕ `caseℕ e₁ [zero⇒ e₂ |suc x ⇒ e₃ ] ⦂ τ
-- In file src/Languages/PCF/Typing.agda
module Languages.PCF.Typing where
infix 4 _⊢_⦂_
data _⊢_⦂_ : Context → Expr → Type → Set₁
open import Features.Naturals.Typing _⊢_⦂_ public
open import Features.Variables.Typing _⊢_⦂_ public
open import Features.Lambda.Typing _⊢_⦂_ public
open import Features.Fixpoint.Typing _⊢_⦂_ public
data _⊢_⦂_ where
  ↑⊢ℕ : ∀ {Γ} {e : Exprℕ} {τ : Type}
    → Γ ⊢ℕ e ⦂ τ
    → Γ ⊢ apply e ⦂ τ
  ↑⊢V : ∀ {Γ} {e : ExprV} {τ : Type}
    → Γ ⊢V e ⦂ τ
    → Γ ⊢ apply e ⦂ τ
  ↑⊢λ : ∀ {Γ} {e : Exprλ} {τ : Type}
    → Γ ⊢λ e ⦂ τ
```

```
    → Γ ⊢ apply e ⦂ τ
  ↑⊢F : ∀ {Γ} {e : ExprF} {τ : Type}
    → Γ ⊢F e ⦂ τ
    → Γ ⊢ apply e ⦂ τ
-- ... Omitted instances of Γ ⊢X e ⦂ τ <: Γ ⊢ apply e ⦂ τ ...
```

A.3.5 Substitution

```
-- In file src/Shared/TypeSignaturesSub.agda
module Shared.TypeSignaturesSub where
tp-rename : Set₁
tp-rename = ∀ {Γ Δ}
  → (∀ {x τ} → Γ ∋ x ⦂ τ → Δ ∋ x ⦂ τ)
  → (∀ {e τ} → Γ ⊢ e ⦂ τ → Δ ⊢ e ⦂ τ)

tp-ext : Set
tp-ext = ∀ {Γ Δ}
  → (∀ {x A} → Γ ∋ x ⦂ A → Δ ∋ x ⦂ A)
  → (∀ {x y A B} → (Γ ▷ y ⦂ B) ∋ x ⦂ A → (Δ ▷ y ⦂ B) ∋ x ⦂ A)

tp-weaken : Set₁
tp-weaken = ∀ {Γ M A}
  → ∅ ⊢ M ⦂ A
  → Γ ⊢ M ⦂ A

tp-drop : Set₁
tp-drop = ∀ {Γ x M A B C}
  → (Γ ▷ x ⦂ A ▷ x ⦂ B) ⊢ M ⦂ C
  → (Γ ▷ x ⦂ B) ⊢ M ⦂ C

tp-swap : Set₁
tp-swap = ∀ {Γ x y M A B C}
  → x ≢ y
  → (Γ ▷ y ⦂ B ▷ x ⦂ A) ⊢ M ⦂ C
  → (Γ ▷ x ⦂ A ▷ y ⦂ B) ⊢ M ⦂ C
```

```
tp-subst : tp-[:=] → Set₁
tp-subst _[_:=_] = ∀ {Γ x e₁ e₂ τ δ}
  → ∅ ⊢ e₂ ⦂ τ
  → (Γ ▷ x ⦂ τ) ⊢ e₁ ⦂ δ
    ----------------------
  → Γ ⊢ e₁ [ x := e₂ ] ⦂ δ

-- In file src/Features/Lambda/Substitution.agda
module Features.Lambda.Substitution
  (_[_:=_] : Expr → Id → Expr → Expr)
  (rename : tp-rename)
  (subst : tp-subst _[_:=_])
  (ext : tp-ext)
  (weaken : tp-weaken)
  (drop : tp-drop)
  (swap : tp-swap)
  where

infix 9 _[_:=_]λ
_[_:=_]λ : Exprλ → Id → Expr → Expr
(λ y ⇒ e₁) [ x := e₂ ]λ with y ≟ x
... | yes _ = apply (λ y ⇒ e₁)
... | no  _ = apply (λ y ⇒ (e₁ [ x := e₂ ]))
(e₁ · e₁′) [ x := e₂ ]λ = apply ((e₁ [ x := e₂ ]) · (e₁′ [ x := e₂ ]))

