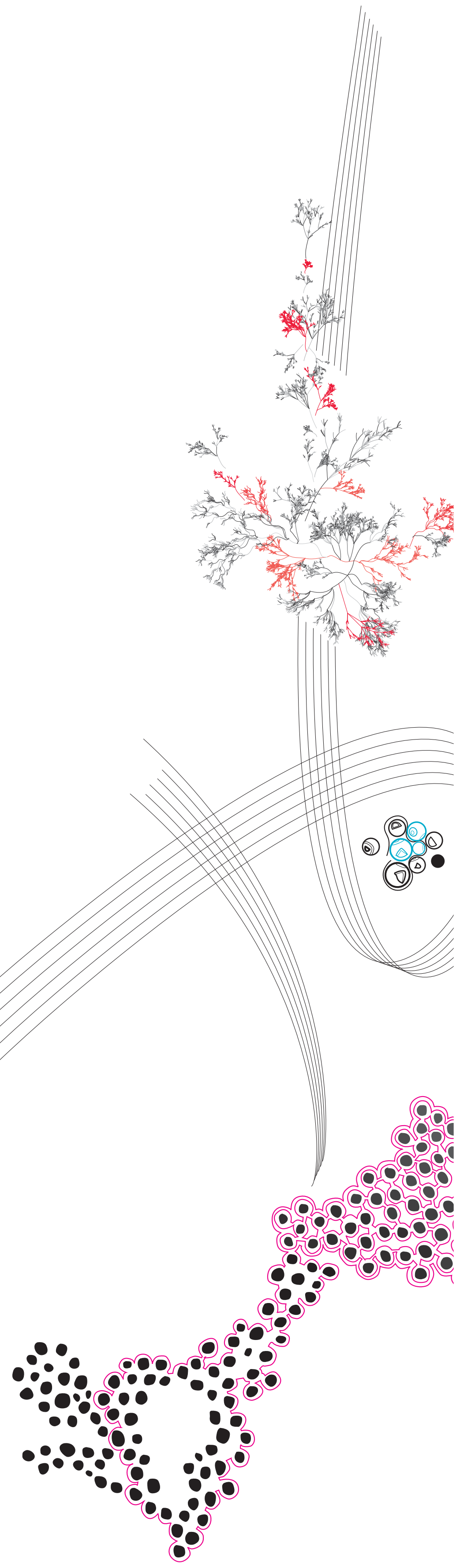
MSc Computer Science
Final Project



Verified Parser- and Printer-Combinator Bidefinition in the Isabelle Proof Assistant

Matthias Sleurink

Supervisor: Dr. Peter Lammich

October, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Parsers and printers are commonly used to transfer data, for example, using the JSON or XML formats. The correctness of these parsers and printers is pertinent to the functioning of the systems that use them to communicate. The separate design and programming of parsers and printers can cause slight differences in behaviour over time due to programming mistakes. By programming the parser and printer simultaneously, using the same expression, we can be sure their behaviours never deviate. Software verification is the practice of verifying the correctness of software. If a piece of software is verified, the verifier might need to verify the parsers and printers in the software as well. In this work, we introduce both a parser and printer bidefinition-combinator library and systems to verify that the bidefinitions created with it are compatible, created in the Isabelle proof assistant. To define what compatible behaviour means, we will describe various efforts made in the past to determine this and explain our choice of definition.

Keywords: Parsing, Printing, Parser Combinators, Software Verification

Chapter 1

Introduction

The separate definition of parsers and printers may cause deviations over time, as changes are made. This might cause issues in parser and printer compatibility, which can cause large issues in software systems. The ubiquitous server-client architecture for websites and desktop software depends on parsers and printers being compatible such that no confusion occurs. As such, we introduce a library that focuses on creating parsers and printers in one, using a parser-combinator-style mechanism. The following few sections will introduce the necessary background knowledge. First, we introduce the idea of a parser and explain some methods of creating parsers. We will then focus on parser combinators, one specifically relevant method of creating parsers. When those have been introduced the counterpart of the parser, the printer, is introduced. Some examples of printers with different goals are called upon, to compare the various ways that the word is used.

With parsers and printers introduced we focus on the relationship between them, showing that choices made in the implementation of the printer influence the way that the parser must read the printed text. Then, we introduce the idea of software verification, which can help us verify statements about the ways that parsers and printers can influence each other.

With these elements as a basis, the research goal for this thesis is introduced, together with subgoals that help in reaching the overarching goal.

The Isabelle code for this thesis is on github in repository `https://github.com/Matthias-Sleurink/bidefinition-thesis-theories`.

1.1 Parsers

A parser transforms input data into a format that a program can understand. This might be source code to an AST, text to JSON, or bytes to an image that can be displayed. They are pervasive in communication between computers, where data is put into some form (be that JSON, XML, or even raw bytes) then transmitted, and upon being received, parsed back into the object that was sent. Their functionality is used in all webpages, for the conversion of the HTML, CSS, and JS text into objects that the browser can use to display the webpage. And also in many native applications, as they communicate to the servers that store the data they show the user.

Parsers can be constructed using many methods. Some examples include parser generators, which create parsers from a grammar description, PEGs [3], with which the to-be-parsed language is described in a parsing expression grammar from which a parser can be generated, and parser combinators. This last example is especially relevant to our work, so will be explained below.

1.1.1 Parser Combinators

Parser combinators are a method of flexibly creating parsers from core building blocks. These building blocks can be split into two groups. That being the parsers and the combinators. Parsers are the functions that can parse some input, and combinators are functions that take some parsers or other input and produce new parsers.

For example, say we have the parser `one_character`, which parses any single character, and the combinator `then` which takes two parsers and returns a parser that applies both parsers to the input in order. With this, we can create a parser that parses two subsequent characters like so: `then one_character one_character`.

An important advantage of parser combinators is that combinators create parsers that can be put back into combinators to create more complex parsers, and so on. This allows the programmer to make complex parsers from simple building blocks, and combine those complex parsers in the same way that they combined the simpler parsers.

1.2 Printers

Printers, also known as pretty-printers or serialisers, transform some input object into text or binary data. Examples include Google protobuf, Javas `Serializable`, and Pythons pickle. Though serialisers and printers can be seen as similar, they have different goals. Where a serialiser often intends to write in a format that is fast to read by a computer, a pretty printer might focus more on readability by humans. For example, the before-mentioned protobuf software writes encodings of data in the shape of variable-length byte arrays. In contrast, the JSON serialiser built for the Kotlin language has a mode to pretty-print data, adding newlines between elements and adding indentation for nested structures.

1.3 The Relationship Between Parsers and Printers

If a server serialises some object to be transmitted to a client it expects that client to be able to read the object without any data loss (modulo networking issues). If this goes wrong in any way, it could cause very significant problems for the software package as a whole. Imagine for example a messaging application. The client sends the current timestamp and the message text to the server. It first writes the current timestamp as 8 consecutive bytes, and after that writes the Unicode bytes of the text. On the server side, the server wrongly starts reading the text and interprets the last 8 bytes as the timestamp. This is the kind of issue that can occur when parsers and printers are developed separately.

Another issue like this is the printing and parsing of variable-length lists of elements. How does the parser know when it should stop parsing list elements? We lay out various ways of resolving this:

First, the printer might print the amount of elements in the list before the list. It might print some special terminating element after the last element in the list, though it has to be sure that the terminator cannot be confused for a list element. It may print a specific marker before each element of the list, with a different marker being printed when there are no more elements to be parsed. It may encode the elements in such a way that whatever is printed after the last list element can never be interpreted as another list element.

This last option is especially interesting because it does not require adding extra data into the printed stream, but it is also especially difficult, because it might be hard to tell when the element printed after the list cannot be parsed by the parser that parses the list elements. Because this is difficult to tell, it may be good to prove that this confusion can indeed never happen. For this, we use Software Verification.

1.4 Software Verification

Software verification is the practice of verifying the correctness of software. It may be done by relating the input and output of some procedure to each other with mathematical equations. For example, a function that increases a number might look like $f\,i = i+1$. Does the function adhere to predicate $f\,i > i$? Yes, assuming that numbers are implemented in such a way that they do not wrap around. So, numbers implemented as Peano numbers would adhere to the predicate, but an 8-bit integer like in the x86 computer architecture does not.[1]

1.4.1 Software Verification, Testing, and Bounded Model Checking

Software verification might seem similar to software testing. However, there is a difference that makes the guarantees given by testing significantly weaker than guarantees given by software verification. For example, let us go back to the function $f\,i = i + 1$. If we wanted to test that this function adheres to the predicate $f\,i > i$ we might design a few tests. We would at least want to test at -1 and 1, and then possibly also at 0 and other interesting values. But with software testing, we can never cover *every* input. A core advantage of software verification is that a predicate is proven for every possible input, not shown for a few inputs. Even if the amount of possible inputs is so large it could never be enumerated in a lifetime.

Bounded model checking is closer to software verification than testing, but also has its limits. CBMC is described by its authors as "a tool for the formal verification of ANSI-C programs using Bounded Model Checking"[1]. CBMC creates a model of code and checks that that model adheres to some given predicates. This model, however, may be infinitely large, and can only be checked up to a certain bound. Thus, the "bounded" part of the model checking limits the guarantees this software gives to being weaker than the guarantees that can be given by software verification as a whole.

1.4.2 Interactive Theorem Provers

Interactive theorem provers are programs that can verify the correctness of mathematical proofs. The interactive part in the name denotes that the software provides an IDE-style interface to the user. Which is used to "program" the proofs. The IDE shows the user the assumptions that they can use, and the goal that they need to prove from them. Examples of interactive theorem provers include Coq [15], Isabelle [12], and Lean [9].

1.4.3 Isabelle Proof Assistant

The Isabelle Proof Assistant is software that can help the user create and verify the correctness of mathematical proofs. It is built upon the idea of a trusted core [12]. Upon this trusted core a large amount of "correct by construction" proof methods and tools are built. One of those tools is Isabelle/HOL. Isabelle/HOL is especially useful for software verification because it can be used as a functional programming language. In fact, the manual for Isabelle/HOL [11] introduces HOL with the equation HOL = Functional Programming +

[1]An overflow error caused the standard library binary search implementation of the JDK to throw unexpectedly between 2004 and 2006 [14]

Logic. When using HOL an Isabelle user has access to all the things a programmer expects from a functional programming language. These facilities are built into the mathematical language that Isabelle uses to create lemmas. This makes the Isabelle Proof Assistant, especially together with the Isabelle/Jedit IDE, an effective tool for software verification.

1.5 Research Goal

The main research goal for this thesis is to '*define an easy-to-use library for verified parser- and printer-combinators.*' To create a verification system for parsers and printers we must first define what makes a parser and printer pair well-formed. For this, we define our first subgoal: '*Defining a notion of "well-formed" for a parser and printer pair.*'

When we have defined a notion of well-formedness we need to formalise this as a predicate in a proof assistant. For us, the proof assistant of choice is Isabelle/HOL. So, our second subgoal is: '*Formalising this well-formedness in Isabelle.*'

Separate from the creation and formalisation of the well-formedness criteria we need to actually create the library. We need several parser-printer pairs (bidefinitions) and combinators that operate on these pairs (bicombinators). For this, we define our third subgoal: '*Creating a library of parser and printer bidefinitions and bicombinators.*'

The next step after creating the formalisation and the bidefinitions and bicombinators is to create lemmas that show that the bidefinitions and bicombinators are well-formed. This includes statements like 'the bidefinition for a single character is well-formed', but also like 'to prove that these two bidefinitions when composed consecutively are well-formed we need to show that...' This process is captured in the fourth subgoal: '*Creating lemmas to prove the well-formedness of these bidefinitions and bicombinators*'

Once we have this library, and the rules to prove the elements are well-formed, we need to show that the library is usable. For this, we will create and verify the well-formedness of a bidefinition for the DIMACS file format. This process will then fulfil the last subgoal: '*Show the usability of the library by creating a bidefinition for an example file format and proving that it is well-formed.*'

Chapter 2

Literature review

The bidefinition of parsers and printers has been investigated multiple times:

In 2010 T. Rendel and K. Ostermann created a system of parsers, printers and combinators that together allow the user to create more complex parser and printer combinations. They called them invertible syntax descriptions. Their system can properly handle associativity and priorities for infix operators, as well as variable whitespace [13].

Zirin Zhu, et al. created the BiYacc [19] library for Haskell, based on bidirectional transformers, it allows the user to write code once, and get a parser and printer. They include a system of normalisers such that changes to the AST (like from optimizers) are reversed in the printing step. They aim not to have the input and output equal but to have syntactic characteristics like brackets to stay the same. As such, they do not make any strict claims about the relationship between the parsed and printed text.

Freed Wiedijk, in his work on Pollack Consistency, investigated the relationship between parsed and printed texts [18]. The claim Wiedijk makes is that the parser and the printer of a proof assistant must be connected such that the printer never prints a true statement in such a way that it is parsed as a false statement by the same proof assistant. For Isabelle, the counter-example given is a proof for "False", which uses specific instructions to the Isabelle parser to accept the text "False" as notation for the boolean value true. This notation context is not printed when terms that were originally parsed by Isabelle using the notation are printed. As such, Isabelle will happily print "False" for the boolean value true without this notation context. This, in turn, means that the Isabelle parser will parse the boolean value false from the text "False" and will not be able to prove this equal to true.

The verification of parsers has also been discussed before. A. Korpowski and H. Binsztok present TRX, their parser interpreter that is developed and verified in Coq [5]. They extract a parser from a PEG [3], and prove that this extracted parser is terminating, and correct with respect to the original grammar and the semantics of PEGs.

Also in Coq, S. Lasser, et al. present the fast and verified ALL(*) parser Costar [7]. Their parser is sound, complete for non-left-recursive grammars, and provides a termination guarantee for those non-left-recursive grammars.

J. Xiaodong, et al. present a parser generator and recognizer, also verified in Coq [4]. Their system is proven correct with respect to the grammar, and the time complexity of the parsing algorithm is also verified with respect to the grammar.

We aim to improve upon these works by including our printer in the definitions of "verified". Our work is also different in that it focuses on parser combinators as a method of creating parser printer pairs, instead of using a PEG or grammar description to base the parser and printer on.

The importance of the verification of parsers can also be seen in other work. In 2024 P. Lammich presents a verified UNSAT certificate checking algorithm that is written and verified completely in Isabelle [6]. The project verifies every part of the certificate checking, including the parsing step. This clearly shows that a generalised library for the parsing step of a verification project would be helpful to verification efforts.

In 2024 S. Tilscher and S. Wimmer present an LL(1) parser generator for use in Isabelle [16]. Their system can, from a given LL(1) grammar, generate a parser which is proven correct, sound, and terminating. The release announcement for this tool contains the following statement: "Great news for everyone who uses Isabelle to build practical tools that involve parsing an input (i.e., all practical tools?)." [17] This, again, clearly shows that there is indeed a need for a generalised library for the parsing step of a verification project.

2.1 Comparison of Existing Bidefinition Techniques

Two existing methods for bidefinitions have been noted above in the literature review. They are Invertible Syntax Descriptions [13], and BiYacc [19]. Since our work claims to improve upon these works it makes sense to compare various features.

First, what kinds of guarantees do the authors make about the created bidefinitions? For Invertible Syntax Descriptions, the authors only show a few examples that show inversion, but no claims about parsers being compatible with printers are ever made. Similarly, their system for two-way functions has no safeguards against misuse.

The BiYacc system claims that its parsers and printers are fully compatible. The authors state that "The pairs of parsers and printer generated by BiYacc are thus always guaranteed to satisfy the usual round-trip properties." Which they later describe as ensuring that the parsers can parse a string into exactly the same AST as was used as input by a printer, and vice versa. There is a limitation to this method of compatibility. They assert that `parse(print(o)) = o` and that `print(parse(t)) = t`, which means that, for example, a parser that can parse a variable amount of whitespace between two characters, may not allow the printer to print a different default amount. This style of un-parsing is not a goal for our library. More about why we believe this is a disadvantage for bidefinitions in Section 3.5 which describes what we believe is a better option for parser and printer compatibility.

