Swaraj Dash

On the Formal Unification of Parsers and Pretty Printers

Computer Science Tripos – Part II

Trinity Hall

13 May, 2016

Proforma

Name:	**Swaraj Dash**
College:	**Trinity Hall**
Project Title:	**On the Formal Unification of Parsers and Pretty Printers**
Examination:	**Computer Science Tripos – Part II**
Word Count:	**11980**
Project Originators:	Dr Dominic Mulligan and Dr Jeremy Yallop
Supervisors:	Dr Dominic Mulligan and Dr Jeremy Yallop

Original Aims of the Project

The project aims to create a library for describing parsers and pretty printers via a single definition, along with built-in mechanically verified proofs of correctness in the dependently typed programming language Agda. This extends previous unverified work in unifying parsing and pretty printing.

Work Completed

I have met all my proposed success criteria and have implemented one extension. I successfully formalised partial isomorphisms and implemented combinators for parsing and pretty printing. Following this milestone, I proved the correctness of my parser and printer combinators by showing that they are inverses of each other, and used these combinators to describe the syntax of primitive recursive functions and arithmetic expressions. As an extension, I proved my combinators to satisfy a set of algebraic laws, demonstrating their well-behavedness as algebraic objects. I have extended prior research by proving more general properties to hold for my implementation.

Special Difficulties

None.

Declaration

I, Swaraj Dash of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date 11 May, 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parsers turn strings into structured data; pretty printers turn structured data into strings (Figure 1.1). This dissertation describes the construction of a library for unified parsing and pretty printing, along with a machine-checked proof of correctness in the dependently typed programming language *cum* theorem prover Agda. It is clear from the dual nature of parsing and pretty printing, and their relation to a common grammar, that specifying them both separately introduces redundancy, which is a source for potential errors and inconsistencies. This redundancy can be avoided if there is a means to specify parsers and printers through a single definition. This project makes it possible to define a correct-by-construction parser and pretty printer from a single description of the syntax to be parsed and pretty printed.

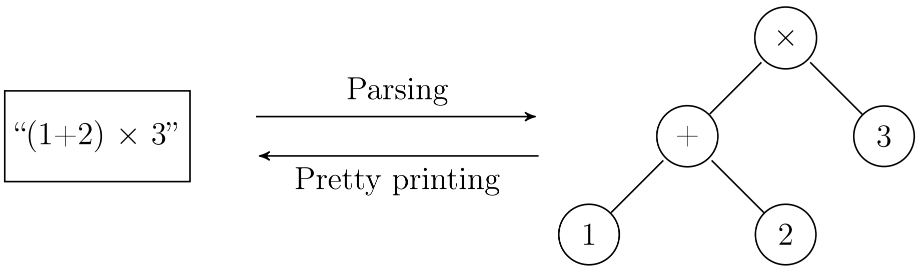


Figure 1.1: Parsing and pretty printing

1.1 Project summary

The approach I took to combining parsing and pretty printing is described by Rendel and Ostermann in their paper *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing* [23]. In this paper, the authors define a set of combinators (higher-order functions) that make it possible to describe a parser (§2.2) and a printer (§2.3) via a single definition. These parsers and printers are linked to each other by partial isomorphisms (§2.4.2), which are functions allowing reversible computation. I describe these concepts in detail in §2.

I have extended Rendel and Ostermann's *unverified* work by proving the correctness of my implementation in Agda. I accomplished this by providing machine-checked proofs that parsers and printers constructed using the method described in *Invertible Syntax Descriptions* are inverses of each other. That is, I proved that any string that is the result of printing some abstract syntax can be parsed to obtain the original abstract syntax unmodified. This guarantee of invertibility is absent in Rendel and Ostermann's implementation.

I have additionally proved my combinators to satisfy the algebraic laws known as functor and monoid laws (§4.3) as extensions to my project. These laws indicate my combinators to be algebraically well-behaved.

1.2 Related work

Combined parsing and printing: Jansson and Jeuring [15][14] take an alternative approach for combined parsing and printing by constructing parsers and printers from polytypic functions (a generalisation of polymorphic functions).

Alimarine et al. [2] accomplish unified parsing and printing by introducing the notion of bi-arrows, which are invertible functions similar to partial isomorphisms. This approach is different from Rendel and Ostermann's due to the difference in the role invertibility plays. While Alimarine et al. implement their parsers and printers directly as bi-arrows, Rendel and Ostermann define their combinators as a transformation on top of partial isomorphisms.

A proprietary tool used in industry is the DMS Software Reengineering Toolkit[1], which provides means for defining language grammars such that parsers and pretty printers can automatically be produced.

Mechanised theorem proving: Theorem provers based on higher-order logic include Isabelle [19] and HOL4 [24], both of which were originally developed in the Computer Laboratory, Cambridge. Agda [20], Coq [18], Idris [5], and Matita [3] belong to the family of type-theory based proof assistants.

Parser and printer combinators have been separately formalised in Agda by Danielsson [6][7]. In *Total Parser Combinators*, Danielsson constructed a parsing library with termination guarantees, which he accomplished by mixing induction and coinduction (a technique for defining and proving properties of infinite data structures, such as streams). *Correct-by-Construction Pretty-Printing* describes a library for pretty printing in Agda. In this paper Danielsson makes use of dependent types to ensure that, for each abstract syntax, the constructed document is correct with respect to the abstract syntax and a given grammar. This was done by encoding specific properties in the type signatures of the combinators.

[1]`http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html`

As far as I am aware, the only research in mechanically verifying Rendel and Ostermann's work was done by Affeldt, Nowak, and Oiwa [1], where they constructed a similar library in Coq. However, while Affeldt et al. prove their statement of invertibility only for unambiguous grammars, the work I describe here generalises their result, and shows invertibility to hold even for ambiguous grammars. Finally, Affeldt et al. only postulate the property of monotonicity (§3.5) for their parsers, which is a key assumption in the proof of invertibility, and do not prove it. In my implementation I proved all my parsers to be monotonic, and rely on no postulates.

1.3 Summary of completed work

Recall my success criteria (Appendix C):

- To formally define partial isomorphisms (defined in §2.4.2).
- To implement a set of combinators to perform combined parsing and pretty printing (implemented in §3.3).
- To describe arithmetic expressions and primitive recursive functions using these combinators (done in §4.4).
- To prove the correctness of my implementation by providing mechanically verified proofs of my parser and printer combinators being inverses of each other (proved in §3.6).

I have explicitly met all my success criteria for this project. As an extension I proved that my combinators satisfy algebraic laws known as the functor and monoid laws, which I detail in §4.3.

Chapter 2

Preparation

In this Chapter I survey material required to understand the rest of the report. I explain the entirety of this report using Agda, which I introduce in §2.1. In §2.2 and §2.3 I detail the concepts of parsing and pretty printing using combinators. Following the introductions I explain the work done by Rendel and Ostermann in unifying parsing and pretty printing in §2.4.

2.1 Agda

2.1.1 Introduction to Agda

Agda [20] is a dependently typed purely functional programming language, where types can depend on values. This ability of types to reference values enables more expressive types than those in non-dependently typed languages, by allowing them to encode properties of programs in their type signatures. With dependent types, a range of bugs (such as "out of bounds" errors for arrays) can be statically eliminated, simply by ensuring program typeability. Under the Curry-Howard correspondence [10], programs in Agda correspond to proofs in higher-order intuitionistic logic and types correspond to formulae.

I begin by constructing some familiar algebraic data types (ADTs) in Agda. Agda permits the declaration of **inductive families of types** [8]. Similar to ADTs, inductive families are characterised by a set of constructors. Unlike ADTs, however, the return types of constructors of inductive families are allowed to depend on values (also called *indexing by values*), and the return type of each constructor may differ.

The type Bool contains only two constructors, true and false.

```
data Bool : Set where
  true : Bool
  false : Bool
```

Set is the type of all small types in Agda. I will say no more about this here other than to note that the reader may assume, as an approximation, that it is the type of types.

The type of lists parametrised by type A has two constructors: the empty list [] and the "cons" constructor (_::_), which prepends elements to lists. It is given by the following definition:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

The **underscores** surrounding "::" indicate the locations of expected arguments; in our definition, the cons constructor is infix. For example, here is a list of three Boolean values:

```
bool-list-example : List Bool
bool-list-example = true :: false :: true :: []
```

Maybe is a polymorphic type in Agda that represents the addition of a new value to a type. I provide its definition, along with two examples, below:

```
data Maybe (A : Set) : Set where
  just : A → Maybe A
  nothing : Maybe A

maybe-example₁ : Maybe Bool
maybe-example₁ = just true

maybe-example₂ : Maybe (List Bool)
maybe-example₂ = just (true :: false :: [])
```

Maybe types have two constructors – just and nothing – and are used as the return type of functions which may or may not return meaningful values as output. Meaningful outputs of type A are wrapped in the just constructor, whereas non-meaningful values are represented by nothing. The Standard ML equivalent of Maybe is `option`, with the constructors `SOME` and `NONE`.

Functions on inductive families can be defined by pattern matching.

```
not : Bool → Bool
not true  = false
not false = true

_∧_ : Bool → Bool → Bool
true  ∧ b = b
false ∧ b = false
```

The not and _∧_ functions above pattern match on Bool.

Recall that the return types of constructors of inductive families can, unlike ADTs, depend on values. A consequence of this feature is the ability to define **propositional equality** in Agda. Two elements of the same type are propositionally equal if they can be shown to reduce to the same value. Consider the following definition:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Propositional equality in Agda is defined as an inductive type depending on a value for which we want to show equality. There is only one constructor refl (short for reflexive), which asserts that all objects are equal to themselves. Terms enclosed in curly braces are known **implicit arguments**, which the type checker should try to infer. Here are some examples of propositional equality being shown:

```
prf₁ : 1 ≡ 1
prf₁ = refl

prf₂ : 3 + 4 ≡ 7
prf₂ = refl

prf₃ : [ 4 ] ≡ (2 + 2) ∷ []
prf₃ = refl
```

In each example, Agda's type checker reduces both sides of the equality until they are equal. We use refl to indicate that both sides reduce to the same value.

The final Agda feature necessary for the understanding of this project is record creation. Records bundle terms and types together in a convenient manner — they are tuples of named fields where the types of later fields can depend on earlier ones. In this project, I made use of records to define syntax description combinators and partial isomorphisms, and structure them based on their mutual dependencies.

We can construct product types using records. In the example below, the field keyword introduces the type of each field, and the constructor keyword introduces syntax to describe our records. I make use of the constructor syntax in both examples below.

```
record _×_ (A B : Set) : Set where
  constructor
    _,_
  field
    fst : A
    snd : B

pair₁ : ℕ × ℕ
pair₁ = 1 , 5

pair₂ : Bool × ℕ
pair₂ = false , 0
```

Logic		$\iff$		**Type theory**
Truth	T	$\iff$	$\top$	Unit type
Falsity	F	$\iff$	$\bot$	Bottom type
Conjunction	$\wedge$	$\iff$	$\times$	Product type
Disjunction	$\vee$	$\iff$	$\uplus$	Sum type
Implication	$\implies$	$\iff$	$\rightarrow$	Function type
Universal quantification	$\forall$	$\iff$	Π	Dependent function type
Existential quantification	$\exists$	$\iff$	Σ	Dependent pair
Modus ponens		$\iff$		Function application
Provability		$\iff$		Inhabitation

Table 2.1: Curry-Howard correspondence for intuitionistic logic

2.1.2 Curry-Howard correspondence in Agda

Having introduced a few features of Agda, I list their correspondences to notions in intuitionistic logic in Table 2.1. This is also known as the *propositions-as-types* interpretation of the Curry-Howard correspondence.

This correspondence enables us to use Agda as a theorem prover. I demonstrate some of these correspondences in Agda below. I explain existential quantification in detail in §3.2.1, and provide a proof involving universal quantification in §2.1.3.

The Agda definitions for the type theory equivalent of truth (⊤) and falsity (⊥) are given below:

```
data ⊤ : Set where
  tt : ⊤

data ⊥ : Set where
```

⊤ only has a single constructor, whereas ⊥ has no constructor at all. This way it is impossible to construct values of type ⊥ in a consistent context, which ensures the consistency of Agda's logic.

Recall the product type _×_ shown previously. Given elements of types A and B, which here we interpret as being proofs of A and B interpreted as formulae, the type $A \times B$ represents the conjunction of A and B, again interpreted as a formula.

```
conjunction : {A B : Set} → A → B → A × B
conjunction a b = a , b
```

Disjunction corresponds to the sum type in Agda (also known as disjoint union). It is defined as:

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

The constructors inj₁ and inj₂ correspond to the introduction rules for disjunctions. Using these definitions it is possible to prove statements in propositional logic. The following are proofs demonstrating the commutativity of conjunction and disjunction:

```
×-comm : {A B : Set} → A × B → B × A
×-comm (a , b) = (b , a)

⊎-comm : {A B : Set} → A ⊎ B → B ⊎ A
⊎-comm (inj₁ x) = inj₂ x
⊎-comm (inj₂ x) = inj₁ x
```

The proof of commutativity of conjunction is given by constructing an element of type $B \times A$. This is done by pattern matching on the input element of type $A \times B$ to result in values a and b, which we then swap. The proof for disjunction follows similarly.

Modus ponens is represented as:

$$\frac{A \quad A \to B}{B}$$

To prove *modus ponens* we need to construct an element of type B from, say, elements a and f of types A and $A \to B$. We accomplish this by applying f to a. Under the Curry-Howard correspondence, therefore, *modus ponens* is equivalent to function application, as demonstrated by the proof below:

```
modus-ponens : {A B : Set} → A → (A → B) → B
modus-ponens a f = f a
```

All the examples above demonstrate that the inhabitation of a type corresponds to provability of the corresponding proposition in logic.

2.1.3 Theorem proving in Agda by example

In this Section I demonstrate how we can prove the property of associativity of list concatenation in Agda. I first present a mathematical proof of this result, which I later translate into a proof in Agda.

Recursively define list concatenation as:

```
_++_ : {A : Set} → List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Lemma 2.1.1 *For all lists xs, ys, and zs, $xs +\!\!+ (ys +\!\!+ zs) \equiv (xs +\!\!+ ys) +\!\!+ zs$.*

I prove this lemma by structural induction on xs.

Base case: In the base case, xs is the empty list. Then we need to demonstrate that

$$ys +\!\!+ zs \equiv ys +\!\!+ zs,$$

which is true by definition.

Inductive step: Let $xs = w :: ws$. We need to show that –

$$(w :: ws) ++ (ys ++ zs) \equiv ((w :: ws) ++ ys) ++ zs).$$

By definition of _++_, the left-hand side is equivalent to

$$w :: (ws ++ (ys ++ zs)).$$

Similarly, the right-hand side is equivalent to

$$w :: ((ws ++ ys) ++ zs).$$

These expressions are equal since we have by the inductive hypothesis that

$$ws ++ (ys ++ zs) \equiv (ws ++ ys) ++ zs.$$

This completes the proof by structural induction. ■

I now present the corresponding proof in Agda, which I detail line-by-line. First, the complete proof:

```
++-assoc : {A : Set} → (xs ys zs : List A) → xs ++ (ys ++ zs) ≡ (xs ++ ys) ++ zs
++-assoc [] ys zs = refl
++-assoc (w :: ws) ys zs = cong (_::_ w) (++-assoc ws ys zs)
```

1. The type signature corresponds to the statement of the lemma. Lists xs, ys, and zs are universally quantified.

We now pattern match on xs to split it into its two cases:

2. `++-assoc [] ys zs = refl`

   Agda performs the necessary reductions and finally requires a proof for $ys ++ zs \equiv ys ++ zs$. This is given by refl.

3. `++-assoc (w :: ws) ys zs = cong (_::_ w) (++-assoc ws ys zs)`

   This line makes use of the cong function from Agda's standard library.

   ```
   cong : {A B : Set} → {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
   ```

   cong states that all Agda functions are well-defined with respect to propositional equality. In other words, applying a transformation f to both sides of an equality preserves the equality.

   Using cong we arrive at a proof for the inductive step by appending (_::_ w) to both sides of the inductive hypothesis (given by ++-assoc ws ys zs).

This proves the lemma of associativity of list concatenation in Agda. ■

From this example we observe that under the Curry-Howard correspondence, **induction corresponds to recursion**. Type-checking the program for ++-assoc corresponds to checking the proof of the given proposition.

This concludes the brief introduction to Agda, where I showed the relation between proofs and programs and introduced concepts that I make use of in the implementation later on.

2.2 Parsing via combinators

Parser combinators are an approach to building recursive descent parsers by modelling parsers as functions and defining higher-order functions (i.e. combinators) to implement common operations found in formal grammars, such as sequencing and choice.

Work on parser combinators was originally motivated by the idea of recursive descent parsing, and has been popularised by Wadler [25], Hutton [12], and Hutton and Meijer [13], amongst others. Implementations of parser combinator libraries include Parsec[1] in Haskell, boost::spirit[2] in C++, and JParsec[3] in Java.

I begin by defining a parser in Agda. As stated earlier, parsers are modelled as functions that take strings as input and return parsed values as output. The Agda definition is based on the following properties of parsers:

- Ambiguous grammars may have multiple parses.
- Parsers may fail.
- Parsers need not return abstract syntax trees and can return any desired data type.
- Parsers need not consume all the input.

We therefore model parsers using the following type:

