L21 Interactive Formal Verification
Michaelmas Term, Academic Year 2017–2018
Assessed Exercise 1: Combinator Parsing

Dr. Dominic P. Mulligan

October 5, 2017

Contents

theory *Assessed-Exercise-One-Questions*
imports *Main* ~~/*src*/*HOL*/*Library*/*Monad-Syntax*
begin

1 Introduction

This is the first assessed exercise for the L21 "Interactive Formal Verification" MPhil course in academic year 2017–2018. In this exercise you will write some parser combinators and thereafter prove some simple properties about these combinators. The exercise will test your ability to write recursive and non-recursive functions in Isabelle/HOL, work with sets and quantifiers, and prove properties about functions using induction and simple automation. Note that for this first exercise, structured proofs written in Isar are not required, though for extra credit more ambitious students could try to convert some of their apply-style proofs into Isar. Concretely, the marking scheme for this exercise is as follows (out of a total of 100):

- **50 marks** for correct definitions and lemmas, in accordance with the distribution of marks outlined in the body of this document,

- **30 marks** for beautiful proofs and definitions, some use of Isar, extra proofs of properties about the combinators presented in this exercise, or more useful combinators and properties about them, or the use of features of Isabelle not lectured in class. Any reasonable evidence that you have gone "above and beyond" in your use of Isabelle will be considered here, and obviously the more ambitious you are, the more marks you accrue,
- **20 marks** for a nice writeup detailing all decisions made in your choice of lemmas, proof strategies, and so on, and an explanation as to what extensions you have decided to implement, or novel features of Isabelle that you have used, to merit some of the 30 marks mentioned above.

Your submissions should be submitted electronically on the L21 Moodle site before 4pm Cambridge local time on submission day. See the course website for submission dates for the two assessed exercises. Submissions should consist of an edited version of the theory file that this document is based on (also available from the course website) that includes your solutions to the exercises contained within, along with a PDF document containing your writeup. Your writeup need not be especially long—2 sides of A4 will suffice—but it should be detailed, describing auxilliary lemmas and definitions that you introduced, if any, design decisions, and so on. Late submissions will of course be penalised, and as always students should not confer with each other when answering the sheet.

Before beginning: for those who have not encountered parser combinators before, it may be a good idea for you to skim-read Hutton's article on the subject before proceeding with the exercise to gain some background.[1] However, the exercises have been written in such a way that consulting any such background material on combinator parsing is not strictly necessary, with enough background information embedded within this document to help you through.

[1] See http://eprints.nottingham.ac.uk/221/1/parsing.pdf

2 Parsers, and some useful combinators

Parsing is a common task when developing computer systems. But what actually are parsers? Abstractly, parsers take a string of characters and either fail, or return a result derived from a prefix of that string paired with a continuation string to parse further. Note here that "characters" need not be concrete ASCII or Unicode characters, but may be (and often are) lexical tokens of the language being parsed, derived from a separate lexing step. Running with this idea, we can model parsers as functions that take as input an arbitrary list of generalised characters and return a set of parsed values, paired with continuation lists of characters yet to parse. Using this scheme, not only are we parametric in the underlying character and return value types, but failure and parse ambiguity can be handled uniformly: a parser returning an empty set of results signals failure, and ambiguity can be handled by simply returning the set of all possible parse results.

Concretely, in Isabelle/HOL, we can capture this view of parsing as a datatype, like follows:

datatype $('a, 'b)$ *parser*
  $=$ *Parser* $'a$ *list* $\Rightarrow$ $('a$ *list* $\times$ $'b)$ *set*

Here, the type variable `'a` stands for the type of elements in the list being parsed—in most typical applications, usually lexical tokens, ASCII or Unicode characters, as mentioned—and the type variable `'b` stands for the return type of the parser. For instance, a combinator that parses lists of characters and returns an integer would have type `(char, int) parser`, whereas a parser that parser lists of tokens to produce a boolean value would have type `(token, bool) parser` for some token type, `token`.