Secondly, what does the user need to do to describe their parser and printer? For Invertible Syntax Descriptions, the syntax is similar to the syntax of the Applicative type class in Haskell. Actions are sequenced using operators $<*>$, $<*$, and $*>$. Where the first applies both and returns from parsing (and takes as input for printing) the result of both, and the second and third return the result of only the left and right respectively. From these composition operators, more complex combinators like many and foldl can be created, which are used in the same way as parser combinators are. When a user wants to operate on the results of the parser (and on the input to printers) they use a combinator that takes a function for both cases separately.

For BiYacc, the user must implement three things. An abstract data type, which is what is returned from the parser, a concrete syntax definition (very similar to a grammar), which is what describes the shape of the parsed and printed string, and a set of actions that describe how to transform the abstract data type into the concrete syntax definition returned by the parser.

Chapter 3

Solution Design

In this section, we describe the choices made to define the parsers, printers, and bidefinitions. We follow that up with a technical explanation of how to define bidefinitions that might run forever (like `many`) in Isabelle, which does not allow us to trivially define non-terminating functions. After that, we describe the methods used to make statements about parser and printer results and errors. Based on these methods we build our formalisation of well-formedness.

3.1 Defining Parsers and Printers

A parser is a function from a string to a result, since they do not always succeed, they need an error case. Since a parser might not consume the whole input, we also need to return the leftover string if there is a successful result. Lastly, for reasons that will be explained in Section 3.3, the partial function setup needed for possibly non-terminating parsers requires another optional to wrap this result type. So, the type is as follows:

```
'a parser = string ⇒ ('a × string) option option
```

Returning `None` means the parser did not terminate for that input, returning `Some None` means the parser had an error for that input, and returning `Some Some (A, L)` means the parser succeeded in parsing object `A` from the input string and did not consume string `L`.

A is a function from an object to a string, they do not always succeed, so need an error case. Lastly, for the same reason that will be explained later in Section 3.3 the partial function setup needed for possibly non-terminating printers requires another optional to wrap this result type. So, the type is as follows:

```
'a printer = 'a ⇒ string option option
```

Where returning `None` means the printer did not terminate for that input, returning `Some None` means the printer had an error for that input, and returning `Some Some T` means the printer succeeded with the result string `T`.

3.2 Bidefinitions

To create a parser combinator-style system to create parsers and printers at the same time we must first define such a bidefinition. We define this as an object that we can ask to provide a parser and a printer. So, we make the type `bidefinition` with at least two accompanying functions: `parse :: 'a bidefinition => 'a parser` and `print :: 'a bidefinition => 'a printer`

Since we have no infrastructure for bidefinitions yet, our first bidefinitions will be made by combining a parser and a printer into a bidefinition. This can be done with a tuple, a GADT, or any other method. In our case, we choose a function: `bdc :: 'a parser => 'a printer => 'a bidefinition`.

So, with as an example a bidefinition for any single character, we will explain the idea of what we need to do. First, the parser, it needs to take a string input, error on empty, and return the head as a result if not:[1]

```
fun a_char_parser :: "char parser" where
  "a_char_parser []      = Some None"
| "a_char_parser (c#cs) = Some (Some (c, cs))"
```

The printer function takes a character, and returns a string containing only that character:

```
fun a_char_printer :: "char printer" where
  "a_char_printer c = Some (Some [c])"
```

Now the bidefinition can be created as follows:

```
definition a_char :: "char bidef" where
  "a_char = bdc a_char_parser a_char_printer"
```

Of course, our end goal is to make parsers and printers at once. So, we will now create a combinator that can directly take bidefinitions. For example, let us try to create a bicombinator that receives two bidefinitions and applies them subsequently. In a grammar or a PEG this would be the composition operator.

The parser for this combinator needs to apply the input string to the parser of the first bidefinition. Then if the first parser fails the combined parser should also fail. If the first parser succeeds, the combined parser needs to apply the leftover from the first parse operation to the second parser. If the second parser fails, the combined parser should also fail. Otherwise, if the second parser succeeds, the combined parser should return both results in a pair.

```
fun compose_parser :: "'a parser ⇒ 'b parser ⇒ ('a × 'b) parser" where
  "compose_parser A B i = (case A i of
      None ⇒ None
    | Some None ⇒ Some None
    | Some (Some (ra, i')) ⇒ (case B i' of
        None ⇒ None
      | Some None ⇒ Some None
      | Some (Some (rb, l)) ⇒ Some (Some ((ra, rb), l))))"
```

Since the parser returns a pair of results, it makes sense for the printer to receive a pair of objects to print. So, we apply the left input to the first printer. Similar to the parser, if the first printer fails we fail too. Otherwise, the right input is applied to the right printer. If the second printer fails, the composition printer fails too. Otherwise, the right print output is appended to the left print output and the complete string is returned as the combined print result.

```
fun compose_printer :: "'a printer ⇒ 'b printer ⇒ ('a × 'b) printer" where
  "compose_printer A B (a,b) = (case A a of
      None ⇒ None
    | Some None ⇒ Some None
    | Some (Some ra) ⇒ (case B b of
        None ⇒ None
      | Some None ⇒ Some None
      | Some (Some rb) ⇒ Some (Some (ra @ rb))))"
```

[1]This is real code, located in the `example_small_examples.thy` file.

Then we can use the same mechanism used above to create a bidefinition from the parser and printer.

```
definition compose :: "'a bidef ⇒ 'b bidef ⇒ ('a × 'b) bidef" where
  "compose A B = bdc (compose_parser (parse A) (parse B))
                     (compose_printer (print A) (print B))"
```

Now, we can pass the character bidefinition to the compositing combinator twice to get a bidefinition that parses (and prints) any two characters.

```
definition two_chars :: "(char × char) bidef" where
  "two_chars = compose a_char a_char"
```

It's not satisfying to create every bidefinition and bicombinator as a pair of parser and printer that needs to be combined. So, we create a set of basic bidefinitions and bicombinators that use this low-level creation mechanism and then build all others on top of these basic elements.

3.2.1 The Basic Elements

Similar to the implementation of PEG [3] we split up the elements of the library into the basic and the derived groups. The basic group is built out of a parser and printer, which is then combined with the bidefinition constructor into a bidefinition. The derived group is built on top of this basic group. The basic elements are:

1. `one_char :: char bidefinition`
   Parses and prints any one character.

2. `return :: 'a => 'a bidefinition`
   Parser consumes nothing and returns the given value. Printer prints nothing if the value is the same as the passed-in value, and fails if not.

3. `fail :: 'a bidefinition`
   Parser always fails, printer always fails.

4. `if_then_else`
   `:: 'a bidefinition => ('a => 'b bidefinition)`
   `=> 'c bidefinition => ('b => 'a)`
   `=> ('b + 'c) bidefinition`
   Tries the first bidefinition, on success it uses the result to create the second and runs that, if the first bidefinition failed it runs the third bidefinition.

5. `ftransform`
   `:: 'a bidefinition => ('a => 'b option) => ('b => 'a option)`
   `=> 'b bidefinition`
   Transforms the result of the inner bidefinition. If the transformation fails the combinator fails too.

6. `peek :: 'a bidefinition => 'a bidefinition`
   Runs the inner bidefinition, fails if it does, on success it returns the result but acts like nothing was consumed or printed.

We can create our whole library of bidefinitions and bicombinators on top of these seven. This is good, because it allows us to create most of the library without needing to create parsers and printers separately. In essence, in creating our library we are experiencing our library as well.

We can now see that the composition bicombinator that we created from scratch above can be implemented as a special case of the `if_then_else` bicombinator.

3.2.2 Examples

To display how the derived elements are made, we will show a few bidefinitions that use these basic elements as building blocks.

Character for Predicate

With the `fallible_transform` bicombinator and the `one_char` bidefinition we can create a bidefinition that parses any character as long as a boolean predicate holds for that character.

```
fun char_for_predicate :: (char => bool) => char bidefinition where
  char_for_predicate P = fallible_transform one_char
                                (\c. if P c then Some c else None)
                                (\c. if P c then Some c else None)
```

At parse time the `fallible_transform` bicombinator applies the `one_char` parser, and it parses a character. Then, the transformation shows that only if the predicate holds for the given character the parser succeeds. If the predicate does not hold the parser fails. The same occurs on the printing side.

Note that this bidefinition is used for many parsers that are available in parser combinator libraries. For example, the `this_char`, `char_from_set`, and the `not_this_char` parsers.

Two of the same character

This is a bidefinition that can be made in various ways.

First, we will consider the variant made with the `dependent_then` bicombinator. The type of which shows how it functions fairly well: `dependent_then :: 'a bidefinition => ('a => 'b bidefinition) => ('b => 'a) => 'b bidefinition`. We give it a bidefinition, and the second bidefinition is created from the parse result of the first. This bidefinition is then also applied. The result of the second bidefinition is the result of the bicombinator.

With this combinator, we can create the `char_twice` bidefinition.

```
definition char_twice :: char bidef where
  char_twice = dependent_then
                 one_char
                 (\c. char_for_predicate (\c'. c' = c))
                 id
```

Another way to construct this bidefinition is with the composition and fallible transform bicombinators.

```
definition char_twice :: char bidef where
  char_twice = fallible_transform
                 (one_char then one_char)
                 (\cs. if (fst cs = snd cs) then Some fst cs else None)
                 (\c. Some (c, c))
```

The first lambda passed to the transform bicombinator is the parse time transformation. Here we check if the characters are equal, and if so return one as a parse result. The second lambda is the print time transformation, it needs to duplicate the one passed in character so that the `then` bicombinator can pass them along to the inner bidefinitions for printing.

Both of these bidefinitions have the same behaviour.[2]

3.3 Partial Function Setup

Any function defined in Isabelle needs to have a termination proof. Most often this proof is generated automatically by the underlying system defining the function. But, in some cases, this is not enough. For us, for example, the `many :: 'a bidef => 'a list bidef` bicombinator provides an issue for these automated proofs. The issue comes from the infinitely recursive nature of the bicombinator. It is defined as follows:[3]

```
many a =
  transform
    (sum_take :: 'a + 'a ⇒ 'a)
    (λl. if l = [] then Inr [] else Inl l)
    (if_then_else
      a
      (λr. dep_then (many a) (λ rr. return (r#rr)) tl)
      (return [])
      (hd))
```

This equation is infinitely recursive. The automatic termination proof system finds that there is only one parameter and tries to prove that this parameter shrinks every recursive call. Since this does not happen, the automatic system cannot prove that this is a terminating recursive call, so it is rejected. So, how do we define this combinator?

For this, we make use of the partial function package. Specifically, the Complete Partial Order 2 package in the HOL library. This package allows defining recursive functions using chain complete partial orders.

To use this package we define an ordering and a least upper bound for bidefinitions. The ordering we chose is a combined ordering on the parser and the printer. Two bidefinitions are ordered if and only if both the parser and printer are ordered. The parser and the printer, in turn, are ordered such that returning `None`, which means nontermination, is smaller than returning an error or a result. Following this same technique, the least upper bound of a set of bidefinitions can be found by creating a bidefinition from the least upper bound of all the parsers and the least upper bound of the printers.

From this definition of the least upper bound it follows that the bottom bidefinition is the one where the parser and printer both always return nontermination.

So, to define the many bicombinator, we can use the same definition as above, only instead of having the `function` package define it, we define it via the `partial_function` package, which does not make the termination argument over the bidefinition object, which is infinitely recursive, but over it being applied to the input.

We still need one more thing. For the partial function package to be convinced the function body is indeed terminating we have to show that it is monotonic. For this, we can again use the idea of the basic elements as laid out in Section 3.2.1. We prove that every basic element is monotone, and then we can build the monotonicity proofs on top of those. In practice, this means that every monotonicity proof for a bidefinition or a bicombinator looks as follows:[4]

```
lemma mono_then[partial_function_mono]:
  assumes ma: "mono_bd A"
  assumes mb: "mono_bd B"
```

[2]The proof for this is in the Isabelle sources; file example_char_twice.thy

[3]Note that the real definition in `derived_many.thy` is slightly different, but it works the same way.

[4]Slightly simplified for clarification, but no assumptions have been left out.

```
  shows "mono_bd (then A B)"
  unfolding b_then_def using ma mb
  by pf_mono_prover
```

The `pf_mono_prover` method is the method that the partial function system uses to prove the monotonicity of the function bodies that are defined in a partial function. It uses the rules in the `partial_function_mono` set and splits any if and case statements recursively. So, by adding our rules to this set, we can ensure that the method can prove the monotonicity of partial function bodies that use our custom combinators. For the many definition, as shown above, the following rules are used:[5]

```
mono_bd A ⟹ mono_bd (transform f f' A)
mono_bd A; ∀ a. mono_bd (B a); mono_bd C ⟹ mono_bd (if_then_else A B C f)
mono_bd A; ∀ a. mono_bd (B a) ⟹ mono_bd (dep_then A B f)
mono_bd (return i)
```

3.3.1 Induction Over Partial Functions

The partial function package provides an induction rule for every function defined using it. The intuition of the rule is that every meaningful instantiation of the function has some bottom that is never called upon because the 'level' above it, the function returns. If the predicate holds for this lowest level, and also for any level above it, the predicate holds in general. Precisely this means: We want to prove some predicate P for bidefinition A defined as A = B A. First, we require that P is admissible. In general, this means that P is of the right shape to be handled by the induction setup. Precisely, it says that: for every possible chain that emerges from the ordering, the predicate holds for the Least Upper Bound of that chain. Now, if we can show that P holds for the bottom bidefinition and that from the assumption that P A' holds, we can show that P (B A') holds, we get that P holds for A.

This is usable, for example, in well-formedness proofs.

There are limits to this technique though, as it requires the bottom bidefinition to adhere to the predicate in question. This causes issues for predicates that directly state that the parser must succeed with some result, or fail in some way. Section 3.4.3 lays out the idea of partial correctness, which might be a way to resolve this difficulty.

3.4 Proving Statements About the Result and Error States of Parsers and Printers

Before we explain how specific statements about parsers and printers can be made we will first lay out two methods that Isabelle users can use to prove statements in general.

The first of those is simplification. Simplification is the application of equality rules in any place they can, until a state is reached where no rule applies. A simplification rule is simply any rule of the shape `LHS = RHS`. When this rule is applied to an equation, any instance of `LHS` is replaced by `RHS`. A much-used example has the following four simplification

[5]These rules are slightly simplified, but no assumptions are left out.

rules:

1. $0 + n = n$
2. $(Suc\ m) + n = Suc\ (m + n)$
3. $(Suc\ m \leq Suc\ n) = (m \leq n)$
4. $(0 \leq m) = True$

Now we will use these rules to show that $0 + (Suc\ 0) \leq (Suc\ 0) + x$:

$$\begin{aligned} 0 + Suc\ 0 \leq Suc\ 0 + x &\overset{(1)}{=} \\ Suc\ 0 \leq Suc\ 0 + x &\overset{(2)}{=} \\ Suc\ 0 \leq Suc\ (0 + x) &\overset{(3)}{=} \\ 0 \leq 0 + x &\overset{(4)}{=} \\ True & \end{aligned}$$

The second relevant system is introduction rule application. The rules for rule application are in the form of $[\![A_1; ...; A_n]\!] \Longrightarrow A$. This means that to prove A it suffices to show all assumptions $A_1...A_n$. So, to prove a subgoal C using this rule Isabelle must unify A and C, and then replace the subgoal C with the new subgoals $A_1...A_n$. Taking care to replace all the unified variables and sub-expressions during unification in the assumptions as well.

To proof things about parsers and printers we need to relate the input of parsers and printers to their output. It would be ideal if we could take some proof about a bicombinator with its parameters and split it up into smaller proofs about the parameters themselves. This allows us to split up these proofs until we get to a level of abstraction that allows us to perform the proof.

For this, we devised the NER (Nontermination, Error, Result) scheme. Where we make rules for each bicombinator and bidefinition that relate their input to their output state. So, a predicate for when it has a result, returns an error, or returns nontermination. We focus especially on removing one "layer" of bicombinator per rule so that proofs are split up into smaller sub-proofs that can be solved using the same mechanism again recursively.