renameλ : ∀ {Γ Δ}
  → (∀ {x τ} → Γ ∋ x ⦂ τ → Δ ∋ x ⦂ τ)
  → (∀ {e τ} → Γ ⊢λ e ⦂ τ → Δ ⊢λ e ⦂ τ)
renameλ ρ (⊢λ ⊢e) = ⊢λ (rename (ext ρ) ⊢e)
renameλ ρ (⊢· ⊢e₁ ⊢e₂) = ⊢· (rename ρ ⊢e₁) (rename ρ ⊢e₂)

substλ : ∀ {Γ x} {e₁ : Exprλ} {e₂ : Expr} {τ δ}
    → ∅ ⊢ e₂ ⦂ τ
    → (Γ ▷ x ⦂ τ) ⊢λ e₁ ⦂ δ
```

```
    ----------------------------------
    → Γ ⊢ apply (e₁ [ x := e₂ ]λ) ⦂ δ
substλ {x = y} ⊢e₂ (⊢λ {x = x} ⊢e) with x ≟ y
... | yes refl = apply (⊢λ (drop ⊢e))
... | no  x≢y  = apply (⊢λ (subst ⊢e₂ (swap x≢y ⊢e)))
substλ ⊢e₂ (⊢· x y) = apply (⊢· (subst ⊢e₂ x) (subst ⊢e₂ y))

-- In file src/Features/Naturals/Substitution.agda
module Features.Naturals.Substitution
  (_[_:=_] : Expr → Id → Expr → Expr)
  (rename : tp-rename)
  (subst : tp-subst _[_:=_])
  (ext : tp-ext)
  (weaken : tp-weaken)
  (drop : tp-drop)
  (swap : tp-swap)
  where

_[_:=_]ℕ : Exprℕ → Id → Expr → Expr
nat n [ x := v ]ℕ = apply (nat n)
sum e₁ e₂ [ x := v ]ℕ = apply (sum (e₁ [ x := v ]) (e₂ [ x := v ]))

renameℕ : ∀ {Γ Δ}
  → (∀ {x τ} → Γ ∋ x ⦂ τ → Δ ∋ x ⦂ τ)
  → (∀ {e τ} → Γ ⊢ℕ e ⦂ τ → Δ ⊢ℕ e ⦂ τ)
renameℕ ρ ⊢ℕ = ⊢ℕ
renameℕ ρ (⊢+ ⊢e₁ ⊢e₂) = ⊢+ (rename ρ ⊢e₁) (rename ρ ⊢e₂)

substℕ : ∀ {Γ x} {e₁ : Exprℕ} {e₂ : Expr} {τ δ}
  → ∅ ⊢ e₂ ⦂ τ
  → (Γ ▷ x ⦂ τ) ⊢ℕ e₁ ⦂ δ
    --------------------
  → Γ ⊢ apply (e₁ [ x := e₂ ]ℕ) ⦂ δ
substℕ ⊢v ⊢ℕ = apply ⊢ℕ
substℕ ⊢v (⊢+ ⊢e₁ ⊢e₂) = apply (⊢+ (subst ⊢v ⊢e₁) (subst ⊢v ⊢e₂))
```

```
-- In fiole src/Features/Variables/Substitution.agda
module Features.Variables.Substitution
  (_[_:=_] : Expr → Id → Expr → Expr)
  (rename : tp-rename)
  (subst : tp-subst _[_:=_])
  (ext : tp-ext)
  (weaken : tp-weaken)
  (drop : tp-drop)
  (swap : tp-swap)
  where