A parser can be "run", or "executed" on a list of characters by simply unwrapping it, and applying the underlying parsing function to produce a set of results:

definition *run* $::$ $('a, 'b)$ *parser* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $('a$ *list* $\times$ $'b)$ *set* **where**
  run p xs $\equiv$ *case p of Parser f* $\Rightarrow$ *f xs*

There exist two very simple but very important parsers: the parser that always succeeds, and the parser that always fails. The parser that always fails is the simpler of the two, so we consider that parser first. No matter what its input, this parser always returns an empty set of results. This can be modelled easily in Isabelle/HOL, as follows:

definition *fail* $::$ $('a, 'b)$ *parser* **where**
  fail $\equiv$ *Parser* $(\lambda xs.\ \{\})$

The parser that always succeeds is a little more complex. For every list of characters to parse, this parser will simply return a singleton set containing that list as its continuation. The parser cannot invent a value to return—as we are parametric in the type of values, so have no idea how to obtain a defined element of type `'b`—so we must pass it one explicitly, instead:

definition *succeed* $::$ $'b \Rightarrow ('a, 'b)$ *parser* **where**
  succeed x $\equiv$ *Parser* $(\lambda xs.\ \{(xs, x)\})$

Lastly, we consider how to make choices when parsing, which is useful: for instance, in a programming language a number may either be deemed to be a finite list of digits *or* a finite list of digits preceded by a negation sign. This sort of choice can easily be captured as a higher-order function (or combinator), which we call `choice`. The combinator takes

two parsers as arguments and produces another parser as a result, and produces its resulting parser by simply running both argument parsers on the parser's input and collecting together all results using the set union operator:

```
definition choice :: ('a, 'b) parser ⇒ ('a, 'b) parser ⇒ ('a, 'b) parser (infixr ⊕ 65) where
  choice p1 p2 ≡ Parser (λxs. run p1 xs ∪ run p2 xs)
```

2.1 Slightly more complex combinators

Aside from failure, success, and choice, there are some other commonly reoccurring patterns that you may notice are common to many parsing tasks, and can therefore be factored out into a series of reusable combinators.

In particular, when parsing we often want to indicate that the string being parsed must have an exact form. This is especially true when parsing keywords or other syntactic elements in a programming language, for instance. The following combinator, `satisfy`, takes a predicate on characters and produces a parser. This resulting parser fails if its input is empty, or else if the supplied predicate does not hold on the first character of its input list. Otherwise it succeeds, returning a singleton set containing the head of the input list as its value, and the tail of the input list as its continuation:

```
definition satisfy :: ('a ⇒ bool) ⇒ ('a, 'a) parser where
  satisfy p ≡
    Parser (λxs.
      case xs of
        [] ⇒ {}
      | x#xs ⇒ if p x then {(xs, x)} else {})
```

Admittedly, this combinator is not very interesting on its own. Rather, it is a combinator that can be used to derive more interesting combinators. For instance, parsing an exact character is now straightforward using the `satisfy` combinator:

```
definition exact :: 'a ⇒ ('a, 'a) parser where
  exact x ≡ satisfy (λy. y = x)
```

Below, we will also see another combinator, `exacts`, which allows us to parse an entire string of characters exactly, rather than a single one, which also uses `exact` as a subprocedure.

Previously, we discussed a parser combinator that captured a notion of "choice". Another important concept when parsing is sequencing: often we wish to parse a keyword immediately followed by an identifier, or something similar. This is a simple notion of sequencing, but sometimes the pattern of sequencing can be more complex. For example, we may wish to parse a number and then parse different things depending on whatever number we obtained from that initial parse. This is especially true when parsing binary file formats, for instance, which often tell you in advance in a header entry at the start of the file how many table elements of some sort are to follow. This more complex notion of sequencing can be captured by a parser combinator, called `bind`. This combinator takes a parser `p` as an input and a function `f`, and returns a new parser as a result. Concretely, `f` accepts the return value of `p` and produces a new parser as a result, capturing this idea that sequencing can branch depending on the intermediate results of previous parses.