3.4.1 Printers: Nontermination, Error, and Result

As mentioned above, because of the Partial Function setup for bidefinitions, printers can be non-terminating, denoted by the predicate `p_is_nonterm :: 'a printer => 'a => bool`. Then, if and only if a printer returns an error for some input the following predicate holds: `p_is_error :: 'a printer => 'a => bool`. Then we can complement this with a predicate for having a result: `p_has_result :: 'a printer => 'a => string => bool`, which holds if and only if the printer, when given said input, prints said string.

Example: `then` has result

Now, say that we have the bicombinator `then`, which runs two bidefinitions subsequently. What can we say about the predicate `p_has_result` for `then`? Well, we can make a rule like so: [6]

[6]Proof for this rule is in the Isabelle sources. In file derived_then.thy.

```
p_has_result (print (then A B)) (Av, Bv) t ⟷
  (∃ ta tb. ta@tb = t
    ∧ p_has_result (print A) Av ta
    ∧ p_has_result (print B) Bv tb)
```

Stating that the printer that prints A and B subsequently has result *t* if and only if the two sub-printers have results that when appended create *t*.

Example: printer never prints empty

Let us say, for example, that we want to prove that some printer always prints at least one character. First, let us imagine some bidefinition to prove this for:

```
apple = b_then (this_string "apple") (optional (this_char 's'))
```

This bidefinition will parse either the text apple or the text apples.

Then, the predicate that we want to prove.

```
¬ p_has_result (print apple) i []
```

The relevant lemmas are the `p_has_result` lemmas for `then` and `this_string`.[7]

```
p_has_result (print (b_then ab bb)) (va, vb) t ⟷
  (∃ta tb. ta@tb = t
      ∧ p_has_result (print ab) va ta
      ∧ p_has_result (print bb) vb tb)

  p_has_result (print (this_string s)) i r ⟷ r = s ∧ i = s
```

We can apply these two rules to prove the lemma. The simplified proof, with simplified proof states interspersed between the statements, looks as follows.[8]

```
lemma ¬ p_has_result (print apple) i []
unfolding apple_def
  ¬p_has_result print b_then(this_string "apple")(optional this_char 's') i []
apply (rule b_then_p_has_result)
  ∄ta tb. ta@tb=[] ∧ p_has_result print this_string "apple" (fst i) ta ...
apply (rule this_string_p_has_result)
  ∄ta tb. ta @ tb = [] ∧ (ta = "apple" ∧ fst i = "apple") ...
by blast
```

This shows that on the printer side, this idea of peeling away the bicombinators one by one via these rules is quite effective. Each rule application simplifies the proof goal in such a way that another rule matches with it. Then, when a state is reached where no bicombinator or bidefinition rules apply, the proof can be finished by the existing proof automation facilities in Isabelle.

3.4.2 Parsers: Nontermination, Error, and Result

Just like printers, parsers also have the nontermination, error, and result states. The predicates are as follows:

```
is_nonterm :: 'a parser => string => bool
is_error :: 'a parser => string => bool
has_result :: 'a parser => string => 'a => string => bool
```

The first two are obvious, the last takes three parameters, an input string, the result, and the leftover after parsing.

[7]Real code; in files derived_then.thy and derived_this_string.thy

[8]Real proof; in file example_small_examples.thy

Example: `or` has result

The bicombinator `or :: 'a bidefinition => 'b bidefinition => ('a + 'b) bidefinition` takes two bidefinitions and uses the first if it succeeds, or the second if the first fails. The type shows that the result of the parser is either the result from the first parameter, or the result of the second parameter. The rules that govern when it has a certain result look as follows:[9]

```
is_error (parse (or p1 p2)) i (Inl lr) l ⟷ has_result (parse p1) i lr l
is_error (parse (or p1 p2)) i (Inr rr) l ⟷ is_error (parse p1) i ∧
                                             has_result (parse p2) i rr l
```

This rule too shows the focus on peeling away one level of bicombinator per rule application. If we want to prove that the result came from the first bidefinition passed in, and we apply this rule, the second bidefinition passed in disappears from the proof state. Note that in practice there also exists a rule that is generic over if the left or right parser succeeds, which is useful in places where a proof needs to be made regardless of which side succeeds.

Example: Proving Parsers Error on Empty Input

One of the bicombinators in the library we created is `optional :: 'a bidefinition => 'a option bidefinition`. Here, when the inner bicombinator fails, instead of failing, it returns an empty optional. To prove that this bicombinator is well-formed (more about this later on in Section 3.5) we need to prove that the bidefinition that is inserted returns an error for an empty input. So, let us see how we prove that.

First, let us imagine some bidefinition to prove this for:

```
apple = b_then (this_string "apple") (optional (this_char 's'))
```

This bidefinition will parse either the text apple or the text apples.

Then, the predicate that we want to prove. We state that the parser extracted from our bidefinition returns an error result when given an empty input.

```
is_error (parse apple) []
```

Relevant for us here are the `is_error` rules for `b_then` and `this_string`:[10]

```
is_error (parse (b_then ab bb)) i ⟷
  is_error (parse ab) i ∨
  (∃ r l. has_result (parse ab) i r l ∧
          is_error (parse bb) l)

is_error (parse (this_string (c#cs))) i ⟷
  i = [] ∨
  (hd i ≠ c ∨
   is_error (parse (this_string cs)) (tl i))
```

We can apply these two rules to prove the lemma. The proof, with simplified proof states interspersed between the statements, looks as follows.[11]

```
lemma is_error (parse apple) []
unfolding apple_def
  is_error parse (b_then (this_string "apple") (optional this_char 's')) []
apply (rule b_then_is_error)
  ... ⟹ is_error (parse (this_string ''apple'')) []
apply (rule this_string_is_error)
```

[9] Real code; in file derived_or.thy

[10] Real code; in files derived_then.thy and derived_this_string.thy.

[11] Real proof; in file example_small_examples.thy

```
  ... ⟹ [] = [] ∨ hd [] ≠ 'a' ∨ is_error parse this_string "pple" (tl [])
by clarsimp
```

This shows that the idea of peeling away one bicombinator at a time with proof rules is also effective on the parser side. After applying the first rule the proof state has simplified significantly, as there is no mention of the right-hand side of the `b_then` bicombinator anymore. Then, when the second rule is applied, we see the input string being compared to the parameter to `this_string` directly. From this state, many built-in proof automation tools are capable of finishing the proof.

3.4.3 Total and Partial Correctness

The `has_result` predicate models the idea of total correctness. The alternative to this, partial correctness, defines the idea that a parser has a certain result, or it does not terminate, for a given input. This idea comes from the induction rule that the partial function setup (explained in Section 3.3.1) provides.

This fixpoint induction rule has a big limitation for our use. For the induction to work, the predicate needs to hold for the bottom bidefinition, which is the bidefinition that does not terminate for any parse or print input.

This is not a problem for predicates that first assume that there is some result, and then require something about that result like so:

```
has_result P i r l ⟶ predicate i r l
```

Because the assumption of the implication is false for the bottom parser the predicate is trivially true. Examples of predicates like this include well-formedness (3.5), PNGI (4.3.1) and does not peek past end (4.2.1).

Predicates that it does cause issues for, are predicates that directly assert some result or failure. For example, the claim that a parser returns an error on the empty input, or that a printer succeeds on any input. These claims are not true for the bottom bidefinition, so we cannot use fixpoint induction to prove them.

This comes up, for example, in well-formedness proofs with the many bicombinator. For the well-formedness of the many bicombinator, it is necessary to prove that the bidefinition passed in does not parse whatever is printed after the list of elements it is actually meant the parse. For example, if a list parser parses the closing bracket as a list element it would not be well-formed, since it parses a list element that was not printed. So, this proof eventually involves a subgoal that states that `is_error P i`. This does not hold for the bottom bidefinition, so it cannot be used in the induction step of the proof.

From this problem, a possible solution emerges in partial correctness. Since we cannot make statements about there being a result or error state, why not try making the statement that there is either a result or nontermination? Experiments with this technique show that this is a meaningful approach. Though it of course has the limitation of not making very strong statements about the results of parsers. To get back these statements it would be necessary to create a proof technique that asserts termination for inputs.

On the other hand, the parser induction rule for the many combinator shows that it is possible to create a meaningful and usable 'total' induction rule for combinators defined via the partial function package. The many combinator runs a parser as often as it can, and when the parser fails, it returns the list of results. Intuitively the 'failure condition', that is, when the parser would be non-terminating, is when the inner parser does not consume anything but does succeed. So, if we can assert that the parser does not behave in that manner, we can create a total induction rule. This rule is formalised as follows:

```
lemma many0_induct:
  assumes pasi: "PASI (parse bd)"
  assumes step: "∀ i r l. has_result (parse bd) i r l ⟶
                      (∀rr l'. (length l < length i ∧ Q l rr l') ⟶
                               Q i (r # rr) l')"
  assumes last_step: "∀ i. is_error (parse bd) i ⟶ Q i [] i"
  shows "has_result (parse (many bd)) i r l ⟶ Q i r l"
  (proof...)
```

`PASI` where means that the Parser Always Shrinks Input. That is: `has_result p i r l ⟶ (∃ c. (i = c @ l ∧ c ≠ []))` This asserts that there is no 'step' in the parsing process that consumes nothing, yet succeeds, which would cause an infinite loop. The step assumption is the induction step, taking the predicate from the existing list of elements to a list one longer. Then the last step assumption is to show that the predicate must hold when the parser fails. From this, we can show that some predicate Q holds for any parse result of the many bicombinator when the given bidefinition is applied to it.

This predicate shows that it is viable to make statements about total correctness for parsers, even if they are defined via the partial function setup. The idea presented in this predicate, that is, that induction is possible on the size of the input string, can be transposed to any application of a partial function defined bidefinition.

3.5 Well-formedness in the Context of Bidefinitions

To show that a pair of printer and parser are compatible, or, well-formed, it makes sense to want a few things: One, the parser should be able to parse anything that the printer can print, creating the same value as inserted into the printer. Second, the printer should be able to print anything that the parser can parse. Intuitively it sounds sensible to say that the printer should print out the exact same text that the parser parsed for the object, but for our intentions that is not a goal.

For example, parsing a list of elements with spaces in between. It seems appreciable for the parser to allow a variable amount of whitespace between the elements. But that means that there is no unique source text to create each parsed list. For example, the text `'a_b'` and `'a__b'` both parse to the list `['a','b']`. So, for the printer to be able to print the exact source text, the parser would also have to return some information on the source text, like the amount of whitespace. This kind of exact 'unparsing' is not a goal of this thesis. As such, we choose not to require the printed text to be the same as the parsed text.

So, we can formalise this well-formedness by stating that a bidefinition is well-formed if the following two predicates hold for its parser and printer:[12]

```
parser_can_parse_print_result par pri ⟷
    ∀t pr. p_has_result pri t pr → has_result par pr t []
printer_can_print_parse_result par pri ⟷
    ∀t i l. has_result par i t l → (∃pr. p_has_result pri t pr)
```

Parsers Should Not Change Characters They do not Consume

As an implementation detail, we also require that the parser of a well-formed bidefinition does not change the characters it does not consume. Which is formalised in the predicate:

```
has_result p i r l ⟶ (∃ c. i = c @ l)
```

[12]Real code; in file types.thy

This means that for every possible result of the parser, it must hold that the input can be split into two strings, the consumed characters and the leftover. In the below discussion of proofs for well-formedness, this facet will not be elaborated here, because it is a predicate for which the proof is easily automated. So, it will be discussed under the name PNGI in Section 4.3, which describes the proof automation results.

3.5.1 Well-formedness of Bidefinitions

With the predicates above defined, we can start to prove well-formedness for our basic bidefinitions.

`fail` is trivially well-formed, as it never succeeds.

`return` is well-formed too, not as trivially, but since it always prints the empty string, and always succeeds in parsing it is obvious that the predicates hold.

`one_char` is well-formed too: For `parser_can_parse_print_result` we know that the printer always prints a string with just the character that should be printed and that the parser always succeeds if there is at least one character in the input, and that the parse result is the head of the input string, so we know it holds. For `printer_can_print_parse_result` we know that the printer can print any character, so it holds.

The proofs for these statements are also formalised in the files for the respective bidefinitions.

3.5.2 Well-formedness of Bicombinators

Just like in the NER system explained above it would be ideal if we could peel away at well-formedness by having rules for each bicombinator. This works fine, but being well-formed is more involved than one might intuitively think. To show why, we will show the rules needed to prove correct well-formedness for the bidefinitions created by three relatively simple bicombinators:

3.5.3 Example: `optional` Well-formed

`optional :: 'a bidefinition => 'a option bidefinition` is a relatively simple bidefinition. If the passed-in bidefinition fails to parse an input the outer bidefinition still succeeds, but it returns an empty optional. Then, during printing, an empty optional is printed as the empty string, and an optional with content has the content passed along to the inner bidefinition.

To prove that it is well-formed we need to prove both inner predicates. We will proof both in order:

```
parser_can_parse_print_result (parse (optional b)) (print (optional b))
```

We will unfold the definition:

```
∀t pr. p_has_result (print (optional b)) t pr ⟶
                has_result (parse (optional b)) pr t []
```

Then we can apply the `p_has_result` rule for `optional`.

```
∀t pr. (case t of None ⇒ pr = []
                | Some rr ⇒ p_has_result (print b) rr pr) ⟶
                  has_result (parse (optional b)) pr t []
```

Now we see that a None is printed to an empty string, so when it is parsed it also needs to turn into None. Applying the `has_result` rule for `optional` will show what needs to be true for that to happen.

```
∀t pr. (case t of None ⇒ pr = []
                | Some rr ⇒ p_has_result (print b) rr pr) ⟶
         (case t of None ⇒ is_error (parse b) pr ∧ pr = []
                  | Some r ⇒ has_result (parse b) pr r [])
```

So, we will split into two cases, either t is None, or it is not. We will also run the simplifier on both subgoals:

```
1. ⟦pr = []; t = None⟧ ⟹ is_error (parse b) []
2. ⋀a. ⟦p_has_result (print b) a pr; t = Some a⟧ ⟹
                              has_result (parse b) pr a []
```

Now it has become clear what we require of bidefinition b. It needs to error on an empty input to resolve the first subgoal, and for the second subgoal, we need it to satisfy the `parser_can_parse_print_result` predicate.

For the second sub-predicate:

```
printer_can_print_parse_result (parse (optional b)) (print (optional b))
```

We will unfold the definition:

```
∀t i l. has_result (parse (optional b)) i t l ⟶
  ∃ ta. (p_has_result (print (optional b)) t) ta
```

Now we apply both the printer and parser `has_result` rules for optional:

```
∀t i l. (case t of None ⇒ is_error (parse b) i ∧ i = l
                 | Some r ⇒ has_result (parse b) i r l) ⟶
     (∃ta. case t of None ⇒ ta = []
                 | Some rr ⇒ p_has_result (print b) rr ta)
```

From this we see that if t is None, there needs to exist some ta that is the empty string, which is trivially true. So, we split into two subgoals, one where t is None and one where t is not None. The simplifier proves the first subgoal for us, and we are left with the following:

```
⟦has_result (parse b) i a l; t = Some a⟧ ⟹ ∃t. p_has_result (print b) a t
```

Now we see what we require from bidefinition b. If `parse b` can parse some text `i` into result `a` with leftover `l`, we need there to be some text `t` that `print b` can print element `a` into. This is covered by predicate `printer_can_print_parse_result`.

We can formalise these found requirements into the following rule:

```
lemma optional_well_formed:
  assumes "is_error (parse b) []"
  assumes "bidef_well_formed b"
  shows "bidef_well_formed (optional b)"
  (proof)
```

So, for `optional b` to be well-formed we need the parser for `b` to error on the empty input, and we need `b` to be well-formed. What we have shown here[13] is an example of a rule where we can peel away the bicombinator to get two 'smaller' subgoals that we can then attack recursively using the same tactics.

