Program Extraction Applied to Monadic Parsing

Ulrich Berger, Alison Jones, and Monika Seisenberger

Computer Science, Swansea University, Swansea SA2 8PP, Wales, UK
{u.berger,481549,m.seisenberger}@swansea.ac.uk
Phone: +441792{602131,513380}

Abstract

This paper outlines a proof-theoretic approach to developing correct and terminating monadic parsers[1]. Using modified realizability, we extract formally verified and terminating programs from formal proofs. By extracting both primitive parsers and parser combinators, it is ensured that all complex parsers built from these are also correct, complete and terminating for any input. We demonstrate the viability of our approach by means of two case studies: we extract (1) a small arithmetic calculator and (2) a non-deterministic natural language parser. The work is being carried out in the interactive proof system Minlog.

Keywords: program extraction, verification, monads, parsing algorithms, natural language processing, termination, left recursion.

1 Introduction

Parsing plays a significant role in many different contexts, each of which can present its own unique problems. Despite differences, however, there are three key properties that all parsers should ideally have. Firstly, they should be *correct*, i.e., they should only parse input strings that are valid in accordance with the grammar and output only valid results. Secondly, they should be *complete*, i.e., they should recognize and parse *all* such input strings and, if a grammar is ambiguous, they should parse these valid strings in all possible ways. Thirdly, parsers should *terminate* with any input. The importance of termination is demonstrated by the well-known left-recursion problem [11].

In this paper we focus on monadic parsers, which allow us to generate parse trees as "side-effects" in pure functional programming languages such as Haskell. Another advantage is that monadic parser combinators naturally accommodate non-determinism and ambiguous input by returning one or more parse results in a list. Monadic parsers can be directly implemented in Haskell. This is the approach taken in much of the literature (see [13], [27], [28]). Testing these parsers could check (to a degree) that they work as expected. A more rigorous approach would be to verify them by formalizing the implementations and proving that they work as expected. A third option, and

[1] An earlier version of this work has been presented at Natural Language and Computer Science, NLCS'14 [3], and at Proof, Structure and Computation, PSC 2014. The research was partly funded by the EU project Correctness by Construction, CORCON, 2014-2017, IRSES 612638.

the one followed and promoted here, is to extract sound parsers directly from formal proofs. Starting from a set-theoretic specification of basic monadic parsers and parser combinators, we prove that certain termination-preserving properties hold not only for the basic parsers, but also when they are combined. We extract monadic parsing algorithms from these proofs – using a modified realizability interpretation (see Section 5.2) – in the interactive proof system Minlog [19]. An important point is that in the mathematical development we can work in a representation-free way with abstract mathematical objects. For example, in the Minlog formalization we are free to identify sets with boolean functions, as is common in classical mathematics. The long-term goal is to apply this method to parsing natural languages specifically. An overall first case study in this direction is carried out in section 8 where we extract parser programs from (left recursive) natural language grammars.

2 Related work

In the following we discuss some related work concerned with the development of formally verified monadic parsers, and comment on its relation to our own. We then outline some work on left-recursion and natural language parsing which relates to Section 8.

Since Moggi's seminal paper [20], monads have become a standard method for capturing side-effects when using pure functional programming languages like Haskell (see [27], [13]). Schröder and Mossakowski have used HasCasl (a higher-order specification and programming language based on Casl) to formally verify arbitrary monadic programs. To guarantee partial correctness, a Hoare calculus was used [23], whilst total correctness and termination were proven using dynamic logic [22].

Danielsson [9] defines a monadic parser combinator library that has been formally verified in Agda. Because Agda is a total language, parsers defined using this library are automatically guaranteed to terminate with any finite input string, without manual verification. What distinguishes Danielsson's work from earlier work on termination is that he allows certain forms of left-recursion. Specifically, he uses a mixture of induction (for the choice combinator) and guarded coinduction (for the bind combinator). Like others cited by Danielsson (e.g., [25]), Danielsson uses the property of input strings getting smaller (i.e., parsers not accepting empty string) when he defines total parser combinators. He refers to parsers with this property as "nullable". In our paper, we call them "hungry". Like Danielsson, we can also guarantee the correctness and termination of parsers written in Minlog without a posteriori formal verification. What differentiates our work from [9] is that we *extract* complete and terminating monadic parsers directly from proofs.

Koprowski and Binsztok [15] extract recursive descent parsers in Coq. Their parsers are generated automatically from Parsing Expression Grammars (PEGs), which are extended with type information for meaningful parsing (to form XPEGs). In contrast to some other grammars (e.g., Context Free Grammars), PEGs are unambiguous. This

is because all parsers generated from them are greedy – i.e., consume as much input as possible – and use prioritised choice – i.e., if A succeeds, don't try B. This makes them unsuitable for some natural language parsing tasks, because natural language is inherently ambiguous. To guarantee the total completeness of a PEG, the grammars are written without left-recursion (direct or indirect). In contrast to [15], we allow non-deterministic parsers and extract monadic operators that can be applied to *various* grammars.

In relation to existing research on left-recursive monadic parsing for natural language, the following two implementations are of interest. Frost et al [10] use counters to constrain the number of times a left-recursive rule calls itself. They also use a technique called *memoization*, in which a table is used to keep track of which parsers have already been applied at each index of the segmented input string. This is done to prevent rework, which increases the efficiency of the algorithm. They implement their parsers directly in Haskell.

In [17], Lickman formally verifies another technique for implementing monadic combinator parsers that can handle left-recursion in arbitrary context-free grammars. This research is based on an unpublished idea of Wadler's. Lickman defines a least fixed point function `fixP` in Haskell, and proves that `(fixP h)` is productive for any continuous function `h:Parser a -> Parser a`. By applying the `fixP` function to the right-hand side of left-recursive grammar rules, the resulting parsers terminate. Instead of carrying out formal verification, we will be extracting a natural language parser program directly from proofs.

3 The parsing problem

Throughout this paper, we differentiate between three types of "parsers". At the logical level, we define parsers as *parser relations* or *parser specifications* (which we call parsers for short). Intuitively, a parser relation P of type A consists of all pairs $(s, a) \in (\mathrm{Str}, A)$ such that a complete parse of s with P yields the parse result a. Secondly, we assign to a parser relation P a *parsing function* $[P]$ that assigns to each string the set of partial parsing results with unconsumed rest strings (see Section 3.1). Thirdly, a *parser program* is the executable, proof-extracted program for a specified parser relation/parsing function (see Section 3.2).

We use sets to formalize parser relations and parsing functions. Let $\mathbb{P}(X)$ denote the classical powerset of X (i.e. the set of all subsets of X including $\emptyset$ and X). Let $\mathbb{P}_{<\omega}(X) \subseteq \mathbb{P}(X)$ denote its finite subsets. For the purposes of program extraction, it is most convenient to define $\mathbb{P}_{<\omega}(X)$ as an inductively defined subset of $\mathbb{P}(X)$:

(F1) $\emptyset \in \mathbb{P}_{<\omega}(X)$

(F2) If $S \in \mathbb{P}_{<\omega}(X)$ and $x \in X$,
then $\{x\} \cup S \in \mathbb{P}_{<\omega}(X)$

The set of all parsers of type A is defined as

$$\mathbb{P}(\mathrm{Str} \times A)$$

where Str is the set of strings, and strings are considered as lists of characters (Char^*). Adopting Haskell-style syntax, we use [] to denote the empty list and $c : cs$ to denote a non-empty list with head c and tail cs. We write $[x_0, x_1, \ldots, x_n]$ for $x_0 : (x_1 : \ldots : (x_n : []) \ldots)$. The concatenation of two lists s and t is denoted by $s +\!\!+ t$. The length of a list s is written $|s|$.

Later on, we will make use of the standard symbol for the monadic bind ($>\!\!>\!\!=$) and the Haskell do-notation (as explained in [12]).

3.1 Formalizing the Parsing Problem

For any parser $P \subseteq \mathrm{Str} \times A$ the *parsing problem* can be stated as follows: given a string $s \in \mathrm{Str}$, find all a such that $(s, a) \in P$.

In the *partial parsing problem*, we only require an initial segment of s to be parsed: given a string $s \in \mathrm{Str}$, find all pairs (a, r) such that s is equal to $s' +\!\!+ r$ and $(s', a) \in P$ for some string s'. This partial parsing problem can be represented by a parsing function $[P] : \mathrm{Str} \to \mathbb{P}(A \times \mathrm{Str})$, defined by

$$[P](s) := \{(a, r) | \exists s'.s = s' +\!\!+ r \wedge (s', a) \in P\} \tag{1}$$

i.e., the input string s is partially parsed into a result a and the remaining unparsed string r.

Usually one is interested in parsers with parsing functions that always yield a finite set of results. We call such parsers *finitely branching*:

$$\mathrm{FB}_A(P) :\equiv \forall s \in \mathrm{Str}\, [P](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str}) \tag{2}$$

3.2 Extracting Parsing Programs

The reason why we are interested in finitely branching parsers is that, from a constructive proof that a parser P is finitely branching, we can extract a program `p` that implements the parsing function $[P]$. Using Haskell notation, `p` will have the type:

```
p :: String -> [(a, String)]
```

where `a` is the (unspecified) type of representations of parsing results for P and the square-bracketed type `[(a,String)]` is the type of finite lists of pairs. Note that the type of `p` is the type of monadic parsers in [13]. Due to the way in which finite sets have been inductively defined, the set of results is represented by a finite list of results at the program level. A more detailed explanation of this process of representation – which is technically called realizability – will be given in Section 5. Essentially, we extract monadic parsers in the form of [13], with the additional benefit of providing a formal proof of correctness, completeness and termination. Instead of building complex parsers from simple ones at the *program* level, we do so at the *proof* level.