_[_:=_]V : ExprV → Id → Expr → Expr
var x [ y := v ]V with x ≟ y
... | yes _ = v
... | no  _ = apply (var x)

renameV : ∀ {Γ Δ}
  → (∀ {x τ} → Γ ∋ x ⦂ τ → Δ ∋ x ⦂ τ)
  → (∀ {e τ} → Γ ⊢V e ⦂ τ → Δ ⊢V e ⦂ τ)
renameV ρ (⊢` ∋x) = ⊢` (ρ ∋x)

substV : ∀ {Γ x} {e₁ : ExprV} {e₂ : Expr} {τ δ}
  → ∅ ⊢ e₂ ⦂ τ
  → (Γ ▹ x ⦂ τ) ⊢V e₁ ⦂ δ
    --------------------
  → Γ ⊢ apply (e₁ [ x := e₂ ]V) ⦂ δ
substV {x = x} v (⊢` {x = y} Z) with x ≟ y
... | yes _  = weaken v
... | no x≢y = ⊥-elim (x≢y refl)
substV {x = y} v (⊢` {x = x} (S x≢y ∋x)) with x ≟ y
... | yes refl = ⊥-elim (x≢y refl)
... | no  _    = apply (⊢` ∋x)

module Features.Fixpoint.Substitution
  (_[_:=_] : Expr → Id → Expr → Expr)
```

```
  (rename : tp-rename)
  (subst : tp-subst _[_:=_])
  (ext : tp-ext)
  (weaken : tp-weaken)
  (drop : tp-drop)
  (swap : tp-swap)
  where

infix 9 _[_:=_]F
_[_:=_]F : ExprF → Id → Expr → Expr
(μ x ⇒ e₁) [ y := e₂ ]F with x ≟ y
... | yes _ = apply (μ x ⇒ e₁)
... | no  _ = apply (μ x ⇒ (e₁ [ y := e₂ ]))

renameF : ∀ {Γ Δ}
  → (∀ {x τ} → Γ ∋ x ⦂ τ → Δ ∋ x ⦂ τ)
  → (∀ {e τ} → Γ ⊢F e ⦂ τ → Δ ⊢F e ⦂ τ)
renameF ρ (⊢μ ⊢e) = ⊢μ (rename (ext ρ) ⊢e)

substF : ∀ {Γ x} {e₁ : ExprF} {e₂ : Expr} {τ δ}
    → ∅ ⊢ e₂ ⦂ τ
    → (Γ ▷ x ⦂ τ) ⊢F e₁ ⦂ δ
      ----------------------------------
    → Γ ⊢ apply (e₁ [ x := e₂ ]F) ⦂ δ
substF {x = y} ⊢v (⊢μ {x = x} ⊢e) with x ≟ y
... | yes refl = apply (⊢μ (drop ⊢e))
... | no   x≢y = apply (⊢μ (subst ⊢v (swap x≢y ⊢e)))

-- In file src/Languages/PCF/SubstitutionTemplate.agda
module Languages.PCF.SubstitutionTemplate where
infix 9 _[_:=_]
_[_:=_] : Expr → Id → Expr → Expr
rename : tp-rename
subst : tp-subst _[_:=_]
```

```
ext : tp-ext
ext ρ Z          = Z
ext ρ (S x≢y ∋x) = S x≢y (ρ ∋x)

weaken : tp-weaken
weaken {Γ} ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → ∅ ∋ z ⦂ C
      ---------
    → Γ ∋ z ⦂ C
  ρ ()
drop : tp-drop
drop {Γ} {x} {M} {A} {B} {C} ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → (Γ ▷ x ⦂ A ▷ x ⦂ B) ∋ z ⦂ C
      -------------------------
    → (Γ ▷ x ⦂ B) ∋ z ⦂ C
  ρ Z                  =  Z
  ρ (S x≢x Z)          =  ⊥-elim (x≢x refl)
  ρ (S z≢x (S _ ∋z))  =  S z≢x ∋z
swap : tp-swap
swap {Γ} {x} {y} {M} {A} {B} {C} x≢y ⊢M = rename ρ ⊢M
  where
  ρ : ∀ {z C}
    → (Γ ▷ y ⦂ B ▷ x ⦂ A) ∋ z ⦂ C
      --------------------------
    → (Γ ▷ x ⦂ A ▷ y ⦂ B) ∋ z ⦂ C
  ρ Z                    =  S x≢y Z
  ρ (S z≢x Z)            =  Z
  ρ (S z≢x (S z≢y ∋z))  =  S z≢y (S z≢x ∋z)

---!!!(INLINE_MODULE Features.NaturalsPCF.Substitution)
---!!!(INLINE_MODULE Features.Variables.Substitution)
```