```
data Parser (A : Set) : Set where
  parser : (String → List (A × String)) → Parser A
```

In this definition we parametrise parsers by the type of the value they return. The above property of being able to return multiple results translates to the result being a list of values. The empty list by convention denotes parsing failure – this enables failure to be handled in terms of list operations. Since parsers are not required to consume the entire input string, each result is given by a pair consisting of the parsed value of type A and the part of the string that remains unconsumed.

This representation of parsers returning multiple results with their respective unconsumed strings makes it possible to build parsers piecewise from smaller parsers. Below I list the type signatures of combinators which allow us to combine parsers together. Each combinator represents a standard operation on formal grammars. I discuss the operations of each combinator in §2.4.

[1]https://hackage.haskell.org/package/parsec
[2]http://boost-spirit.com/home/
[3]https://github.com/jparsec/jparsec

Pure	pure	:	Parser A
Token	token	:	Parser Char
Empty	empty	:	Parser A
Choice	_⟨|⟩_	:	Parser $A \to$ Parser $A \to$ Parser A
Functor	_⟨\$⟩_	:	$(A \to B) \to$ Parser $A \to$ Parser B
Sequence	_⟨•⟩_	:	Parser $A \to$ Parser $B \to$ Parser $(A \times B)$
Repetition	⟨⋆⟩_	:	Parser $A \to$ Parser (List A)

Table 2.2: Parser combinators

Functors are families of types with a general "`map`" operation. Lists, for example, are an instance of a functor with their map function. I detail functors in §4.3.

As an example, a parser combinator for parsing the grammar $A ::= (\ A\ A\ )\ |\ \varepsilon$ would look like the following:

$$A = \text{token } `(` \ \langle\bullet\rangle\ A\ \langle\bullet\rangle A\ \langle\bullet\rangle\ \text{token } `)` \ \langle|\rangle\ \text{empty}$$

The code fragment above illustrates the use of combinators in building up parsers. The token combinator constructs parsers for the single tokens '(' and ')'. The sequence combinator ⟨•⟩ runs the first parser on its input and threads the unconsumed string to the second parser. Using this combinator, the parser for A is defined recursively as the sequencing of two parsers for A, which are enclosed in the token combinators to parse the brackets. The empty token corresponds to ε in the grammar. These two parsers are finally combined using the choice combinator, which returns the union of the results.

The complete parses of a parser are those values for which the entire string has been consumed. They can be extracted by applying the parse function to a parser and a string:

```
parse-filter : {A : Set} → List (A × String) → List A
parse-filter [] = []
parse-filter ((x , []) :: xs) = x :: parse-filter xs
parse-filter ((x , (y :: z)) :: xs) = parse-filter xs

parse : {A : Set} → Parser A → String → List A
parse (parser p) s = parse-filter (p s)
```

parse applies the internal parser p to the input string s, and then filters out results in which the entire string has been consumed using parse-filter (i.e., when the remainder string is the empty string), returning a list of parsed results of type A. I compare the application of p to s with and without parse-filter in Figure 2.1. We observe in the second case that ast_n is filtered since str_n is non-empty. Compared to the unfiltered case, only those parses where the entire string has been consumed are returned when parse is applied.

The function unParser extracts the inner parsing function.

```
unParser : {A : Set} → Parser A → String → List (A × String)
unParser (parser p) = p
```

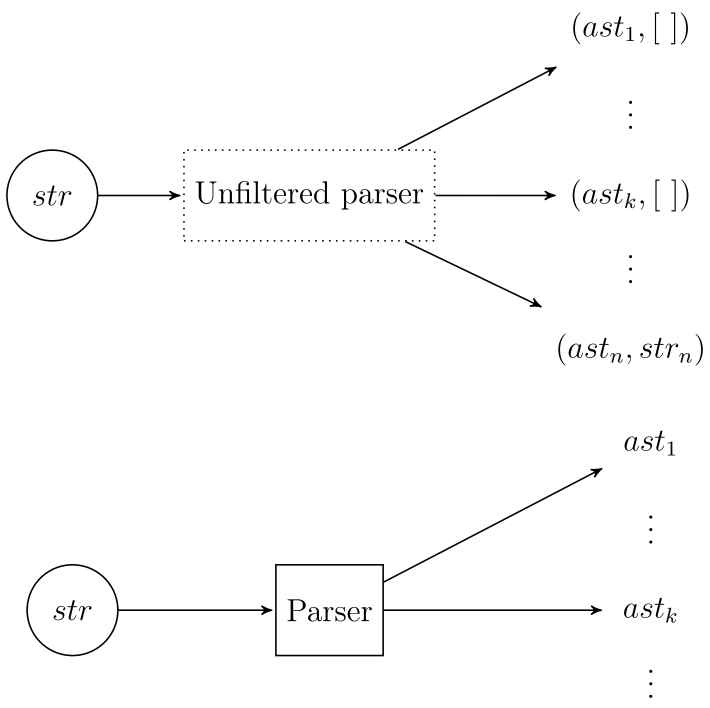


Figure 2.1: Top: application of unfiltered parser. Bottom: application of `parse`.

Parser combinators provide the following benefits over traditional parser implementations:

- Due to the functional nature of parser combinators, they are very compositional; large complicated parsers can be constructed by combining smaller simpler parsers with different combinators.
- Unlike parser generators, parser combinators are not separate from programming languages – they are first-class citizens of the language they have been implemented in. This enables users to define combinators by making full use of the power of the programming language.
- Parser combinators have been shown to satisfy several algebraic properties (such as having monadic and applicative instances [13]), which allows us to reason about their behaviour, and aggressively refactor them as they satisfy various algebraic laws.

2.3 Pretty printing via combinators

A pretty printer turns structured data into strings (for example, converting trees to strings, i.e., “flattening” trees). Pretty printers are defined as:

```
data Printer (A : Set) : Set where
  printer : (A → Maybe (List Char)) → Printer A
```

Maybe is the option type introduced in §2.1. Using it in the definition allows us to handle cases of failure in terms of operating on values of type Maybe A. As with parsers, the above definition permits the construction of combinators for printers. Work on pretty printer combinators has been popularised by Hughes [11], Oppen [21], and Wadler [26]. Implementations of pretty printing combinators include the pretty[4] and FPretty[5] libraries in Haskell.

Similar to Table 2.2, I provide the type signatures for printer combinators in Table 2.3. I compare the operations of both sets of combinators in Table 2.4.

Pure	pure	:	Printer A
Token	token	:	Printer Char
Empty	empty	:	Printer A
Choice	_⟨|⟩_	:	Printer $A \to$ Printer $A \to$ Printer A
Cofunctor	_⟨\$⟩_	:	$(B \to A) \to$ Printer $A \to$ Printer B
Sequence	_⟨•⟩_	:	Printer $A \to$ Printer $B \to$ Printer $(A \times B)$
Repetition	⟨⋆⟩_	:	Printer $A \to$ Printer (List A)

Table 2.3: Printer combinators

Printers can be applied to input objects by making use of the print function:

```
print : {A : Set} → Printer A → A → Maybe (List Char)
print (printer p) a = p a
```

This function applies the internal printing function p to the input a.

2.4 Syntax descriptions – unifying parsing and printing

Having observed how parsing and pretty printing can be decomposed into relatively simple combinators, we arrive at the main idea underlying the unification of parsing and printing, which I have based this project on. This idea comes from Rendel and Ostermann in their paper titled *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing* [23], where they accomplished this unification in Haskell.

Rendel and Ostermann make use of two observations in this paper. The first is that a common set of combinators can be used to represent parsing and printing combinators. This is achieved by overloading the combinators with the two respective functionalities of parsing and printing and invoking the required methods accordingly, which they call "syntax description combinators". The second observation links the unification of the functor and cofunctor combinators for parsers and printers to the notion of partial isomorphisms. I describe both these observations below.

[4]https://hackage.haskell.org/package/pretty
[5]https://hackage.haskell.org/package/FPretty

2.4.1 Overloading combinators

Previously, I presented a set of example combinators for parsers and pretty printers from which other combinators can be constructed. I list these combinators in Table 2.4 and compare their actions on parsers and pretty printers. The operation of the functor and cofunctor combinators has been omitted and is discussed in §2.4.2.

Combinator	δ = Parser	δ = Printer
pure : $\delta\ A$	pure c is the elementary parser which always succeeds, returning c as the result on any input (which remains unconsumed)	pure c compares the input to c, and returns just "" if true, and nothing otherwise
token : δ Char	token fails if there is no input, and returns the first character of the input otherwise	token simply returns its input character as a single-character string
empty : $\delta\ A$	empty is the elementary parser that always fails, i.e., returns the empty list on any input	empty is the elementary printer that always fails, i.e., returns nothing on any input
$\langle\vert\rangle : \delta\ A \to \delta\ A \to \delta\ A$	$\langle\vert\rangle$ takes two parsers and returns their concatenated individual results for any input	$\langle\vert\rangle$ takes two printers; given an input, it uses either printer that doesn't fail, starting with the first one
$\langle\bullet\rangle : \delta\ A \to \delta\ B \to \delta\ (A \times B)$	$\langle\bullet\rangle$ takes two parsers; it runs its input string through the first parser, threading the remainder string to the second parser and returning the pair of values parsed	$\langle\bullet\rangle$ takes two printers; if either printer fails on the input then it fails overall, otherwise it returns the concatenated results of the printers
$\langle\star\rangle : \delta\ A \to \delta$ (List A)	given a parser p, $\langle\star\rangle\ p$ is a parser that runs p sequentially on the input until it reaches the end of the strings	given a printer p, $\langle\star\rangle\ p$ is a printer that can be applied to a list of items to print sequentially; it fails overall if any one fails to get printed

Table 2.4: Comparison of parser and printer combinators

Comparing the actions of the combinators on parsers and printers, we can observe that there is a symmetry behind how they work. This observation motivates the idea of reducing the redundancy of providing both, a parser and a printer, by assigning a common set of combinators, which we shall call "syntax descriptions". By using syntax description combinators, the syntax of the language will only need to be detailed *once*, from which its parser and printer, respectively, can be derived automatically. This is achieved by function overloading (ad-hoc polymorphism), where each combinator is given a parsing and a printing implementation.

Formally, we define a type constructor δ to be an instance of a syntax description if it is possible to construct the six combinators below with the given type signatures:

$$
\begin{aligned}
&\mathsf{pure} : A \to \delta\ A \\
&\mathsf{token} : \delta\ \mathsf{Char} \\
&\mathsf{empty} : \delta\ A \\
&\_\langle|\rangle\_ : \delta\ A \to \delta\ A \to \delta\ A \\
&\_\langle\bullet\rangle\_ : \delta\ A \to \delta\ B \to \delta\ (A \times B) \\
&\langle\star\rangle\_ : \delta\ A \to \delta\ \mathsf{List}\ A
\end{aligned}
$$

In the next Section I extend this idea to the functor and cofunctor combinators.

2.4.2 Partial isomorphisms

Recall the type signatures of the parser (subscript a) and printer (subscript i) functor and cofunctor combinators, respectively.

$$
\begin{aligned}
&\_\langle\$\rangle_a\_ : (A \to B) \to \mathsf{Parser}\ A \to \mathsf{Parser}\ B \\
&\_\langle\$\rangle_i\_ : (B \to A) \to \mathsf{Printer}\ A \to \mathsf{Printer}\ B
\end{aligned}
$$

Both combinators take a function f and a parser (resp. printer) p as input.

The functor combinator for parsers first parses an input string, returning values of type A. These values are then transformed using f to values of type B, which are the final results.

Conversely, the cofunctor combinator for printers first applies f to the input argument of type B, transforming it to a value of type A. This value of type A is then printed by using p, which is the final result of the printer.

Combining these two combinators, then, would require us to combine a function of type $A \to B$ with that of type $B \to A$. Unifying this pair of types implies the ability to use a function in a bidirectional manner, i.e., it would need to be invertible (also known as an isomorphism). Since not all functions expressible in general programming languages are invertible, we define the notion of a partial isomorphism such that only functions which can be used forwards and backwards can be considered.

Formally, a partial isomorphism between types A and B is defined as a pair of functions $f : A \to \mathsf{Maybe}\ B$ and $g : B \to \mathsf{Maybe}\ A$ so that

$$f\ a \equiv \mathsf{just}\ b \iff g\ b \equiv \mathsf{just}\ a$$

(where a and b are values of types A and B respectively). A partial isomorphism is then defined as a pair of functions of type $A \to \mathsf{Maybe}\ B$ and $B \to \mathsf{Maybe}\ A$ satisfying these two properties.

Partial isomorphisms enable us to use a function in its forward direction when we are working in the context of parsers, and its backward direction when we are working in a printer context. With this notion of partial isomorphisms, all our syntax descriptions can be unified.

2.4.3 Overall

Finally, the complete definition for syntax descriptions is given by a set of functions satisfying the following type signatures for both δ = Parser and δ = Printer. Define Iso A B to be a partial isomorphism between A and B.

```
pure : A → δ A
token : δ Char
empty : δ A
_⟨|⟩_ : δ A → δ A → δ A
_⟨$⟩_ : Iso A B → δ A → δ B
_⟨•⟩_ : δ A → δ B → δ (A × B)
⟨⋆⟩_ : δ A → δ List A
```

Call _⟨\$⟩_ the *iso*functor combinator.

With this definition it is now possible to represent the parser and printer for any grammar by providing a single implementation with the syntax description combinators above. I describe the syntax for the language of primitive recursive functions and the language of arithmetic expressions in §4.4 using these combinators.

2.5 Requirements analysis

Analysing the requirements of my project indicated that the development process would be split into two phases – the combinator construction phase, and the correctness proving phase.

As part of the first phase of development, my aim was to create a platform in Agda which could facilitate the construction of syntax description combinators. This included implementing a library of frequently used functions and defining partial isomorphisms. The second phase, concerned with validating these combinators, would provide machine checked proofs of their invertibility.

The sequential nature of this project where the second phase could only be worked on after the completion of the first phase inspired me to follow the iterative development model. The first phase recursively followed an iterative model for each combinator that I implemented. This development model proved to be beneficial as a result of the relatively small size of each combinator. This enabled me to first ensure their correctness by running them through a set of tests, and finally guaranteeing this correctness by proving it in Agda in the second phase.

Apart from the use of Agda's standard library, this project was implemented from scratch.

2.6 Summary

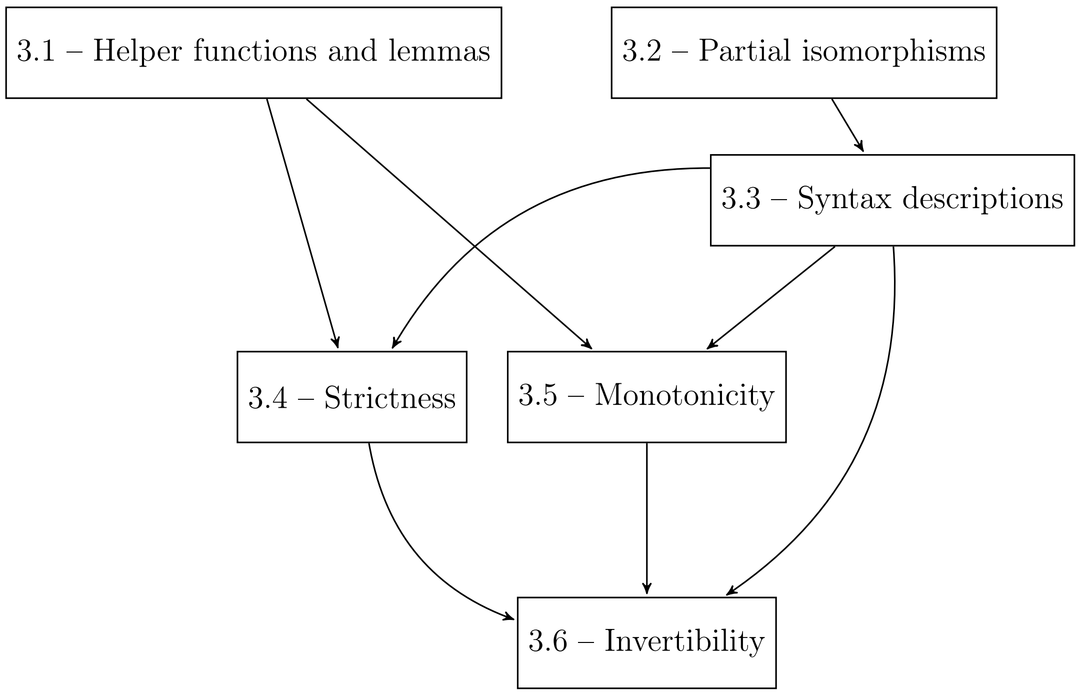
In this Chapter I have discussed the work undertaken before I started the implementation. I began by introducing Agda (§2.1), and showed how it can be used as a theorem prover by presenting an example proof. I followed this by describing how the actions of parsing (§2.2) and printing (§2.3) can be accomplished using a set of combinators (namely, the pure, token, empty, choice, isofunctor, sequence, and repetition combinators). Finally, I detailed Rendel and Ostermann's technique of combining the set of combinators for parsers and printers into a common set called "syntax description combinators" (§2.4).

In the next Chapter I give my implementation of syntax description combinators, following which I prove the correctness of my implementation.

Chapter 3

Implementation

In this Chapter I detail the construction of syntax description combinators and prove their invertibility. I begin this Chapter by presenting a set of helper functions and lemmas in §3.1 that I used throughout the implementation. Using these functions and the Agda definition of partial isomorphisms which I discuss in § 3.2, I describe how I implemented all my syntax description combinators in §3.3. In §3.4 and §3.5 I introduce two algebraic properties, namely, strictness and monotonicity, which I show to hold for all parsers. Using these algebraic properties and previously demonstrated lemmas, I state and prove my theorem of invertibility in Agda, which certifies the correctness of my implementation. I summarise this in the dependency graph below:


3.1 – Helper functions and lemmas
3.2 – Partial isomorphisms
3.3 – Syntax descriptions
3.4 – Strictness
3.5 – Monotonicity
3.6 – Invertibility

3.1 Helper functions and lemmas

In this Section I describe the implementation of list comprehensions in Agda (used in implementing parsers), reasoning about list membership (used in proofs of monotonicity and invertibility), defining functions to operate on Maybe types (used in implementing printers), and show results related to partitioning of lists (used in proofs relating to the repetition combinator).

3.1.1 Defining list comprehensions in Agda

List comprehensions — a feature available in some programming languages, such as Python and Haskell — are expressions that are used for defining lists based on some decidable property. While the syntax of list comprehensions can vary across languages, it is primarily based on set comprehension notation (also known as set-builder notation), such as:

$$\{x + y \mid x, y \in [0, 100],\ x^2 \equiv 3 \pmod 4,\ \sin(\pi y) = -1\}$$

List comprehensions in Haskell have a syntax similar to that of set comprehensions. The above example in Haskell would look like:

$$[\ x + y \mid x \leftarrow [0..100],\ y \leftarrow [0..100],\ x * x \text{ `mod` } 4 == 3,\ \sin\,(pi * y) == -1\ ]$$

In general, a Haskell list comprehension is of the form $[\ X \mid Y\ ]$, where X is an expression in terms of the variables introduced in Y, and Y is a comma separated list of generators ($x \leftarrow [0..100]$) and predicates ($\sin\,(pi * y) == -1$).

Rendel and Ostermann made extensive use of list comprehensions in their work. When porting their Haskell code in the syntax descriptions library to Agda, I needed to ensure that my definitions matched those of the authors'. This implied the heavy use of list comprehensions in my codebase, a feature which Agda does not provide by default. Therefore, one of my initial aims in this project was to create a means of being able to describe and use list comprehensions in Agda, which I detail below.

My implementation of list comprehensions was inspired from the one used in Haskell [9], where the list comprehension notation is simply syntactic sugar for a combination of two functions that operate on lists: the monadic bind function on lists $\gg\!=_l$ (implemented using concat and map) and return. They are defined as:

```
concat : {A : Set} → List (List A) → List A
concat [] = []
concat (xs :: xss) = xs ++ concat xss

_≫=ₗ_ : {A B : Set} → List A → (A → List B) → List B
xs ≫=ₗ f = concat (map f xs)
```

$\mathsf{return} : \{A : \mathsf{Set}\} \to A \to \mathsf{List}\ A$
$\mathsf{return}\ x = [\ x\ ]$

Before showing how these functions can form list comprehensions, I apply an initial simplification to the task: I consider list comprehensions that consist solely of generators. Such an assumption is valid because predicate clauses present inside comprehensions can always be converted to filtering functions which can be applied outside the comprehension. This shows that generator-only comprehensions and full-fledged list comprehensions are equally expressive. As an example, the list comprehension

$$[\ x \mid x \leftarrow [0..100],\ \texttt{isOdd}\ x\ ]$$

can be written in a generator-only style as

$$\texttt{filter isOdd}\ [\ x \mid x \leftarrow [0..100]\ ]$$

And so (without any loss of generality) to be able to describe list comprehensions I made use of the observation that comprehensions of the form:

$$[\ f\ x\ y\ ...\ z \mid x \leftarrow xs,\ y \leftarrow ys,\ ...,\ z \leftarrow zs\ ]$$

can be desugared to the following expression.

$xs \gg\!\!=_l (\lambda\ x \to$
$ys \gg\!\!=_l (\lambda\ y \to$
$...$
$zs \gg\!\!=_l (\lambda\ z \to \mathsf{return}\ (f\ x\ y\ ...\ z))...))$

I will use “list comprehension expression” to refer to the equivalent representation of a list comprehension obtained by desugaring it into its basic functions. From here on I will write $[\ x\ ]$ instead of $\mathsf{return}\ x$ in list comprehension expressions.

Using this method I was able to express all the original definitions that made use of list comprehensions in terms of bind, return, and their respective filtering functions. Note, however, that although this technique makes describing list comprehensions convenient, it makes reasoning about them difficult. This is due to the complexity of the expression that represents the given list comprehension, which contains many nested invocations of the bind function.

With a view to make reasoning about such expressions feasible, I proved a set of lemmas which demonstrate how certain list comprehension expressions are equivalent to more familiar forms, which are easier to work with. I discuss one such lemma in this section.

Lemma 3.1.1 *For any list xs and function f, the expression*

$$xs \gg\!\!=_l (\lambda\ y \to [\ f\ y\ ])$$

is equivalent to

$$\mathit{map}\ f\ xs.$$

The proof of this lemma follows by structural induction on xs and makes use of cong, which was introduced in §2.1. Recall its type signature:

```
cong : {A B : Set} → {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
```

cong states that any Agda function f is well-defined with respect to propositional equality.

This lemma can be expressed as the following type signature:

```
≫=l-map : {A B : Set} → (xs : List A) → (f : A → B) →
          xs ≫=l (λ y → [ f y ]) ≡ map f xs
```

Our base case is the case when xs is the empty list – its proof follows trivially since Agda automatically reduces both sides, leaving us to show that [] ≡ [].

```
≫=l-map [] f = refl
```

For the inductive step, let the list be of the form $x :: xs$. Assume that a proof for xs exists, which is given by ≫=$_l$-map xs. It is then required to prove that

$$(x :: xs) \gg=_l (\lambda\ y \to [\ f\ y\ ]) \equiv \mathsf{map}\ f\ (x :: xs).$$

Agda automatically reduces the expressions on both sides as much as possible according to their definitions, and the expression we are left to prove is

$$(f\ x) :: (xs \gg=_l (\lambda\ y \to [\ f\ y\ ])) \equiv (f\ x) :: \mathsf{map}\ f\ xs$$

This expression is similar to the inductive hypothesis, but with $f\ x$ prepended to both sides. So, this lemma is finally proved by making use of cong to prepend $f\ x$ to both sides of the inductive hypothesis.

```
≫=l-map : {A B : Set} → (xs : List A) → (f : A → B) →
          xs ≫=l (λ y → [ f y ]) ≡ map f xs
≫=l-map [] f = refl
≫=l-map (x :: xs) f = cong (_::_ (f x)) (≫=l-map xs f)
```

This lemma can alternatively be verified by sugaring the list comprehension expression into list comprehension notation. On conversion, the list comprehension obtained looks like:

$$[\ f\ y \mid y\ \leftarrow xs\ ].$$

We can see that this is indeed equal to map $f\ xs$. ■

A result similar to the lemma above, but one that I did not need to prove was:

$$([\ x\ ] \gg=_l f) \equiv [\ f\ x\ ]$$

This equality automatically follows by definition of concat and map, and so no proof was required by Agda. I encountered such expressions in proofs of edge cases.

Overall, I made use of list comprehension expressions throughout my implementation. In addition to other similar lemmas, $\gg\!=_l$-map proved to be useful in efficiently reasoning about them.

I proved the results in the remainder of this section in a similar inductive manner.

3.1.2 List membership

Proofs of monotonicity and invertibility (§3.5 and §3.6) require reasoning about list membership. I now define the list membership relation in Agda and present some useful lemmas.

Intuitively, an element is a member of a list if it is either the head of the list or is a member of the tail of the list. This inductive definition is the motivation behind the way the list membership predicate has been defined:

```
data _∈_ {A : Set} : A → List A → Set where
  here  : (x : A) → (xs : List A) → x ∈ (x :: xs)
  there : (x₁ x₂ : A) → (xs : List A) → x₁ ∈ xs → x₁ ∈ (x₂ :: xs)
```

The constructor here allows the programmer to construct a proof of $x \in (x :: xs)$, for arbitrary xs, if the head of a list x is a member of the list (which is then, by definition x :: xs). The there constructor is used to construct proofs of $x_1 \in (x_2 :: xs)$, provided $x_1 \in xs$, for arbitrary x_2.

Given a list and an element in it, the membership of that element can be proved using only the here and there constructors. For example:

```
example-prf : 3 ∈ (1 :: 2 :: 3 :: 4 :: [])
example-prf = there _ _ _ (there _ _ _ (here _ _))
```

The underscores are arguments that can be inferred by Agda and therefore need not be passed explicitly. In general, the proof that an element is present in the n^{th} position in a list will consist of $n-1$ uses of there and one use of here.

I list below some results based on proving list membership which I made use of in proofs later on. All these results were proved by induction on the structure of the lists involved, similar to the proof of $\gg\!=_l$-map.

Mapping a function over a list: Consider x and xs such that $x \in xs$. From this we can conclude that $(f\ x) \in (\text{map}\ f\ xs)$. The type signature of the proof is given by:

```
∈-map : {A B : Set} → (x : A) → (xs : List A) → (f : A → B) →
          x ∈ xs → f x ∈ map f xs
```

Concatenating two lists: Consider an element x and lists xs and ys such that $x \in xs$. From this we can conclude that $x \in xs +\!\!+ ys$. The proof has the following type signature:

```
∈-++¹ : {A : Set} → (x : A) → (xs ys : List A) → x ∈ xs → x ∈ (xs ++ ys)
```

Similarly, if $x \in ys$, then we have $x \in xs +\!\!+ ys$.

```
∈-++ʳ : {A : Set} → (x : A) → (xs ys : List A) → x ∈ ys → x ∈ (xs ++ ys)
```

Computing the concat of a list: If $[\,x\,] \in xs$, then $x \in$ concat xs as the concat function concatenates all the sublists together. This was shown by proving:

```
∈-concat : {A : Set} → (x : A) → (xs : List (List A)) → [ x ] ∈ xs → x ∈ concat xs
```

Filtering lists: If (just $x, y) \in xs$ then $(x, y) \in$ ⟨$⟩-filter xs (see §3.3.3 for the operation of ⟨$⟩-filter).

```
∈-⟨$⟩-filter : {A : Set} → (a : A) → (b : String) → (xs : List (Maybe A × String)) →
               (just a , b) ∈ xs → (a , b) ∈ ⟨$⟩-filter xs
```

3.1.3 `Maybe` functions

Recall the definition of Maybe:

```
data Maybe {A : Set} : Set → Set where
  just : A → Maybe A
  nothing : Maybe A
```

Maybe is a type constructor. An example use case for Maybe is to introduce a value to an existing type, to represent failed computations on that type, without reserving any particular value from the original type.

In this Section I look at functions that make it possible to operate on values of type Maybe without having to explicitly pattern match on the inputs.

The monadic bind function on Maybe, $\gg\!=_m$ (pronounced "maybe bind"), allows passing values of type A to functions f when only values of type Maybe A are available. It is defined as:

```
_≫=ₘ_ : {A B : Set} → Maybe A → (A → Maybe B) → Maybe B
just x ≫=ₘ f = f x
nothing ≫=ₘ f = nothing
```

In the case of the input being a meaningful value wrapped in the just constructor, the bind function applies f to the inner value x. Otherwise it returns nothing. In other words, the bind function permits sequencing of Maybe-valued functions.

The bind function is associative. One interpretation of the bind function is that it introduces sequentiality in code. Under this interpretation, the associativity of the bind operator implies that it only cares about the order of computations, and not about their nesting. The proof of associativity follows by pattern matching the argument to split it into two cases. In each case Agda can deduce that the left-hand side and right-hand side reduce to the same value, and so the equality holds.

```
>>=m-assoc : {A B C : Set} → (m : Maybe A) →
             (f : A → Maybe B) → (g : B → Maybe C) →
             (m >>=m f) >>=m g ≡ m >>=m (λ x → f x >>=m g)
>>=m-assoc (just m) f g = refl
>>=m-assoc nothing f g = refl
```

Functions of the form A → Maybe B can be composed by using the Kleisli composition function _>=>m_ defined below:

```
_>=>m_ : {A B C : Set} → (A → Maybe B) → (B → Maybe C) →
         (A → Maybe C)
_>=>m_ f g a = f a >>=m g
```

It is common to apply an operator expecting arguments of type A to values of type Maybe A. liftM2 enables such a functionality, by evaluating to nothing if either argument is nothing, and otherwise computing the result and wrapping it in the just constructor.

```
liftM2 : {A B C : Set} → (A → B → C) → Maybe A → Maybe B → Maybe C
liftM2 f (just x) (just y) = just (f x y)
liftM2 f (just x) nothing = nothing
liftM2 f nothing y = nothing
```

The mplus function returns either argument if the argument has a just constructor (if both arguments satisfy this criterion, then the first is returned), and returns nothing otherwise.

```
mplus : {A : Set} → Maybe A → Maybe A → Maybe A
mplus (just x) _ = (just x)
mplus nothing y = y
```

mplus is an associative function, which I proved in Agda. The proof follows by case splitting the first argument so that Agda can deduce that in both cases, the left-hand side and right-hand side reduce to the same value.

```
mplus-assoc : {A : Set} → (x y z : Maybe A) →
              mplus x (mplus y z) ≡ mplus (mplus x y) z
mplus-assoc (just x) y z = refl
mplus-assoc nothing y z = refl
```

I made use of all these functions and their properties in the implementation related to printers.

3.1.4 Partitions of a list

Partitioning a list into contiguous sublists played an important role in the implementation of the repetition combinator ⟨⋆⟩. Since one of my aims was to prove the invertibility of ⟨⋆⟩, proving certain properties about my partitioning function consequently followed. Here I discuss one such lemma that I had to prove: the set of all partitions of a list contains the list itself.

Definition 3.1.1 *Define a partition of a list to be a list of non-empty contiguous sublists, which when concatenated together form the original list. Let the function parts return all such partitions of the list.*

For example, parts $[1, 2, 3] = [[[1], [2], [3]],\ [[1], [2, 3]],\ [[1, 2], [3]],\ [[1, 2, 3]]]$. Note that the input list is contained in the results.

parts can be defined recursively.