4 Monadic parsers and their properties

In this section we introduce some basic parsers and parser combinators at the set-theoretic level, and prove that they are finitely branching. We end the chapter by establishing that parsers naturally carry the structure of a monad.

4.1 Basic parsers

We define the return operator, which maps $x : A$ to the parser that consumes no input and returns x, as well as two basic monadic parsers: the failure parser, which fails with any input, and the item parser, which parses one "unit" of input.

$\text{return} : A \to \mathbb{P}(\text{Str} \times A)$
$\text{return}\, x = \{([], x)\}$

$\text{fail} : \mathbb{P}(\text{Str} \times A)$
$\text{fail} = \emptyset$

$\text{item} : \mathbb{P}(\text{Str} \times \text{Char})$
$\text{item} = \{([c], c) | c \in \text{Char}\}$

The corresponding parsing functions can be easily calculated:

$[\text{return}] : A \to \text{Str} \to \mathbb{P}(A \times \text{Str})$
$[\text{return}](x)(s) = \{(x, s)\}$

$[\text{fail}] : \text{Str} \to \mathbb{P}(A \times \text{Str})$
$[\text{fail}](s) = \emptyset$

$[\text{item}] : \text{Str} \to \mathbb{P}(\text{Char} \times \text{Str})$
$[\text{item}]([]) = \emptyset$
$[\text{item}](c : cs) = \{(c, cs)\}$

We now turn our attention to the fundamental monadic combinators, which enable non-deterministic and sequential parsers to be built from simple ones in a modular fashion.

4.2 Union

The union of two parsers P and Q with the same type corresponds to the choice between two parsers:

$$P \cup Q = \{(s, a) | (s, a) \in P \vee (s, a) \in Q\}$$

The corresponding parsing function is the point-wise union of the parsing functions P and Q:

$$[P \cup Q](s) = [P](s) \cup [Q](s)$$

4.3 Composition

The composition or sequencing of two parsers can be defined as follows. Suppose that parser P is of type A, and that Q_a is a family of parsers of type B, parametrized over a value of type A. Their composition results in a parser of type B, called the monadic bind of P and Q.

$$P >\!\!>\!\!= Q = \{(s +\!\!+ r, b) | \exists a \,.\, (s, a) \in P \wedge (r, b) \in Q_a\}$$

The corresponding parsing function is:

$$[P >\!\!>\!\!= Q](s) = \{(b, t) | \exists a, r.(a, r) \in [P](s) \wedge (b, t) \in [Q_a](r)\}$$

4.4 The basic parsers and parser combinators are finitely branching

In this section, we prove that the basic parsers and parser combinators are finitely branching. Due to the formalization of parsers as sets, the finitely branching property can be proven very simply using some of the basic properties of finite sets. These properties are given in Lemma 1. The proofs in this section are given in such detail as to allow a direct formalization in the Minlog proof system. The source files can be found in [2].

Lemma 1.

(a) The union of two finite sets S and T is finite.

Proof. By induction on the finiteness of S. *Base Case:* T is the empty set. $S \cup \emptyset \equiv S$. S is finite, therefore so is $S \cup T$. *Step Case:* Assume $S \cup T$ is finite. Show that $(\{x\} \cup S) \cup T$ is finite. Since union is associative, this can be rewritten as $\{x\} \cup (S \cup T)$. By the induction hypothesis, $S \cup T$ is finite. By **F2**, the union of this set and $\{x\}$ is also finite. □

(b) The union of a finite family of finite sets is finite. More precisely, if S is a finite set and T_x is a finite set for each $x \in S$, the union of these sets – denoted $\bigcup_{x \in S} T_x$ – is also finite.

Proof. By induction on S. *Base Case:* S is the empty set. It follows that $\bigcup_{x \in \emptyset} T_x \equiv \emptyset$, which is finite by **F1**. *Step Case:* Assume that $\bigcup_{x \in S} T_x$ is finite. Show that $\bigcup_{x \in (\{x'\} \cup S)} T_x$ is finite. This set is equivalent to $T_{x'} \cup (\bigcup_{x \in S} T_x)$. By Lemma 1 (a),

this union set is finite if its two subsets are. $\bigcup_{x \in S} T_x$ is finite by the induction hypothesis. The finiteness of $T_{x'}$ follows by assumption. □

(c) Singleton sets are finite.

Proof. This follows directly from the inductive definition of finite sets. By **F1**, we know that the empty set is finite. A singleton set $\{x\}$ is equivalent to $\{x\} \cup \emptyset$. By **F2**, this is finite. □

We are now able to show that the parsers and parser combinators we defined earlier are finitely branching.

Lemma 2. *The basic parser relations are finitely branching.*

(a) $\forall x \in A\ \mathrm{FB}_A(\mathrm{return}\, x)$.

Proof. Using (2), this can be rewritten as:

$$\forall x, s\ [\mathrm{return}\, x](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str})$$

For any x and s, this is the singleton set $\{(x, s)\}$, which is finite by Lemma 1 (c). □

(b) $\mathrm{FB}_A(\mathrm{fail})$.

Proof. $[\mathrm{fail}](s) = \emptyset$ which is finite. □

(c) $\mathrm{FB}_{\mathrm{Char}}(\mathrm{item})$.

Proof. Using (2), this is equivalent to:

$$\forall s\, [\mathrm{item}](s) \in \mathbb{P}_{<\omega}(\mathrm{Char} \times \mathrm{Str})$$

Proof by cases on the input string s. *Case 1:* s is the empty string. The resulting set for this parsing function is the empty set, which is finite by **F1**. *Case 2:* s is a composed string $c : cs$, which results in the singleton set $\{(c, cs)\}$. This is finite by Lemma 1 (c). □

Lemma 3. *Parsers built using union (the monadic choice combinator) and composition (the monadic bind combinator) are finitely branching.*

(a) $\mathrm{FB}_A(Q) \to \mathrm{FB}_A(P) \to \mathrm{FB}_A(P \cup Q)$

Proof. Using (2), this can be rewritten as:

$$\forall s([Q](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str}) \wedge [P](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str})) \to \forall s\, [P \cup Q](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str})$$

Using the definition of $[P \cup Q]$ in Section 4.2, this becomes an instantiated version of Lemma 1 (a), where S is the set $[P](s)$ and T is the set $[Q](s)$. □

(b) $\mathrm{FB}_A(P) \to \forall a \in A\, \mathrm{FB}_B(Q_a) \to \mathrm{FB}_B(P >\!\!>\!\!= Q)$

Proof. As before, we can use (2) to rewrite the proof goal as:

$$\forall s\, [P](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str}) \to \forall a, s\, [Q_a](s) \in \mathbb{P}_{<\omega}(B \times \mathrm{Str}) \to \forall s\, [P >\!\!>\!\!= Q](s) \in \mathbb{P}_{<\omega}(B \times \mathrm{Str})$$

We then prove the goal as follows. Assume $\forall s\, [P](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str})$ and $\forall a, s\, [Q_a](s) \in \mathbb{P}_{<\omega}(B \times \mathrm{Str})$. Instantiate Lemma 1 (b) with $[P](s)$ and $\lambda(a, r)[Q_a](r)$. This reduces the proof goal to showing that the set of parsing results $[P](s)$ is finite, and that for any $(a, r) \in [P](s)$ the set $[Q_a](r)$ is finite, which both follow by assumption. □

For the inductive proofs in Sections 7 and 8, we will use a bounded version of Lemma 3 (b) that requires the finiteness property of Q_a only for strings s' with $|s'| + k \leq |s|$ provided P is *k-hungry* in the sense defined below.