3.5.4 Example: `or` Well-formed

`or :: 'a bidefinition => 'b bidefinition => ('a + 'b) bidefinition` is a slightly more complex bidefinition. If the first passed in bidefinition fails to parse something then the second bidefinition may try, if that also fails then the whole bidefinition fails. During

[13]Real proof; in file derived_optional.thy

printing the sum type variant that is filled in chooses the bidefinition that is used. To prove that it is well-formed we need to prove both inner predicates. We will show how to proof both, in order:

```
parser_can_parse_print_result (parse (or b1 b2)) (print (or b1 b2))
```

We unfold the definition of the predicate, and also the `has_result` and `p_has_result` rules for `or`:

```
case t of Inl lr ⇒ p_has_result (print b1) lr pr
        | Inr rr ⇒ p_has_result (print b2) rr pr ⟹
    case t of Inl lr ⇒ has_result (parse b1) pr lr []
            | Inr rr ⇒ is_error (parse b1) pr ∧
                        has_result (parse b2) pr rr []
```

Then we split into two subgoals, for if t is `Inl` or `Inr`, and we run the simplifier on both new goals:

```
⟦p_has_result (print b1) a pr; t = Inl a⟧ ⟹ has_result (parse b1) pr a []
⟦p_has_result (print b2) b pr; t = Inr b⟧ ⟹ is_error (parse b1) pr ∧
                                             has_result (parse b2) pr b []
```

The first subgoal is resolved if `b1` satisfies `parser_can_parse_print_result`. The second subgoal shows us that `b2` needs to satisfy the same predicate, and also `parse b1` must error on any output of `print b2`.

Now for the second predicate:

```
printer_can_print_parse_result (parse (or b1 b2)) (print (or b1 b2))
```

We unfold the definition of the predicate and the `has_result` and `p_has_result` rules for `or`:

```
case t of Inl lr ⇒ has_result (parse b1) i lr l
        | Inr rr ⇒ is_error (parse b1) i ∧
                   has_result (parse b2) i rr l ⟹
    ∃l. case t of Inl lr ⇒ p_has_result (print b1) lr l
                | Inr rr ⇒ p_has_result (print b2) rr l
```

Again we split the proof into two subgoals for the possible states of t and then run the simplifier:

```
⟦t = Inl a; has_result (parse b1) i a l⟧ ⟹ ∃l. p_has_result (print b1) a l
⟦t = Inr b; is_error (parse b1) i; has_result (parse b2) i b l⟧ ⟹
    ∃l. p_has_result (print b2) b l
```

These subgoals show that `b1` and `b2` both must satisfy `printer_can_print_parse_result`.

These found requirements can be formalised into the following rule:

```
lemma or_well_formed:
  assumes "bidef_well_formed b1"
  assumes "bidef_well_formed b2"
  assumes "∀ v t. p_has_result (print b2) v t ⟶ is_error (parse b1) t"
  shows "bidef_well_formed (or b1 b2)"
  (proof)
```

So, we see that `or b1 b2` is well-formed if both `b1` and `b2` are well-formed, and if `parse b1` always errors on the results of `print b2`.[14] Note that just like for `optional`, this rule is an example of a rule that peels away the bicombinator to create smaller subgoals that can be dispatched recursively using the same tactics.

[14] Real proof; in file derived_or.thy

3.5.5 Example: `then` Well-formed

`then :: 'a bidefinition => 'b bidefinition => ('a × 'b) bidefinition` is the composition bicombinator. It runs the first bidefinition first, and if it succeeds it runs the second bidefinition on the leftover. To prove that it is well-formed we need to prove both inner predicates. We will show how to proof both, in order:

```
parser_can_parse_print_result (parse (b_then b1 b2)) (print (b_then b1 b2))
```

We unfold the definition and apply the parser and printer `has_result` lemmas:

```
⟦p_has_result (print b1) a ta; p_has_result (print b2) b tb⟧ ⟹
  ∃l'. has_result (parse b1) (ta @ tb) a l' ∧ has_result (parse b2) l' b []
```

Note how the parser for b2 needs to parse the leftover of b1. So, we need to add the assumption that the first parser does not eat into the 'territory' of the second parser. We do that by adding the assumption that `b1` and `b2` satisfy the following predicate:

```
no_collision b1 b2 = (∀t1 pr1 t2 pr2.
  p_has_result (print b1) t1 pr1 ∧ p_has_result (print b2) t2 pr2 ⟶
    has_result (parse b1) (pr1 @ pr2) t1 pr2)
```

We use the fact that we assume `b2` to be well-formed to get that it can parse its own print result `pr2`. This is convenient in some proofs, but not always. Later on, we discuss ways that this assumption breaks the proof, and alternatives that we have. Regardless, by using this assumption and the assumption that `b1` and `b2` are well-formed we can solve the proof goal.

The second predicate is proven easier:

```
printer_can_print_parse_result (parse (b_then b1 b2)) (print (b_then b1 b2))
```

We unfold the definition and the parse and print lemmas for `b_then` to gain the following two subgoals:

```
1. ⟦has_result (parse b1) i a l'; has_result (parse b2) l' b l⟧ ⟹
     ∃ta. p_has_result (print b1) a ta
2. ⟦has_result (parse b1) i a l'; has_result (parse b2) l' b l⟧ ⟹
     ∃ta. p_has_result (print b2) b ta
```

As we intuitively expect, the printers are not connected in the same way that the parsers are. We need to prove that they both have a result, but those results do not need to be compatible in any way, as that is proven on the parser side. So, we can solve these two subgoals using the assumption that both `b1` and `b2` are well-formed.

This gives rise to the following formalisation of the requirements found above:

```
lemma b_then_well_formed:
  assumes "bidef_well_formed b1"
  assumes "bidef_well_formed b2"
  assumes "pa_does_not_eat_into_pb_nondep b1 b2"
  shows   "bidef_well_formed (b_then b1 b2)"
  (proof)
```

Where the predicate in the third assumption is defined as follows:

```
pa_does_not_eat_into_pb_nondep ba bb ⟷ (
  ∀ t1 pr1 t2 pr2. p_has_result (print ba) t1 pr1 ∧
                   p_has_result (print bb) t2 pr2
      ⟶ has_result (parse ba) (pr1@pr2) t1 pr2)
```

This way of proving that `then` is well-formed is not the only method. In Section 4.2 we will discuss the ways to prove that parsers do not interfere with each other and a way that parsers can interfere with each other but still get the correct result.

Chapter 4

Results

This section starts with the introduction of all the bidefinitions and bicombinators created for the library and their well-formed rules. From that, several patterns are extracted which are looked at closely in sections about consecutive parsers (4.2) and proof automation (4.3). After that, we explain for each bidefinition and bicombinator which of the described predicates hold for it, and any preconditions needed. We conclude this section with an example, we will create a bidefinition for the DIMACS file format. The process of creating the bidefinition will be laid out step by step, and various proof techniques will be displayed.

4.1 Enumeration of Bidefinitions, Bicombinators, and Their Well-formed Rules

Now that we have explained the derivation of a few example well-formed rules we will enumerate all well-formed rules in the library. In this enumeration, a pattern of predicates will appear. After this pattern is noted the various oft-recurring predicates will be laid out in Section 4.2. The bidefinitions, bicombinators, and well-formed proofs shown below can be found in their respective Isabelle source files. For the basic elements they are called `basic_<name>.thy`, and for the derived elements the name follows the pattern `derived_<name>.thy`.

Fail

The bidefinition `fail :: 'a bidef` always fails. It is a basic bidefinition, trivially well-formed, as it can never print or parse a result, and it always terminates.

Fallible Transform

The bicombinator `ftransform :: ('a => 'b option) => ('b => 'a option) => 'a bidef => 'b bidef` allows the user to transform the result of the passed in bidefinition. It can also be used to select a subset of results by not transforming the result, but returning `None` for those elements of the type that are not in the wanted set. It is part of the set of basic bicombinators.

Intuitively, this bicombinator is well-formed when the transformation functions are the inverse of each other. This is true, and it can be used to prove well-formedness, but it is an overestimation of the requirements actually needed. Since the passed in bidefinition might not be able to create each element of its type with its parser, we can constrain the inverse requirement to only needing to hold in those cases where the parameter bidefinition can actually parse or print the elements. This gives rise to the following well-formed rule:

```
lemma ftransform_well_formed:
  assumes funcs: "well_formed_ftransform_funcs f f' b"
  assumes wf: "bidef_well_formed b"
  shows "bidef_well_formed (ftransform f f' b)"
  (proof...)
```

Where the predicate `well_formed_ftransform_funcs` models the idea of two functions that are each other's inverses in the cases where the parser and printer can handle the object. Which is formalised as follows:

```
well_formed_ftransform_funcs f f' b ⟷ (
    (∀ i v l v'. has_result (parse b) i v l ∧ Some v' = f v ⟶
        f' v' = Some v)
  ∧ (∀ pr t. p_has_result (print (ftransform f f' b)) t pr ⟶
        (∃t'. f' t = Some t' ∧ f t' = Some t)))
```

One Char

The bidefinition `one_char :: char bidef` is a basic bidefinition that parses and prints any single character. It only fails when the parse input is empty, never fails to print, and never returns the nontermination result. It is well-formed, as it can parse any printed character, and it can print any parsed character.

Peek Result

The bicombinator `peek :: 'a bidef => 'a bidef` is a basic bicombinator that runs the passed-in parser or printer, and if it succeeds, returns the result without printing any text and consuming any parse input. The peek combinator is intuitively not easy to be well-formed, because well-formedness requires the parser to be able to parse a print result, but the printer prints nothing, yet the parser does expect the passed in bidefinition to parse successfully. Hence, the well-formed rule for this bicombinator works by requiring that the passed-in bidefinition fails on anything that is not the empty string, which is the only thing that peek will ever print.

```
lemma peek_well_formed_empty:
  assumes "bidef_well_formed b"
  assumes "∀ t. t ≠ [] ⟶ is_error (parse b) t"
  shows "bidef_well_formed (peek b)"
  (proof...)
```

Return

The bidefinition `return :: 'a => 'a bidef` is a basic bidefinition that always succeeds parsing with the parameter as parse result, and succeeds printing only if the value to print is the same as originally passed in as a parameter. It never returns the nontermination result, and never fails to parse. It is always well-formed, as it parses and prints only the empty string.

If Then Else

The bicombinator `if_then_else :: 'a bidef => ('a => 'b bidef) => 'c bidef => ('b => 'a) => ('b + 'c) bidef` is a basic bicombinator which is the basis of control flow in the library. At parse time it runs the first parser, if it succeeds it creates the second bidef with the result from the first and runs it on the leftover input. If the first failed it runs the third parameter on the initial parse input. At print time it either uses the printer for `'c`, or uses the transformation function to create an `'a` from the `'b` object to be printed, uses that to create the bidefinition for `'b`, and then prints the `'a` and `'b` objects in order.

The if-then-else bicombinator has a well-formed rule, but most uses of this bicombinator in the library do not use this rule, because their instantiations stub out some parameters, which means that it is easier for them to work on the definition directly. Intuitively, what we need for well-formedness is a few things.

Firstly, if a printed text is printed by the else parameter of the bidefinition, it must not be parsable by the if parameter. Secondly, if a printed text is printed by the if and then parameters of the bidefinition, it must not be the case that the if parameter fails to parse it. Lastly, the passed-in parameters must be well-formed, and all the 'then' bidefinitions that can be made from the 'if' bidefinitions results must be well-formed as well. This is formalised in the following rule:

```
lemma if_then_else_well_formed:
  assumes "bidef_well_formed ab"
  assumes "∀ i r l. has_result (parse ab) i r l ⟶ bidef_well_formed (a2bb r)"
  assumes "bidef_well_formed cb"
  assumes "b2_res_trans_is_b1_res ab a2bb b2a"
  assumes "b1_then_b2_print_parse_loop ab a2bb b2a"
  assumes "b1_cannot_parse_b3_print_result ab cb"
  assumes "pa_does_not_eat_into_pb ab a2bb"
  shows "bidef_well_formed (if_then_else ab a2bb cb b2a)"
  (proof...)
```

The first three assumptions regard the well-formedness of the passed in bidefinitions. The fourth assumption states that any result that can be created by the 'then' bidefinition when transformed back into the type of the 'if' bidefinition, can be a result of the parser of that bidefinition. This is to ensure that the transformed back elements stay 'inside' the set of elements that well-formedness makes statements about. The fifth assumption states that the print results from the 'if' and 'then' bidefinitions, when appended together, must be parsable into the same objects as were printed originally. The fifth assumption ensures that the 'if' parser fails on prints from the 'else' printer. Then the last assumption ensures that the 'if' parser does not eat into the printed text of the 'then' printer. A variant of this last predicate will return in many proofs that concern consecutive bidefinitions, and will thus be laid out in detail in Section 4.2, which explains the various ways to handle consecutive parsers.

Or

The bicombinator `or` is a derived bicombinator which attempts to parse using the first parameter first, and only if that fails, runs the second. At print time the type of the print input is used to decide which printer is used. It is defined as follows:

```
definition or :: "'a bidef ⇒ 'b bidef ⇒ ('a + 'b) bidef" where
  "or a b = if_then_else a return b (id :: 'a ⇒ 'a)"
```

Intuitively, for it to be well-formed it must be the case that both parameter bidefinitions are well-formed, and the first bidefinition should not be able to parse print results from the second bidefinition. This is formalised in the following rule:

```
lemma or_well_formed:
  assumes "bidef_well_formed b1"
  assumes "bidef_well_formed b2"
  assumes "∀ v t. p_has_result (print b2) v t ⟶ is_error (parse b1) t"
  shows "bidef_well_formed (or b1 b2)"
  (proof...)
```

Transform

The bicombinator `transform` is a derived bicombinator used to transform a value from one type to another if that transformation can never fail. This is usable, for example, in cases where a type constructor needs to be called with the parse result. Its definition is as follows:

```
definition transform :: "('a ⇒ 'b) ⇒ ('b ⇒ 'a) ⇒'a bidef ⇒'b bidef" where
  "transform t t' bi = ftransform (Some o t) (Some o t') bi"
```

Intuitively the transform combinator requires the transformation functions to be each other's inverse, but, for the same reason as in the fallible transform case described above, (Section 4.1) this inverse property only has to hold for the values of the type that the inner bidefinition can actually parse and print. This is formalised in the following rule:[1]

```
lemma transform_well_formed:
  assumes funds: "well_formed_transform_funcs f f' b"
  assumes wf: "bidef_well_formed b"
  shows "bidef_well_formed (transform f f' b)"
  (proof...)
```

Where the predicate `well_formed_transform_funcs` is defined as follows:

```
well_formed_transform_funcs f f' b ⟷ (
    (∀ i v l. has_result (parse b) i v l ⟶ f' (f v) = v)
  ∧ (∀ pr t. p_has_result (print (transform f f' b)) t pr ⟶ f (f' t) = t))
```

Dependent Then

The bicombinator `dep_then` is a derived bicombinator that fulfils the function of the bind function in a monad. It is meant for running two bidefinitions in order, where the second depends on the result of the first. The definition is as follows:

```
definition dep_then :: "'a bidef ⇒('a⇒'b bidef) ⇒('b⇒'a) ⇒'b bidef" where
  "dep_then ab a2bb b2a = (transform projl Inl
                                (if_then_else ab a2bb fail b2a))"
```

Well-formedness here requires, just like if then else before this, that when parsing, the first bidefinition does not eat into the text that the second bidefinition has printed. This intuition is formalised in the following rule:

```
lemma dep_then_well_formed:
  assumes "bidef_well_formed ba"
  assumes "∀ i r l. has_result (parse ba) i r l ⟶ bidef_well_formed (a2bb r)"
  assumes "reversed_b2_result_is_b1_result ba a2bb b2a"
  assumes "pa_does_not_eat_into_pb ba a2bb"
  shows "bidef_well_formed (dep_then ba a2bb b2a)"
  (proof...)
```

All assumptions are similar to the ones for if then else described in 4.1. The passed-in bidefinitions need to be well-formed, the transformation function needs to create the correct results, and the parser of the first parameter must not eat into the print result of the second bidefinition.

[1]This rule is actually called `transform_well_formed4`, and the predicate is similarly changed in name.