Exercise (6 marks): complete the definition of `bind` by replacing the `consts` declaration below with a complete definition. Use the type signature of the function to guide you in its implementation. Make sure you properly thread all of the continuation lists of intermediate parses through the definition. Some of the properties that you will later prove below may fail to hold should you get the implementation wrong initially, so you may need to come back and re-examine your implementation later. You can also use the examples in Section 4 to test whether your definition is reasonable.

consts *bind* :: $('a, 'b)$ *parser* $\Rightarrow$ $('b \Rightarrow ('a, 'c)$ *parser*$) \Rightarrow ('a, 'c)$ *parser*

Now that we have a notion of sequencing together two parsers to produce a new parser, we can define a combinator that iteratively applies a parser a fixed number of times in sequence—call it `biter`. This combinator accepts two arguments—call them `m` and `p`—where `m` is a natural number indicating the number of times to apply parser `p` in sequence. Ideally, we would also like `biter` to return a list of the elements that it has parsed, too.

Concretely, what should the `biter` combinator do? If we are iterating zero times, then the resulting parser should succeed, albeit returning the empty list as its value. Otherwise, if iterating `Succ m` times, then we should first parse using `p` and then immediately after parse using the `m`-fold iteration of `p`, combining the two intermediate results to produce a list as the final value.

Exercise (3 marks): complete the definition of the `biter` combinator by replacing the `consts` declaration below with a complete definition. Again, use the type signature of the function to guide you in its definition. Some of the properties that you will later prove below may fail to hold should you get the implementation wrong initially, so you may need to come back and re-examine your implementation later.

consts *biter* :: *nat* $\Rightarrow$ $('a, 'b)$ *parser* $\Rightarrow$ $('a, 'b$ *list*$)$ *parser*

A slightly different, but related notion of iteration, is often useful. Suppose we want to parse a keyword in a programming language. How can we do that, given the combinators that we already have? Using the previously discussed `exact` combinator, we can parse a single character of the keyword at a time, in sequence, until there are no more characters of the keyword left to parse. The combinator `exacts` accepts a list of characters `cs` to parse one by one, and as its return value it produces the list of characters that it has parsed.

Exercise (3 marks): complete the definition of the `exacts` combinator by replacing the `consts` declaration below with a complete definition. Again, use the type signature of the function to guide you in its definition. Some of the properties that you will later prove below may fail to hold should you get the implementation wrong initially, so you may need to come back and re-examine your implementation later.

consts *exacts* :: $'a$ *list* $\Rightarrow$ $('a, 'a$ *list*$)$ *parser*

Lastly, suppose we wish to parse numbers for a programming language interpreter we are writing. One way to do this would be to parse a list of digits, returning a string, and then take this string and apply a function that maps strings of digits into some numeric type, returning that as our result. We can capture this pattern using a notion of "mapping", similar to the map function on lists from standard functional programming:

definition *map* :: $('a, 'b)$ *parser* $\Rightarrow$ $('b \Rightarrow 'c) \Rightarrow ('a, 'c)$ *parser* **where**
map p f $\equiv$ *Parser* $(\lambda xs.\ (\lambda x.\ (fst\ x, f\ (snd\ x)))\ `\ (run\ p\ xs))$

3 Some properties of this library

We now prove some important properties of our parser combinators to ensure that they behave correctly. Many of these properties can be expressed as equivalences between parsers. But first, we must define what it means for two parsers to be equivalent, or equal.

3.1 Equivalence of parsers

What is a suitable notion of equivalence for parsers? Recall that parsers are essentially functions in disguise, and as you know functions are equal when they agree on all inputs. Two parsers should therefore be considered equivalent when executing them both on the same arbitrary input leads to the same result.