A parser relation $P \in \mathbb{P}(\mathrm{Str} \times A)$ is *k-hungry* if for all $(t, a) \in P$ the length of t is at least k. A family of parser relations $Q : A \to \mathbb{P}(\mathrm{Str} \times B)$ is called k-hungry if Q_a is k-hungry for all $a \in A$. We call a parser P k-hungry at s ($\mathrm{Hungry}_A(P, k, s)$) if the property above holds for all $(t, a) \in P$ such that t is an initial segment of s. Equivalently, $\mathrm{Hungry}_A(P, k) := \forall s\, \mathrm{Hungry}_A(P, k, s)$ where

$$\mathrm{Hungry}_A(P, k, s) := \forall a, r\,.\,(a, r) \in [P](s) \to |r| + k \leq |s| \tag{3}$$

Lemma 4. *If $[P](s)$ is finite, P is k-hungry at s and $[Q_a](s')$ is finite for all a, s' with $|s'| + k \leq |s|$, then $[P >\!\!>\!\!= Q](s)$ is finite.*

Proof. As in Lemma 3, this is an instance of 1 (b). □

Lemma 5. *(a)* $\forall P, s\, \mathrm{Hungry}_A(P, 0, s)$

Proof. Assume P,s. Using (3), this can be rewritten as:

$$\forall a, r.(a, r) \in [P](s) \to |r| \leq |s|$$

By (1), we know that the set $[P](s)$ consists of all (a, r) such that

$$\exists s'.s = s' +\!\!+ r \wedge (s', a) \in P$$

It follows that $|r| \leq |s|$ □

(b) $\forall P, s, k, l.\mathrm{Hungry}_A(P, k, s) \wedge l \leq k \to \mathrm{Hungry}_A(P, l, s)$

Proof. Immediate, by monotonicity of addition. □

(c) $\forall P, Q, s, k.\mathrm{Hungry}_A(P, k, s) \wedge \mathrm{Hungry}_A(Q, k, s) \to \mathrm{Hungry}_A(P \cup Q, k, s)$

Proof. Immediate, since $[P \cup Q](s) = [P](s) \cup [Q](s)$. □

(d)

$$\forall P, Q, s, k, l \,.\, \mathrm{Hungry}_A(P, k, s) \wedge (\forall a, r.(a, r) \in [P](s) \to \mathrm{Hungry}_B(Q_a, l, r)) \\ \to \mathrm{Hungry}_B(P >\!\!>\!\!= Q, k + l, s)$$

Proof. By definition, $[P >\!\!>\!\!= Q](s)$ is the set of all (b, t) such that

$$\exists a, r.(a, r) \in [P](s) \wedge (b, t) \in [Q_a](r)$$

By assumption, we know that

$$\forall a, r\, (a, r) \in [P](s) \to |r| + k \leq |s|$$

and

$$\forall b, t\, (b, t) \in [Q_a](r) \to |t| + l \leq |r|$$

Since we know that $|t| + l \leq |r|$, it follows that $(|t| + l) + k \leq |s|$ for any (b, t) in the goal set. □

4.5 The parser monad

The operations return and $>\!\!>\!\!=$ endow the type of parsers $\mathbb{P}(\mathrm{Str} \times A)$, viewed as a functor of the result type A, with the structure of a monad in the category of sets and functions, considering $\mathbb{P}(\cdot)$ as a (co-variant) functor. Monads were first connected with computation by [20] and are now widely used in programming. The type of parsing functions, $\mathrm{Str} \to \mathbb{P}(A \times \mathrm{Str})$, viewed again as a functor of A, is a monad as well. Its structure maps are [return] (as defined in Section 4.1) and [>>=] where

$$(F[>\!\!>\!\!=]G)(s) = \{(b, t) | \exists a, r.(a, r) \in F(s) \wedge (b, t) \in G_a(r)\}$$

Hence $[\mathrm{return}(x)] = [\mathrm{return}](x)$ and $[P >\!\!>\!\!= Q] = [P]\ [>\!\!>\!\!=]\ [Q]$ (where $[Q]_a := [Q_a]$), which means that the map $P \mapsto [P]$, sending a parser (relation) to its parsing function is a monad homomorphism. The monad of parsing functions, $\mathrm{Str} \to \mathbb{P}(A \times \mathrm{Str})$, looks similar to the Haskell type of parsers, $\mathrm{Str} \to [(A, \mathrm{Str})]$ in [13], but is in fact quite different. This is due to the use of the full set-theoretic function space and powerset, as opposed to Haskell's computable functions and lists. In the following we will write return and $>\!\!>\!\!=$ (instead of [return] and [>>=]) for the structure maps of the parsing function monad $\mathrm{Str} \to \mathbb{P}(A \times \mathrm{Str})$.

The monadic structure allows us to use the popular "do-notation":

$$\begin{array}{lll} \mathrm{do}\,\{\, a \leftarrow P\,;\, Q_a \,\} & \text{stands for} & P >\!\!>\!\!= \lambda a \,.\, Q_a \\ \mathrm{do}\,\{\, P\,;\, Q \,\} & \text{stands for} & P >\!\!> Q \end{array}$$

where $P >\!\!> Q$ stands for $P >\!\!>\!\!= \lambda a \,.\, Q$ (a not in Q). These definitions will be used for both the parser and the parsing function monad.

5 Formalization and program extraction in Minlog

We pause the mathematical development of monadic parsers to give some background information about the Minlog system and explain the formalization and program extraction process by means of the parsers we have seen so far. The development of monadic parsers will be resumed in Section 6. Minlog ([19], [1]) is a proof system based on a first order natural deduction calculus. It is interactive, using a backwards reasoning style. Minlog implements Heyting Arithmetic in finite types enriched with free predicate variables, program constants (denoting computable functionals) with term-rewriting rules, inductive data types (freely generated algebras), and inductively and coinductively defined predicates. It is not a type-theoretic system, but keeps formulas and proofs separate from types and terms and has a simple domain-theoretic semantics: the partial continuous functionals. Minlog has an elaborate module for program extraction, which will be described in more detail in Section 5.2. The theoretical background of Minlog is explained in [24].

5.1 Formalization

In the following, we discuss some relevant aspects of the formalization in Minlog. So far, the development of the theory has been independent from the representation of sets. We have decided to treat sets as boolean functions in Minlog, since this is closest to standard mathematical practice. A set of elements of type α therefore has the type $\alpha \to$ Boole. If S is a set and a is an element of type α, $S\,a$ represents the statement $a \in S$. In our example, α will be instantiated by the type $A \times \mathrm{Str}$, i.e., pairs of parsing results and strings.

It is worth noting that, by using boolean functions and the existential quantifier in our formalization, we make our proofs non-constructive. This is due to the fact that the universal and existential quantifiers become boolean functionals that perform the uncomputable tests as to whether a given boolean function (with possibly infinite domain) is constant true, or not constant false, respectively. However, as we only compute not with the extracted programs but the proofs themselves this is not an issue. The realizability interpretation used in the Minlog system lets us use such aspects of classical logic in proofs whilst still extracting constructive programs.

We define the empty set (`Em`) and the operations insertion (`Ins`) for inserting an element into a set and (`Un`) for computing the union of two sets as program constants in Minlog with appropriate term-rewriting rules.

With these constants, we can define an inductive predicate `Fin:alpha=>boole` (where `alpha` is a type variable), containing all finite sets (cf. $\mathbb{P}_{<\omega}(X)$ in Section 3). `Fin` has two closure axioms: one for the base case of the empty set (**F1**), and one for the step case of a finite set with an additional inserted element (**F2**). In Minlog, we give names to these closure axioms: `InitFin` and `GenFin`. Reverting to the set-theoretic syntax, these axioms are:

InitFin: $\emptyset \in \mathbb{P}_{<\omega}(X)$
GenFin: $\forall S.S \in \mathbb{P}_{<\omega}(X) \to \forall x \in X.\{x\} \cup S \in \mathbb{P}_{<\omega}(X)$

In Minlog, this inductive definition reads as follows:

```
(add-ids
 (list (list "Fin" (make-arity (py "alpha=>boole")) "list"))
  '("Fin (Em alpha)" "InitFin")
  '("allnc S (Fin S -> all x Fin ((Ins alpha) x S))" "GenFin")
```

Here `x` is a variable of type `alpha`, `S` is a variable of type `alpha=>boole`, and `allnc` is a "non-computational" quantifier (explained below).

As an example we now discuss the proof of Lemma 3 (a), which essentially states that the union of two finitely branching parsing functions is also finitely branching. In Minlog syntax, we write our initial goal as:

```
(set-goal (pf "allnc qfun (all st Fin (qfun st) ->
   allnc pfun (all st Fin (pfun st) ->
     all st Fin ((Un alpha@@list char) (pfun st) (qfun st)))")))
```

where `pfun` and `qfun` have the type $\text{Str} \to \mathbb{P}(A \times \text{Str})$, or, in Minlog syntax:

```
list char => (alpha@@list char)=>boole
```

and `st` is of type Str (`list char`). We then proceed by instantiating the proof of Lemma 1 (a), where S and T become the sets of parsing results for $[P](s)$ and $[Q](s)$ respectively. Once the proof is finished, we can save this lemma in Minlog with the command:

```
(save "FinLemma3a")
```

Noteworthy in this example is the use of the quantifier `allnc` in the goal and the inductive definition, which is logically equivalent to the `all` quantifier. It may be used if the quantified variable does not occur freely in any term in the proof at a computationally relevant position (i.e., in particular not in the introduction rule for the existential quantifier, or the elimination rule for the `all` quantifier). In the programs we extract the input string `st` *is* computationally relevant, hence the use of the `all` quantifier for this variable.

5.2 Program extraction

Through the Curry-Howard correspondence, we can associate formulas in intuitionistic logic with data types. For instance, the conjunction of two formulas corresponds to the Cartesian product of two data types; implication corresponds to the type of functions. Furthermore, we can represent natural deduction proofs as terms written in the simply typed lambda calculus. From these proof terms we extract the programs, essentially by removing the computationally irrelevant parts.

To extract programs from proofs, the system uses a modified realizability interpretation (see [16], [24] for details). This interpretation translates formulas and logical rules into types and operations respectively. There are two sources of constructive content in MINLOG proofs: $\forall\exists$-formulas and inductive definitions. In the following, we briefly explain how to realize inductive definitions.