Character for Predicate

The bidefinition `char_for_predicate` is a derived bidefinition that parses and prints any character that satisfies the passed-in predicate. It is always well-formed, regardless of the predicate. The definition is as follows:

```
definition char_for_predicate :: "(char ⇒ bool) ⇒ char bidef" where
  "char_for_predicate p = dep_then one_char
                                  (λfound. if p found then return found
                                                   else fail)
                                  id"
```

Any from set

The bidefinition `any_from_set` is a derived bidefinition that parses and prints any character that is in the given set.

```
definition any_from_set :: "char set ⇒ char bidef" where
  "any_from_set s = char_for_predicate (λfound. found∈s)"
```

It is well-formed for any set. This bidefinition forms the basis for the bidefinitions for specific character sets listed below:

1. Any digit character
2. Any lowercase alphabet character
3. Any uppercase alphabet character
4. Any alphabet character
5. Any alphanumeric character
6. Any whitespace character

This Character

The bidefinition `this_char` is a derived bidefinition that parses and prints only the passed-in character. It is well-formed for any character, and is defined as follows:

```
definition this_char :: "char ⇒ char bidef" where
  "this_char c = any_from_set {c}"
```

This String

The bidefinition `this_string` is a derived bidefinition that parses and prints only the passed-in string. It is well-formed for any string, and is defined as follows:

```
definition this_string :: "char list ⇒ char list bidef" where
  "this_string = m_map this_char"
```

Character not in Set

The bidefinition `char_not_in_set :: char set => char bidef` is a derived bidefinition that parses and prints any character that is not in the given set. It is well-formed for any set, and is defined as follows:

```
definition char_not_in_set :: "char set ⇒ char bidef" where
  "char_not_in_set s = char_for_predicate (λfound. found∉s)"
```

Then

The bicombinator `b_then` is a derived bicombinator which runs the two parameter bidefinitions consecutively.

```
definition b_then :: "'a bidef ⇒ 'b bidef ⇒ ('a × 'b) bidef" where
  "b_then ab bb = dep_then ab (λa. transform (Pair a) snd bb) fst"
```

Intuitively, it is well-formed when both parameter bidefinition are well-formed, and the first bidefinition can parse the print output from the second appended to the first without breaking the second parser. This intuition, described in more detail above in the section that introduces well-formedness above (3.5.5), is formalised as follows.

```
lemma b_then_well_formed:
  assumes "bidef_well_formed b1"
  assumes "bidef_well_formed b2"
  assumes "pa_does_not_eat_into_pb_nondep b1 b2"
  shows   "bidef_well_formed (b_then b1 b2)"
  (proof...)
```

Where the predicate in the third assumption is defined as follows:

```
pa_does_not_eat_into_pb_nondep ba bb ⟷ (
  ∀ t1 pr1 t2 pr2. p_has_result (print ba) t1 pr1 ∧
                   p_has_result (print bb) t2 pr2
      ⟶ has_result (parse ba) (pr1@pr2) t1 pr2)
```

This is not the most general way to formalise the well-formedness of `then`. Note how the definition of the predicate above requires the leftover after the parser for `ba` has run, to be the exact same text as was printed by `bb`, this is not strictly necessary. More about this, and other ways of ensuring consecutive parsers can work, in Section 4.2.

Many

The bicombinator `many` is a derived bicombinator which runs a parser as often as it can and returns the full list of results. It is defined by the partial function setup as described in Section 3.3.

```
partial_function (bd) many :: "'a bd ⇒ 'a list bd":
"many a = transform
             sum_take
             (λl. if l = [] then Inr [] else Inl l)
             (if_then_else
               (dep_then a (λr. dep_then (many a) (λ rr. return (r#rr)) tl) hd)
               return
               (return [])
               (id)
               )"
```

The well-formedness of the many bicombinator intuitively depends on the parameter being well-formed, and not 'eating into' its own print results. Sadly, using the same predicate as we use for `then` (Section 4.1) is not enough. Mainly because if a parser does not eat into itself it may still eat into two of itself. Possible solutions for this include requiring that the parser does not eat into many of itself, or requiring that the parser does not peek past its end, or its own first character.

There are four rules for the well-formedness of `many A` for any `A`. They have three assumptions in common:

```
lemma well_formed_many...:
  assumes "bidef_well_formed A"
  assumes "PASI (parse A)"
  assumes "¬is_nonterm (parse A) [] ∨ is_error (parse A) []"
  assumes "Differs Each Time"
  shows "bidef_well_formed (many A)"
  (proof...)
```

Then, the four different predicates are:

```
"parse_result_cannot_be_grown (parse b)"
"does_not_peek_past_end (parse A)"
"parse_result_cannot_be_grown_by_printer (parse b) (print (many b))"
"∀i c. first_printed_chari (print A) i c
              ⟶ does_not_consume_past_char3 (parse A) c"
```

The first two are very similar. They both express the idea that if a piece of text can be parsed into a result, it must also be parsable into that same result when the leftover text is changed. The third expresses the intuition that even if a parse result *can* be changed by some text, it can never be grown by text that can be printed by many times itself. Then, lastly, we specialise the third statement by saying that if A does not consume past its own first character then it can make many well-formed. The second and fourth are explained in more detail in Section 4.2 about consecutive parsers.

Lastly, there is a specialization for many where the bidefinition inserted is an application of then:

```
lemma WF_many_then:
  // same common assumptions as above
  assumes "∀i c. first_printed_chari (print B) i c
              ⟶ does_not_consume_past_char3 (parse A) c"
  assumes "∀i c. first_printed_chari (print A) i c
              ⟶ does_not_consume_past_char3 (parse B) c"
  shows "bidef_well_formed (many (b_then A B))"
```

This specialisation is especially useful for cases where showing that the two bidefinitions, when combined, do not eat into some characters is harder than doing it separately, which is almost always the case.

Many1

The bicombinator `many1` is a derived bicombinator that parses with the passed in bidefinition as long as it succeeds and fails if zero elements were parsed. Similarly, the printer fails on the empty list. It is defined as follows:

```
definition many1 :: "'a bidef ⇒ 'a list bidef" where
"many1 a = ftransform
              (λ (r, rs). Some (r#rs))
              (λ l. if l = [] then None else Some (hd l, tl l))
              (b_then a (many a))"
```

For well-formedness, the library shows that proving well-formedness of `many A` also shows well-formedness of `many1 A` with the following rule:

```
lemma many1_well_formed_from_many:
  assumes "bidef_well_formed (many b)"
  shows "bidef_well_formed (many1 b)"
  (proof...)
```

Natural Number

The bidefinition `nat_b` is a derived bidefinition that parses and prints any natural number. That is, any whole number larger than or equal to zero. It is well-formed, and defined as follows.

```
definition nat_b :: "nat bidef" where
  "nat_b = transform
             nat_from
             print_nat
             (many1 digit_char)"
```

Integer

The bidefinition `int_b` is a derived bidefinition that parses and prints any integer. That is, any whole number. It is well-formed, and defined as follows:

```
definition int_b :: "int bidef" where
  "int_b = transform
            (λInl i⇒ − (int i) | Inr n ⇒ int n)
            (λn. if n < 0 then Inl (nat (−n)) else Inr (nat n))
            (if_then_else
              (this_char CHR ''−'')
              (λ_. nat_b)
              nat_b
              (const CHR ''−''))"
```

Separated By

The bicombinator `separated_by` is a derived bicombinator that parses and prints any amount of instances of the first parameter, separated by the second parameter. Since the bidefinition does not return the parse results of the separator, it takes in an additional parameter which is used as print input for the separator. This bicombinator works on the basis that a list of elements separated by separators can be parsed by first parsing one of the elements, and then parsing pairs of separators and elements together. It needs to take special care of the case where the first element parse fails though, as it has to ensure that this means the overall parser does not fail but returns empty. Accordingly, it is defined as:

```
definition separated_by :: "'b bd ⇒ 'a bd ⇒ 'b ⇒ 'a list bd" where
"separated_by sep elem sep_oracle =
  transform
    (λm_al. case m_al of None ⇒ [] | Some (a, l) ⇒ a#(map snd l))
    (λl. case l of [] ⇒ None | (a#as) ⇒ Some (a, map (Pair sep_oracle) as))
    (optional (b_then elem (many (b_then sep elem))))"
```

Similarly to the `many` rule described above (4.1) there are multiple ways of being well-formed for separated by. The two most relevant rules follow the same pattern. The only way they differ is in how they show that the element and the separator parser do not interfere with each other. With one rule using the does not consume past character method, and the other using the does not peek past end method. The first one is displayed here:

```
lemma separated_by_well_formed_no_consume_past_char:
  assumes "∃r. p_has_result (print sep) oracle r"
  assumes "bidef_well_formed elem"
  assumes "bidef_well_formed sep"
  assumes "is_error (parse elem) []"
  assumes "is_error (parse sep) []"
  assumes "PASI (parse elem) ∨ PASI (parse sep)"
  assumes "∀i c. first_printed_chari (print elem) i c ⟶
              does_not_consume_past_char3 (parse sep) c"
  assumes "∀i c. first_printed_chari (print sep) i c ⟶
              does_not_consume_past_char3 (parse elem) c"
  assumes "∀i c. first_printed_chari (print (b_then sep elem)) i c ⟶
              does_not_consume_past_char3 (parse (b_then sep elem)) c"
  shows "bidef_well_formed (separated_by sep elem oracle)"
  (proof...)
```

The first three assumptions are obvious. The oracle needs to be printable, and the element and separator bidefinitions need to be well-formed. Then, the fourth is to ensure that empty parses fail correctly. That is, if there is no printed text, then the first element parser must not return a result. The fifth is to ensure that there is no way for the element parser to see another printed element text right after itself. That is, since the 'does not consume past char' assumptions are all based on the 'next' bidefinition we must ensure that it cannot look at itself in the text. The sixth is to ensure that the combinator application `b_then sep elem` is PASI. That is, it always consumes at least one character. This is to ensure that the bicombinator never enters a loop where the separator and element parse correctly, but nothing is consumed, so they again parse correctly, and so on. The last three assumptions are the parts where we ensure that the parsers never look past their own 'territory'. However, we limit that requirement to only those cases where the character after their own 'territory' is printable by the other bidefinition.

Separated By 1

The `separated_by1` is a derived bidefinition to parse elements separated by separators, ensuring that at least one element is parsed and printed. This bidefinition is not derived from the `separated_by` bicombinator, as that one's implementation is complicated by needing to support the 0 element case. As such, it is defined as follows:

```
definition separated_by1 :: "'a bidef ⇒'b bidef ⇒'b ⇒ 'a list bidef" where
"separated_by1 elem sep oracle =
  ftransform
    (λ(x,xs). Some (x#(map snd xs)))
    (λi. case i of [] ⇒ None | x#xs ⇒ Some (x, map (Pair oracle) xs))
    (b_then elem (many (b_then sep elem)))"
```

As described above in the section about the well-formedness of `separated_by` and `b_then`, the well-formedness of bicombinators that operate their parameters subsequently can be resolved in many ways. This is the reason that this bidefinition does not have a complex rule, but rather dispatches the proof to the `b_then` and `many` combinators like so:

```
lemma separated_by1_well_formed:
  assumes "∃t. p_has_result (print sep) oracle t"
  assumes "bidef_well_formed elem"
  assumes "bidef_well_formed (b_then elem (many (b_then sep elem)))"
  shows "bidef_well_formed (separated_by1 elem sep oracle)"
  (proof...)
```

Drop

The bicombinator `drop` is a derived bicombinator that turns the result type of a bidefinition to unit, and always prints the same value. This can be useful for any proofs that are inductive over the elements of the result type, where this bicombinator ensures that there is only one element in the result type. An example of a place like this is ignorable whitespace parsing, where the specific amount of whitespace parsed does not matter. It is defined as follows:

```
definition drop :: "'a bidef ⇒ 'a ⇒ unit bidef" where
  "drop A oracle = transform (const ()) (const oracle) A"
```

The well-formedness rule is obvious:

```
lemma drop_well_formed:
  assumes wf_A: "bidef_well_formed A"
  assumes good_oracle: "∃t. p_has_result (print A) oracle t"
  shows "bidef_well_formed (drop A oracle)"
  (proof...)
```

Then drop first (and Then drop second)

The bicombinators `then_drop_first` are derived bicombinators to remove elements from the result tree that are not important. They both take in two bidefinitions and some element to print in place of the one that is ignored in the result tree. They are defined as follows:

```
definition then_drop_first :: "'a bidef ⇒ 'b bidef ⇒ 'a ⇒ 'b bidef" where
  "then_drop_first ab bb a = transform
                              (snd :: ('a×'b) ⇒ 'b)
                              ((λ b. (a, b)) :: 'b ⇒ ('a×'b))
                              (b_then ab bb :: ('a×'b) bidef)"
definition then_drop_second :: "'a bidef ⇒ 'b bidef ⇒ 'b ⇒ 'a bidef" where
  "then_drop_second ab bb b = transform
                              (fst :: ('a×'b) ⇒ 'a)
                              ((λ a. (a, b)) :: 'a ⇒ ('a×'b))
                              (b_then ab bb :: ('a×'b) bidef)"
```

This can for example be useful in places where some element needs to be pre- or succeeded by a variable amount of whitespace, but the amount of whitespace does not matter for the parse result. Which would be implemented as `then_drop_first (many1 whitespace) (AST) '' '' :: AST bidef` which results in a parser that parses one or more elements of whitespace and then some AST, returns only the AST as a result, and at print time, uses the default one space character to fill in the whitespace.

And so, for both bicombinators, the rules look like this:

```
lemma then_drop_first_well_formed:
  assumes "bidef_well_formed b1"
  assumes "bidef_well_formed b2"
  assumes "pa_does_not_eat_into_pb_nondep b1 b2"
  assumes "∃i. has_result (parse b1) i a []"
  shows   "bidef_well_formed (then_drop_first b1 b2 a)"
  (proof...)
```

The first three assumptions are directly lifted from the `b_then` well-formed rule, and the last is to ensure that the oracle element passed in to replace the ignored results can be parsed, which, via the well-formedness of its bidefinition, also means that it can be printed.

ws char ws, ws char, and char ws

These tree bidefinitions `ws_char_ws, char_ws, ws_char` are derived bidefinitions that ignore any whitespace on the left or right side of the parsed character, print no whitespace, and return a unit. This unit is returned for the reason described in the section about `drop` above, it is useful for proofs that are generic over the elements of the return type.

`char_ws` is well-formed. The other two bidefinitions are well-formed if the passed-in character is not in the whitespace set.

They are defined as follows:

```
definition char_ws :: "char ⇒ unit bidef" where
  "char_ws c = drop (b_then (this_char c) (many whitespace_char)) (c, [])"

definition ws_char :: "char ⇒ unit bidef" where
  "ws_char c = drop (b_then (many whitespace_char) (this_char c)) ([], c)"

definition ws_char_ws :: "char ⇒ unit bidef" where
  "ws_char_ws c = drop (b_then (many whitespace_char)
                               (b_then (this_char c)
                                       (many whitespace_char)))
                       ([], c, [])"
```

End Of File

The bidefinition `EOF :: unit bidef` is a derived bidefinition that succeeds in parsing only at the end of the file, and always prints the empty string. It is well-formed.

```
definition eof :: "unit bidef" where
  "eof = transform
          (λr. case r of Inl _ ⇒ undefined | Inr () ⇒ ())
          Inr
          (if_then_else one_char (λ_. fail) (return ()) id)"
```

Optional

The bicombinator `optional` is a derived bicombinator that returns `None` if the parameter bidefinition fails to parse, and if it succeeds it returns the value inside a `Some`.

```
definition optional :: "'a bidef ⇒ 'a option bidef" where
  "optional b = transform
                 (λr. case r of Inl v⇒ Some v | Inr _ ⇒ None)
                 (λr. case r of None ⇒ Inr () | Some v ⇒ Inl v)
                 (if_then_else b return (return ()) id)"
```

As discussed in more detail in Section 3.5.3 the well-formedness of the optional bicombinator depends on the parameter being well-formed, and the passed in bidefinition failing on the empty string, which is the print output of the `None` case. As such, the rule is formalised as follows:

```
lemma optional_well_formed:
  assumes "is_error (parse b) []"
  assumes "bidef_well_formed b"
  shows "bidef_well_formed (optional b)"
  (proof...)
```

Peek Boolean

The bicombinator `peek_bool` is a derived bicombinator that executes the given parser but does not consume anything. It returns false if the passed-in parser failed, and true if it succeeded. It always prints the empty string. As discussed in the section about the peek bicombinator that does return the result (4.1), it does not make sense to have a well-formed rule for it. It is defined as follows:

```
definition peek_bool :: "'a bidef ⇒ 'a ⇒ bool bidef" where
  "peek_bool a oracle = transform (Not o Option.is_none)
                                  (λb. if b then Some oracle else None)
                                  (peek (optional a))"
```

Monadic Map

The bicombinators `m_map` is a derived bicombinator that creates a list of bidefinitions from a list of parameters with a given function and executes them all consecutively. Its recursive definition is as follows:

```
fun m_map :: "('a ⇒ 'b bidef) ⇒ 'a list ⇒ 'b list bidef" where
  "m_map f []      = return []"
| "m_map f (a#as) = ftransform
                        (Some)
                        (λ[] ⇒ None
                         |(i#is) ⇒ Some (i#is))
                        (dep_then (f a)
                                  (λB. dep_then (m_map f as)
                                                        (λBs. return (B#Bs))
                                                        tl)
                                  hd)"
```

To be well-formed it first makes sense to require that all the parameters in the list, when applied to the function, create a well-formed bidefinition. Then, the created bidefinitions must not eat into the bidefinitions created from the elements that come after it in the list. This is formalised, less precisely, in the following predicate:

```
lemma m_map_well_formed:
  assumes "∀i∈set is. bidef_well_formed (a2bi i)"
  assumes "∀i is. pa_does_not_eat_into_pb_nondep (a2bi i) (m_map a2bi is)"
  shows "bidef_well_formed (m_map a2bi is)"
  (proof...)
```

This rule is significantly less precise than the one lined out in the intuition. That is because the only consumer in our library is the `this_string` bidefinition, which creates `this_char` bidefinition with the function, and that bidefinition never peeks past its own end, so it trivially does not eat into the bidefinitions created by the rest of the input list.