Exercise (2 marks): define a binary relation `peq` on parsers that captures when two parsers are equivalent by replacing the `consts` declaration below with a complete definition. Some of the properties that you will later prove below may fail to hold should you get the implementation wrong initially, so you may need to come back and re-examine your implementation later.

consts *peq* :: $('a, 'b)$ *parser* $\Rightarrow$ $('a, 'b)$ *parser* $\Rightarrow$ *bool*

We can check that the putative equivalence relation above behaves somewhat correctly by checking that it is indeed an equivalence relation, i.e. that it is reflexive, symmetric, and transitive.

Exercise (3 marks, 1 mark each): prove that `peq` is *reflexive*, *symmetric*, and *transitive* by stating and proving three relevant lemmas.

3.2 Parsers have a commutative monoidal structure under choice

Now we have a notion of parser equivalence, we can state and prove some interesting properties of our combinators. First, we examine how `choice` and the always-failing parser, `fail`, interact. It should be intuitively obvious that `fail` acts as a *neutral* (or identity) element for the `choice` combinator.

Exercise (2 marks, 1 mark each): prove that `fail` is a left- and right-neutral element for `choice` by proving the following two lemmas. That is, replace the `oops` commands below with complete proofs.

lemma *choice-ident-fail1*:
shows *peq* (*fail* $\oplus$ *p1*) *p1*
oops

lemma *choice-ident-fail2*:
shows *peq* (*p1* $\oplus$ *fail*) *p1*
oops

Moreover, it should be obvious that choice is commutative and associative—it does not and should not matter in which order you choose to apply parsers under the choice combinator, as the choice combinator simply collects together all possible parses in one big set.

Exercise (2 marks, 1 mark each): prove that `choice` is commutative and associative by proving the following two lemmas. That is, replace the `oops` commands below with complete proofs.

lemma *choice-comm*:
shows *peq* (*p1* ⊕ *p2*) (*p2* ⊕ *p1*)
oops

lemma *choice-assoc*:
shows *peq* (*p1* ⊕ (*p2* ⊕ *p3*)) ((*p1* ⊕ *p2*) ⊕ *p3*)
oops

3.3 Map is functorial, and has an alternative definition

The familiar "map" function on lists enjoys a number of properties. A few of these properties are particularly important, namely:

- mapping the identity function *id* over a list results in the same list,
- mapping the composition of two functions, $f \circ g$, over a list is the same as first mapping g over that list followed by mapping f over the resulting list.

These two properties taken together are sometimes known as "functoriality", and a great number of similar "map" functions on different types possess them. Indeed, the map function that we have defined on parsers also possesses these functoriality properties.

Exercise (1 mark): show that mapping the identity function over a parser `p` is equivalent to `p` by stating and proving a relevant lemma.

Exercise (3 marks): show that mapping the composition of two functions over a parser is equivalent to first mapping one function, and then the other, over that same parser by stating and proving a relevant lemma.

Earlier we gave a direct definition of `map` in terms of the set image function. It was rather "low level", requiring us to deal directly with the underlying representation of parsers. In fact, having defined the `bind` and `succeed` combinators already, it was already possible to give an alternative definition of `map` in terms of those operations that did not require us to deal directly with the underlying representation of parsers.

Exercise (2 marks): show that `map` can be given an alternative definition in terms of the `bind` and `succeed` combinators by proving the following lemma. That is, replace the `oops` command below with a complete proof.

lemma *map-alternative-def*:
shows *peq* (*map p f*) (*bind p* (λ*x*. *succeed* (*f x*)))
oops

3.4 Bind and succeed satisfy the monad laws (and other properties)

Next, we examine how the `bind` and `succeed` combinators interact with each other. First, we show that `succeed` acts as a right-neutral element for `bind`.

Exercise (1 mark): show that `succeed` is a right-neutral element for `bind` by stating and proving a relevant lemma.

Moreover, succeed also acts as a kind of left-neutral element for bind, albeit in a slightly messier way than for the right-neutral case. As a result, I will provide the lemma statement.