At the program level, the inductively defined predicate `Fin` corresponds to the algebra `list`. The elements of this algebra are terms built from two constructors, which correspond to the closure axioms of the inductive definition. In this case, `InitFin` is realized by the constructor called `cInitFin` of type `(list alpha)`. `GenFin` is realized by the constructor:

```
cGenFin:  (list alpha) => alpha => (list alpha)
```

It is worth noting that this corresponds to the standard cons constructor for lists, but with the order of the arguments reversed. *In general, any proof that a parsing function is finitely branching yields, via program extraction, a program that lists all of the finitely many parsing results.*

Finally, the induction principle for the inductive definition corresponds to recursion on the program level.

5.3 Extracted program for the union of two parsers

From an inductive proof that the union of two finitely branching parsers is finite, we can automatically extract a recursive program in MINLOG using the following command:

```
(define choice (rename-variables
   (proof-to-extracted-term
    (theorem-name-to-proof "FinLemma3a"))))
```

where `choice` is a name we have chosen for the program. To print the extracted program, we use:

```
(pp choice)
```

The output is shown below:

```
[qfunr,pfunr,st]
 (cFinLemma1a alpha@@list char)(qfunr st)
 (pfunr st)
```

Here, the corner brackets denote lambda abstraction. On the proof level, the parsing functions `pfun` and `qfun` had the type

```
list char => (alpha@@list char)=>boole
```

On the program level, the first two input arguments are realizers for these parsing functions with the type

```
list char => list(alpha@@list char)
```

i.e., the type of parser programs shown in Section 3.2.

The program `choice` takes a string as its third input argument. It then takes the MINLOG term for Lemma 1 (a) (instantiated with type $(A \times \mathrm{Str})$) and applies it to the sets `(qfunr st)` and `(pfunr st)` of that type.

`cFinLemmala` defines the following program:

```
[r1,r2]
 (Rec list alpha=>list alpha)r2 r1
 ([r3,r4,x]
   (CGenFin alpha)x::r4)
```

where `r2` and `r1` are variables of type `(list alpha)` representing the finiteness of the sets `S` and `T`. The program runs by recursion on the finiteness of `S`, i.e., recursion on lists. In the base case, when `S` is empty, the finiteness of the union follows from the finiteness of `T` (hence `r1` will be returned). In the step case, where `S` equals `Ins x S'`, `CGenFin` produces a realizer for the finiteness of `Un (Ins x S')T` using `r4` and `x`. Essentially, the program implements the concatenation of lists.

Note that the sets `S` and `T` themselves do not appear in this program. This is due to our use of the non-computational quantifier `allnc` in the proof.

Programs have also been extracted from MINLOG proofs of Lemmas 1 (b) and 1 (c) (see [2]). `cFinLemma1c` maps any element `x` to a singleton list `[x]`. `cFinLemma1b` behaves like the `concat` function in Haskell, which concatenates the elements of a list of lists into a single list.

5.4 Soundness of the extracted program

MINLOG will automatically generate a soundness proof stating the correctness of the extracted program. In the case discussed in the previous section, it makes use of an inductively defined predicate FinMR with two arguments (a term r and a set S) stating that r is a realizer for the finiteness of S which means that r is a finite listing of the elements of S (possibly with repetitions). The soundness theorem then states that if we have realizers for the finiteness of S and T, then the extracted program applied to these realizers realizes that the union of S and T is finite. In mathematical notation this statement reads as follows, where p is the extracted program, S and T are sets of type α and r_1 and r_2 are realizers:

$$\forall T, r_1\,(\mathrm{FinMR}\, r_1\, T \to \forall S, r_2\,(\mathrm{FinMR}\, r_2\, S \to \mathrm{FinMR}\,(p\, r_1\, r_2)(S \cup T)))$$

5.5 Example run of an extracted parser combinator

Now we illustrate how the previous proofs can be used to extract correct and terminating parser programs using combinators.

In Section 4.2, we formalized the parser combinator for choice in terms of set union. In Section 5.3, we extracted the choice combinator from an instantiated proof that the

union of two finitely branching sets is finite. To extract parser programs that use the choice combinator in MINLOG, we just need to instantiate `pfunr` and `qfunr` with concrete parsers, i.e., with proofs that the parser relations corresponding to these parser programs (`pfun` and `qfun`) are finitely branching.

As an example of this, we will extract a program that combines the item parser and the return operator. Extracting such a program is straightforward. After proving that both parsing functions are finitely branching, we use these proofs with `FinLemma3a` to prove:

$$\forall c.\mathrm{FB}_{\mathrm{Char}}(\mathrm{item} \cup \mathrm{return}\ c)$$

In accordance with Section 3.1, a parser is finitely branching if its parsing function yields a finite set of results with any input string s. At the proof level, this follows from the finite union proof and the finiteness of the individual parsers. At the program level, it is realized by a concatenated list of parsing results.

The extracted program takes a (return) character `'c'` and a string `st` as input. It can be run in MINLOG as follows. With the argument string `"aba"` and character `'c'`, the program yields a list with two elements: `('a',"ba")` from the item parser and `('c',"aba")` from the return function. In MINLOG, this is displayed as:

```
(a@b::a:)::(c@a::b::a:):
```

where `::` is the cons operator for lists and `@` is used for pair terms.

As a second example run, the empty string could be given as input. The item parser would fail, so the result from the return function would be returned as a singleton list:

```
(c@(Nil char)):
```

where `(Nil char)` is the empty list of type `Char`.

Similarly to the parsers and parser combinators item, return and choice featuring in the example above, we carried out formalization and extraction for the bind parser combinator and the recursive iteration combinator. In the following section, these steps will not be shown in full detail. Instead, we will focus on the theory and proofs used for extracting recursive programs.

6 Recursive monadic parser combinators

We now return to the mathematical development and look at recursive parsers. Nearly all non-trivial parsers use some form of recursion. In our setting recursive parsers are relations that are defined inductively. As an example we consider finite iterations of a parser. For a parser $P \in \mathbb{P}(\mathrm{Str} \times A)$ we define the parser $\mathrm{many}(P) \in \mathbb{P}(\mathrm{Str} \times A^*)$ inductively as follows:

- $([], []) \in \mathrm{many}(P)$
- If $(s, a) \in P$ and $(t, as) \in \mathrm{many}(P)$, then $(s +\!\!+ t, a : as) \in \mathrm{many}(P)$.

Equivalently, many(P) can be defined as the least solution of the equation

$$\text{many}(P) = \{([]; [])\} \cup (P >>= \lambda a.\text{many}(P) >>= \lambda as.\text{return}(a : as))$$

or, using do-notation,

$$\text{many}(P) = \{([]; [])\} \cup \text{do}\,\{\, a \leftarrow P;\ as \leftarrow \text{many}(P);\ \text{return}(a : as)\,\}$$

In general, many(P) need not be finitely branching, even if P is. For example, if $P = \text{return } 0 = \{([], 0)\}$, then $\text{many}(P) = \{([], 0^n) | n \in N\}$, which is not finitely branching. However, if we exclude parsers accepting the empty string, then the operator many preserves the finite branching property.

Clearly many(P) is only 0-hungry, but its variant

$$\text{many1}(P) := \text{do}\,\{\, a \leftarrow P\,;\ as \leftarrow \text{many}(P)\,;\ \text{return}(a : as)\,\}$$

is k-hungry, provided that P is k-hungry (and finitely branching).

Our definition of many corresponds to the standard Haskell implementation of the `many` combinator (see [13]). We can prove the following in Minlog:

$$\forall P.\, \text{FB}_A(P) \to \forall k, s\, \text{Hungry}_A(P, k, s) \to \text{FB}_{A^*}(\text{many}(P))$$

Instead of proving this, however, it will be more useful to work with a generalization of the many operator[2] with an arbitrary result type B:

$$\text{manyGen}_{A,B} \ :\ \mathbb{P}(\text{Str} \times A) \to (A \to B \to B) \to B \to \mathbb{P}(\text{Str} \times B)$$

$$\text{manyGen}_{A,B}\, P\, op\, b \ := (\text{return}\, b) \cup \text{do}\,\{\, a \leftarrow P;\ \text{manyGen}_{A,B}\, P\, op\, (op\, a\, b)\,\} \qquad (4)$$

In addition to a parser $P \in \mathbb{P}(\text{Str} \times A)$, this operator also takes an arbitrary function $op : A \to B \to B$ as a parameter. This is applied in place of the (:) function used in the standard many combinator. Hence many is a special case of this more general operator. Because the return type is not necessarily A^*, the base case is not necessarily the set $\{([], [])\}$. To incorporate the base case result for any type B, it must be given to the combinator as a third parameter argument b. In Sections 7 and 8, we will see examples of how the generalized many combinator can be used to extract a recursively defined calculator program (Section 7) and a non-deterministic natural language grammar (Section 8). In the lemma below we use the abbreviation

$$\text{FB}_A(P, n) := \forall s\,.\,|s| \le n \to [P](s) \in \mathbb{P}_{<\omega}(A \times \text{Str})$$

Lemma 6. *If P is a parser relation which is finitely branching for strings of length $\le n$ and k-hungry where $k > 0$, then for any operator op and starting value b the parser relation* $\text{manyGen}_{A,B}\, P\, op\, b$ *is finitely branching for strings of length $\le n$, formally:*

$$\forall P, n\,.\,\text{FB}_A(P, n) \to \forall k\,.\,0 < k \to \text{Hungry}_A(P, k) \to \forall op, b\,.\,\text{FB}_B(\text{manyGen}_{A,B}\, P\, op\, b, n)$$