```
---!!!(INLINE_MODULE Features.Lambda.Substitution)
---!!!(INLINE_MODULE Features.Fixpoint.Substitution)

↑exprℕ a [ x := b ] = a [ x := b ]ℕ
↑exprV a [ x := b ] = a [ x := b ]V
↑exprλ a [ x := b ] = a [ x := b ]λ
↑exprF a [ x := b ] = a [ x := b ]F

rename ρ (↑⊢ℕ wt) = ↑⊢ℕ (renameℕ ρ wt)
rename ρ (↑⊢V wt) = ↑⊢V (renameV ρ wt)
rename ρ (↑⊢λ wt) = ↑⊢λ (renameλ ρ wt)
rename ρ (↑⊢F wt) = ↑⊢F (renameF ρ wt)

subst ⊢e1 (↑⊢ℕ ⊢e2) = substℕ ⊢e1 ⊢e2
subst ⊢e1 (↑⊢V ⊢e2) = substV ⊢e1 ⊢e2
subst ⊢e1 (↑⊢λ ⊢e2) = substλ ⊢e1 ⊢e2
subst ⊢e1 (↑⊢F ⊢e2) = substF ⊢e1 ⊢e2
```

A.3.6 Value

```
module Features.Lambda.Value (Value : Expr → Set) where
data Valueλ : Exprλ → Set where
  V-λ : ∀ {x e}
    → Valueλ (λ x ⇒ e)

module Features.Naturals.Value (Value : Expr → Set) where
data Valueℕ : Exprℕ → Set where
  v-zero : Valueℕ `zero
  v-suc : ∀ {e : Expr}
    → Value e
    → Valueℕ (`suc e)

module Features.Variables.Value (Value : Expr → Set) where
data ValueV : ExprV → Set where
  empty : ∀ {e} → ⊥ → ValueV e

module Features.Fixpoint.Value (Value : Expr → Set) where
```

```
data ValueF : ExprF → Set where
  empty : ∀ {e} → ⊥ → ValueF e

module Languages.PCF.Value where
data Value : Expr → Set
open import Features.Naturals.Value Value public
open import Features.Variables.Value Value public
open import Features.Lambda.Value Value public
open import Features.Fixpoint.Value Value public
data Value where
  ↑val-ℕ : {e : Exprℕ} → Valueℕ e → Value (apply e)
  ↑val-V : {e : ExprV} → ValueV e → Value (apply e)
  ↑val-λ : {e : Exprλ} → Valueλ e → Value (apply e)
  ↑val-F : {e : ExprF} → ValueF e → Value (apply e)
-- ... Omitted intances of ValueX e <: Value (apply e) ...
```

A.3.7 Reduction Relation

```
-- In file src/Features/Lambda/Step.agda
module Features.Lambda.Step (_—→_ : Expr → Expr → Set₁) where
data _—→λ_ : Exprλ → LazyCoercion Expr → Set₁ where
  ξ-·₁ : ∀ {e₁ e₁′ e₂}
    → e₁ —→ e₁′
      -------------------------------
    → (e₁ · e₂) —→λ delay (e₁′ · e₂)
  ξ-·₂ : ∀ {e₁ e₂ e₂′}
    → e₂ —→ e₂′
      -------------------------------
    → (e₁ · e₂) —→λ delay (e₁ · e₂′)
  β-λ : ∀ {e₁ e₂ : Expr} {x : Id}
    → Value e₂
      ---------------------------------------------------
    → (apply (λ x ⇒ e₁) · e₂) —→λ delay (e₁ [ x := e₂ ])

-- In file src/Features/Naturals/Step.agda
module Features.Naturals.Step (_—→_ : Expr → Expr → Set₁) where
```