4.1.1 Patterns From all Bidefinitions and Bicombinators

From this listing, we can see two patterns. Firstly, the rules for combinations all make some requirements on the relationship between the passed in bidefinitions and then make well-formedness requirements on the passed in bidefinitions. This follows a pattern of 'peeling away' layers (bicombinators) to eventually reach the bottom of the definitions. This ties up well with the goals laid out in Section 3.4 which introduced the ideas of `has_result` and the other related predicates. The usability of this technique is expanded upon later, in Section 4.3 which describes various applications of proof automation.

Secondly, we can see that every bicombinator that deals with multiple bidefinitions being run consecutively has to deal with the parsers not eating into each others' 'territory' somehow. The various ways that this can be done have shortly been noted in the above enumeration and will be expanded upon more in Section 4.2, which focuses on various predicates created to prove that parsers do not interfere with the results of later parsers.

4.2 Consecutive Parsers

In this section, we describe the various ways that we use to describe that two parsers can be used consecutively without interfering with each other. This section is general across any two consecutive parsers, but it is best read by keeping in mind the context of Section 3.5.5 and 4.1. The first describes how a well-formed proof for the `then` bicombinator is made, and the second, the enumeration of all bidefinitions, bicombinators, and their well-formed rules.

In practice, we want to show that, if we have two printed texts A and B, parsing AB gets the same two results as parsing A and B separately. Or, more formally:[2]

```
p_has_result (print A_bd) A_object A ∧ p_has_result (print B_bd) B_object B
  ⟶ has_result (parse A_bd) (A@B) A_object C
    ∧ has_result (parse B_bd) C B_object []
```

4.2.1 Does Not Peek Past End

The most restricted way that a parser can fulfil this requirement is by not peeking past its end. That is, for any successfully parsed text, it must not matter how the leftover characters are changed, which is formalised with the following predicate:

```
does_not_peek_past_end p ⟷ (∀ c r l l'.
  has_result p (c@l) r l ⟶ has_result p (c@l') r l')
```

Intuitively this means that the parser only looks at the characters that are consumed and that any changes in the leftover are not seen.

How to use does not peek past end in a proof goal

So, if a printer appends two print results, and the first adheres to this predicate, then the well-formedness of both the parsers shows us that they can be parsed together as well. The formal proof goal, and the proof to show this are as follows:[3]

```
⟦p_has_result (print A) a ar; p_has_result (print B) b br⟧ ⟹
  has_result (parse A) (ar @ br) a br ∧ has_result (parse B) br b []
```

Now we can split this conjunction into two proofs. The first is as follows:

```
⟦p_has_result (print A) a ar; p_has_result (print B) b br⟧ ⟹
  has_result (parse A) (ar @ br) a br
```

Now we bring in the assumption that A does not peek past its end and the relevant part of well-formed:

```
⟦p_has_result (print A) a ar;
 p_has_result (print B) b br;
 has_result (parse A) ar a [] ⟹ has_result (parse A) (ar @ br) a br;
 p_has_result (print A) a ar ⟹ has_result (parse A) ar a []
⟧ ⟹ has_result (parse A) (ar @ br) a br
```

Now we can resolve the last assumption with the first, and then that can be resolved with the third assumption, which in turn solves the proof goal. The second subgoal created above is simpler to prove.

```
⟦p_has_result (print A) a ar; p_has_result (print B) b br⟧ ⟹
  has_result (parse B) (br) b []
```

This can be resolved with the assumption that `B` is well-formed.

[2]This is an example predicate, the actual proof goals fulfilled by the below predicates are varied but similar.

[3]Real proof; in file types.thy named `consecutive_parses_proof_for_thesis`

Examples

Examples of parsers that adhere to this predicate include parsers that only parse one character, or that parse some set amount of characters. Also included are parsers like `(this_char '{') then (many (this_char 'A')) then (this_char '}')` which internally do not adhere to the predicate, but as a whole do.[4]

4.2.2 Does Not Peek Past Character

Does not peek past character is a slightly less constrained variant of does not peek past end. Instead of requiring that the parser does not peek past what it itself has parsed, we allow it one character to 'stop' its parsing process. So, intuitively, a parser that adheres to this predicate returns the same result for the same text, regardless of changes in the leftover part, as long as the leftover part starts with the specific characters given. We define this formally as:

```
does_not_consume_past_char p ch ⟷ (∀c r l l'.
  has_result p (c@l) r l ⟶ (has_result p c r [] ∧
                              has_result p (c@(ch#l')) r (ch#l')))
```

We prove in Isabelle that if and only if this holds for every character the parser also adheres to `does_not_peek_past_end`.[5]

Examples of parsers that have a character they do not peek past include `many this_char 'A'`, which does not peek past any character but 'A', and `many (this_char 'A' then this_char 'B')` which also does not peek past any character that is not 'A', as well as the `nat` and `int` bidefinitions, which do not peek past any characters that are not digits. [6]

4.2.3 First Printed Character

Now that we have a way of saying that a parser does not peek past a character, it makes sense to want to formalise a way of gathering which characters could be the first character for a bidefinition. This is what the predicate `first_printed_chari :: 'a printer => 'a => char => bool` is for. It is true if and only if the given printer prints the given object successfully into a string that starts with the passed-in character.

We can use this predicate in proof states that look like the following:

```
⟦p_has_result (print A) a ar; p_has_result (print B) b br⟧ ⟹
  has_result (parse A) (ar@br) a br
```

Let us assume that we can show that A does not peek past some character `c`. We would like to prove that the first character of `br` is `c`. This is exactly what this predicate is made for. It is defined as follows:

```
first_printed_chari p i c ⟷ (∃t. p_has_result p i t ∧ t≠[] ∧ (hd t) = c)
```

Example: The first printed characters for `then`

What are the possible first characters of `A then B` when printing some object `(a, b)`? They must be the first characters of A when printing `a`, or, if that is empty, the first characters of B when printing `b`. So, we can formalise the `fpci` rule for `then` as follows:

[4]Proof; in file `example_small_examples.thy` named `example_does_not_peek_past_char`

[5]Proof; in file `types.thy` named `does_not_consume_past_any_char3_eq_not_peek_past_end`

[6]Proof; in file `example_small_examples.thy` named `many_two_chars_no_peek_past_anything_but_A` and `many _this _char _no _peek _past _any _other _char`, and in the files `derived _nat.thy` and `derived _int.thy`.

```
fpci (print (b_then A B)) (a, b) c ⟷ (
  if p_has_result (print A) a [] then
    (fpci (print B) b c)
  else
    (fpci (print A) a c ∧ (∃t. p_has_result (print B) b t)))
```

Note that in the case that `a` is not printed empty we need to be sure that `b` can actually be printed. Otherwise, the overall printer for `then` might fail, which means there is no overall print result to take the first character from. We see in this rule again the application of 'peeling away' one bicombinator to gain new proof goals, which can be attacked using the same technique again.

4.2.4 First Parsed Character

When showing how fpci works we showed a proof goal that included a premise which decided the value of the string by going through the printer. In the cases that the premise includes a clause that decides the value of the string by going through a parser, we have `fpc`. Which stands for First Parsed Character. This predicate holds when a parser can parse a given object starting from a given character:

```
fpc p t c ⟷ (∃cs l. has_result p (c#cs@l) t l)
```

So, if we have the following proof state:

```
⟦p_has_result (print A) a ar; has_result (parse B) br b []⟧ ⟹
  has_result (parse A) (ar@br) a br
```

We can try to find all possible characters `c` for which `fpc (parse B) b c` holds. If we have that, we can prove that `parse A` does not peek past those characters to satisfy this proof goal. But also, in short, to show that this result holds for whatever comes after `br`. No matter what other things we append to this input text, `parse A` will still parse the same result `a`.

Example: The first parsed characters for `or`

What are the possible first characters for `A or B` when parsing some object `i`? If the A parser succeeds it must be the first parsed characters for A, else if B succeeds it must be the first parsed characters for B. As such, the fpc rule for `or` is as follows:

```
fpc (parse (or A B)) i c ⟷ (
  case i of Inl i' ⇒ fpc (parse A) i' c
          | Inr i' ⇒ fpc (parse B) i' c)
```

Again you see that we peel away one bicombinator in this rule to split the proof goal into smaller subgoals that can be proven recursively using the same method.

4.2.5 Consecutive parsers that do eat into each other

Now that we have seen two ways of showing that parsers do not eat into the coming text, it makes sense to think; What if a parser does eat into the text of another bidefinition, but the other bidefinition's result is not impacted?

Commonly, parsers need to ignore any whitespace after or before parsing something. Imagine for example a parser for a common list literal syntax. It parses the '[' character to start the list and then needs to drop whitespace until the first element. Now imagine the parser has parsed the last element. We parse any leftover whitespace and then the closing ']'. Assuming that there is some way using the above methods to resolve the conflicts between those parsers we only need to resolve the case where the list is empty. The opening bracket parser parses '[' and whitespace, and the closing bracket parser parses whitespace and then ']'. There clearly is a conflict here. The first parser looks past its end, and it looks past whitespace characters, which can be the first characters that the closing parser parses. So, our existing predicates cannot solve this case.

What we need is a predicate that shows that the first parser might interfere with the second, but in such a way that the result does not change. The predicate takes the shape of:

```
⟦has_result (parse A) (ca @ cb @ l) a (cb @ l);
 has_result (parse B) (cb @ l) b l⟧ ⟹
  ∃l'. has_result (parse A) (ca @ cb @ l'') a l' ∧
       has_result (parse B) l' b l''
```

This predicate works similarly to does not peek past character, but states that it does not peek past the parsed text of another parser.

A simple parser that shows behaviour where this predicate is useful is:

```
  empty_list = b_then (char_ws CHR ''['') (ws_char CHR '']'')
```

Here the bidefinitions `char_ws` and `ws_char` parse one character and then whitespace, and whitespace and then one character respectively. The proof that this parser is well-formed and that it does not peek past its own end (even if it has an inner parser that does peek past its own end) are in file `example _small_examples.thy` named `empty_list_wf` and `empty_list_no_peek_past_end`.

4.3 Proof Automation

This section describes the various proof automation methods that we have created and applied. These methods run on the basis of simplification and introduction rules, which were both explained in Section 3.4. For each proof automation we will explain the rules involved, and what the idea is behind their applications.

4.3.1 Parsers Should not Change Characters They do not Consume

This property of a parser describes that for any successful parse action, the input can be split into two strings, the consumed part, and the leftover. Or, formally:

```
PNGI p ⟷ (∀ i r l. has_result p i r l ⟶ (∃ c. i = c @ l))
```

`PNGI` originally stood for Parser Never Grows Input. This meaning fell by the wayside, but the abbreviation stuck.
This rule system is built upon a few core rules for the basic bicombinators and bidefinitions.

```
PNGI (parse fail)
PNGI (parse one_char)
PNGI (parse (peek_result b))
PNGI (parse (return v))
PNGI (parse bi) ⟹ PNGI (parse (ftransform t t' bi))
⟦PNGI (parse ab); PNGI (parse cb);
 ∀ i r l. has_result (parse ab) i r l ⟶ PNGI (parse (a2bb r))⟧ ⟹
  PNGI (parse (if_then_else ab a2bb cb b2a))
```

Note that all the basic bidefinitions are PNGI, and the bicombinators are PNGI if their parameters are. From these rules, we can create rules for derived bidefinition and bicombinators by combining the rules for all the elements used in the definition of the derived elements.

The only place that this becomes somewhat hard is the derived bicombinator `many :: 'a bidef => 'a list bidef`. Its (significantly simplified) implementation is:

```
many a = b_then a (many a)
```

Now imagine that we want to prove that this bicombinator creates PNGI bidefinitions if we give it a PNGI bidefinition. We get the proof goal: `PNGI (b_then a (many a))`. We apply the `b_then` rule (`PNGI A; PNGI B ⟹ PNGI (b_then A B)`), and get the proof goal: `PNGI a; PNGI (many a)`. It quickly becomes clear that this is not a viable proof method for bicombinators that are infinitely recursive.

We need to use the fixpoint induction that the partial function implementation (explained in Section 3.3) provides. With this we can, instead of unfolding the definition infinitely often, use the induction principle that every meaningful usage of the function does not recurse infinitely deep. We prove that PNGI is admissible (see Section 3.3.1 that explains the induction rule in detail). Then we show that the bottom bidefinition is PNGI. This proof is trivial, as the bottom bidefinition never has a result. Then the induction step requires us to show that adding one 'layer' of the `many` definition preserves PNGI-ness. Since it, following the `b_then` rule, clearly does, we have now shown that:

```
  PNGI A ⟹ PNGI (many A)
```

Now that we have these basic rules, we can derive a new PNGI rule for each newly created bidefinition or bicombinator, and register it into the set of rules. Then, any time we need to prove PNGI of some bidefinition we apply these intro rules over and over again until we are done. The technique as described has one caveat. For example, sometimes the goal has the shape of:

```
PNGI (parse A) ⟹ PNGI (parse (b_then A (this_char ‘C’)))
```

Now when the automation applies the intro rule for `b_then` it gets the following two subgoals:

```
PNGI (parse A) ⟹ PNGI (parse A)
PNGI (parse A) ⟹ PNGI (parse (this_char ‘C’))
```

We have a rule in the set that shows that `this_char C` is PNGI, so the recursive solving setup can dispatch that subgoal. But, the first subgoal cannot be resolved because the knowledge is in the assumptions. Accordingly, in our recursive setup we must first check if we can prove the subgoal with an assumption, and only if that is not possible do we apply rules and recurse. In Isabelle, this can be written as follows:

```
method pngi_solver = (repeat_new ⟨ assumption | rule PNGI_intros ⟩)
```

This is written in the method builder syntax. `repeat_new` means that this method is tried on the goal, and that for every *new* subgoal, the method is applied recursively. The `assumption` method tries to solve the goal using only assumptions. The vertical bar syntax means to use the left method first, and then the right method only if the left method fails. The `rule` method takes a set of rules as a parameter and applies them.

We know that this method will always work on proof goals that only include the basic or derived bidefinitions and bicombinators in the library because those all have intro rules that show PNGI if the parameters are also PNGI.