[2]This was motivated by the `chainl` combinator in [13]

Proof. Using (2) and (3) and swapping the universal quantifiers $\forall b$ and $\forall s$ in the conclusion, this can be unfolded as:

$$\begin{aligned}&\forall P, n\,.\,(\forall s\,.\,|s| \le n \to [P](s) \in \mathbb{P}_{<\omega}(A \times \mathrm{Str})) \to\\ &\quad \forall k\,.\,0 < k \to (\forall s, a, r\,.\,(a, r) \in [P](s) \to |r| + k \le |s|) \to\\ &\qquad \forall op, s, b\,.\,|s| \le n \to [\mathrm{manyGen}_{A,B}\ P\ op\ b](s) \in \mathbb{P}_{<\omega}(B \times \mathrm{Str})\end{aligned}$$

Assume that $[P](s)$ is finite for all s with $|s| \le n$ and that P is k-hungry where $k > 0$. Fix an operator op. Show $\forall s, b.|s| \le n \to [\mathrm{manyGen}_{A,B}\ P\ op\ b](s) \in \mathbb{P}_{<\omega}(B \times \mathrm{Str})$ by induction on the length of s. Hence, we assume as induction hypothesis

$$\forall s'.|s'| < |s| \to \forall b\,.\,|s'| \le n \to [\mathrm{manyGen}_{A,B}\ P\ op\ b](s') \in \mathbb{P}_{<\omega}(B \times \mathrm{Str})$$

and $|s| \le n$ and have to show $[\mathrm{manyGen}_{A,B}\ P\ op\ b](s) \in \mathbb{P}_{<\omega}(B \times \mathrm{Str})$. By (4), the new goal can be rewritten as

$$[\mathrm{return}\ b](s) \cup (\mathrm{do}\,\{\, a \leftarrow [P]; [\mathrm{manyGen}_{A,B}\ P\ op\ (op\ a\ b)]\,\})(s) \in \mathbb{P}_{<\omega}(B \times \mathrm{Str})$$

By Lemma 3 (a), the proof that this union is finite reduces to showing that the two component sets are finite. $[\mathrm{return}\ b](s)$ is finite by Lemma 2 (a). By Lemma 4, the proof that

$$(\mathrm{do}\,\{\, a \leftarrow [P];\ [\mathrm{manyGen}_{A,B}\ P\ op\ (op\ a\ b)]\,\})(s)$$

is finite reduces to a proof that $[P](s)$ is finite and that, for all elements $(a, r) \in [P](s)$ such that $|r| + k \le |s|$, $[\mathrm{manyGen}_{A,B}\ P\ op\ (op\ a\ b)](r)$ is also finite. The finiteness of $[P](s)$ follows by assumption. Since P is more than 0-hungry for any $(a, r) \in [P](s)$, the string r must be shorter than s (since $|r| + k \le |s|$). Therefore, by the induction hypothesis (used with $b = op\ a\ b$), $[\mathrm{manyGen}_{A,B}\ P\ op\ (op\ a\ b)](r)$ is finite. □

The following program was extracted from this proof in Minlog. Here general recursion, `GRecGuard`, corresponds to the use of induction on the length of the string. Note that the variables P, n, k are non-computational and do not show up in the program. In the following `btlist` is a variable for type `list(beta@@list char)`, and `manypfunr` is of type `list char=>beta=>list(beta@@list char)`. The program is similar to Haskell's `foldr` (fold-right); it repeatedly applies the parser P and combines the results using the operator *op* with the starting value b in a right-associative way.

```
[op,pfunr,st]
 (GRecGuard list char beta=>list(beta@@list char))([st0]Lh st0)st
 ([st0,manypfunr,b]
   ([btlist^]btlist^)
   ((cFinLemmaFoura beta)st0((cFinLemmaTwoa beta)b st0)
    ((cFinLemmaFourb alpha beta)st0
        (pfunr st0)([a,st1]manypfunr st1(op a b)))))
 True
```

7 Case Study: Extracting a calculator

As an example, we use the generalized many parser combinator to extract a calculator program. This is a slightly simplified version of an example in [13]. To make it easier to follow, we have restricted the operators to addition and multiplication. As in [13], however, the approach can be applied to the non-associative operations subtraction and division. Details are provided in the source files [2].

In Backus Naur Form (BNF), the grammar for our parser is represented as follows.

$$\begin{aligned} expr &::= expr\ \text{'}+\text{'}\ term \mid term \\ term &::= term\ \text{'}*\text{'}\ factor \mid factor \\ factor &::= digit \mid \text{'(' } expr\ \text{')'} \\ digit &::= \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \end{aligned}$$

This grammar can be directly interpreted as a simultaneous inductive definition of parser relations by using the choice combinator in place of vertical bars and the bind combinator for sequencing.

$$\begin{aligned} \text{expr} &= \text{do}\,\{\, e \leftarrow \text{expr};\ \text{addP};\ t \leftarrow \text{term};\ \text{return}(e+t)\,\} \cup \text{term} \\ \text{term} &= \text{do}\,\{\, t \leftarrow \text{term};\ \text{mulP};\ n \leftarrow \text{factor};\ \text{return}(t*n)\,\} \cup \text{factor} \\ \text{factor} &= \text{do}\,\{\, d \leftarrow \text{digit};\ \text{return}\ d\,\} \cup \text{do}\,\{\, \text{leftP};\ e \leftarrow \text{expr};\ \text{rightP};\ \text{return}\ e\,\} \end{aligned}$$

Here addP, mulP and digit are the parser relations for the characters '+' and '*' and the integers 1–4 respectively. I.e.,

$$\begin{aligned} \text{addP} &:= \{([\text{'+'}], ())\} \\ \text{mulP} &:= \{([\text{'*'}], ())\} \\ \text{digit} &:= \{(\text{'1'}, 1), (\text{'2'}, 2), (\text{'3'}, 3), (\text{'4'}, 4)\} \end{aligned}$$

where () represents the dummy value. leftP and rightP are the parser relations for parsing the characters '(' and ')' respectively.

$$\begin{aligned} \text{leftP} &:= \{([\text{'('}], ())\} \\ \text{rightP} &:= \{([\text{')'}], ())\} \end{aligned}$$

For simplicity, we assume that the input doesn't contain space characters, although space robustness could be added easily. We also output the numerical value of the parsed strings and not their syntax trees (as is done in [13]).

Note that despite the "left-recursiveness" of the parsers expr and term they are well-defined relations (because the recursive calls occur at positive positions). This would not be the case if the definitions were interpreted as Haskell programs (replacing union by concatenation) since these would not terminate.

One can prove that the parser relations expr and term are finitely branching. First one defines for a set $X \in \mathbb{P}(\text{Nat} \times \text{Str})$ and number d, $X^{\geq d} := \{(n, r) \in X \mid |r| \geq d\}$, and then shows that the sets [expr](s), [term](s), [factor](s) are finite by induction on s using a side induction on $|s| - d$ to prove that the sets [expr]$(s)^{\geq d}$, [term]$(s)^{\geq d}$, [factor]$(s)^{\geq d}$ are finite (the side induction is needed to cope with left-recursion). Since this can be proven constructively, one could extract parser programs. However, these would be rather inefficient. In order to obtain reasonably efficient programs we follow the standard rewriting method (see [21], [17]), which works as follows.

Take a left-recursive rule of the form

$$A ::= A\,\alpha_1 \mid ... \mid A\,\alpha_r \mid \beta_1 \mid ... \mid \beta_s$$

where A is a non-terminal in the grammar and for $1 \leq i \leq r$ ($1 \leq i \leq s$) each α_i (β_i) is a sequence of terminals and non-terminals. In addition, the β_i do not start with the symbol A. Remove the left-recursive clauses from the rule for A, and add the non-terminal category A' to the end of each β-sequence:

$$A ::= \beta_1\,A' \mid ... \,\beta_s\,A'$$

where A' is either empty (ϵ) or an α-sequence followed by A':

$$A' ::= \alpha_1\,A' \mid ... \mid \alpha_r\,A' \mid \epsilon$$

With this approach one obtains a grammar that is *weakly equivalent* to the original one. Weakly equivalent grammars can be used to recognise and parse the same set of sentences as their left-recursive counterparts, but will not generate the same set of derivation trees in all cases.