```
data _—→ℕ_ : Exprℕ → LazyCoercion Expr → Set₁ where
  ξ-suc : {e₁ e₁′ : Expr}
    → e₁ —→ e₁′
      ----------------------------------------
    → (`suc e₁) —→ℕ  delay (`suc e₁′)
  ξ-caseℕ : ∀ {x} {e₁ e₁′ e₂ e₃ : Expr}
    → e₁ —→ e₁′
      ----------------------------------------
    → (`caseℕ e₁ [zero⇒ e₂ |suc x ⇒ e₃ ]) —→ℕ delay (`caseℕ e₁′ [zero⇒
    ↪  e₂ |suc x ⇒ e₃ ])
  β-zero : ∀ {x} {e₂ e₃ : Expr}
      ----------------------------------------------------
    → (`caseℕ (apply `zero) [zero⇒ e₂ |suc x ⇒ e₃ ]) —→ℕ (delay e₂)
  β-suc : ∀ {x} {e₁ e₂ e₃ : Expr}
    → Value e₁
      ----------------------------------------------------
    → (`caseℕ (apply (`suc e₁)) [zero⇒ e₂ |suc x ⇒ e₃ ]) —→ℕ (delay (e₃ [
    ↪  x := e₁ ]))

-- In file src/Features/Variables/Step.agda
module Features.Variables.Step (_—→_ : Expr → Expr → Set₁) where
data _—→V_ : ExprV → LazyCoercion Expr → Set₁ where
  empty : ∀ {e e′} → ⊥ → e —→V e′

-- In file src/Features/Fixpoint/Step.agda
module Features.Fixpoint.Step (_—→_ : Expr → Expr → Set₁) where
data _—→F_ : ExprF → LazyCoercion Expr → Set₁ where
  β-μ : ∀ {e : Expr} {x : Id}
      ------------------------------------------------
    → (μ x ⇒ e) —→F delay (e [ x := apply (μ x ⇒ e) ])

-- In file src/Languages/PCF/Step.agda
data _—→_ : Expr → Expr → Set₁

open import Features.Naturals.Step _—→_ public
open import Features.Variables.Step   _—→_ public
```

```
open import Features.Lambda.Step     _—→_ public
open import Features.Fixpoint.Step   _—→_ public

data _—→_ where
  ↑stepℕ : {e₁ : Exprℕ} {e₂ : LazyCoercion Expr}
    → e₁ —→ℕ e₂
    → apply e₁ —→ coerce e₂
  ↑stepV : {e₁ : ExprV} {e₂ : LazyCoercion Expr}
    → e₁ —→V e₂
    → apply e₁ —→ coerce e₂
  ↑stepλ : {e₁ : Exprλ} {e₂ : LazyCoercion Expr}
    → e₁ —→λ e₂
    → apply e₁ —→ coerce e₂
  ↑stepF : {e₁ : ExprF} {e₂ : LazyCoercion Expr}
    → e₁ —→F e₂
    → apply e₁ —→ coerce e₂
-- ... Omitted intances of e₁ —→X e₂ ↑ apply e₁ —→ coerce e₂ ...
```

A.3.8 Preservation

```
module Features.Lambda.Preservation (preservation : tp-preservation) where
preservationλ : ∀ {e : Exprλ} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→λ e′
  → ∅ ⊢λ e ⦂ τ
    --------------------
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationλ (ξ-·₁ e₁—→e₁′) (⊢· ⊢e₁ ⊢e₂) =
  apply (⊢· (preservation e₁—→e₁′ ⊢e₁) ⊢e₂)
preservationλ (ξ-·₂ e₂—→e₂′) (⊢· ⊢e₁ ⊢e₂) =
  apply (⊢· ⊢e₁ (preservation e₂—→e₂′ ⊢e₂))
preservationλ (β-λ v-e₂) (⊢· ⊢f ⊢e₂) with canonical-forms-λ (apply V-λ) ⊢f
... | _ , ⊢λ ⊢e₁ , _ , refl , refl , refl = subst ⊢e₂ ⊢e₁

module Features.Naturals.Preservation (preservation : tp-preservation) where
preservationℕ : ∀ {e : Exprℕ} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→ℕ e′
```