4.3.2 Print Empty: Can This Printer Print Empty Text?

When trying to figure out what the first printed character of a printer can be it often matters if a printer can print empty. For example, if we run two printers consecutively and the first prints empty, then the first printed character comes from the second printer, not the first. For this we create a ruleset of printer has result rules (introduced in 3.4) that focus purely on the first character. This automation uses simp rules, a few examples are shown here:

```
p_has_result (print nat_b) n [] = False
p_has_result (print (many b)) [] [] = True
p_has_result (print (many b)) (i # is) [] = p_has_result (print b) i [] ∧
                                            p_has_result (print (many b)) is []
p_has_result (print (b_then A B)) (ia, ib) [] = p_has_result (print A) ia []
                                                ∧ p_has_result (print B) ib []
```

When a subgoal matches a rule like this, and the right-hand sides of these rules only create easily dispatched subgoals, or subgoals that match rules in this form, they can be unfolded automatically to a true or false result. Assuming that there are rules for all the bidefinitions and bicombinators involved. In the library, we indeed see that these rules work very well. However, some combinators can make this harder.

The `ftransform` bicombinator allows the user to provide a parse and print time function to transform the element which can also cause the parser and printer to fail. So, to make it print empty both the transformation function must succeed and the result of that must be printed empty. This second part is the simple case, but since the bicombinator allows any function to be used to transform the given element this part of the proof can be as difficult as the function needs.

Any bicombinator that needs a printer to succeed, but ignores the result, has a similar issue. For example, the `peek` bicombinator, which parses but does not consume, and uses the printer only to check if it succeeds or fails. The only possible print result of peek is the empty string, but it will only succeed in printing if the inner printer succeeds on the given parameter. The rule we create thus looks as follows:

```
p_has_result (print (peek b)) i [] ⟷ (∃t. p_has_result (print b) i t)
```

Note how the left-hand side only mentions printing to empty lists, but the right-hand side does not constrain itself to that. At this point, we have 'escaped' the subset of printer rules that only print to empty, and must use the more general printer simp set or other proof methods.

4.4 The Library

The following is a table of all bidefinitions and bicombinators, with information on the various predicates that they adhere to. The table is split into two, first the basic bidefinitions and bicombinators and then the derived elements. To save space the columns are strongly condensed. An explainer for each bidefinition and bicombinator is given in Section 4.1 which contains an enumeration of all elements and their well-formed rules, in detail. The WF column contains 'Y' if there is a rule, or it is proven well-formed, in the case of bidefinitions. A 'N' is given when there is no rule, or the rule is such that it excludes any useful bidefinitions. The DNPPE column describes, for the 'Does Not Peek Past End' rule: 'Y' if there is a rule for it, 'Parameter' if the rule is such that it purely defers to the parameters, and 'N' if there is no rule for it, which can either be because it cannot be proven, or because there is no rule. In the DNPPC column for the 'Does Not Peek Past Character' rule, the bidefinitions have a description of what characters they do not peek past, and bicombinators, same as before, state if the rule exists, and if it defers to the parameters. The FPCI column, for the 'First Printed Character' rule, describes for bidefinitions what their first printed character is, and bicombinators will note the number of rules defined.

Basic Elements	WF	DNPPE	DNPPC	FPCI
fail	Y	Y	Any	None
ftransform	Y	Parameter	Parameter	Parameter
one_char	Y	Y	Any	Parameter
peek	N	Parameter	Parameter	Does not print
return	Y	Y	Any	Does not print
if_then_else	Y	Parameter	N	Parameter
Derived Elements	WF	DNPPE	DNPPC	FPCI
or	Y	Parameter	Parameter	Parameter
transform	Y	Parameter	Parameter	Parameter
dep_then	Y	Y	N	Parameter
char_for_predicate	Y	Y	Any	Parameter
The above also holds for instantiations of this rule, as noted in 4.1.				
any_from_set	Y	Y	Any	Parameter
this_char	Y	Y	Any	Parameter
this_string	Y	Y	Any	Parameter
char_not_in_set	Y	Y	Any	Parameter
b_then	Y	Y	Y	Parameter
many	Y	N	Y	Y
many1	Y	N	N	Y
nat_b	Y	N	Not digit	Digit
int_b	Y	N	Not digit	- or digit
separated_by	Y	N	Y	Parameter
separated_by1	Y	N	N	Parameter
then_drop_first	Y	Y	N	Parameter
then_drop_second	Y	Y	N	Parameter
drop	Y	Parameter	Parameter	Oracle
char_ws	Y	N	Not whitespace	Parameter
ws_char_ws	Y	N	Not whitespace	Parameter
ws_char	Y	Y	Any	Parameter
eof	Y	Y	Any	Does not print
optional	Y	Parameter	Parameter	Parameter
peek_bool	N	Parameter	Parameter	Does not print
m_map	Y	Parameter	N	Parameter

4.5 Example: DIMACS

The DIMACS format is used by SAT solvers to read formulas to solve. [2] A DIMACS file contains a header and then a line for each clause. The header line starts with `p cnf` and then two numbers. First the number of variables, and then the number of clauses, separated by a space character. Each line representing a clause contains positive and negative instances of variables denoted by positive and negative numbers separated by whitespace, and ending in a 0. An example of a file like this is:

```
p cnf 12 4
5 6 7 −8 −9 10 11 12 0
2 3 4 6 0
1 5 −6 0
2 6 9 10 12 0
```

Following this specification we will create and verify a bidefinition using the above-described library. We will first create a bidefinition for the header, then one for a line, and then combine these to create one for the whole file.

4.5.1 The DIMACS File Header

We will first show the definition of the header, then explain it, and then describe the well-formed proof.

```
definition DIMACS_header :: "(nat × nat) bidef" where
  "DIMACS_header = then_drop_first
                     (this_string ''p cnf '')
                     (b_then
                       (nat_b)
                       (then_drop_first
                         (this_char CHR '' '')
                         (nat_b)
                         (CHR '' '')))
                     ''p cnf ''"
```

We use the bicombinator `then_drop_first` to parse two elements but only use the second in the result. It takes two bidefinitions as parameters and as a third parameter the value that the first should print. This helps us simplify the datatype that the bidefinition parses and prints. First, we parse the start of the header, which is always static text, and remove it from the datatype. After that we parse a natural number, then a space character is ignored, after which the last number is parsed.

Well-formed Proof

The following section does not aim to be a direct transcription of an Isabelle proof session. We aim to provide the main steps of the proof and show the intuition involved in the proof steps. Largely duplicate proof steps will only be noted shortly. The proof text itself will be, for context, provided at the end of this section.

We start with the following proof goal:

```
  bidef_well_formed DIMACS_header
```

Then, we unfold the definition of `DIMACS_header` and get:

```
bidef_well_formed (then_drop_first (this_string ''p cnf '')
                    (b_then nat_b (then_drop_first (this_char CHR '' '')
                      nat_b CHR '' '')) ''p cnf '')
```

As described in the section that introduced the well-formed rules (4.1) we apply a rule corresponding to the outermost combinator. In this case that is the `then_drop_first` combinator. It requires both parameters to be well-formed, and that the first does not eat into the second.

```
1. bidef_well_formed (this_string ''p cnf '')
2. bidef_well_formed (b_then nat_b
                        (then_drop_first (this_char CHR '' '')
                          nat_b CHR '' ''))
3. pa_does_not_eat_into_pb_nondep (this_string ''p cnf '')
                                     (b_then nat_b
                                       (then_drop_first (this_char CHR '' '')
                                         nat_b CHR '' ''))
4. ∃i. has_result (parse (this_string ''p cnf '')) i ''p cnf '' []
```

The bidefinition `this_string` is known to be well-formed. The collision between `this_string` and the rest of the bidefinition can be solved via the idea of a parser not peeking past its own end, as described in Section 4.2.1. Intuitively we say that the `this_string` parser does not peek past its end, so it does not matter what is printed after, it will never be consumed. The fourth subgoal fulfils the idea that the third parameter to the `then_drop_first` bicombinator, that is, the object it must print, is actually printable. It is trivially solved via the has result ruleset.

We focus on the second subgoal and apply the well-formed rule for `b_then`. Again we see that the two bidefinitions passed in must be well-formed and that they must not eat into each other.

```
1. bidef_well_formed nat_b
2. bidef_well_formed (then_drop_first (this_char CHR '' '') nat_b CHR '' '')
3. pa_does_not_eat_into_pb_nondep nat_b
                        (then_drop_first (this_char CHR '' '') nat_b CHR '' '')
```

The natural number bidefinition is known to be well-formed. The third subgoal inhabits a different case for the collision between parsers. We know that the natural number parser does peek past its end. Nonetheless, we know that it only does this for characters that are in the digits set. So, since the first character that is printed by the bidefinition coming after it is not a digit character, we know that there is no collision. (This mechanism is described in more detail in Section 4.2.2, which introduces the idea of not peeking past a character.) The proof that this first printed character is not a digit character is provided by the fpci concept introduced in Section 4.2.3. Since this is the first time we see it in this proof it will be shown in detail. When we apply the rule to use this technique, then see this subgoal:

```
first_printed_chari
  (print (then_drop_first (this_char CHR '' '') nat_b CHR '' '')) i c
    ⟶ does_not_consume_past_char3 (parse nat_b) c
```

This tells us that we need to use the fpci rules to see what the value of c is, and then prove that the natural number parser does not peek past it. The application of these rules, in addition to the print empty rules (shown in 4.3.2), tells us that the only character that can be the first in a print result of the given bidefinition is the space character. Since space is not a digit, and we know that the natural number parser only peeks past digits, we know that this collision is resolved.

Following this, for subgoal two, we apply the well-formed rule for `then_drop_first`. It asks us to show that the two parameters are well-formed, which `nat_b` and `this_char` are. Lastly, as above, it asks us to show that `this_char CHR '' ''` can print the space character, which it can.

There are two styles that this proof can be written with. The first is a more explorative method, where rules are applied one-by-one, and where the created subgoals are easier for the user to interpret. The second is a proof style more applicable to a knowledgeable user, rules are applied in one fell swoop, and created subgoals have no clear origin.

The previous proof, when written in the first style in Isabelle, looks as follows:

```
lemma header_wf_proof_for_thesis:
  "bidef_well_formed DIMACS_header"
  unfolding DIMACS_header_def
  apply (rule then_drop_first_well_formed)
  subgoal by (rule this_string_wf)
  subgoal
    apply (rule b_then_well_formed)
    subgoal by (rule nat_b_well_formed)
    subgoal
      apply (rule then_drop_first_well_formed)
      subgoal by (rule this_char_well_formed)
      subgoal by (rule nat_b_well_formed)
      subgoal
        unfolding pa_does_not_eat_into_pb_nondep_def
        apply (clarsimp simp add: fp_NER)
        by (clarsimp simp add: NER_simps)
      subgoal by (clarsimp simp add: NER_simps)
      done
    subgoal by (auto simp add: pa_does_not_eat_into_pb_nondep_def
                               NER_simps fp_NER takeWhile_tail)
    done
  subgoal
    apply (rule does_not_peek_past_end_implies_does_not_eat_into)
    subgoal by (rule this_string_does_not_peek_past_end)
    subgoal by (rule this_string_wf)
    done
  subgoal by (clarsimp simp add: NER_simps)
  done
```

This might seem like a lot of work to prove that this bidefinition is well-formed. Note how most rule applications are done by hand, and how the subgoal keyword is used very often to simplify the presented proof state, which makes the origin of new subgoals clearer. The above proof shows the process of a user applying rules one by one to get a good look at what proof goals come up, and how they want to solve those rules. That process is more exploratory than an experienced user would do. Once a user has gained more experience with which rules are available this proof can be done significantly simpler:

```
lemma header_WF:
  "bidef_well_formed DIMACS_header"
  unfolding DIMACS_header_def
  by (auto intro!: then_drop_first_well_formed b_then_well_formed
         simp add: this_string_wf nat_b_well_formed this_char_well_formed
                   pa_does_not_eat_into_pb_nondep_def fp_NER  NER_simps
                   takeWhile_tail)
```

This proof largely applies the same rules, but those rules are applied all at once. With no regard given for accidental applications, or more streamlined methods. Note how, for example, the two rules that are used to dispatch collision proofs (`does_not_eat_into`...) are disregarded, in favour of proving the predicate directly. This proof method requires more knowledge of what the various proof methods can do, but in turn, is more streamlined and faster to write. A disadvantage of this proof method is that sometimes a rule can be applied in a place where it is not intended to be applied. The solution for this is to extract the subgoal to a different lemma to prove, such that this rule application site does not appear when the wrong rule is available.

4.5.2 A Line in a DIMACS File

When the header has been parsed, the next step is to parse the clauses. Each clause is one line, so it makes sense to create a bidefinition for just one line. Let us remind ourselves: each line in the file is a clause of positive and negative numbers split by whitespace. So, the bidefinition looks like this:

```
definition DIMACS_line :: "int list bidef" where
  "DIMACS_line = separated_by1 int_b (this_char CHR '' '') (CHR '' '')"
```

Here, `separated_by1` takes a bidefinition, and then another bidefinition that should be in between each instance of the result bidefinition. The third parameter is the value that should be printed by the separator printer. So, this bidefinition will parse numbers separated by space characters.

Well-formed Proof

This proof description will follow the same general structure as the proof description above. It will shortly mention, but not explain again, proof methods and proofs that have already been shown in said proof description above.

After unfolding the definition we get the following well-formed proof goal:

```
bidef_well_formed separated_by1 int_b (this_char CHR '' '') (CHR '' '')
```

The `separated_by1` well-formed rule does not do a lot. We need to show that the given oracle can be printed by the separator. Since the oracle is a space character, it can be printed by the `this_char CHR '' ''` bidefinition. Secondly, it requires the element bidefinition to be well-formed, and `int_b` is. The last requirement gives a view into the definition of the bidefinition.

```
bidef_well_formed (b_then int_b (many (b_then (this_char CHR '' '') int_b)))
```

We see that the bidefinition is defined using the idea that parsing an element separated by a separator is the same as parsing one element, and then a separator and another element as long as it is possible to do so.

From the `b_then` well-formed rule we get that `int_b` and the other side of the bidefinition must be well-formed, as well as that the parser for `int_b` must not eat into the 'territory' of the rest of the bidefinition. The third requirement is shown by the fact that the integer parser does not peek past anything but digit characters and that the other side of the `b_then` bidefinition has as the first printed character, always a space character, which is not a digit. So, the collision between the two parsers is resolved.

Now what is left is the well-formedness of the second parameter to the `b_then` bicombinator.

```
bidef_well_formed (many (b_then (this_char CHR '' '') int_b))
```

The outermost combinator is the `many` combinator. There are a few rules to show that it is well-formed, of which one is a specialisation specifically made for when the bidefinition passed into many is an application of `b_then`. The idea of this rule is that the fact that each iteration of the parser does not interfere with the next is proven by showing that `this_char` does not eat into `int_b`, and vice versa. Which can sometimes be easier than showing that one iteration of `b_then (this_char CHR '' '') int_b` does not eat into itself. This rule gives us the following new subgoals:

```
1. is_error (parse int_b) []
2. is_error (parse (this_char CHR '' '')) []
3. ¬ PASI (parse (this_char CHR '' '')) ⟹ PASI (parse int_b)
4. ∀i c. first_printed_chari (print int_b) i c ⟶
     does_not_consume_past_char3 (parse (this_char CHR '' '')) c
5. ∀i c. first_printed_chari (print (b_then (this_char CHR '' '') int_b)) i c
   ⟶ does_not_consume_past_char3 (parse (b_then (this_char CHR '' '')
                                                                      int_b)) c
```

The first three subgoals are there to ensure that the 'separator, element' parse loop does not go on forever. That is, it eventually fails, and never succeeds without consuming something. The third subgoal uses the predicate PASI to show that second part, it stands for Parser Always Shrinks Input, the case being made is that either the separator, or the element parser must always shrink the input when succeeding. The fourth and fifth subgoals concern the idea that the element and the separator do not eat into each other. The fourth can be resolved by using the idea that `this_char CHR '' ''` does not peek past its end, and thus does not consume past any character. The fifth subgoal can be simplified by applying the fpci and print empty rulesets. From this appears the following goal:

```
does_not_consume_past_char3 (parse (b_then (this_char CHR '' '') int_b))
                                    CHR '' ''
```

Now we can resolve this proof goal by applying that `this_char` does not peek past its own end, and `int_b` does not consume past space characters.