Using this method, the left-recursive BNF grammar for arithmetic expressions can be rewritten as follows.

$$\begin{aligned}
expr &::= term\ expr' \\
expr' &::= \text{'}+\text{'}\ term\ expr' \mid \text{'}\epsilon\text{'} \\
term &::= factor\ term' \\
term' &::= \text{'}*\text{'}\ factor\ term' \mid \text{'}\epsilon\text{'} \\
factor &::= digit \mid \text{'(' } expr \text{ ')'} \\
digit &::= \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'}
\end{aligned}$$

Interpreting 'ϵ' as the empty list, expr$'$ essentially parses lists of (values of) terms interspersed with the symbol '+'. The resulting list of values is then used by expr, together with (the value of) a leading term, to compute the final result, which is the sum of all these values. Following the approach in [13] we will accomplish this whole task in one go by using the manyGen combinator (which replaces the combinator `chainl` in [13]). We define the following parsers for this purpose:

$$\text{addTermP, mulFactorP} : \mathbb{P}(\text{Str} \times \text{Nat})$$

$$\text{addTermP} = \text{do}\,\{\,\text{addP};\ \text{term}\,\}$$
$$\text{mulFactorP} = \text{do}\,\{\,\text{mulP};\ \text{factor}\,\}$$

Using these parsers with the generalized many combinator, we can directly "implement" the modified parts of the BNF grammar shown above, i.e., $expr'$ and $term'$. By binding the resulting parsers to term and factor respectively, we get a direct representation of the modified grammar as parser relations:

$$\begin{array}{l}\text{expr} : \mathbb{P}(\text{Str} \times \text{Nat})\\ \text{expr} := \text{do}\,\{\,n \leftarrow \text{term};\ \text{manyGen}_{\text{Nat,Nat}}\ \text{addTermP}\ (+)\ n\,\}\end{array}$$

$$\begin{array}{l}\text{term} : \mathbb{P}(\text{Str} \times \text{Nat})\\ \text{term} := \text{do}\,\{\,n \leftarrow \text{factor};\ \text{manyGen}_{\text{Nat,Nat}}\ \text{mulFactorP}\ (*)\ n\,\}\end{array}$$

$$\begin{array}{l}\text{factor} : \mathbb{P}(\text{Str} \times \text{Nat})\\ \text{factor} := \text{digit} \cup \text{do}\,\{\,\text{leftP};\ \text{expr};\ \text{rightP}\,\}\end{array}$$

To prove that $[\text{term}](s)$ is finite for any string s, we use instantiated proofs of Lemmas 4, 5 and 6. The only remaining work is to prove that the component parsers are finitely branching. At the lowest level, this reduces to simple proofs that the basic parsers digit, mulP, addP, leftP and rightP are finitely branching. The modified parser definitions are correct since they define the same parsing functions as in [13]. This can be shown using similar inductions to those described above, but we omit the proof since we do not wish to extract a program from it. However, correctness relies on the fact that these parsers are deterministic, i.e., all result sets are either empty or singletons. Otherwise, one might miss out some results (that this can indeed happen will be demonstrated in the second case study (see Section 8)).

We can extract a precedence-preserving arithmetic calculator from a proof that the expression parsing function is finitely branching and hungry for any input string by induction on the length of the input. To do this, we use the following lemmas. The complete proofs can be found in [2].

Lemma 7. *All the parser relations shown above are 1-hungry.*

Proof. The proof is very easy in most cases. As an example, we give details of the proof that the parser relation factor is hungry. We show

$$\forall s\, \text{Hungry}_{\text{Nat}}(\text{factor}, 1, s)$$

by induction on $|s|$. Assume as induction hypothesis

$$\forall s'.|s'| < |s| \to \text{Hungry}_{\text{Nat}}(\text{factor}, 1, s')$$

Using the definition of expr and Lemma 5 (c), we need to prove

$$\text{Hungry}_{\text{Nat}}(\text{digit}, 1, st) \wedge \text{Hungry}_{\text{Nat}}((\text{do}\,\{\,\text{leftP};\ \text{expr};\ \text{rightP}\,\}), 1, st)$$

Left: Proof omitted here. *Right:* Proven with Lemma 5 (b) (with k instantiated with 3) and Lemma 5 (d) to prove

i. $\text{Hungry}_{\text{Char}}(\text{leftP}, 1, st)$
Proof omitted here.

ii. $\forall c, r.(c, r) \in [\text{leftP}](s) \to \text{Hungry}_{\text{Nat}}(\text{expr}, 1, r))$
With the definition of expr and Lemma 5 (d), this reduces to showing that

$$\text{Hungry}_{\text{Nat}}(\text{term}, 1, r)$$

and

$$\forall n, r'.(n, r') \in [\text{term}](r) \to \text{Hungry}_{\text{Nat}}((\text{manyGen}_{\text{Nat,Nat}}\ \text{addTermP}\ (+)\ n), 0, r')$$

The latter follows from Lemma 5 (a)[3]. The former (with the definition of term and Lemma 5 (d)) reduces to showing that

$$\text{Hungry}_{\text{Nat}}(\text{factor}, 1, r)$$

and

$$\forall n, r'.(n, r') \in [\text{factor}](r) \to \text{Hungry}_{\text{Nat}}((\text{manyGen}_{\text{Nat,Nat}}\ \text{mulFactorP}\ (*)\ n), 0, r')$$

Again, the latter follows from Lemma 5 (a). To prove $\text{Hungry}_{\text{Nat}}(\text{factor}, 1, r)$, we use the induction hypothesis and the fact that $|r| < |s|$.

iii. $\forall n, r'.(n, r') \in [\text{expr}](r) \to \text{Hungry}_{\text{Char}}(\text{rightP}, 1, r'))$
Proof omitted here.

□

Lemma 8. $\text{FB}_{\text{Nat}}(\text{factor})$

Proof. Using (2), this can be rewritten as

$$\forall s\ .[\text{factor}](s) \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$

Proof by induction on $|s|$. Assume s. Assume

$$\forall s'.|s'| < |s| \to [\text{factor}](s') \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$

Show

$$[\text{factor}](s) \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$

Unfolding the definition and using Lemma 3 (a), the proof reduces to showing that

[3] As was noted in Section 6, we can only prove 0-hungriness of the generalized many combinator, as it is defined as a union with the return parser (which is only 0-hungry).

i. $[\text{digit}](s) \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$
Proof omitted here.

ii. $(\text{do}\,\{\,[\text{leftP}];\ [\text{expr}];\ [\text{rightP}]\,\})(s) \in \mathbb{P}_{<\omega}(\text{Char} \times \text{Str})$

A proof is given by using Lemma 4 (where k is instantiated with 1). $[\text{leftP}](s)$ is finite and 1-hungry. By Lemma 4 (again instantiated with 1), for any $(c, r) \in [\text{leftP}](s)$ the set $(\text{do}\,\{\,[\text{expr}];\ [\text{rightP}]\,\})(r)$ is finite if $\text{Hungry}_{\text{Nat}}(\text{expr}, 1, r)$ (which follows from Lemma 7) and

$$[\text{expr}](r) \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$

and

$$\forall n, t.(n, t) \in [\text{expr}](r) \to [\text{rightP}](t) \in \mathbb{P}_{<\omega}(\text{Char} \times \text{Str})$$

The latter follows naturally from the definition of rightP. Unfolding [expr] (and its constituent [term]) in the same way, we eventually prove

$$[\text{factor}](r)$$

using the induction hypothesis and the fact that $|r| < |s|$. □

Lemma 9. $\text{FB}_{\text{Nat}}(\text{term})$

Proof. Unfolding the definition, use Lemma 4 (where k is instantiated with 1) and prove

i.
$$[\text{factor}](s) \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$
Follows from Lemma 8.

ii.
$$\text{Hungry}_{\text{Nat}}(\text{factor}, 1, s)$$
Follows from Lemma 7.

iii.
$$\forall n, r.|r| + 1 \leq |s| \to [\text{manyGen}_{\text{Nat,Nat}}\ \text{mulFactorP}\ (*)\ n](r) \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$
Assume n, r, and bounded length of r. Prove
$$[\text{manyGen}_{\text{Nat,Nat}}\ \text{mulFactorP}\ (*)\ n](r) \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$
using Lemma 6 (where n and k are instantiated with $|r|$ and 1 respectively). Since $[\text{mulFactorP}](s)$ is 1-hungry for any s (by Lemma 7) and $0 < k$, all that remains is to show that
$$\forall t.|t| < |r| \to [\text{mulFactorP}](t) \in \mathbb{P}_{<\omega}((\text{Nat} \to \text{Nat} \to \text{Nat}) \times \text{Str})$$

Unfolding the definition, we know that the set [mulP](t) is finite. All that remains to show is that

$$\forall f, t'.(f, t') \in [\text{mulP}](t) \to [\text{factor}](t') \in \mathbb{P}_{<\omega}(\text{Nat} \times \text{Str})$$

This follows from Lemma 8.

□

Proposition 10. $\text{FB}_{\text{Nat}}(\text{expr})$

Proof. Similar to the proof for Lemma 9, but using [addTermP](t) and [term](t') (where the finiteness of the latter follows from Lemma 9). □

We have extracted proof terms for Lemmas 8 and 9 in Minlog. Note that the measure function must be explicitly given to carry out general induction in Minlog. For Lemmas 7 and 8, this is the length function (`Lh`):

```
(gind (pt "[st]Lh st"))
```

Proposition 10 is proven from the following goal:

```
(set-goal (pf "all st. Fin(Expr st)"))
```

The resulting program is shown below.

```
[st]
 ([ntlist^]ntlist^)
 ((cFinLemmaFourb nat nat)st(cFinTerm st)
   ([n,st0]
    (cFinLemmaSix nat nat)AddOp
    ([st1]
      ([manyfnr]
        manyfnr st1(cFinAddP st1)
        ([f,st2]
          (cFinLemmaOneb (nat@@list char) (nat@@list char))
              (cFinTerm st2)
              ([nst](cFinLemmaTwoa nat)left nst right nst)))
              (cFinLemmaFourb nat=>nat=>nat nat))
    st0
    n))
```

Here `ntlist` has the type `list(nat@@list char)`, i.e., it represents a list of parsing results for the calculator. We also use variables `n:nat`, `nst:(nat@@list char)`

and `f:nat=>nat=>nat`. The extracted program takes a string as input[4]. As an example, suppose that we give the input string `"2*3+4"`. The output would be:

```
(10@(Nil char))::(6@+::4:)::(2@*::3::+::4:):
```

As with the standard many combinator, it returns a list of partial parse results, starting from the most complete.