```
parts-filter : {A : Set} → (y : A) → (xs : List (List (List A))) → List (List (List A))
parts-filter _ [] = []
parts-filter y ([] :: xs) = parts-filter y xs
parts-filter y ((x :: xs) :: xss) = ((y :: x) :: xs) :: parts-filter y xss

parts : {A : Set} → List A → List (List (List A))
parts [] = [] :: []
parts (x :: xs) = (map (_::_ (x :: [])) (parts xs)) ++ parts-filter x (parts xs)
```

In the case of an empty list, there is only one partition which is the list containing only []. When the list is non-empty, its partitions are defined in terms of the head and the partitions of the tail.

With the definition established, we can begin to prove the following lemma.

Lemma 3.1.2 *The set of all partitions of a list also contains the list itself.*

$$[\ xs\ ] \in \textit{parts}\ xs$$

By careful analysis and experimentation of the parts function it can be concluded that the original list always appears last in the list of all partitions. With the help of this observation I could transform the proof of showing that [xs] is the last element of parts xs to an inductive proof. The proof by induction is based on the invariant that at each recursive stage if xs is at the end of parts xs, then it must be present in the second half of the definition, i.e., xs must be a member of parts-filter x (parts xs). This idea is illustrated in Figure 3.1, where I show how the statement "the element lies in the second half" remains invariant iff it is the last element.

The following sub-lemma was crucial in being able to prove this result:

```
∈-filter-parts : {A : Set} → (x : A) → (xs : List A) → (ys : List (List (List A))) →
                 [ xs ] ∈ ys → [ x :: xs ] ∈ parts-filter x ys
```

∈-filter-parts shows that if a singleton list [xs] is in ys, then the list [$x :: xs$] is in parts-filter x ys. The list membership result this lemma states is very similar to the definition of parts, which is why ∈-filter-parts proved to be useful. I proved it by case-splitting ys; each case either held trivially or followed by induction on the list.

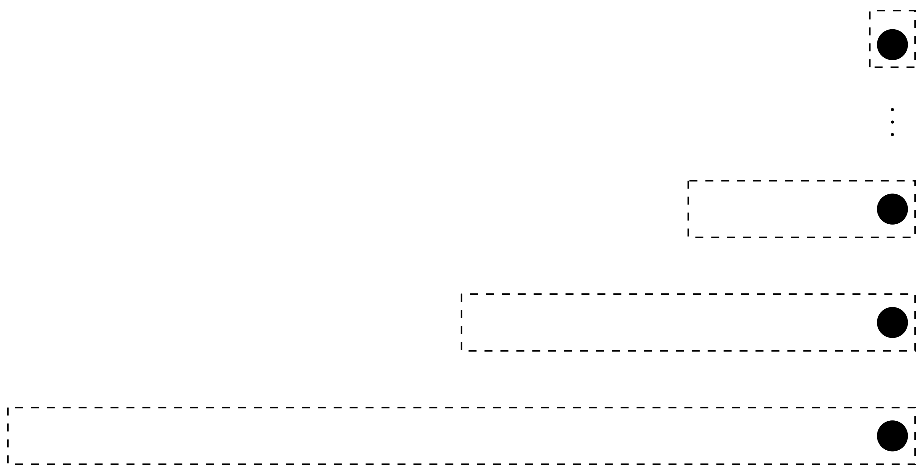


Figure 3.1: Inductive hypothesis for `parts` proof

The outline of the inductive step of the proof is as follows:

1. Assume $[\ xs\ ] \in$ parts xs.
2. Then by ∈-filter-parts, we have $[\ x :: xs\ ] \in$ parts-filter x (parts xs)
3. $[\ x :: xs\ ] \in$ parts-filter x (parts xs) $\Longrightarrow$
   $[\ x :: xs\ ] \in$ (map ($[\ x\ ]$ ::) (parts xs)) ++ (parts-filter x (parts xs))
   (Concatenating preserves the elements in both lists.)
4. Therefore, $[\ x :: xs\ ] \in$ parts $(x :: xs)$

Point (3) made use of the $\in$-$++^r$ result shown earlier.

I arrived at the complete the proof for $[\ xs\ ] \in$ parts xs by invoking all the sub-lemmas above.

```
∈-parts : {A : Set} → (xs : List A) → xs ≢ [] → [ xs ] ∈ parts xs
```

■

Another result which was crucial to proving the monotonicity and invertibility of the repetition combinator was parts-map, which states that map (_::_ x) (parts xs) is an order-preserving sublist of parts (x ++ xs) (for non-empty xs).

```
parts-map : {A : Set} → (xs ys : List A) → xs ≢ [] →
            map (_::_ xs) (parts ys) ⊆ parts (xs ++ ys)
```

parts-map was useful in simplifying proofs where I needed to show the existence of an element in parts (xs ++ ys). Demonstrating this is not easy as parts (xs ++ ys) is not defined solely in terms of the partitions of xs and ys. map (_::_ xs) (parts ys), on the other hand, satisfies this criterion, and so is easier to reason about.

3.2 Partial isomorphisms

Recall partial isomorphisms from §2.4.2. In this section I present their definition in Agda, extending the Haskell definition given in [23] to include their invertibility property.

Definition 3.2.1 *A partial isomorphism is a pair of functions – $f : A \to \mathsf{Maybe}\ B$ and $g : B \to \mathsf{Maybe}\ A$ – such that for all $a : A$ and $b : B$*

$$f(a) \equiv \mathsf{just}\ b \iff g(b) \equiv \mathsf{just}\ a$$

I implemented partial isomorphisms in Agda via records, as dependent records make it possible to package functions together with their properties. I split my definition of partial isomorphisms into two records – one to contain the functions (Iso), and the other to contain the two properties (IsIso). Such a separation of objects from their properties is standard practice in Agda's libraries[1].

```
record IsIso {A B : Set} (f : A → Maybe B) (g : B → Maybe A) : Set where
  constructor
    _,_
  field
    prop₁ : (a : A) (b : B) → f a ≡ just b → g b ≡ just a
    prop₂ : (b : B) (a : A) → g b ≡ just a → f a ≡ just b

record Iso (A B : Set) : Set where
  constructor
    _,_⟨_⟩
  field
    f  : A → Maybe B
    g  : B → Maybe A
    is-iso : IsIso f g
```

The constructor field makes it possible to define records in mixfix syntax. That is, given functions f and g and properties p_1 and p_2 with the appropriate type signatures, we can construct its partial isomorphism as:

$$f\ ,\ g\ \langle\ p_1\ ,\ p_2\ \rangle$$

We can then define the following functions to use partial isomorphisms:

```
inverse : {A B : Set} → Iso A B → Iso B A
inverse (f , g ⟨ p₁ , p₂ ⟩) = g , f ⟨ p₂ , p₁ ⟩

apply : {A B : Set} → Iso A B → A → Maybe B
apply (f , _ ⟨ _ ⟩) = f
```

[1]The Algebra library for example: `www.cse.chalmers.se/~nad/listings/lib-0.7/Algebra.html`

```
unapply : {A B : Set} → Iso A B → B → Maybe A
unapply (_ , g ⟨ _ ⟩) = g
```

The above definition of partial isomorphisms forces programmers to prove their claims about the invertibility of their functions before they can use them. This is unlike most other languages, where the programmer can only *assume* such claims, but cannot mechanically verify them.

3.2.1 Algebra of partial isomorphisms

Partial isomorphisms are central to programming with invertible syntax descriptions. I discuss here how programming with partial isomorphisms can be accomplished while maintaining invertibility guarantees and maximising convenience of use. Although programmers do have the option of writing new partial isomorphisms each time one is required, this approach necessitates proving the invertibility of every isomorphism defined, which soon becomes cumbersome.

Rather than defining partial isomorphisms from scratch every time, it is more convenient to have a set of defined primitive partial isomorphisms along with some functions that let us combine them in different ways. Rendel and Ostermann provide such functions in [23]. I ported these functions to my partial isomorphism library and extended them by providing mechanically checked proofs of correctness, showing that their resultant values are indeed always partial isomorphisms. I list below two such functions; the complete set can be found in Appendix A.

The identity partial isomorphism:

```
id : {A : Set} → Iso A A
id = just , just ⟨ (λ a b prf → sym prf) , (λ b a prf → sym prf) ⟩
```

id is the identity partial isomorphism (it is analogous to the identity function $f(x) = x$). Both the functions of id simply wrap their argument in a just constructor. Its proofs of invertibility require us to show that $x \equiv y \implies y \equiv x$ and vice-versa, i.e., that propositional equality is symmetric, which is shown by the standard library function sym. I prove in §4.3 that id applied to any syntax description combinator returns the unmodified combinator.

Composition:

```
_∘_ : {A B C : Set} → Iso B C → Iso A B → Iso A C
_∘_ (g , g⁻¹ ⟨ gp₁ , gp₂ ⟩) (f , f⁻¹ ⟨ fp₁ , fp₂ ⟩) = (f >=>ₘ g) , (g⁻¹ >=>ₘ f⁻¹) ⟨ ... ⟩
```

_∘_ is the composition function for partial isomorphisms. The implementation of _∘_ uses the Kleisli composition operator discussed in §3.1.3. I proved the invertibility of composition by making use of two lemmas: compose and interpolate.

```
compose : {X Y Z : Set} → (x : X) → (y : Y) → (z : Z) →
          (u : X → Maybe Y) → (v : Y → Maybe Z) →
          (u x ≡ just y) → (v y ≡ just z) →
          (u >=>m v) x ≡ just z

interpolate : {X Y Z : Set} → (x : X) → (z : Z) →
              (u : X → Maybe Y) → (v : Y → Maybe Z) →
              (u >=>m v) x ≡ just z →
              ∃ λ y → (u x ≡ just y) × (v y ≡ just z)
```

interpolate is an existential proof which shows the existence of an intermediate value y when the Kleisli composition of two Maybe values is given. Recall from §2.1 that proofs of existential statements in Agda are encoded as pairs where the second value depends on the first. The statement "$\exists x.P(x)$" (where P is a predicate) is written in Agda as:

```
existential : {A : Set} → (P : A → Set) → ∃ λ x → P x
```

A proof for this statement would be a pair where the first element is a satisfying value of x (say x_0) and the second element is a proof demonstrating that $P(x_0)$ holds. Proving interpolate consisted of presenting such a pair. I showed this by case-splitting on the result of $u\ x$ in order to indicate to Agda what the interpolating value could be.

I pass the value y, computed using interpolate, to compose which uses it to glue two other Kleisli compositions together, thereby completing the proof.

3.3 Syntax description combinators

In §2.4 I presented all the syntax description combinators and their respective meanings. Here I show the details of how I constructed these combinators in Agda. The implementations of the elementary, isofunctor, choice, and sequence combinators are based on the ones given in *Invertible Syntax Descriptions* [23], whereas the repetition combinator was designed and implemented from scratch. I implemented all the combinators using functions introduced in §3.1.1 and §3.1.3. §4 shows these combinators in use.

3.3.1 Elementary combinators
`pure`, `token`, and `empty`

Pure The pure elementary combinator is parametrised by an existing value x. Its resulting parser and printer relate the empty input with x.

A pure x parser returns x without consuming any input.

```
parser-pure : {A : Set} → Decidable _≡_ → A → Parser A
parser-pure dec x = parser (λ y → [ (x , y) ])
```

A pure x printer discards values that are equal to x. Equality is tested by *dec*, a function of type Decidable _≡_ which compares x to y, given an equality relation (_≡_) for them, and returns explicit proofs of whether or not x and y are equal. Values of type Decidable have two constructors: yes and no, corresponding to whether or not the two arguments are equal. Then, by pattern matching on the comparison, the appropriate results are returned.

```
pure-internal : {A : Set} → Decidable _≡_ → A → A → Maybe (List Char)
pure-internal dec x y with dec x y
pure-internal dec x y | yes p = just []
pure-internal dec x y | no ¬p = nothing

printer-pure : {A : Set} → Decidable _≡_ → A → Printer A
printer-pure dec x = printer (λ y → pure-internal dec x y)
```

Token The token combinator relates each character with itself.

As a parser, token returns the first character of its input string. parser-token therefore succeeds if a single character string is given to it, and fails otherwise.

```
token-internal : String → List (Char × String)
token-internal [] = []
token-internal (x :: xs) = [ x , xs ]

parser-token : Parser Char
parser-token = parser token-internal
```

The printer instance just returns its input character as a single-character string.

```
printer-token : Printer Char
printer-token = printer (λ x → just [ x ])
```

Empty The empty combinator always fails. Both its parser and printer instances reflect this by returning their respective failure values – the parser returns the empty list and the printer returns nothing.

```
parser-empty : {A : Set} → Parser A
parser-empty = parser (λ s → [])

printer-empty : {A : Set} → Printer A
printer-empty = printer (λ a → nothing)
```

3.3.2 Choice combinator ⟨|⟩

Parsing The choice parser combinator, ⟨|⟩, takes two parsers and returns their concatenated individual results for any input.

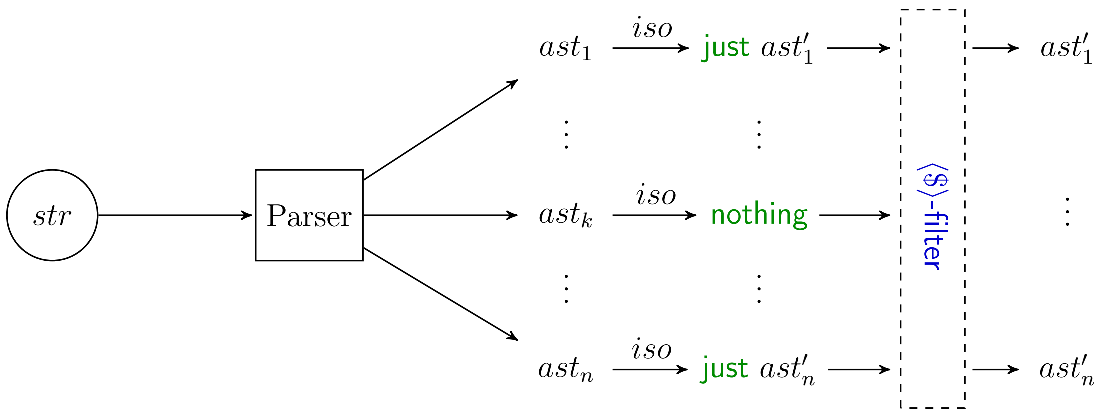


Figure 3.2: Operation of isofunctor parser combinator

```
_⟨|⟩_ : {A : Set} → Parser A → Parser A → Parser A
parser p₁ ⟨|⟩ parser p₂ = parser (λ s → p₁ s ++ p₂ s)
```

Printing The choice printer combinator, ⟨|⟩, takes two printers; given an input, it uses either printer that doesn't fail, starting with the first one.

```
_⟨|⟩_ : {A : Set} → Printer A → Printer A → Printer A
printer p₁ ⟨|⟩ printer p₂ = printer (λ a → mplus (p₁ a) (p₂ a))
```

See §3.1.3 for the definition of mplus.

3.3.3 Isofunctor combinator ⟨\$⟩

Parsing Given a partial isomorphism *iso* and a parser p, the parser p ⟨\$⟩ *iso* runs its input through p and transforms the resultant values by applying *iso* in the forward direction.

```
_⟨$⟩_ : {A B : Set} → Iso A B → Parser A → Parser B
iso ⟨$⟩ (parser p) = parser (λ s → ⟨$⟩-filter (p s                         ≫=ₗ (λ x →
                                                [ (apply iso) (proj₁ x) ] ≫=ₗ (λ y →
                                                [ y , (proj₂ x) ]))))
```

The above code can be viewed alternatively as a list comprehension in Haskell (see §3.1.1 for an explanation of the conversion):

$$\langle\$\rangle\text{-filter}\ [\ (y,\ proj_2\ x)\ |\ x \leftarrow (p\ s),\ y \leftarrow (apply\ iso)\ (proj_1\ x)\ ]$$

Description: the pair x contains the results of applying p to the input string s. The partial isomorphism is applied in the forward direction to each parsed value (note that apply *iso* returns the first function of the pair). Finally, all the nothings are removed and the values inside just constructors are extracted using ⟨\$⟩-filter. I illustrate this process in Figure 3.2.

Printing Given a partial isomorphism *iso* and a printer p, the printer p ⟨\$⟩ *iso* transforms its input by applying *iso* in the reverse direction, and then prints it using p.

```
_⟨$⟩_ : {A B : Set} → Iso A B → Printer A → Printer B
iso ⟨$⟩ (printer p) = printer (λ b → (unapply iso) b ≫=ₘ p)
```

Description: the printer is given a value b of type B as input. It applies the partial isomorphism in the reverse direction (unapply *iso*), transforming it to type Maybe A, which is passed to p via the bind function (discussed in §3.1.3).

3.3.4 Sequence combinator ⟨•⟩

Parsing The sequence parser combinator, ⟨•⟩, takes two parsers; it runs its input string through the first parser, threading the remainder string to the second parser and returning the pair of values parsed.

```
_⟨•⟩_ : {A B : Set} → Parser A → Parser B → Parser (A × B)
parser p₁ ⟨•⟩ parser p₂ = parser (λ s → (p₁ s)  ≫=ₗ (λ x →
                                  (p₂ (proj₂ x))  ≫=ₗ (λ y →
                                  [ ((proj₁ x) , proj₁ y) , proj₂ y ])))
```

This parser makes use of the technique to describe list comprehensions (discussed in §3.1.1). It can alternatively be viewed as:

$$[\ ((proj_1\ x,\ proj_1\ y),\ proj_2\ y) \mid x \leftarrow (p_1\ s),\ y \leftarrow p_2\ (proj_2\ x)\ ]$$

Printing The sequence printer combinator, ⟨•⟩, takes two printers; if either printer fails on the input then it fails overall, otherwise it returns the concatenated results of the printers.