```
  → ∅ ⊢ℕ e ⦂ τ
    ---------
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationℕ (ξ-suc st) (⊢suc wt) = apply (⊢suc (preservation st wt))
preservationℕ (ξ-caseℕ e₁—→e₁′) (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃) =
  apply (⊢caseℕ (preservation e₁—→e₁′ ⊢e₁) ⊢e₂ ⊢e₃)
preservationℕ β-zero (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃) = ⊢e₂
preservationℕ (β-suc v-v) (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃)
  with canonical-forms-ℕ (apply (v-suc v-v)) ⊢e₁
... | _ , ⊢suc ⊢v , _ , refl , refl , refl = subst ⊢v ⊢e₃

module Features.Fixpoint.Preservation (preservation : tp-preservation) where
preservationF : ∀ {e : ExprF} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→F e′
  → ∅ ⊢F e ⦂ τ
    --------------------
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationF β-μ (⊢μ ⊢e) = subst (apply (⊢μ ⊢e)) ⊢e

module Features.Variables.Preservation (preservation : tp-preservation) where
preservationV : ∀ {e : ExprV} {e′ : LazyCoercion Expr} {τ : Type}
  → e —→V e′
  → ∅ ⊢V e ⦂ τ
    ---------
  → ∅ ⊢ (coerce e′) ⦂ τ
preservationV (empty ())

module Languages.PCF.PreservationTemplate where
preservation : tp-preservation
---!!!(INLINE_MODULE Features.Naturals.Preservation)
---!!!(INLINE_MODULE Features.Variables.Preservation)
---!!!(INLINE_MODULE Features.Lambda.Preservation)
---!!!(INLINE_MODULE Features.Fixpoint.Preservation)
preservation (↑stepℕ st) (↑⊢ℕ wt) = preservationℕ st wt
preservation (↑stepV st) (↑⊢V wt) = preservationV st wt
preservation (↑stepλ st) (↑⊢λ wt) = preservationλ st wt
preservation (↑stepF st) (↑⊢F wt) = preservationF st wt
```

A.3.9 Progress

```
module Features.Lambda.Progress
  (progress : tp-progress)
 where
progressλ : {e : Exprλ} {τ : Type}
  → ∅ ⊢λ e ⦂ τ
  → Prog (apply e)
progressλ (⊢λ _) = done (apply V-λ)
progressλ (⊢ · ⊢f ⊢v) with progress ⊢f
... | step f—→f′ = step (lift (ξ- ·₁ f—→f′))
... | done val-f with canonical-forms-λ val-f ⊢f | progress ⊢v
...   | _ | step v—→v′ = step (lift (ξ- ·₂ v—→v′))
...   | (λ x ⇒ t) , _ , _ , refl , _ , _ | done val-v = step (lift (β-λ val-v))

module Features.Naturals.Progress
  (progress : tp-progress)
  where
progressℕ : {e : Exprℕ} {τ : Type}
  → ∅ ⊢ℕ e ⦂ τ
  → Prog (apply e)
progressℕ ⊢zero   = done (apply (v-zero))
progressℕ (⊢suc ⊢e) with progress ⊢e
... | done v-e   = done (apply (v-suc v-e))
... | step e—→e′ = step (lift (ξ-suc e—→e′))
progressℕ (⊢caseℕ ⊢e₁ ⊢e₂ ⊢e₃) with progress ⊢e₁
... | step e₁—→e₁′ = step (lift (ξ-caseℕ e₁—→e₁′))
... | done v-e₁ with canonical-forms-ℕ v-e₁ ⊢e₁
...   | `zero  , _ , _ , refl , refl , refl = step (lift (β-zero))
...   | `suc _ , _ , v-suc v , refl , refl , refl = step (lift (β-suc v))

module Features.Variables.Progress (progress : tp-progress) where
progressV : {e : ExprV} {τ : Type}
  → ∅ ⊢V e ⦂ τ
  → Prog (apply e)
progressV (⊢` ())
```

```
module Features.Fixpoint.Progress
  (progress : tp-progress)
 where
progressF : {e : ExprF} {τ : Type}
  → ∅ ⊢F e ⦂ τ
  → Prog (apply e)
progressF (⊢μ ⊢e) = step (lift β-μ)
```

```
module Languages.PCF.ProgressTemplate where
progress : tp-progress
---!!!(INLINE_MODULE Features.Naturals.Progress)
---!!!(INLINE_MODULE Features.Variables.Progress)
---!!!(INLINE_MODULE Features.Lambda.Progress)
---!!!(INLINE_MODULE Features.Fixpoint.Progress)
progress (↑⊢ℕ wt) = progressℕ wt
progress (↑⊢V wt) = progressV wt
progress (↑⊢λ wt) = progressλ wt
progress (↑⊢F wt) = progressF wt
```