In the example shown above, the parser can calculate the result recursively from left to right. But suppose that we swap the ordering to `4+3*2`:

```
(10@(Nil char))::(7@*::2:)::(4@+::3::*::2:):
```

As this second output list illustrates, the parser initially calculates `4+3`, but then backtracks to return the correct result calculated from (`4 + (expr "3::*::2:")`).

If we run the program with the input string `"(4+3)*2"`, however, the expression inside the brackets is calculated first:

```
(14@(Nil char))::(7@*::2:):
```

8 Case Study: Extracting a parser for natural language

In the introduction, we indicated that our long term goal is to apply our method of extracting monadic parsers to natural language. In this case study, we demonstrate that our method can also deal with ambiguous Context Free Grammars (CFGs). Additionally, we show that monadic parser combinators provide an elegant way of capturing these ambiguities (i.e., by facilitating backtracking and returning all possible parses of ambiguous input in a list). As in Section 7, we translate left-recursive grammar rules into weakly equivalent ones that are not left-recursive.

For programming languages, automated parser generators such as Yacc [14] and Happy [18] provide a convenient and fast means of turning BNF representations of certain grammars into parser programs. Yacc accepts LALR(1) grammars and produces shift-reduce bottom-up parsers written in C. The Haskell parser generator Happy accepts both GLR grammars and attribute grammars, which means that it can handle ambiguous input. It can also be used to generate top-down monadic parsers. However, Happy does not accept left-recursive grammar rules or CFGs. This means that neither of the two parser generators is suitable for some NLP tasks.

Ideally, we would like to use Minlog to extract parsers that correspond directly to left-recursive grammars such as the following CFG[5], where *sent*, *np*, *vp*, *pp*, *det*, *pnoun*,

[4] To return parse trees instead of calculating values, simply comment out the computation rules for `AddOp` and `MulOp` in the Minlog source file. For the first example shown below, this would return

```
((AddOp 4(MulOp 2 3))@(Nil char))::((MulOp 2 3)@+::4:)::(2@*::3::+::4:):
```

[5] CFG shown in [10], but with conjunctions.

conj and *prep* represent the categories for sentences, noun phrases, verb phrases, prepositional phrases, determiners, proper nouns, conjunctions and prepositions respectively.

Grammar A:

$$
\begin{aligned}
\mathit{sent} &::= \mathit{np}\ \mathit{vp} \mid \mathit{sent}\ \mathit{pp} \\
\mathit{np} &::= \mathit{det}\ \mathit{noun} \mid \mathit{pnoun} \mid \mathit{np}\ \mathit{conj}\ \mathit{np} \mid \mathit{np}\ \mathit{pp} \\
\mathit{pp} &::= \mathit{prep}\ \mathit{np} \\
\mathit{vp} &::= \mathit{verb}\ \mathit{np} \\
\mathit{det} &::= \text{the} \\
\mathit{noun} &::= \text{cat} \mid \text{telescope} \mid \text{saw} \\
\mathit{pnoun} &::= \text{Annie} \mid \text{Beth} \\
\mathit{conj} &::= \text{and} \mid \text{or} \\
\mathit{prep} &::= \text{with} \\
\mathit{verb} &::= \text{saw}
\end{aligned}
$$

A parser for this grammar should take an input string and attempt to parse it as a sentence s. Some valid sentences can be parsed in multiple ways (see, e.g., Figure 1), so all possible parses should be returned as a set of derivation trees (also known as parse trees). A parser for this grammar should *also* be able to parse noun phrases that have a smaller noun phrase as their left-most constituent. Currently, our proof-extracted implementation does both of these things. However, it skirts the issue of left-recursion by using the weakly equivalent grammar shown below.

Grammar B:

$$
\begin{aligned}
\mathit{sent} &::= \mathit{np}\ \mathit{vp}\ \mathit{sent}' \\
\mathit{sent}' &::= \mathit{pp}\ \mathit{sent}' \mid \epsilon \\
\mathit{np} &::= \mathit{det}\ \mathit{noun}\ \mathit{np}' \mid \mathit{pnoun}\ \mathit{np}' \\
\mathit{np}' &::= \mathit{conj}\ \mathit{np}\ \mathit{np}' \mid \mathit{pp}\ \mathit{np}' \mid \text{'}\epsilon\text{'} \\
\mathit{pp} &::= \mathit{prep}\ \mathit{np} \\
\mathit{vp} &::= \mathit{verb}\ \mathit{np} \\
\mathit{det} &::= \text{the} \\
\mathit{noun} &::= \text{cat} \mid \text{telescope} \mid \text{saw} \\
\mathit{pnoun} &::= \text{Annie} \mid \text{Beth} \\
\mathit{conj} &::= \text{and} \mid \text{or} \\
\mathit{prep} &::= \text{with} \\
\mathit{verb} &::= \text{saw}
\end{aligned}
$$

For the sake of readability in the formalization stage, we use the following abbreviations:

$$npsingular ::= det\ noun \mid pnoun$$
$$nprec ::= conj\ np \mid pp$$

Our parsers will produce derivation trees as results. These are finite trees whose leaves carry words from the set

$$\text{Word} := \{\text{Annie}, \text{Beth}, \text{cat}, \text{telescope}, \text{saw}, \text{the}, \text{with}, \text{and}, \text{or}\}$$

and whose inner nodes are labelled with names of syntactic categories from the set

$$\text{Cat} := \{\text{S}, \text{S}', \text{NP}, \text{NP}', \text{NPSing}, \text{VP}, \text{PP}\}$$

(where S and S′ stand for *sent* and *sent′*). We let Tree denote the set of all derivation trees. Elements of this set are formally defined by two constructors: one for leaves and one for branches.

$$\text{Leaf} : \text{Word} \to \text{Tree}$$
$$\text{Branch} : \text{Cat} \to [\text{Tree}] \to \text{Tree}$$

An example of a valid tree for Grammar A is

$$\begin{aligned}&\text{Branch S}[\text{Branch NP }[\text{Leaf Beth}],\\ &\qquad\text{Branch VP }[\text{Leaf saw},\\ &\qquad\qquad\text{Branch NP }[\text{Leaf the}, \text{Leaf cat}]]]\end{aligned}$$

In addition to the two constructors shown above, we need a function for appending a single tree to a list of trees

$$\text{cnsTree} : \text{Tree} \to [\text{Tree}] \to [\text{Tree}]$$

The interpretation of Grammar B as a simultaneous inductive definition of parser relations is straightforward. To shorten the exposition, we skip the lexical analysis phase and represent the words in the set Word by the arbitrary characters 'A', 'B', 'C', 'L', 'P', 'D', 'M', 'R','O' in that order.

The primitive parsers $\text{detP}, \text{nounP}, \text{pnounP}, \text{prepP}, \text{conjP}, \text{verbP} : \mathbb{P}(\text{Str} \times \text{Tree})$ are defined below.

$$\begin{aligned}\text{detP} &:= \{(['\text{D}'], \text{Leaf the})\}\\ \text{nounP} &:= \{(['\text{C}'], \text{Leaf cat}), (['\text{L}'], \text{Leaf telescope}), (['\text{P}'], \text{Leaf saw})\}\\ \text{pnounP} &:= \{(['\text{A}'], \text{Leaf Annie}), (['\text{B}'], \text{Leaf Beth})\}\\ \text{prepP} &:= \{(['\text{M}'], \text{Leaf with})\}\\ \text{conjP} &:= \{(['\text{R}'], \text{Leaf and}), (['\text{O}'], \text{Leaf or})\}\\ \text{verbP} &:= \{(['\text{P}'], \text{Leaf saw})\}\end{aligned}$$

These primitive parsers form the basis for the following composite parsers.

$$\text{npsingularP}, \text{nprecP}, \text{ppP} : \mathbb{P}(\text{Str} \times \text{Tree})$$

$$\begin{aligned}\text{npsingularP} := \ & \text{do}\,\{\, d \leftarrow \text{detP};\ n \leftarrow \text{nounP};\ \text{return}(\text{Branch NPSing}\ [d, n])\,\} \cup \\ & \text{do}\,\{\, n \leftarrow \text{pnounP};\ \text{return}\ (\text{Branch NPSing}\ [n]\ )\,\}\end{aligned}$$

$$\begin{aligned}\text{nprecP} := \ & \text{do}\,\{\, c \leftarrow \text{conjP};\ np \leftarrow \text{npP};\ \text{return}(\text{Branch NP}'\ [c, np])\ \,\} \cup \\ & \text{do}\,\{\, pp \leftarrow \text{ppP};\ \text{return}\ (\text{Branch NP}'\ [pp]\ )\,\}\end{aligned}$$

$$\begin{aligned}\text{ppP} := \ & \text{do}\,\{\, p \leftarrow \text{prepP};\ np \leftarrow \text{npP}; \\ & \qquad \text{return}(\text{Branch PP}\ [p, np])\,\}\end{aligned}$$