```
_⟨•⟩_ : {A B : Set} → Printer A → Printer B → Printer (A × B)
_⟨•⟩_ (printer p₁) (printer p₂) = printer (λ ab → liftM2  _++_
                                                          (p₁ (proj₁ ab))
                                                          (p₂ (proj₂ ab)))
```

See §3.1.3 for the definition of liftM2.

3.3.5 Repetition combinator ⟨⋆⟩

Parsing Given a parser p, ⟨⋆⟩ p is a parser that runs p sequentially on the input, threading the remaining string to itself until it reaches the end.

$$\begin{tikzcd}
{[} & ast_1 \arrow[d, "p"] & , & ast_2 \arrow[d, "p"] & , & \cdots & , & ast_n \arrow[d, "p"] & {]} & \\
 & str_1 & ++ & str_2 & ++ & \cdots & ++ & str_n & \equiv & str
\end{tikzcd}$$

Figure 3.3: Repetition printer combinator

```
parseList : {A : Set} → Parser A → List (String) → List (List A × String)
parseList _ [] = []
parseList (parser p) (str ∷ []) = map (λ x → [ proj₁ x ] , (proj₂ x)) (p str)
parseList par (str ∷ strs) = (parse par str)      ≫=ₗ (λ x →
                             (parseList par strs) ≫=ₗ (λ ys →
                             [ (x ∷ (proj₁ ys)) , (proj₂ ys) ]))
```

```
⟨⋆⟩_ : {A : Set} → Parser A → Parser (List A)
⟨⋆⟩_ p = parser (λ str → (parts str) ≫=ₗ (λ s → parseList p s))
```

The repetition combinator makes use of the partitioning function defined in §3.1.4. It partitions the input string into all possible tokens and passes that list to the parseList function. The parseList function parses each token with the given parser resulting in a set of parses for each token. Finally it computes the Cartesian product of all those sets and returns the list of results. The reason I implemented this combinator in this manner and not in a recursive one was because the recursive definition does not terminate if given a non-consuming combinator. Due to the finiteness of my definition, non-consuming combinators simply fail when provided to the repetition combinator, which matches their expected semantics (since in the case for parsing, non-termination is equivalent to failure).

Printing Given a printer p, $\langle\star\rangle\ p$ is a printer that can be applied to a list of items to print sequentially; it fails overall if any one element fails to get printed.

```
⟨⋆⟩_ : {A : Set} → Printer A → Printer (List A)
⟨⋆⟩_ (printer p) = printer (λ as → foldr (liftM2 _++_) (just []) (map p as))
```

The repetition printer first applies the given printer to each element of the list. The resultant list of Maybe Strings is then printed by concatenating all the values inside just constructors together, which makes use of the standard foldr function and liftM2 (discussed in §3.1.3). Note that if any value in the list is nothing, the entire result of concatenation will be nothing due to the definition of liftM2. I illustrate this in Figure 3.3.

3.4 Strictness

The property of strictness of parsers demonstrates that consuming parsers, when given the empty input, return the empty string. This property shows that parsers are well-behaved on null inputs. I use the strictness of parsers to show the invertibility of the repetition combinator in §3.6.

A function $f : X \to Y$ is defined to be strict if it maps the least value of X to the least value of Y. I show that all consuming parsers p are strict by proving that unParser p [] ≡ []. Recall that unParser extracts the internal parsing function from a parser.

I packaged the property of strictness of a parser in a Strict data type. Strict has one constructor which allows us to show that a parser is strict if the internal function (returned by unParser) applied to the empty list returns the empty list.

```
data Strict {A : Set} : Parser A → Set where
  strict : (p : Parser A) → unParser p [] ≡ [] → Strict p
```

Lemma 3.4.1 (Strictness) *All consuming parsers p : Parser A are strict.*

I proved this lemma by case analysis and showed it holds for each combinator.

The proof of each case followed by definition. This was because Agda automatically performs the necessary reductions when the parsers are given the empty list as input. This resulted in showing for each case that [] ≡ [], which is true by refl. I list all the proofs in Agda below, which are universally quantified for all parsers (p, p_1, p_2) and partial isomorphisms (*iso*).

```
token-strict       : Strict token
empty-strict       : Strict empty
choice-strict      : Strict p₁  → Strict p₂ → Strict (p₁ ⟨|⟩ p₂)
isofunctor-strict  : Strict p   → Strict (iso ⟨$⟩ p)
sequence-strict    : Strict p₁  → Strict p₂ → Strict (p₁ ⟨•⟩ p₂)
repetition-strict  : Strict p   → Strict (⟨⋆⟩ p)
```

We can show any consuming parser to be strict using the cases above.

■

3.5 Monotonicity

An assumption that is central to proving the invertibility of the sequence combinator is the property that parsers only parse characters in order of their appearance in the input string.

This is equivalent to saying that if a parser on input *str* : String parses a value *ast* : A, then on an input of *str* ++ *str′* its list of unfiltered results *must* contain (*ast*, *str′*) for any string str' (that is, str' remains unconsumed in at least one case). In other words, such a property guarantees that the results of a parser cannot get deleted on providing it with additional input. Define this property as the *monotonicity* of a parser:

Definition 3.5.1 (Monotonicity) *A consuming parser p : Parser A is monotonic on input string str : String and parsed value ast : A if for all input strings str' : String,*

$$(ast, str') \in \textsf{unParser}\ p\ (str +\!\!+\ str')$$

Recall that unParser p $(str +\!\!+\ str')$ returns all the parsed results along with their respective remainder strings – in the statement above, ast is the parsed value and str' is the remainder string.

Such a definition translates to the following predicate in Agda:

```
data ⟦_⟧⟨_⇒_⟩ {A : Set} : Parser A → String → A → Set where
  monotonic : (p : Parser A) → (str : String) → (ast : A) →
              ((str′ : String) → (ast , str′) ∈ unParser p (str ++ str′)) →
              ⟦ p ⟧⟨ str ⇒ ast ⟩
```

And so, we can show a parser p to be monotonic on str and ast if we can construct an object of type ⟦ p ⟧⟨ str ⇒ ast ⟩ in Agda. This is read as "the parser p mapping str to ast is monotonic". I illustrate monotonicity in Figure 3.4, where we can see that apart from newly parsed results, the previously parsed results still exist with the new string str' concatenated to the end.

Theorem 3.5.1 (Monotonicity) *All consuming parsers are monotonic.*

I proved this theorem by case analysis. From the elementary combinators, monotonicity only holds for token as it is the only non-consuming non-failing elementary combinator. All the proofs below are universally quantified over their input parsers (p, p_1, p_2), strings (str, str_1, str_2), and parsed values (ast, ast_1, ast_2). I explain in detail here the proof of monotonicity for the isofunctor combinator. The remaining proofs followed similarly.

Token: The proof followed by definition of the token combinator.

```
token-monotonic : (ch : Char) → ⟦ token ⟧⟨ [ ch ] ⇒ ch ⟩
```

Choice: There are two proofs of monotonicity of the choice combinator due to the two branches it can choose between.

```
choice-monotonic₁ : ⟦ p₁ ⟧⟨ str ⇒ ast₁ ⟩ → ⟦ p₂ ⟧⟨ str ⇒ ast₂ ⟩ →
                    ⟦ p₁ ⟨|⟩ p₂ ⟧⟨ str ⇒ ast₁ ⟩

choice-monotonic₂ : ⟦ p₁ ⟧⟨ str ⇒ ast₁ ⟩ → ⟦ p₂ ⟧⟨ str ⇒ ast₂ ⟩ →
                    ⟦ p₁ ⟨|⟩ p₂ ⟧⟨ str ⇒ ast₂ ⟩
```

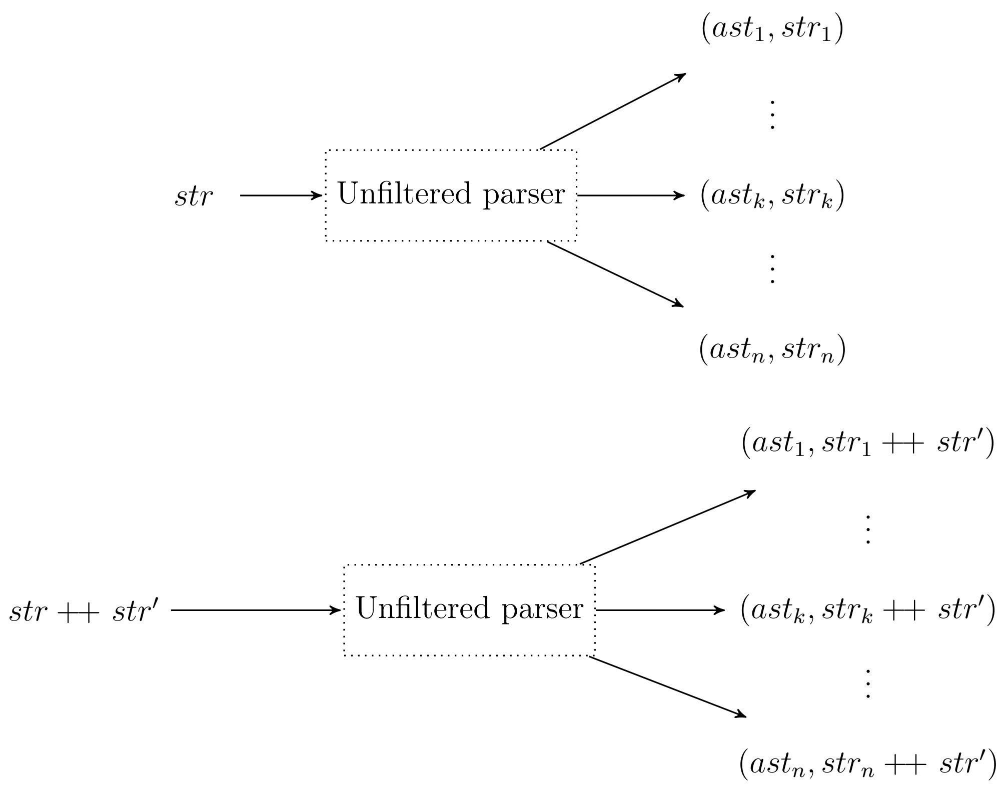


Figure 3.4: Monotonicity of a parser

For both proofs I showed that if an element is a member of one list, then it is also a member of that list concatenated with any other list. This used the lemmas $\in$-++^{l} and $\in$-++^{r} (introduced in §3.1.2).

Isofunctor: I proved that if a partial isomorphism transforms ast to ast', then the transformed parser is also monotonic on the same input string. The proof of this case has the following type signature:

$$\begin{aligned}\text{isofunctor-monotonic} : \ & (iso : \text{Iso}\ A\ B) \to \text{apply}\ iso\ ast \equiv \text{just}\ ast' \to \\ & [\![\ p\ ]\!]\langle\ str \Rightarrow ast\ \rangle \to [\![\ iso\ \langle\$\rangle\ p\ ]\!]\langle\ str \Rightarrow ast'\ \rangle\end{aligned}$$

Given the inductive hypothesis that p mapping str to ast is monotonic, I demonstrated using $\in$-map and $\in$-concat that apply iso (ast, str') is an element of the unfiltered resultant parser. By the assumption that iso when applied to ast returns just ast', I could demonstrate that this value is equivalent to (just ast', str'). The statement to prove is that (ast', str') is a member of the final parser – I showed this by applying $\langle\$\rangle$-filter-lemma (defined in §3.1.2) to the previously proved result, thereby arriving at the required list comprehension expression.

Sequence: If two parsers are monotonic on values ast_1 and ast_2 with inputs str_1 and str_2 respectively, then the combined parser is monotonic on the concatenated input.

$$\begin{aligned}\text{sequence-monotonic} : \ & [\![\ p_1\ ]\!]\langle\ str_1 \Rightarrow ast_1\ \rangle \to [\![\ p_2\ ]\!]\langle\ str_2 \Rightarrow ast_2\ \rangle \to \\ & [\![\ p_1\ \langle\bullet\rangle\ p_2\ ]\!]\langle\ str_1 ++ str_2 \Rightarrow (ast_1\ ,\ ast_2)\ \rangle\end{aligned}$$

This was proved in a manner similar to the monotonicity of the isofunctor combinator. I started out with the assumption of the inductive hypothesis and then invoked ∈-map and ∈-concat as necessary to build the final expression.

Repetition: There are two proofs for the repetition combinator, corresponding to the two cases in the parseList function. In the first case: if a parser p parses *ast* on input *str*, then $\langle\star\rangle\ p$ is monotonic on parse value [*ast*] and the input *str*. This proof made use of the fact that all parsers are strict, which I showed in §3.4.

$$\mathsf{repetition\text{-}monotonic}_1 : \mathsf{Strict}\ p \to [\![\ p\ ]\!]\langle\ str \Rightarrow ast\ \rangle \to [\![\ \langle\star\rangle\ p\ ]\!]\langle\ str \Rightarrow [\ ast\ ]\ \rangle$$

The second case shows how the monotonicity of a parser and a repeated parser can be combined to prove the monotonicity for the repeated parser on a different input.

$$\begin{aligned}\mathsf{repetition\text{-}monotonic}_2 : {} & [\![\ p\ ]\!]\langle\ str \Rightarrow ast\ \rangle \to [\![\ \langle\star\rangle\ p\ ]\!]\langle\ strs \Rightarrow asts\ \rangle \to \\ & [\![\ \langle\star\rangle\ p\ ]\!]\langle\ str \mathbin{+\!+} strs \Rightarrow ast :: asts\ \rangle\end{aligned}$$

I proved both these cases by using ∈-parts, ∈-map, $\in\text{-}{+\!+}^l$, and $\in\text{-}{+\!+}^r$ as required in order to build the final expressions to prove. They were similar to the isofunctor and sequence monotonicity proofs.

With all the individual cases proved, this proof is complete. I have summarised this proof in Figure 4.2, where I evaluate the usefulness of monotonicity. ■

3.6 Invertibility

In this Section I establish the validity of all the parsers and printers that have been constructed so far by proving that they fulfill their intended purpose. Before starting, I first formalise the statement to be proved.

A central aim of this project was to show that for any input the action of printing it as a string and then immediately parsing that string will result in the original input. Call this property of being able to retrieve the original argument after the combined action of printing and parsing the *invertibility* of syntax descriptions. In the formal definition of invertibility, I took the following points into consideration:

- The parse and print functions cannot simply be composed with each other due to their different type signatures. To parse a printed value one needs to unwrap the just constructor first.
- A parser may return multiple parses if the defined grammar is ambiguous.
- A printer can fail returning nothing, in which case a parser will not be able to parse anything.

This brings us to the definition of invertibility of a parser-printer pair. From here on I will refer to the input given to the printer as the “abstract syntax tree” (or AST in short) out of convention – this does not indicate the input necessarily being a tree. Combinators with the subscript a refer to p**a**rsers and those with subscript i refer to pr**i**nters.

Definition 3.6.1 (Invertibility) *A parser par : Parser A and a printer pri : Printer A form an invertible pair with values ast : A and str : String if it can be shown that:*

- *print pri ast ≡ just str – i.e., the printer can successfully print ast to the string str.*
- *ast ∈ parse par str – i.e., ast is included in the set of results returned on parsing the string str.*

I defined the following notation to show that *par*, *pri*, *ast*, and *str* satisfy the property of invertibility – it indicates that the parsed value *ast* and the string *str* are inter-convertible via the parser-printer pair of *par* and *pri*.

$$\langle ast\ [\![par \iff pri]\!]\ str\rangle$$

I provide another visualisation of the property of invertibility in Figure 3.5, where I show what invertibility means for ambiguous and unambiguous grammars. In the ambiguous case, there always exists at least one parsed result (ast_k) which can be printed back as *str*. This definition and notation correspond to the following definition in Agda:

```
data ⟨_⟦_⇔_⟧_⟩ {A : Set} : A → Parser A → Printer A → String → Set where
  invertible : (ast : A) → (par : Parser A) → (pri : Printer A) → (str : String) →
               print pri ast ≡ just str → ast ∈ parse par str →
               ⟨ ast ⟦ par ⇔ pri ⟧ str ⟩
```

This brings us to the main result which certifies the correctness of my implementation.

Theorem 3.6.1 (Invertibility) *For any syntax description combinator, the parser-printer pair extracted from it form an invertible pair for some values ast and str.*

Similar to the proofs before, I proved this theorem by case analysis. As in the proof of monotonicity, all the input values of the statements below have been universally quantified.

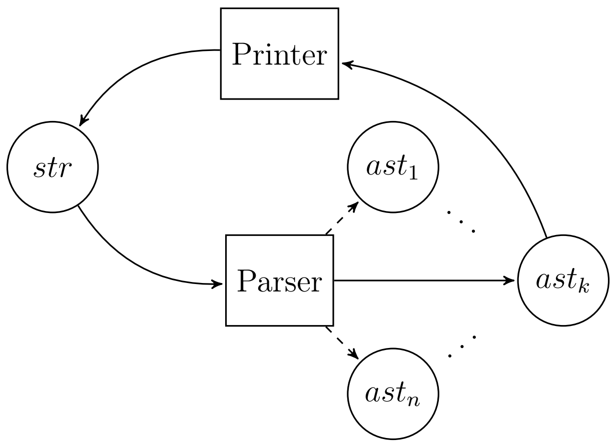
For each case I had to provide two proofs: (1) a proof of the printer successfully printing the AST into a string and (2) a proof of the AST being a member of the list of parsed results.

Empty: There is no proof of invertibility of the empty combinator as it always fails.

Pure: Given *ast* : A (and a function *dec* to compute equality between values of type A), the pure parser and printer form an invertible pair with *ast* and the empty string.