4.5.3 The Header, and then as Many Lines as Possible

Now that we have the header and the line as well-formed bidefinitions we combine them to get a bidefinition for the whole file. For this, we need to apply the header bidefinition first, after that a newline character, and then we parse the line bidefinition as often as possible, separated by a newline character. The object parsed by, and printed by, this bidefinition includes both the result of the header bidefinition and all the lines. It excludes the newline characters separating it.

```
definition DIMACS :: "((nat × nat) × int list list) bidef" where
  "DIMACS = b_then (then_drop_second DIMACS_header
                                     (this_char CHR ''\n'') CHR ''\n'')
                   (separated_by (this_char CHR ''\n'')
                                 DIMACS_line CHR ''\n'')"
```

Well-formed Proof

We can apply the well-formed rules for `b_then` and `then_drop_second`. The subgoals that appear from this are very similar to the ones shown above. We need to show that the header is well-formed, which we have, and that the header does not eat into the newline character, which is easy to show as the header ends in the natural number bidefinition which does not eat into non-digit characters. The two interesting subgoals that appear are the ones that require that the combination of the header and the newline character bidefinition does not eat into the lines, and that the lines themselves are well-formed.

```
1. pa_does_not_eat_into_pb_nondep
       (then_drop_second DIMACS_header (this_char CHR ''\n'') CHR ''\n'')
       (separated_by (this_char CHR ''\n'') DIMACS_line CHR ''\n'')
2. bidef_well_formed (separated_by (this_char CHR ''\n'')
                                   DIMACS_line CHR ''\n'')
```

We prove the first subgoal via the fact that parsing the header and then the newline ends with parsing the newline, which does not peek past its end. Adding to that the point that the header does not eat into the newline character, and we have resolved the first subgoal.

For the second subgoal, we apply the proof rule for the well-formedness of `separated_by`. This gives us 11 total subgoals:

```
1. bidef_well_formed DIMACS_header
2. bidef_well_formed (this_char CHR ''\n'')
3. pa_does_not_eat_into_pb_nondep DIMACS_header (this_char CHR ''\n'')
4. bidef_well_formed DIMACS_line
5. bidef_well_formed (this_char CHR ''\n'')
6. is_error (parse DIMACS_line) []
7. good_separated_by_oracle (this_char CHR ''\n'') CHR ''\n''
8. ¬ PASI (parse (this_char CHR ''\n'')) ⟹ PASI (parse DIMACS_line)
9. ∀i c. first_printed_chari (print DIMACS_line) i c ⟹
     does_not_consume_past_char3 (parse (this_char CHR ''\n'')) c
10. ∀i c. first_printed_chari (print (this_char CHR ''\n'')) i c ⟹
     does_not_consume_past_char3 (parse DIMACS_line) c
11. ∀a b c. first_printed_chari
     (print (b_then (this_char CHR ''\n'') DIMACS_line)) (a, b) c ⟹
         does_not_consume_past_char3 (parse (b_then (this_char CHR ''\n'')
                                                    DIMACS_line)) c
```

The first five subgoals are easy to resolve via methods already seen above. The sixth subgoal is to ensure that, if we had printed zero lines, the parser does not parse one line. (Which would not be well-formed.) It is easily resolved with the has result ruleset. The seventh subgoal requires that the separator can print the given object. In this case, that means that the `this_char CHR ''\n''` can print `CHR ''\n''`, which can easily be resolved with the printer has result ruleset. The eighth subgoal appears for the same reason as shown in the well-formed proof for the line bidefinition. It requires that the 'element, separator' parse loop consumes at least one character, which ensures that it cannot 'spin in place'. The ninth subgoal requires that the line parser does not eat into the newline. This proof is resolved the same way as above, breaking up the parser into smaller pieces and showing that the last does not eat into a newline, that the second to last does not eat into the last, and so on. The tenth subgoal requires that the newline parser does not eat into the line bidefinition, which is trivially proven by the fact that the newline parser only parses one character, so never eats into anything. The last subgoal shows that the 'newline, line' parser does not consume past the first printed character of the 'newline, line' printer. This first character is the newline character. The proof for this is similar to the ones shown above.

Conclusion

The above section has shown both the design and the well-formed proof for a DIMACS bidefinition. It shows that the 'parser combinator' style function application-based building of bidefinitions makes it easy to create bidefinitions in parts, and combine those parts into larger wholes. It also shows one of the drawbacks of this approach, that it can become hard to understand the nesting of the various applications of composition functions like `then`. A possible solution for this, in the shape of a monad syntax, is described in the future work section below (5.1.1.) Another facet of the library that has been displayed is the proof methods for well-formed proving. We've shown how the combinators make it possible to break up proofs into smaller, easier, sub-proofs. With that, we've shown various proof methods introduced earlier (4.2, 4.3) in use in a practical example.

What this means, is that we now have an object that can provide a parser and a printer.

For these two functions it holds that if we have a DIMACS object, we print it, and then we parse it again, this is guaranteed to result in the exact same DIMACS object. Similarly, any object that is parsed from some input can be printed, with the same guarantees for the re-parsing. The last guarantee given is that any text in the input string after the DIMACS data is not changed. All three of these are desirable guarantees for SAT solvers, and any tools that use SAT solvers like proof assistants, including Isabelle, and tools like planning software for classrooms or hotels.

Chapter 5

Conclusion

In this thesis, we laid out our definition of the correctness of bidefinitions, described methods used to prove this, a library that encapsulates these methods, and presented an example bidefinition for the DIMACS file format that is proven correct using the methods presented.

The main research goal (1.5) for this thesis is: "*define an easy-to-use library for verified parser- and printer-combinators.*" The well-formed rule enumeration (4.1), the example bidefinition for the DIMACS format (4.5), and the description of the various proof methods (4.2), show that this has been completed up to a point. In the future work section below, we describe various methods through which both the definition of bidefinitions (5.1.1) and the well-formedness proofs (5.1.3) can be simplified and improved.

A few subgoals have been defined to guide the process. First and second, "*Defining a notion of "well-formed" for a parser and printer pair.*" and "*Formalising this well-formedness in Isabelle.*" We show this in Section 3.5, which describes well-formedness. Our method is different from others that try to gain a full and exact un-parsing of the source string. This has advantages and disadvantages. On the one hand, this means that users of the library do not have to carry around information about the source string needed to un-parse the object. On the other hand, this means that the library does not automatically try to recreate whitespace or other 'separator' text, which can be a downside for pretty-printers and formatters. This notion of well-formedness works well, and gives satisfying guarantees, but there are improvements possible. In the future work section (5.1.2) we describe how the adherence to structural equality for the types could be improved upon in the future.

The third subgoal, "*Creating a library of parser and printer bidefinitions and bicombinators.*" is described mainly in Section 4.4 which enumerates all bidefinitions and bicombinators and describes the rules and predicates that are proven for them. In addition to that, Section 4.1 shows how we fulfilled the fourth subgoal: "*Creating lemmas to prove the well-formedness of these parsers and printers.*"

Then, the use of these lemmas is exemplified in sections 4.5. This fulfils the last subgoal: "*Show the usability of the library by creating a bidefinition for an example file format and proving that it is well-formed.*"

In Section 2.1 we described two existing bidefinition techniques, those being Invertible Syntax Descriptions [13], and BiYacc [19]. We have shown our library to be better than the existing options in various ways. Firstly, Invertible Syntax Descriptions makes no claims about the compatibility of generated parsers and printers. BiYacc does, but their correctness requires that parse and print are each other's inverse, which we show in Section 3.5 is not ideal for our use case. Secondly, the way bidefinitions are created. Invertible Syntax Descriptions have a function application system very similar to ours, BiYacc has three inputs that have to be created and kept synchronised through changes over time.

5.1 Future Work

In this section, we lay out three different categories of improvements to the bicombinator system explained above. The first is about improving the creation process of bidefinitions, the second is about improving the bidefinitions that are made with the combinators, and the last is about improving the process of proving bidefinitions well formed.

5.1.1 Improving the Creation of Bidefinitions

Relevant to the idea of a better creation method for these bidefinitions are elements like `transform`, and `dep_then`. Both of these elements have parse-time and print-time functions which are supposed to reverse each other. Well-formed proofs for these elements require proving that these functions reverse each other correctly. Various methods to ensure reversibility exist, one is included in the parse and un-parse library BiYacc [19] mentioned above, which uses specific methods to create functions that can always be inverted. Other works describing invertible computing include Matsuda K. and Wang Mm, which introduce Sparcl, a programming language for partially invertible computation [8] and Mu S. et al. which introduced a combinator calculus with an implementation that has relational semantics in such a way that it can express only invertible computations [10]. Should a method of invertible computation be included in the way combinators are used this could further unify the parsing and printing sides of the program, and thus simplify the creation of bidefinitions.

The programming of bidefinitions is done via the ‘parser combinator’ style function application. This can be quite confusing when many composition operators are used. An example of this is the example DIMACS bidefinition given in Section 4.5. A different manner of presenting parsers or printers is the more declarative ‘parser monad’. A parser monad, especially in the do notation syntax, is a very natural way of representing parsers, our bidefinition type has a `return` function, and the `dep_then` combinator is similar to `bind`, if the print-time conversion parameter is ignored. This might eliminate most explicit uses of composition combinators like `then`, and `dep_then`. Creating Isabelle syntax rules such that do notation syntax can be used to define bidefinitions might make the library significantly easier to use.

Many parsers include some level of whitespace permissiveness. In the implementation of our library, this comes up in `ws_char`, `ws_char_ws`, and `char_ws`. These kinds of special cases work well, but only in the case that the parser needs exactly one character with whitespace around it. In cases where the whitespace surrounds a larger parser requires more work. The library could be expanded with even more special cases, but looking into more generic methods of skipping whitespace might result in more reusable results.

5.1.2 Improving the Resultant Bidefinitions

The bidefinitions in the library are programmed in a simple and straightforward manner. Their definitions are not optimized and might be quite slow. Future work could evaluate the efficiency of the created parsers, possibly taking into account the various languages that code can be exported to. The results of this evaluation could be used to optimize the library by verifying the equality of parsers (and combinators) with more efficient implementations. For example, one could implement `many (this_char c)` in the form of a `(take/drop) While (=c)` call, which might be faster to execute.

Users of parsers, for example for programming languages and user-facing interfaces, are interested in explaining to users why a parser fails to parse their input. Think for example of messages like ‘parse error on line 17 column 12, found IF where IDENTIFIER was expected.’ In our implementation of the fundamental types of the parser and printer, there is no room for this information. The error case is simply a ‘None’. To make this library more user friendly this functionality could be created. For example, by replacing the existing optional type in the parse result with an either, which could carry around error information.

None of the bicombinators in the library are specific to a character being the token. Only the bidefinitions themselves are specific to the token type. This implies that there might be an abstraction level above the existing bidefinition type which allows parametrisation over the token type. An application of this would allow pre-tokenisation of the input text. As well as the direct parsing of binary files that include non-character bytes.

Our definition of well-formed requires that an object, when parsed from some text T, is the same as the object that was printed into text T. For some domains, this can turn out to not be the right choice. Consider, for example, a compiler. It might parse the text `if a then b else skip` into the AST `IF(a,b,skip)`. The printer may realise that the language supports leaving out the else clause, and prints `if a then b`. This text is parsed into the AST `IF(a,b)`. These two ASTs are not strictly the same. But they can be considered semantically the same. Future research might seek to replace our equality requirement with some semantic equality function (possibly provided by the user of the library.)

5.1.3 Improving the Proving Process

The proof automation Section 4.3 mentions only one method that directly solves proof goals. (`PNGI`) The other proof automations are actually sets of lemmas that together solve most of the goals that can come up in their shapes. Though this is effective, it leaves much to be desired, could other predicates’ proof attempts be automated? To mind comes the section about consecutive parsers, (4.2) which builds up various methods that parsers can work consecutively. It makes sense to imagine that a proof search could at least solve the low-hanging-fruit proof goals in this territory. That is, the proof goals where most if not all sub-bidefinitions adhere to `does_not_peek_past_end` or `does_not_consume_past_char`. This would be valuable to users of the library as it frees up effort from the mechanic proof steps in these proofs for more involved proofs in other steps.

Section 3.3 explains the partial function setup. It is the basis of the `many` combinator, which is used in many places in the library. To resolve the problem with the induction rule from the partial induction setup we have created a custom induction rule for the many combinator, which uses the idea that if a parser always shrinks the input, it will never get stuck in an infinite loop. Generalising this rule to arbitrary bicombinators and bidefinitions defined via the partial function setup would improve the usability of the library when creating bidefinitions and bicombinators with partial functions. Other possible solutions include the idea of partial correctness, as introduced in Section 3.4.3.

5.2 Acknowledgements

Finally, but most importantly, I must extend my deepest appreciation to the University of Twente and its staff, who have excited and supported me from the moment I had my first classes on campus. Especially Peter Lammich, who supervised my thesis, and Marieke Huisman, who introduced me to Program Verification in a most unforgettable lecture series.

Secondly, my parents, other family members, and friends, who supported me through the process, and listened to me sage about things they did not understand in the slightest for far too long. This thesis is the crowning achievement of twenty-one years of schooling and has been a true joy to work on.

Bibliography

[1] Edmund Clarke, Daniel Kroening, and Flavio Lerda. *A Tool for Checking ANSI-C Programs*, pages 168–176. Springer Berlin Heidelberg, 2004. URL: http://dx.doi.org/10.1007/978-3-540-24730-2_15, doi:10.1007/978-3-540-24730-2_15.

[2] SAT Competition. Sat competition 2009: Benchmark submisssion guidelines. https://web.archive.org/web/20190325181937/https://www.satcompetition.org/2009/format-benchmarks2009.html, 2009. online; accessed 08-10-2024; archived 25-05-2019.

[3] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/964001.964011.

[4] Xiaodong Jia, Ashish Kumar, and Gang Tan. A derivative-based parser generator for visibly pushdown grammars. *ACM Trans. Program. Lang. Syst.*, 45(2), May 2023. doi:10.1145/3591472.

[5] Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. *Logical Methods in Computer Science*, Volume 7, Issue 2, June 2011. doi:10.2168/lmcs-7(2:18)2011.

[6] Peter Lammich. *Fast and Verified UNSAT Certificate Checking*, pages 439–457. Springer Nature Switzerland, 2024. URL: http://dx.doi.org/10.1007/978-3-031-63498-7_26, doi:10.1007/978-3-031-63498-7_26.

[7] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. Costar: a verified all(*) parser. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 420–434, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454053.

[8] KAZUTAKA MATSUDA and MENG WANG. Sparcl: A language for partially invertible computation. *Journal of Functional Programming*, 34:e2, 2024. doi:10.1017/S0956796823000126.

[9] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction - CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, pages 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-79876-5_37.

[10] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In Dexter Kozen, editor, *Mathematics of Program Construction*, pages 289–313, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Tobias Nipkow. Programming and proving in isabelle/hol. In *Technical report, University of Cambridge*. University of Cambridge, 2013.

[12] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[13] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. *SIGPLAN Not.*, 45(11):1–12, 9 2010. doi:10.1145/2088456.1863525.

[14] rmT116609. JDK-5045582: (coll) binarySearch() fails for size larger than 1 << 30, May 2004. URL: https://bugs.java.com/bugdatabase/view_bug?bug_id=5045582.

[15] The Coq Development Team. The coq proof assistant, 2024. URL: https://zenodo.org/doi/10.5281/zenodo.1003420, doi:10.5281/ZENODO.1003420.

[16] Sarah Tilscher and Simon Wimmer. Ll(1) parser generator. *Archive of Formal Proofs*, May 2024. https://isa-afp.org/entries/LL1_Parser.html, Formal proof development.

[17] Dmitriy Traytel. https://lists.cam.ac.uk/sympa/msg/cl-isabelle-users/2024-05/8-OGGcK2RjrZ5G62hJoANw, 2024. online; accessed 08-10-2024; archived 10-05-2024.

[18] Freek Wiedijk. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285:85–100, 2012. Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10). URL: https://www.sciencedirect.com/science/article/pii/S157106611200028X, doi:10.1016/j.entcs.2012.06.008.

[19] Zirun Zhu, Hsiang-Shang Ko, Pedro Miguel Ribeiro Martins, João Alexandre Saraiva, and Zhenjiang Hu. Biyacc: Roll your parser and reflective printer into one. In *CEUR Workshop Proceedings*. CEUR-Ws, 7 2015. URL: https://hdl.handle.net/1822/40556.