Exercise (1 mark): show that `succeed` is a left-neutral element for `bind` by proving the following lemma. That is, replace the `oops` command below with a complete proof.

lemma *bind-succeed-collapse*:
  shows *peq* (*bind* (*succeed x*) *f*) (*f x*)
oops

The `bind` combinator also exhibits a kind of "associativity" property which allows one to rearrange a series of nested applications of `bind` from being left-associative to right-associative. Again, this is a rather messy property, so I will provide the lemma statement to prove.

Exercise (1 mark): show that `bind` has an associativity property by proving the following lemma. That is, replace the `oops` command below with a complete proof.

lemma *bind-assoc*:
  shows *peq* (*bind* (*bind p f*) *q*) (*bind p* (λ*x*. *bind* (*f x*) *q*))
  oops

(The previous three properties are sometimes referred to as the "monad laws", and hold for many useful parameterised types that appear naturally in functional programming with suitable definitions for `bind` and `succeed`. For example: lists, sets, the option type, continuations, and so on, all possess the three properties above for suitably chosen implementations of `bind` and `succeed` specific to each type.)

The `bind` combinator also satisfies some other properties not captured by the monad laws. In particular, it interacts well with the always-failing parser, `fail`, and in fact `fail` is a left-annihilating element for `bind`. Intuitively, that is: if you first fail to parse anything, and then try to parse something else, you will always fail.

Exercise (1 mark): show that `fail` is a left-annihilating element for `bind` by proving the following lemma. That is, replace the `oops` command below with a complete proof.

lemma *bind-fail-annihil*:
  shows *peq* (*bind fail f*) *fail*
  oops

In addition, the `bind` and `choice` combintors also interact well, and one may factor `bind` through the `choice` combinator freely.

Exercise (1 mark): show `bind` can be factored through the `choice` combinator by proving the following lemma. That is, replace the `oops` command below with a complete proof.

lemma *bind-choice-split*:
  shows *peq* (*bind* (*p* ⊕ *q*) *f*) (*bind p f* ⊕ *bind q f*)
  oops

Lastly, `bind` has a rather strong interpolation property that not all parameterised types that satisfy the monad laws possess (though others, such as the familiar option type do possess a very similar property).

Exercise (8 marks): show that `bind` exhibits an interpolation property by proving the following lemma. That is, replace the `oops` command below with a complete proof.

lemma *bind-interpolate*:
  assumes *run* (*bind p f*) *xs* = *ps*
  shows ∃ *qs*. *run p xs* = *qs* ∧ ((⋃(*q*, *r*)∈*qs*. *run* (*f r*) *q*) = *ps*)
oops

3.5 Properties of iteration

In a final pair of exercises, we address properties of the iteration combinators that we defined previously.

First, we show that iterating a parser $m + n$ times is the same as iterating a parser m times followed by iterating a parser n times, before succeeding with the append of the two results.

Exercise (5 marks): show that this property holds of the `biter` combinator by proving the following lemma. That is, replace the `oops` command below with a complete proof.

lemma *biter-plus-bind*:
  shows *peq* (*biter* (*m*+*n*) *p*) (*bind* (*biter m p*) (λ*xs*. *bind* (*biter n p*) (λ*ys*. *succeed* (*xs*@*ys*))))
oops

A very similar property holds of the combinator `exacts`. If we try to exactly parse a keyword `xs` appended to another keyword `ys`, then this should be the same as exactly parsing `xs` followed by exactly parsing `ys`, succeeding with the append of the two intermediate results as the return value.

Exercise (5 marks): show that this analogous described property holds for the `exacts` combinator by stating and proving a relevant lemma.

4 Example: parsing a fragment of English

This section is non-assessed, and is included to provide a motivating example, so that you can gain some intuition for what the definitions should do, and as a testing ground for you to use to ensure that your definitions are correct. In particular, we will use our small combinator library to parse a tiny (but ambiguous) fragment of English.