```
pure-invertible : {A : Set} → (ast : A) → (dec : Decidable (_≡_ {A = A})) →
                  let par = parser-pure dec ast
                      pri = printer-pure dec ast
                  in ⟨ ast ⟦ par ⇔ pri ⟧ [] ⟩
```

The proof followed by definition.



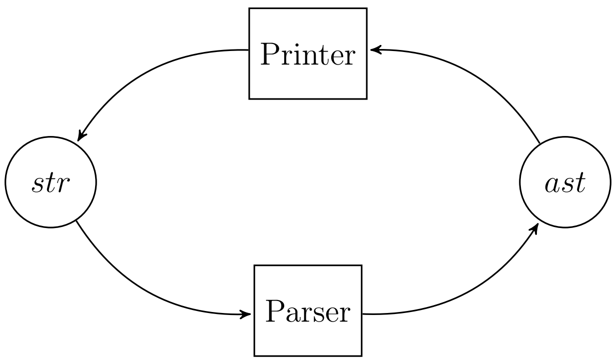


Figure 3.5: Top: ambiguous grammar. Bottom: unambiguous grammar. Undashed arcs indicate paths satisfying invertibility.

Token: Given a character ch, the token parser and printer form an invertible pair with the character ch and the singleton string $[ch]$.

$$\mathsf{token\text{-}invertible} : (ch : \mathsf{Char}) \to \langle\ ch\ [\![\ \mathsf{token}_a \Leftrightarrow \mathsf{token}_i\ ]\!]\ [\ ch\ ]\ \rangle$$

The proof followed by definition.

Choice: The choice combinator has two proofs, one for each branch it can take. Each branch corresponds to the following proof in Agda:

$$\begin{aligned}\mathsf{choice\text{-}inv}_1 :\ & \langle\ ast_1\ [\![\ par_1 \Leftrightarrow pri_1\ ]\!]\ str_1\ \rangle \to \\ & \langle\ ast_2\ [\![\ par_2 \Leftrightarrow pri_2\ ]\!]\ str_2\ \rangle \to \\ & \mathsf{let}\ par = par_1\ \langle|\rangle_a\ par_2 \\ & \quad\ pri = pri_1\ \langle|\rangle_i\ pri_2 \\ & \mathsf{in}\ \ \langle\ ast_1\ [\![\ par \Leftrightarrow pri\ ]\!]\ str_1\ \rangle\end{aligned}$$

$$\begin{aligned}\mathsf{choice\text{-}inv}_2 :\ & \langle\ ast_1\ [\![\ par_1 \Leftrightarrow pri_1\ ]\!]\ str_1\ \rangle \to \\ & \langle\ ast_2\ [\![\ par_2 \Leftrightarrow pri_2\ ]\!]\ str_2\ \rangle \to \\ & \mathsf{let}\ par = par_1\ \langle|\rangle_a\ par_2 \\ & \quad\ pri = pri_1\ \langle|\rangle_i\ pri_2 \\ & \mathsf{in}\ \ \mathsf{print}\ pri_1\ ast_2 \equiv \mathsf{nothing} \to \langle\ ast_2\ [\![\ par \Leftrightarrow pri\ ]\!]\ str_2\ \rangle\end{aligned}$$

The proofs for both branches are similar; we have the precondition in the second proof (print pri_1 ast_2 ≡ nothing) to force it to take the second branch when printing. This precondition is always satisfied as long as the two combinators combined by the choice combinator do not operate on similar inputs (which is the case for unambiguous grammars).

Both the proofs of successful printing followed by definition of mplus. I showed the proofs of membership by invoking ∈-++^{l} and ∈-++^{r}, similar to the proofs of monotonicity.

Isofunctor: The proof of invertibility of the isofunctor combinator states that if *ast* and *str* are inter-convertible using a parser-printer pair, then we can deduce that *ast′* and *str* are inter-convertible for the parser-printer pair on which the appropriate partial isomorphism has been applied. I formulated this in Agda as:

```
isofunctor-inv : ⟨ ast ⟦ par ⇔ pri ⟧ str ⟩ →
                 let par′ = iso ⟨$⟩ₐ par
                     pri′ = iso ⟨$⟩ᵢ pri
                 in  apply iso ast ≡ just ast′ →
                     ⟨ ast′ ⟦ par′ ⇔ pri′ ⟧ str ⟩
```

The proof of successful printing was shown by invoking the invertibility property of partial isomorphisms, which in this case produced: unapply *iso ast′* ≡ just *ast*.

I showed that *ast′* is a result on parsing *str* by making use of the fact that apply *iso ast* ≡ just *ast′* and the results ∈-map and ≫=$_l$-map to arrive at the correct expression (similar to the proof of monotonicity).

Sequence: The proof of invertibility of the sequence combinator shows the desired form of sequential behaviour in parsing and printing, which I show in the Agda snippet below:

```
sequence-inv : ⟦ par₁ ⟧⟨ str₁ ⇒ ast₁ ⟩ →
               ⟨ ast₁ ⟦ par₁ ⇔ pri₁ ⟧ str₁ ⟩ →
               ⟨ ast₂ ⟦ par₂ ⇔ pri₂ ⟧ str₂ ⟩ →
               let par = par₁ ⟨•⟩ₐ par₂
                   pri = pri₁ ⟨•⟩ᵢ pri₂
               in  ⟨ ast₁ , ast₂ ⟦ par ⇔ pri ⟧ str₁ ++ str₂ ⟩
```

The proof of successful printing followed by definition of liftM2 and the induction hypothesis.

Showing that the pair (ast_1, ast_2) was one of the parsed results made use of the fact that all parsers (specifically par_1 in our case) are monotonic. This result followed as a special case of the monotonicity of the sequence combinator.

Repetition: There are two proofs for the repetition combinator, which is due to the two cases in the parseList function. In the first case I demonstrated that applying the repeated parser on the same string returns the original AST as a singleton list. This was written in Agda as:

```
repetition-inv₁ : ⟨ ast ⟦ par ⇔ pri ⟧ str ⟩ →
                  let par′ = ⟨⋆⟩ₐ par
                      pri′ = ⟨⋆⟩ᵢ pri
                  in  ⟨ [ ast ] ⟦ par′ ⇔ pri′ ⟧ str ⟩
```

The printing proof was shown by definition of liftM2, and the parsing proof was demonstrated using ∈-parts and ∈-map to construct the final expression. In the second case I showed that if a parser-printer pair parses and prints a pair of values and if the repeated parser-printer pair parses and prints another pair, then the repeated parser-printer pair should be able to parse and print the combined pairs. In Agda:

```
repetition-inv₂ : ⟨ ast ⟦ par ⇔ pri ⟧ str ⟩ →
                  let par′ = ⟨⋆⟩ₐ par
                      pri′ = ⟨⋆⟩ᵢ pri
                  in  strs ≢ [] →
                      ⟨ asts ⟦ par′ ⇔ pri′ ⟧ strs ⟩ →
                      ⟨ ast :: asts ⟦ par′ ⇔ pri′ ⟧ str ++ strs ⟩
```

The proof of successful printing followed by definition of liftM2 and the inductive hypotheses.

In the proof of parsing I needed to show that $ast :: asts$ is an element of (a transformation of) parts $(str \mathbin{++} strs)$. Since the expression parts $(str \mathbin{++} strs)$ cannot be written as an expression only in terms of parts str and parts $strs$, I considered a sublist of parts $(str \mathbin{++} strs)$. I showed that:

$$(ast :: asts) \in (\mathsf{map}\ (str\ ::)\ (\mathsf{parts}\ strs))$$

which is a sublist of parts $(str \mathbin{++} strs)$, which I showed by parts-map. For this case overall, I made use of ∈-parts and ∈-map as in the first case, which completed the proof.

With all the individual cases proved, this proof is complete. I have summarised this proof in Figure 4.1 where I discuss its effectiveness.

■

3.7 Summary

In this Chapter I have shown the implementations of the basic syntax description combinators. In the second half of the Chapter I provided mechanically checked proofs of their strictness, monotonicity, using which I proved their invertibility. I proved all these results by structural induction.

In the following Chapter I discuss the effectiveness of these proofs and evaluate the practicality of my syntax description library by constructing syntax descriptions for the grammars of primitive recursive functions and arithmetic expressions.

Chapter 4

Evaluation

In §3 I described how I constructed each syntax description combinator and showed that the parsers and printers they generated were inverses of each other. In this Chapter I evaluate my implementation from three different perspectives. In § 4.1 I discuss the effectiveness of the proofs presented at the end of the § 3, and show that they indeed state the invertibility guarantees expected from syntax descriptions. I discuss how I tested and validated my implementation during development (and before formally proving correctness) by unit testing (§ 4.2) and demonstrating the combinators' satisfiability of some algebraic laws (§4.3). Finally, I use these syntax description combinators to describe the syntax of primitive recursive functions and arithmetic expressions (§ 4.4), thereby evaluating the practicality of my combinators.

4.1 Discussion of proofs

Recall the definitions of monotonicity of parsers and invertibility of syntax descriptions from §3.5 and §3.6. A parser is monotonic if the addition of garbage data at the end of a string does not delete any previously parsed values. Such a property is fundamental to any good parser, and is not immediately apparent from the definition of parser combinators. Affeldt et al. [1] only postulated the monotonicity of parsers in their work, and never proved it. A parser and a printer are defined to be invertible if there exists a parsed value *ast* and string *str* such that they are inter-convertible using the parser and the printer. In this project I showed all my parser combinators to be monotonic, and all my syntax description combinators to satisfy the property of invertibility. I summarise the proofs of invertibility and monotonicity in Figures 4.1 and 4.2, where I present all the cases I proved.

The structure of the proof of invertibility additionally yields an operational semantics for syntax descriptions, which shows the relation between the parsed value and printed string at all stages of the computation. *Notation*: In the following proof cases, subscripts of a (resp. i) correspond to p***a***rser (resp. pr***i***nter) combinators. The proofs are quantified for all strings (*str*) and all parsed ASTs (*ast*).

$$(\textbf{PURE})\quad \frac{}{\left\langle ast\ \llbracket pure_a\ ast \Longleftrightarrow pure_i\ ast \rrbracket\ [\,] \right\rangle}$$

$$(\textbf{TOKEN})\quad \frac{}{\left\langle ch\ \llbracket token_a \Longleftrightarrow token_i \rrbracket\ [ch] \right\rangle}$$

$$(\textbf{CHOICE}_1)\quad \frac{\left\langle ast_1\ \llbracket par_1 \Longleftrightarrow pri_1 \rrbracket\ str_1 \right\rangle \qquad \left\langle ast_2\ \llbracket par_2 \Longleftrightarrow pri_2 \rrbracket\ str_2 \right\rangle}{\left\langle ast_1\ \llbracket par_1\ \langle | \rangle_a\ par_2 \Longleftrightarrow pri_1\ \langle | \rangle_i\ pri_2 \rrbracket\ str_1 \right\rangle}$$

$$(\textbf{CHOICE}_2)\quad \frac{\left\langle ast_1\ \llbracket par_1 \Longleftrightarrow pri_1 \rrbracket\ str_1 \right\rangle \qquad \left\langle ast_2\ \llbracket par_2 \Longleftrightarrow pri_2 \rrbracket\ str_2 \right\rangle}{\left\langle ast_2\ \llbracket par_1\ \langle | \rangle_a\ par_2 \Longleftrightarrow pri_1\ \langle | \rangle_i\ pri_2 \rrbracket\ str_2 \right\rangle}$$

$$(\textbf{ISO})\quad \frac{\left\langle ast\ \llbracket par \Longleftrightarrow pri \rrbracket\ str \right\rangle \qquad \texttt{apply}\ iso\ ast \equiv \texttt{just}\ ast'}{\left\langle ast'\ \llbracket iso\ \langle \$ \rangle_a\ par \Longleftrightarrow iso\ \langle \$ \rangle_i\ pri \rrbracket\ str \right\rangle}$$

$$(\textbf{SEQ})\quad \frac{\left\langle ast_1\ \llbracket par_1 \Longleftrightarrow pri_1 \rrbracket\ str_1 \right\rangle \qquad \left\langle ast_2\ \llbracket par_2 \Longleftrightarrow pri_2 \rrbracket\ str_2 \right\rangle}{\left\langle (ast_1, ast_2)\ \llbracket par_1\ \langle \bullet \rangle_a\ par_2 \Longleftrightarrow pri_1\ \langle \bullet \rangle_i\ pri_2 \rrbracket\ str_1 +\!\!+\ str_2 \right\rangle}$$

$$(\textbf{REP}_1)\quad \frac{\left\langle ast\ \llbracket par \Longleftrightarrow pri \rrbracket\ str \right\rangle}{\left\langle\ [ast]\ \llbracket \langle \star \rangle_a\ par \Longleftrightarrow \langle \star \rangle_i\ pri \rrbracket\ str \right\rangle}$$

$$(\textbf{REP}_2)\quad \frac{\left\langle ast\ \llbracket par \Longleftrightarrow pri \rrbracket\ str \right\rangle \qquad \left\langle\ asts\ \llbracket \langle \star \rangle_a\ par \Longleftrightarrow \langle \star \rangle_i\ pri \rrbracket\ strs \right\rangle}{\left\langle\ ast :: asts\ \llbracket \langle \star \rangle_a\ par \Longleftrightarrow \langle \star \rangle_i\ pri \rrbracket\ str +\!\!+\ strs \right\rangle}$$

Figure 4.1: Proofs of invertibility of syntax descriptions

I now discuss the complexity of my proofs, for which I use the metric of source lines of code (SLOC). The SLOC blow-up factor after converting unverified Haskell to verified Agda code provides, to an approximation, an indication of the time and effort expended in proving the correctness of the implementation. I list the SLOC blow-up factors for the proofs of monotonicity and invertibility for each combinator in Table 4.1.

From this Table we can observe that the proofs for the repetition combinator were most complex. This is due to the repetition combinator using the list partitioning function parts, which, as a result of its non-trivial definition, was reasoned about separately (§3.1.4). The monotonicity and invertibility of pure, token, and empty were shown to hold by definition, which resulted in very short proofs. Similarly, the relatively simple choice combinator,

$$(\textbf{TOKEN})\,\frac{}{\llbracket token_a \rrbracket \left\langle [ch] \Rightarrow ch \right\rangle}$$

$$(\textbf{CHOICE}_1)\,\frac{\llbracket p_1 \rrbracket \left\langle str_1 \Rightarrow ast_1 \right\rangle \qquad \llbracket p_2 \rrbracket \left\langle str_2 \Rightarrow ast_2 \right\rangle}{\llbracket p_1 \langle | \rangle\ p_2 \rrbracket \left\langle str_1 \Rightarrow ast_1 \right\rangle}$$

$$(\textbf{CHOICE}_2)\,\frac{\llbracket p_1 \rrbracket \left\langle str_1 \Rightarrow ast_1 \right\rangle \qquad \llbracket p_2 \rrbracket \left\langle str_2 \Rightarrow ast_2 \right\rangle}{\llbracket p_1 \langle | \rangle\ p_2 \rrbracket \left\langle str_2 \Rightarrow ast_1 \right\rangle}$$

$$(\textbf{ISO})\,\frac{\llbracket p \rrbracket \left\langle str \Rightarrow ast \right\rangle \qquad \texttt{apply}\ iso\ ast \equiv \texttt{just}\ ast'}{\llbracket iso\, \langle \$ \rangle\, p \rrbracket \left\langle str \Rightarrow ast' \right\rangle}$$

$$(\textbf{SEQ})\,\frac{\llbracket p_1 \rrbracket \left\langle str_1 \Rightarrow ast_1 \right\rangle \qquad \llbracket p_2 \rrbracket \left\langle str_2 \Rightarrow ast_2 \right\rangle}{\llbracket p_1 \langle \bullet \rangle\, p_2 \rrbracket \left\langle str_1 +\!\!+\ str_2 \Rightarrow (ast_1, ast_2) \right\rangle}$$

$$(\textbf{REP}_1)\,\frac{\llbracket p \rrbracket \left\langle str \Rightarrow ast \right\rangle}{\llbracket \langle \star \rangle\ p \rrbracket \left\langle str \Rightarrow [ast] \right\rangle}$$

$$(\textbf{REP}_2)\,\frac{\llbracket p \rrbracket \left\langle str \Rightarrow ast \right\rangle \qquad \llbracket \langle \star \rangle\ p \rrbracket \left\langle strs \Rightarrow asts \right\rangle}{\llbracket \langle \star \rangle\ p \rrbracket \left\langle str +\!\!+\ strs \Rightarrow ast :: asts \right\rangle}$$

Figure 4.2: Proofs of monotonicity of parsers

which I defined using list concatenation and the mplus function, resulted in straightforward proofs. Both the isofunctor and sequence proofs were of similar complexity as a result of being defined using list comprehensions, which involved proving a separate set of lemmas to reason about them (§3.1.1).

Overall, the implementation of combinators and partial isomorphisms accounts for 20% of my source code – the proofs of correctness form the remaining 80%. Correctness proofs being the bulk of the implementation is not unusual in formal verification. For example, in CompCert – a compiler for a subset of C formally verified in Coq – 76% of the SLOC corresponded to the correctness proofs [17], as of 2009.

To conclude, the syntax description combinators I implemented fit their expected specification of being able to correctly print and parse data from a single description. I showed this by providing machine checked proofs of an invertibility property which states that

Combinator	Monotonicity	Invertibility
pure	—	1×
token	1×	1×
Choice	4×	4×
Isofunctor	60×	99×
Sequence	65×	100×
Repetition	176×	230×

Table 4.1: SLOC blow-up factors for proofs of monotonicity and invertibility

any string that is the result of printing some abstract syntax can be parsed to obtain the original abstract syntax unmodified. This invertibility property completely characterises combined parsing and pretty printing for use cases where each string has a unique AST associated with it (such as in network packet processing), and provides a strong guarantee of correct parsing and printing for other grammars. In §5.2 I introduce another invertibility property (not explored in this project due to time and scope constraints) which, when combined with the invertibility theorem I proved, fully characterises combined parsing and pretty printing for all grammars.

4.2 Unit testing

In the combinator-construction phase of the project, the development of the combinators was guided by unit tests I designed. These tests not only found early bugs in the combinator implementations, but also contributed to increased confidence in their correct construction.

I accomplished unit testing for my project using Agda's type system, where I designed the tests as proofs of equality. Each equality compared the expression I wanted to evaluate with its expected result. By setting the proofs of all these tests to refl, I was able to reduce the process of testing to that of type checking. By this method, Agda would complain if the type checking failed, implying unexpected test results. Here is an example unit test:

```
parse-test : let p   = ⟨⋆⟩ comb
                 str = "AA"
             in  parse p str ≡ (X ∷ X ∷ []) ∷ (Y ∷ []) ∷ []
parse-test = refl
```

In the test above, my combinator comb maps the single string character "A" to the symbol X and "AA" to Y. This test ensures that if the repeated combinator was given the string "AA", then it will parse both [X,X] and [Y].

$$\begin{tikzcd}
 & b \arrow[rd, "g"] \arrow[dd, dashed, "\delta"] & \\
a \arrow[ru, "f"] \arrow[rr, "f \circ g"] \arrow[dd, dashed, "\delta"] & & c \arrow[dd, dashed, "\delta"] \\
 & \delta\ b \arrow[rd, "\langle\$\rangle g"] & \\
\delta\ a \arrow[ru, "\langle\$\rangle f"] \arrow[rr, "\langle\$\rangle(f \circ g)"] & & \delta\ c
\end{tikzcd}$$

Figure 4.3: Functor composition law

4.3 Verification of algebraic laws

Another method of validating my implementation before formally proving its correctness was to check whether or not my combinators satisfied algebraic laws known as the functor and monoid laws, which I explain below. The benefit of showing the satisfaction of these laws is twofold: firstly, it indicates that the combinators truly are algebraically well-behaved (due to functor and monoid research [16]), and secondly it enables programmers to readily use functions and design patterns from functor and monoid libraries, which are present in some languages (such as Haskell [1] [2]). In the proofs below, _≈_ is an equality relation on parsers (resp. printers). Two parsers (resp. printers) are equal iff they evaluate to the same result on every input.

Functor laws Informally, a functor is a type constructor which contains values, and has an associated function `fmap` (also written as $\langle\$\rangle$) which makes it possible to transform all the contained values by applying a function to the entire container. An example of a functor is List, for which `fmap` = map. Formally, a type constructor is said to have a functor instance if there exists a function `fmap` which satisfies two laws: the *identity law*

$$\forall p.\ id\ \langle\$\rangle\ p \equiv p$$

and the *composition law*:

$$\forall p.\ (g \circ f)\ \langle\$\rangle\ p \equiv g\ \langle\$\rangle\ (f\ \langle\$\rangle\ p)$$

where id is the identity function and f and g are arbitrary functions. Figure 4.3 depicts the functor composition law for a functor δ where `fmap` = $\langle\$\rangle$.

I proved that these laws hold for the isofunctor combinator I constructed. The proof statements are given by the type signatures of the proofs. The definitions for id and _∘_

[1] `https://hackage.haskell.org/package/base-4.8.2.0/docs/Data-Functor.html`
[2] `https://hackage.haskell.org/package/base-4.8.2.0/docs/Data-Monoid.html`

are given in §3.2.1. The following proofs are quantified over all partial isomorphisms f and g.

$$
\begin{array}{lll}
\text{parser-id-law} & : & (p : \text{Parser } A) \to \text{id} \ \langle\$\rangle \ p \approx p \\
\text{parser-comp-law} & : & (p : \text{Parser } A) \to (g \circ f) \ \langle\$\rangle \ p \approx g \ \langle\$\rangle \ (f \ \langle\$\rangle \ p) \\
 & & \\
\text{printer-id-law} & : & (p : \text{Printer } A) \to \text{id} \ \langle\$\rangle \ p \approx p \\
\text{printer-comp-law} & : & (p : \text{Printer } A) \to (g \circ f) \ \langle\$\rangle \ p \approx g \ \langle\$\rangle \ (f \ \langle\$\rangle \ p)
\end{array}
$$

Both identity laws followed by definition. I proved the parser composition law by reasoning about nested list comprehensions, for which I required $\gg\!=_l$-map (§3.1.1). The composition law for printers followed by the associativity of the Maybe bind function, which I proved in §3.1.3.

Monoid laws In mathematics, a monoid is a set that is closed under an associative binary operation and has an identity element. For example, the set of natural numbers with the addition function – with zero as the identity element – form a monoid.

This definition translates to type constructors: a type constructor is defined to have a monoid instance if there exists an associative function `mappend` $: A \to A \to A$ and a value `mempty` $: A$ such that `mempty` is the left and right identity for `mappend` (an example is list concatenation and the empty list). Based on this definition, I showed that the choice and empty combinators form a monoid. This resulted in demonstrating the following proofs:

$$
\begin{array}{lllll}
\text{parser-left-id} & : & (p : \text{Parser } A) & \to & (\text{empty} \ \langle|\rangle \ p) \approx p \\
\text{parser-right-id} & : & (p : \text{Parser } A) & \to & (p \ \langle|\rangle \ \text{empty}) \approx p \\
\text{parser-assoc} & : & (x \ y \ z : \text{Parser } A) & \to & x \ \langle|\rangle \ (y \ \langle|\rangle \ z) \approx (x \ \langle|\rangle \ y) \ \langle|\rangle \ z \\
 & & & & \\
\text{printer-left-id} & : & (p : \text{Printer } A) & \to & (\text{empty} \ \langle|\rangle \ p) \approx p \\
\text{printer-right-id} & : & (p : \text{Printer } A) & \to & (p \ \langle|\rangle \ \text{empty}) \approx p \\
\text{printer-assoc} & : & (x \ y \ z : \text{Printer } A) & \to & x \ \langle|\rangle \ (y \ \langle|\rangle \ z) \approx (x \ \langle|\rangle \ y) \ \langle|\rangle \ z
\end{array}
$$

In general, the proofs of these laws for parsers reduced to showing monoidal properties of list concatenation, which is a result proved in the Agda standard library. Similarly, the proofs for the printer functions reduced to showing that nothing is the left and right identity element for mplus and that mplus is associative (proved in §3.1.3). The associativity of the choice combinator is illustrated in Figure 4.3.

4.4 Using syntax descriptions

Using the combinators in the previous chapters, I give a brief overview of the construction of syntax descriptions for the grammar describing primitive recursive functions. The complete set of partial isomorphisms and combinators I discuss here can be found in Appendix B.

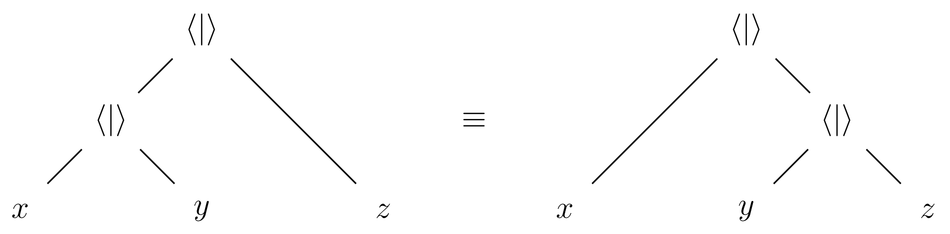


Figure 4.4: Monoid associativity law

The general method of building syntax descriptions involves constructing an ADT representing the grammar. The appropriate partial isomorphisms are created for each constructor of the ADT, which are connected to tokens using the isofunctor combinator. Finally, all the individual combinators are combined using the choice, sequencing, and repetition combinators as required.

To begin, recall the syntax of primitive recursive functions [22][4]:

$$f ::= succ \mid proj_i^n \mid zero^n \mid f \circ [f_1, f_2, \ldots, f_n] \mid \rho^n(f_1, f_2)$$

where i and n range over all the natural numbers. This is an unambiguous grammar.

I represent these non-terminals in text as `succ`, `proj_i^n`, `zero^n`, `f.[f1,f2,...,fn]`, and `rho^n`.

The first step is to convert this grammar into an ADT for which we can create partial isomorphisms. This ADT is given by:

```
data PrimRec : Set where
  Succ : PrimRec
  Proj : Char → Char → PrimRec
  Zero : Char → PrimRec
  Rec : Char → PrimRec → PrimRec → PrimRec
  Comp : PrimRec → List PrimRec → PrimRec
```

Each constructor in the grammar represents a partial isomorphism. Below is the partial isomorphism for the Comp constructor – the rest are defined similarly and can be found in Appendix B. The proofs of invertibility follow trivially by pattern matching, and have been omitted here.

```
comp : Iso (PrimRec × List PrimRec) PrimRec
comp = f , g ⟨ ... ⟩
  where
    f : PrimRec × List PrimRec → Maybe PrimRec
    f (fun , funs) = just (Comp fun funs)

    g : PrimRec → Maybe (PrimRec × List PrimRec)
    g (Comp fun funs) = just (fun , funs)
    g _ = nothing
```

In the following examples, each combinator is parametrised by a variable s : Syntax δ. Once the final syntax description is obtained, its parser (resp. printer) is extracted by passing to it the Syntax record for parsers (resp. printers).

In practice, it is sometimes convenient to parse a string as two sequential parses and then discard one of the parses. This commonly occurs when we are parsing keywords – once they are parsed along with their arguments, the keywords themselves are not required. This is analogous to the case in printers, where specific printers only require the arguments to print, and print keywords on their own. Such functionality can be implemented by the _⟨•_ and _•⟩_ combinators, which return their left and right results respectively. These combinators have been derived from _⟨•⟩_ and are defined in Appendix B. Additionally, I make use of the text combinator, which parses and prints given strings.

I start by defining parens and sqbracks – syntax descriptions which operate on data within a pair of parentheses and square brackets.

```
parens : {A : Set} → δ A → δ A
parens x = (text "(") •⟩ x ⟨• (text ")")

sqbracks : {A : Set} → δ A → δ A
sqbracks x = (text "[") •⟩ x ⟨• (text "]" )
```

Overall, the main description is given by providing descriptions for each non-terminal, which are then combined by the choice combinator. The description for `proj`, for example, looks like the following:

```
proj ⟨$⟩ (text "proj_")  •⟩ token  ⟨• (text "^")  ⟨•⟩ token
```

This code translates to the following procedure: split the input into four parts, discarding the strings `"proj_"` and `"^"`. The two tokens surrounding `"^"` are the required indices (extracted using token), which are passed to the proj partial isomorphism.

Similar descriptions are written for each non-terminal, which combine to result in the final syntax description (Appendix B).

From this syntax description I can extract functions prim-parser and prim-printer which provide parsing and printing functionalities. They can be checked by setting the type signature to an equality and having Agda's type checker mechanically verify it.

```
prim-ex₁ :  prim-printer (Rec '2' Succ (Proj '2' '3')) ≡
            just "rho^2(succ,proj_2^3)"
prim-ex₁ = refl

prim-ex₂ :  prim-parser "proj_1^5" ≡ [ Proj '1' '5' ]
prim-ex₂ = refl
```

With this syntax description I have demonstrated how to build a simple parser and printer without any redundant effort. We can observe the invertibility of the parser-printer pair in the following example, where I successfully print and parse $succ \circ [proj_1^4, zero^3]$.

```
prim-ex₃ : prim-printer (Comp Succ (Proj '1' '4' :: Zero '3' :: [])) ≡
           just "succ.[proj_1^4,zero^3]"
prim-ex₃ = refl

prim-ex₄ : prim-parser "succ.[proj_1^4,zero^3]" ≡
           [ (Comp Succ (Proj '1' '4' :: Zero '3' :: [])) ]
prim-ex₄ = refl
```

Using this method I was also able to produce a syntax description to describe arithmetic expressions. The syntax is given by the grammar below:

$$A ::= n \mid A + A \mid A \times A$$

This grammar corresponds to the following ADT:

```
data Expression : Set where
  Literal : Char → Expression
  AddOp : Expression → Expression → Expression
  MulOp : Expression → Expression → Expression
```

I followed a procedure similar to the one above to extract functions arith-parser and arith-printer for parsing and printing these expressions. I present examples of these functions in use below:

```
arith-ex₁ : arith-parser "1+2*(4+5)" ≡
     AddOp (Literal '1') (MulOp (Literal '2') (AddOp (Literal '4') (Literal '5')))
  :: MulOp (AddOp (Literal '1') (Literal '2')) (AddOp (Literal '4') (Literal '5'))
  :: []
arith-ex₁ = refl
```

arith-ex₁ illustrates how the parser parses the unambiguous expression of `1+2*(4+5)` into two results: `1+(2*(4+5))` and `(1+2)*(4+5)`.

```
arith-ex₂ : arith-printer (AddOp (Literal '1') (Literal '2')) ≡ just "1+2"
arith-ex₂ = refl
```

With this I have shown my implementation to meet the success criterion of being able to describe primitive recursive functions and arithmetic expressions. Due to the proof of invertibility presented in §3.6, we can assert *formally* that all parser-printer pairs extracted from syntax descriptions in this manner form invertible pairs.

4.5 Summary

In this Chapter I have discussed the evaluation of my implementation in terms of testing during the combinator construction phase (unit tests and the verification of functor and monoid laws) and testing after completing construction (proofs of monotonicity and invertibility). The verification of my combinators satisfying the two laws indicates their generality as functions. Additionally, I assessed the practicality of using my implemented combinators by constructing a single description for arithmetic expressions and primitive recursive functions respectively, from which I could derive parsers and printers with no added effort.

Chapter 5

Conclusion

This dissertation has detailed how I designed, implemented, and proved the correctness of a library for combined parsing and pretty printing in the dependently typed language Agda. I have met all the success criteria initially set out and have extended them by proving additional properties about my implementation. Compared to the previous work of Affeldt et al. [1], my proofs are more general as they are not limited to ambiguous grammars, and are entirely axiom-free. Their work, to my knowledge, is the only other research that has been done on the formal verification of invertible syntax descriptions.

5.1 Results

I list below all my success criteria (Appendix C) and describe how I met them.

"[To implement a] complete formalisation of partial isomorphisms."

I have defined and extended partial isomorphisms and functions operating on them to contain proofs of their invertibility, which is a guarantee that could not be shown in the original implementation in Haskell (§3.6).

"Development of an EDSL to be able to describe the necessary languages."

I developed seven combinators (namely, the pure, token, empty, choice, isofunctor, sequence, and repetition combinators) in §3.3, using which I could describe the languages in the following criterion.

"Having the ability to describe [the syntax of] arithmetic expressions [and] primitive recursive functions. I should then, without having to do any extra work, be able to use the associated parser and printer that are generated."

In §4.4 I provided syntax descriptions for arithmetic expressions and primitive recursive functions using the developed combinators and showed their associated parsers and printers to function as required.

"Having a proof of correctness [and] proofs of termination."

I demonstrated the correctness of my implementation by proving in Agda that the parser and printer extracted from any syntax description are inverses of each other in §3.6. In other words, I proved that any string that is the result of printing some abstract syntax can be parsed to obtain the original abstract syntax unmodified. All my implemented combinators terminate on any input – this is attested by Agda's termination checker accepting my definitions.

Extension: I showed my combinators to satisfy the algebraic laws known as functor and monoid laws (defined and proved in §4.3). These laws indicate the universality of my combinators and indicate that they are algebraically well-behaved.

5.2 Further work

In this project I only focused on one type of invertibility property, where a parsed result can be obtained by printing it and then parsing it again. Symbolically, I have shown:

$$\forall a.\ a \in \texttt{parse}\ (\texttt{print}\ a)$$

This property shows that parsing is the left inverse of printing. Another property of invertibility could look like the following:

$$\forall s.\ \texttt{parse}\ (\texttt{print}\ (\texttt{parse}\ s)) \equiv \texttt{parse}\ s$$

where s ranges over all strings. Such a proof could be used to validate code formatters.

5.3 Lessons learned

In the initial stages of proving the correctness of my implementation, I stated the property of invertibility as an equality, such as

$$\mathsf{parse}\ par\ str \equiv [\ ast\ ],$$

rather than as a statement of list membership,

$$ast \in \mathsf{parse}\ par\ str.$$

Although (for individual combinators) statements of equality were tighter than their membership-form counterparts, they could not be combined into one general statement for arbitrary combinators. Additionally, their proofs were less elegant and more complicated. I could only learn this by trying first, but having the insight to look into proofs of list membership (rather than proofs of equality) earlier would have avoided a great deal of refactoring and made proving correctness easier.

Overall, this project has given me a glimpse into the exciting world of formal verification, and I have gained invaluable experience in the art of theorem proving.

Bibliography

[1] Reynald Affeldt, David Nowak, and Yutaka Oiwa. Formal network packet processing with minimal fuss: Invertible syntax descriptions at work. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, pages 27–36, New York, NY, USA, 2012. ACM.

[2] Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: Arrows for invertible programming. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, pages 86–97, New York, NY, USA, 2005. ACM.

[3] Andrea Asperti, Claudio Sacerdoti Coen, and al. Matita. `http://matita.cs.unibo.it/`.

[4] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic.* Cambridge University Press, 2007.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[6] Nils Anders Danielsson. Total Parser Combinators. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 285–296, 2010.

[7] Nils Anders Danielsson. Correct-by-Construction Pretty-Printing. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, DTP '13, pages 1–12, New York, NY, USA, 2013. ACM.

[8] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

[9] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing Back Monad Comprehensions. *SIGPLAN Not.*, 46(12):13–22, September 2011.

[10] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[11] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96, London, UK, UK, 1995. Springer-Verlag.

[12] Graham Hutton. Parsing using combinators. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, pages 353–370, London, UK, UK, 1990. Springer-Verlag.

[13] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[14] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, pages 273–287, London, UK, UK, 1999. Springer-Verlag.

[15] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Sci. Comput. Program.*, 43(1):35–75, April 2002.

[16] Mark Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Journal of Functional Programming*, pages 52–61. ACM Press, 1995.

[17] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[18] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[19] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[20] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[21] Derek C. Oppen. Pretty Printing. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980.

[22] Andrew Pitts. *Computation Theory, 1B Computer Science Tripos, Cambridge University.*

[23] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. *SIGPLAN Not.*, 45(11):1–12, September 2010.

[24] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 28–32, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Philip Wadler. How to replace failure by a list of successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[26] Philip Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

Appendix A

Partial isomorphisms

In this Appendix I list all the partial isomorphisms I used in my project. I omit the proofs of invertibility, which can be found in the source code.

Identity:

```
id : {A : Set} → Iso A A
id = just , just ⟨ … ⟩
```

Composition:

```
_∘_ : {A B C : Set} → Iso B C → Iso A B → Iso A C
_∘_ (g , g⁻¹ ⟨ gp₁ , gp₂ ⟩) (f , f⁻¹ ⟨ fp₁ , fp₂ ⟩) = (f >=>ₘ g) , (g⁻¹ >=>ₘ f⁻¹) ⟨ … ⟩
```

Element:

```
element : {A : Set} → Decidable (_≡_ {A = A}) → A → Iso ⊤ A
element {A} dec x = f , g ⟨ … ⟩
  where
    f : ⊤ → Maybe A
    f tt = just x

    g : A → Maybe ⊤
    g b with dec b x
    … | yes prf = just tt
    … | no ¬prf = nothing
```

Commute:

```
commute : {A B : Set} → Iso (A × B) (B × A)
commute = comm , comm ⟨ … ⟩
  where
    comm : {A B : Set} → (A × B) → Maybe (B × A)
    comm (a , b) = just (b , a)
```

Unit:

```
unit : {A : Set} → Iso A (A × ⊤)
unit {A} = f , g ⟨ ... ⟩
  where
    f : A → Maybe (A × ⊤)
    f a = just (a , tt)

    g : (A × ⊤) → Maybe A
    g (a , tt) = just a
```

Nil:

```
nil : {A : Set} → Iso ⊤ (List A)
nil {A} = f , g ⟨ ... ⟩
  where
    f : ⊤ → Maybe (List A)
    f tt = just []
```

Cons:

```
cons : {A : Set} → Iso (A × List A) (List A)
cons {A} = f , g ⟨ ... ⟩
  where
    f : A × List A → Maybe (List A)
    f (proj₁ , proj₂) = just (proj₁ :: proj₂)

    g : List A → Maybe (A × List A)
    g [] = nothing
    g (x :: xs) = just (x , xs)
```

Associate:

```
associate : {A B C : Set} → Iso (A × (B × C)) ((A × B) × C)
associate = f , g ⟨ ... ⟩
  where
    f : A × (B × C) → Maybe ((A × B) × C)
    f (a , (b , c)) = just ((a , b) , c)

    g : (A × B) × C → Maybe (A × (B × C))
    g ((a , b) , c) = just (a , (b , c))
```

Appendix B

Syntax descriptions

In this Appendix I list all the syntax description combinators I used, and present the syntax describing the primitive recursive functions and arithmetic expressions.

Both the syntax descriptions for primitive recursive functions and arithmetic expressions are bounded by a fixed recursion depth, so that Agda can infer termination of these functions.

B.1 Syntax description combinators

```
_•⟩_ : {A : Set} → δ ⊤ → δ A → δ A
_•⟩_ p q = (inverse unit ∘ commute) ⟨$⟩ (p ⟨•⟩ q)

_⟨•_ : {A : Set} → δ A → δ ⊤ → δ A
_⟨•_ p q = (inverse unit) ⟨$⟩ (p ⟨•⟩ q)

between : {A : Set} → δ ⊤ → δ ⊤ → δ A → δ A
between {A} p q r = (p •⟩ r) ⟨• q

text : List Char → δ ⊤
text [] = pure dec-unit tt
text (x :: xs) = inverse (element dec-unit² (tt , tt)) ⟨$⟩ ((inverse (element dec-char x) ⟨$⟩ token)
                                                      ⟨•⟩ (text xs))
```

dec-unit² and dec-char are functions for checking equality between pairs of ⊤ type values and characters, respectively.

B.2 Arithmetic expressions

```
parens : {A : Set} → δ A → δ A
parens = between (text "(") (text ")")

expression : ℕ → δ Expression
expression zero      =  literal  ⟨$⟩ token
expression (suc n)   =  literal  ⟨$⟩ token
                     ⟨|⟩ addOp   ⟨$⟩ (expression n) ⟨• (text "+") ⟨•⟩ (expression n)
                     ⟨|⟩ mulOp   ⟨$⟩ (expression n) ⟨• (text "*") ⟨•⟩ (expression n)
                     ⟨|⟩ parens (expression n)
```

B.3 Primitive recursive functions

```
parens : {A : Set} → δ A → δ A
parens x = (text "(") •⟩ x ⟨• (text ")")

sqbracks : {A : Set} → δ A → δ A
sqbracks x = (text "[") •⟩ x ⟨• (text "]")

exp : ℕ → δ PrimRec
exp 0        = succ   ⟨$⟩ text "succ"
             ⟨|⟩ proj  ⟨$⟩ text "proj_" •⟩ (token ⟨• text "^") ⟨•⟩ token
             ⟨|⟩ zero  ⟨$⟩ text "zero^" •⟩ token
exp (suc n)  = succ   ⟨$⟩ text "succ"
             ⟨|⟩ proj  ⟨$⟩ text "proj_" •⟩ (token ⟨• text "^") ⟨•⟩ token
             ⟨|⟩ zero  ⟨$⟩ text "zero^" •⟩ token
             ⟨|⟩ comp  ⟨$⟩ (exp n ⟨• text ".")
                              ⟨•⟩ sqbracks (cons  ⟨$⟩ exp n
                                                  ⟨•⟩ ⟨⋆⟩ (text ",") •⟩ (exp n))
             ⟨|⟩ rec   ⟨$⟩ text "rho^"  •⟩ token
                                  ⟨•⟩ (parens ((exp n  ⟨• text ",")
                                                       ⟨•⟩ exp n))
```

Appendix C

Project Proposal

Computer Science Project Proposal

On the Formal Unification of Parsers and Pretty Printers

Swaraj Dash, Trinity Hall

Originators: Dr Dominic Mulligan, Dr Jeremy Yallop

October 23, 2015

Project Supervisors: Dr Dominic Mulligan, Dr Jeremy Yallop

Director of Studies: Prof. Simon Moore

Project Overseers: Prof. Marcelo Fiore, Prof. Ian Leslie

Introduction

Parsing and pretty printing are tasks that people in many areas in (and relating to) computer science come across frequently. What is common to both these tasks is the grammars that they're based on - parsers and pretty printers are almost always programmed together, but are rarely constructed from a common description of the language. This way of deriving them from a single description not only saves a lot of time, but also avoids the possibility of having inconsistent implementations (which could potentially happen if either component were to be individually updated, for example). In this project I will look at unifying parser combinators and pretty printers in a dependently typed language.

Parser combinators

In functional programming languages, parsing can be accomplished in a functional style by making use of parser combinators – these are higher order functions that take as input zero or more parser combinators and return a parser combinator as output.

For example, if we have a grammar `A ::= '(' A A ')' |` ε, the associated parser combinator one would write in Haskell – using a suitable library – is:

```
A = term '(' <*> A <*> A <*> term ')' <|> empty
-- the <*> combinator denotes sequence
-- the <|> combinator denotes choice
```

Let us now consider the slightly more complicated example of constructing a parser for a `List`. `Lists`, as usual, are defined like the following in Haskell:

```
data List a = Nil | Cons a (List a)
```

Then, the parser we could make would look like:

```
newtype Parser a = Parser (String -> [(a, String)])

parseMany :: Parser a -> Parser (List a)
parseMany p
  =  const Nil <$> text ""
 <|> Cons      <$> p
               <*> parseMany p
```

Pretty printers

Pretty printers are functions that take as input user-specified structured datatypes (ASTs, for example) and "flatten" them (by returning strings, for example). Going back to our `List` parsing example, we can construct the relevant printer as well:

```
type Printer a = a -> Doc

printMany :: Printer a -> Printer (List a)
printMany p list
  = case list of
    Nil       -> text ""
    Cons x xs -> p x
              <> printMany p xs
```

Note: Doc here is any defined structured type.

Unification of parser combinators with pretty printers

Given the very similar nature of parsing (using parser combinators) and pretty printing – as can be seen from the similar implementations of `parserMany` and `printMany` (note the use of pattern matching and recursion styles in both functions) – one can easily come to the conclusion that they should really both be written together at the same time since their definitions rely on nearly the same information.

And so we have the idea of unifying parsers and printers into one interface. This idea has been explored with Haskell in the paper "Invertible syntax descriptions" by Rendel and Ostermann, where they do exactly this. Unification in the paper is accomplished on top of another concept the authors developed in Haskell called "partial isomorphisms", which helped them capture the notion of being able to use both, parsers and printers, from the same description of the grammar.

What I would like to do

I would like to create a similar unification of parsers and printers, but in a dependently typed language. Dependent types have two desirable properties for this project: firstly, I will be able to provide automatically constructed proofs of correctness of my implementation. Additionally, since functions in total dependently typed languages that are based on a 'proofs-as-programs' paradigm are required to be total, I will also prove termination of the implementation.

Evaluation of my project will be based on the expressiveness of the language that I'll be able to describe via the embedded domain-specific language (EDSL) I develop. The minimum requirement I've set is to be able to parse simple arithmetic expressions with nested brackets. As another target, I also aim to be able to parse a simple language describing primitive recursive functions. I will also check other metrics that could be used to quantitatively describe my implementation, such as time complexity and performance (although the reader should note that optimising based on these metrics isn't the aim of the project).

As far as I'm aware this hasn't been implemented so far, and so it would be a nice achievement to be able to parse/print a non-regular subset of context free grammars – all while knowing the correctness of the implementation.

Starting point

Over the course of this project I will be building on the following resources:

- ***"Invertible syntax descriptions: Unifying parsing and pretty printing"*** **– Tillmann Rendel and Klaus Ostermann**
  This is a paper in which the authors developed an EDSL for Haskell in order to be able to define grammars in such a way that both, a parser and a printer would get automatically generated. However, neither do they provide any proofs of correctness nor any proofs of termination, both of which are central to my project.
- ***"Total parser combinators"*** **– Nils Anders Danielsson**
  This paper explores parser combinators in the total dependently typed language Agda. Since all the functions in Agda need to be total, this paper contains useful tools and techniques that will aid me in proving termination of my implementation. (Most total dependently typed languages have their own termination checkers but they only work for structural recursion, and not many other exotic forms of recursion.)
- **Summer internship at the Computer Laboratory**
  I interned in the Computer Laboratory from early July to early September 2015, under the guidance of Dr Dominic Mulligan. This internship involved formalising the constructive reals and using them to prove mathematical results in the field of (constructive) real analysis in Agda, which I learned over the course of the internship. This has been very helpful in familiarising myself with dependent types and learning about how I can use them to prove propositions in (intuitionistic) logic. Experience from my summer internship will help me formalise partial isomorphisms due to the similarity of the nature of work involved.

I will be combining the approaches in the two papers mentioned above to achieve my goal. The paper by Danielsson, combined with my experience from last summer has made me decide my language of implementation to be Agda. This seems to be the best option as it avoids having to waste unnecessary time reinventing the wheel, and will let me hit the ground running.

Resources required

For this project I will be using my own computer which runs Arch Linux – this machine will suffice as I have no requirement for high performance. Revision control will be done using `git` and will be hosted online on Github. I'll also have nightly builds, which will be backed up to physical storage media, and pushed to the MCS as another safety measure.

Work to be done

I have broken the project down into the following major components, which I will glue together to reach my final implementation.

- **Developing partial isomorphisms:** The concept of partial isomorphisms is explored in "Invertible syntax descriptions". The authors there use them to capture the notion of bi-directional computing, so that one can combine functions of the type $(\alpha \to \beta)$ with those of the type $(\beta \to \alpha)$ – this way a single combinator used to describe the grammar can be used to construct both parsers as well as printers (that relate to the function of the relevant combinator, of course).

  I will need to implement these in Agda so I can build the unification of parsers and combinators on top of them.

- **Developing a suitable EDSL for Agda:** In "Invertible syntax descriptions", the authors created an embedded DSL for Haskell to allow describing the grammar. I will take a similar approach for users to be able to describe their languages in Agda, but its implementation might differ as this is very language-dependent.

- **Developing combinators for the EDSL:** To be able to describe any language/-grammar, I will need to implement a set of combinators I could use. These may include combinators that would allow me to express choice, concatenation, recursion, etc.

Success criteria for the main result

The following are my criteria for success:

- A complete formalisation of partial isomorphisms.
- Development of an EDSL to be able to describe the necessary languages.
- Having the ability to describe
  - arithmetic expressions that include nested brackets, and
  - a simple language capable of describing primitive recursive functions.

  I should then, without having to do any extra work, be able to use the associated parser and printer that are generated.
- Having a proof of correctness such that it shows that the function `parse ∘ print` is extensionally equivalent to the identity function, and proofs of termination of my implementation.

Possible extensions

I propose the additional feature of being able to assign operator precedences in the description language (the EDSL). This can be very useful in defining other operators (that would potentially take more than two values as input) with ease.

Timetable

- **Michaelmas weeks 2–4**
  Familiarise myself with the two papers mentioned above – "Invertible syntax descriptions" and "Total parser combinators". Start experimenting with different approaches of implementing partial isomorphisms in Agda.
- **Michaelmas weeks 5–6**
  Try to experiment with simple combinators if time permits. Look into the different ways an EDSL could be implemented in Agda.
- **Michaelmas weeks 7–8**
  Some buffer time to catch up with project work in case I fall behind.
  Milestone: Have a good platform to start being able to implement various kinds of combinators needed to reach my goal.
- **Michaelmas vacation**
  Look into how proving termination will be accommodated.
- **Lent weeks 0–2**
  Write progress report and create presentation.
  Milestone: Present progress.
- **Lent weeks 3–5**
  Most design choices (such as style of termination proofs, set of combinators, form of EDSL) to be decided by now.
  Milestone: Finalised implementations of the various combinators developed.
- **Lent weeks 6–8**
  Research into work that has to be done to implement the proposed extension. Start drafting up the introduction and implementation chapters of the dissertation.
- **Easter vacation:**
  Evaluate/implement extensions if time permits. Look into optimising code if possible.
- **Easter term 0–2:**
  Near completion of the dissertation with time to incorporate suggestions from reviewers.
- **Easter term 3:**
  Clean up code and dissertation. Buffer time to proof read my dissertation and address any other problems.
  Milestone: Submission of dissertation.

References

[1] Ostermann K. and Rendel T., Invertible Syntax Decriptions: Unifying Parsing and Pretty Printing. In *Haskell Symposium, 2010.*

[2] Danielsson N. A., Total Parser Combinators. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP 2010).*