The following command is used to set up the "monadic do"-syntax, which allows us to write repeated binds as a "do block" in the form `do { ...}`. This command can be safely ignored, as it only makes the rest of the material below easier to read.

adhoc-overloading *Monad-Syntax.bind bind*

First, we define a small utility combinator that exactly parses an arbitrary character from a supplied set of characters. This will be used below.

definition *one-of* :: *'a set* ⇒ (*'a*, *'a*) *parser* **where**
  one-of ss ≡ *satisfy* (λ*x*. *x* ∈ *ss*)

Next, we give ourselves a supply of common English nouns, verbs, transitive verbs, and determinants. These are the basic building blocks of the sentences that we will try to parse.

Note here that the English word "loves" is classed as both a transitive verb and a plain verb, indicating a degree of ambiguity when parsing is to be expected.

definition *nouns* :: *string set* **where**
  nouns ≡ {*''man''*, *''woman''*, *''child''*}

definition *verbs* :: *string set* **where**
  verbs ≡ {*''runs''*, *''walks''*, *''loves''*}

definition *transitive-verbs* :: *string set* **where**
  transitive-verbs ≡ {*''likes''*, *''loves''*}

definition *determinants* :: *string set* **where**
  determinants ≡ {*''a''*, *''the''*, *''some''*, *''every''*}

A noun phrase in English is a determinant followed by a noun. Note that the sequencing between the two is expressed using `bind`, albeit hidden beneath the `do { ... }` syntax.

definition *noun-phrase* :: (*string*, *string list*) *parser* **where**
  noun-phrase ≡
    do
    { *d* ← *one-of determinants*
    ; *n* ← *one-of nouns*
    ; *succeed* [*d*, *n*]
    }

An alternative, much less readable rendering of `noun_phrase` above, which does not use the `do { ... }` syntax is:

definition *noun-phrase′* :: (*string*, *string list*) *parser* **where**
  noun-phrase′ ≡ *bind* (*one-of determinants*) (λ*d*. *bind* (*one-of nouns*) (λ*n*. *succeed* [*d*, *n*]))

From this, it should be intuitively clear how a `do`-block is translated into nested binds.

A verb phrase in English is either a verb, or a transitive verb followed by a noun phrase. Note here that this example uses *both* choice and sequencing.

definition *verb-phrase* :: (*string*, *string list*) *parser* **where**
  verb-phrase ≡
    (*do*
    { *v* ← *one-of verbs*
    ; *succeed* [*v*]
    }) ⊕
    (*do*
    { *t* ← *one-of transitive-verbs*
    ; *n* ← *noun-phrase*
    ; *succeed* (*t*#*n*)
    })

Lastly, a sentence in our English-language fragment is a noun phrase followed by a verb phrase.

```
definition sentence :: (string, string list) parser where
  sentence ≡
    do { n ← noun-phrase
       ; v ← verb-phrase
       ; succeed (n@v)
       }
```

Note in all cases above, when parsing a fragment of English, our parsers return the words (or a list of them) that were parsed as their return value. We can now test our parsers, to make sure they are behaving as expected. First, some sentences that should be successfully parsed:

```
value run sentence ["some", "man", "likes", "the", "woman"]
value run sentence ["some", "child", "walks"]
```

Ambiguous sentences should also work fine. Note that all parses should be returned, and the continuation lists should look "correct":

```
value run sentence ["some", "woman", "loves", "a", "child"]
```

Multiple sentences can be parsed using iteration. Here we parse two consecutive sentences. Again all possible parses of the sentences should be returned:

```
value run (biter 2 sentence) ["some", "woman", "loves", "a", "child",
  "every", "man", "loves", "a", "child"]
```

Here is a parse that should fail (that is, return an empty list of results):

```
value run sentence ["some", "man", "hates", "a", "horse"]
```

Note that none of these examples will properly evaluate to a set of value-continuation pairs until you supply correct definitions in the exercises above.

end