The parsers above are the building blocks for the recursive parser relation for noun phrases:

$$\text{npP} : \mathbb{P}(\text{Str} \times \text{Tree})$$

$$\begin{aligned}\text{npP} := \ & \text{do}\,\{\, np \leftarrow \text{npsingularP}; \\ & \qquad rnp \leftarrow \text{manyGen}_{\text{Tree},[\text{Tree}]}\ \text{nprecP cnsTree}\ []; \\ & \qquad \text{return}(\text{Branch NP}\ (np : rnp))\,\}\end{aligned}$$

The remaining parser relations for Grammar B are shown below.

$$\text{vpP}, \text{sentrecP}, \text{sentP} : \mathbb{P}(\text{Str} \times \text{Tree})$$

$$\begin{aligned}\text{vpP} := \ & \text{do}\,\{\, v \leftarrow \text{verbP};\ np \leftarrow \text{npP}; \\ & \qquad \text{return}(\text{Branch VP}\ [v, np])\,\}\end{aligned}$$

$$\begin{aligned}\text{sentrecP} := \ & \text{do}\,\{\, trlist \leftarrow \text{manyGen}_{\text{Tree},[\text{Tree}]}\ \text{ppP cnsTree}\ []; \\ & \qquad \text{return}(\text{Branch S}'\ trlist)\,\}\end{aligned}$$

$$\begin{aligned}\text{sentP} := \ & \text{do}\,\{\, np \leftarrow \text{npP};\ vp \leftarrow \text{vpP};\ sp \leftarrow \text{sentrecP}; \\ & \qquad \text{return}(\text{Branch S}\ [np, vp, sp])\,\}\end{aligned}$$

These parser relations are motivated by Grammar B as opposed to Grammar A. One can show that the two grammars are indeed weakly equivalent, but again the proof is not shown here as we don't wish to extract a program from the equivalence.

As in Section 7, we use Lemmas 3, 4, 5 and 6 to prove that the parser for sentences is finitely branching. The full proofs can be found in the MINLOG source files [2].

Lemma 11. *Apart from* sentrecP*, all of the parser relations shown above are 1-hungry.*

Proof. As an example, we prove

$$\forall s\, \mathrm{Hungry}_{\mathrm{Tree}}(\mathrm{npsingularP}, 1, s)$$

Using Lemma 5 (c) this reduces to showing that

i.

$$\forall s\, \mathrm{Hungry}_{\mathrm{Tree}}((\mathrm{do}\,\{\, d \leftarrow \mathrm{detP};\; n \leftarrow \mathrm{nounP};\; \mathrm{return}(\mathrm{Branch\ NPSing}\ [d, n])\,\}), 1, s)$$

Using Lemma 5 (b) (where k is instantiated with 2), we prove that this parser is 2-hungry using Lemma 5 (d) twice. As detP is 1-hungry, the proof reduces to showing that $\mathrm{do}\,\{\, n \leftarrow \mathrm{nounP}; \mathrm{return}(\mathrm{Branch\ NPSing}\ [d, n])\,\}$ is 1-hungry. This follows from the 1-hungriness of nounP and Lemma 5 (a).

ii.

$$\forall s\, \mathrm{Hungry}_{\mathrm{Tree}}((\mathrm{do}\,\{\, n \leftarrow \mathrm{pnounP};\; \mathrm{return}\ (\mathrm{Branch\ NPSing}\ [n]\ )\,\}), 1, s)$$

This is proven using Lemma 5 (d). pnounP is 1-hungry and

$$\mathrm{return}\ (\mathrm{Branch\ NPSing}\ [n]\ )$$

is 0-hungry (by Lemma 5 (a)).

□

Lemma 12. *The helper parser relations are finitely branching.*

(a) $\mathrm{FB}_{\mathrm{Tree}}(\mathrm{npsingularP})$

(b) $\mathrm{FB}_{\mathrm{Tree}}(\mathrm{npP})$

(c) $\mathrm{FB}_{\mathrm{Tree}}(\mathrm{vpP})$

(d) $\mathrm{FB}_{\mathrm{Tree}}(\mathrm{sentrecP})$

Proof.

a. Using (2), this can be rewritten as

$$\forall s\, [\mathrm{npsingularP}](s) \in \mathbb{P}_{<\omega}(\mathrm{Tree} \times \mathrm{Str})$$

Assume s. Unfolding the definition and using Lemma 3 (a), this reduces to showing that

i.

$$(\text{do}\,\{\, d \leftarrow [\text{detP}];\\ n \leftarrow [\text{nounP}];\\ [\text{return}](\text{Branch NPSing}\,[d,n])\,\})(s) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})$$

Using Lemmas 4 and 11, this follows from the fact that detP and nounP are finitely branching, along with Lemma 2 (a).

ii.

$$(\text{do}\,\{\, n \leftarrow [\text{pnounP}];\\ [\text{return}](\text{Branch NPSing}\,[n]\,)\,\})(s) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})$$

Using Lemmas 4 and 11, this follows from the fact that pnounP is finitely branching, along with Lemma 2 (a).

b. To prove that

$$\forall s\,[\text{npP}](s) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})$$

we carry out induction on $|s|$. Assume s and

$$\forall s'.|s'| < |s| \to [\text{npP}](s') \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})$$

Using the definition of npP and Lemma 4 (where k is instantiated with 1), we just need to show that

i.

$$[\text{npsingularP}](s) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})$$

This follows from Lemma 12 (a).

ii.

$$\text{Hungry}_{\text{Tree}}(\text{npsingularP}, 1, s)$$

This follows from Lemma 11.

iii.

$$\forall tr, r.|r| + 1 \le |s| \to\\ (\text{do}\,\{\, rnp \leftarrow [\text{manyGen}_{\text{Tree},[\text{Tree}]}\ \text{nprecP cnsTree}\ []\,];\\ [\text{return}](\text{Branch NP}\ (tr : rnp))\,\})(r) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})$$

Assume r where $|r| + 1 \le s$, and prove using Lemmas 1 (b) and 6 (where, for Lemma 6, n and k are $|r|$ and 1). To prove that the parser nprecP is finitely branching, use the induction hypothesis and the fact that the string fed to the recursive call is shorter (since $\forall s\,\text{Hungry}_{\text{Tree}}(\text{conjP}, 1, s) \wedge \text{Hungry}_{\text{Tree}}(\text{prepP}, 1, s)$). The parser nprecP is 1-hungry by Lemma 11.

c. Using Lemma 4 (b) twice, this follows from the finiteness of [verbP] with any string s, Lemma 12 (b) and Lemma 2 (a).

d. Using the definition of sentrecP and Lemma 1 (b), we use Lemma 6 to reduce the goal to showing that ppP is 1-hungry and finitely branching. Unfolding the definition of ppP, this follows from the hungriness of prepP (by Lemma 11) and the fact that for any string s

$$\begin{aligned}&(\text{do}\,\{\, trlist \leftarrow [\text{manyGen}_{\text{Tree},[\text{Tree}]}\ \text{ppP}\ \text{cnsTree}\ []];\\ &\qquad [\text{return}](\text{Branch}\ \text{S}'\ trlist)\,\})(s) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})\end{aligned}$$

□

Proposition 13. $\text{FB}_{\text{Tree}}(\text{sentP})$

Proof. To prove

$$\forall s\, [\text{sentP}](s) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})$$

assume s and use Lemma 4 (where k is instantiated with 1). Show

i. $\text{FB}_{\text{Tree}}(\text{npP})$ This follows from Lemma 12 (b).

ii. $\text{Hungry}_{\text{Tree}}(\text{npP}, 1, s)$ This follows from Lemma 11.

iii.

$$\begin{aligned}&\forall tr, r.|r| + 1 \le |s| \rightarrow\\ &(\text{do}\,\{\, vp \leftarrow [\text{vpP}];\\ &\qquad sp \leftarrow [\text{sentrecP}];\\ &\qquad [\text{return}](\text{Branch}\ \text{S}\ [np, vp, sp])\,\})(r) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})\end{aligned}$$

Assume r s.t. $|r|+1 \le |s|$. Using Lemma 1 (b), the proof follows from the finiteness of the set $[\text{vpP}](r)$ (by Lemma 12 (c)) and a proof that for all $(tr', t) \in [\text{vpP}](r)$

$$\begin{aligned}&(\text{do}\,\{\, sp \leftarrow [\text{sentrecP}];\\ &\qquad [\text{return}](\text{Branch}\ \text{S}\ [np, vp, sp])\,\})(t) \in \mathbb{P}_{<\omega}(\text{Tree} \times \text{Str})\end{aligned}$$

This follows from Lemmas 1 (b), 12 (d) and 2 (a).

□

A monadic parser program for Grammar B can be extracted from a proof with the following initial goal statement, which corresponds to Proposition 13:

```
(set-goal (pf "all st Fin(SentP st))
```

The extracted program is shown below, where `trlistr` is a variable for lists of parse results and `trst` represents a single parse result.

```
[st]
 ([trlistr]trlistr)
 ((cFinLemmaFourb (rtree word cat)
         (rtree word cat))st(cFinNP st)
  ([tr,st1]
    (cFinLemmaOneb (rtree word cat@@list char)
           (rtree word cat@@list char))
    (cFinVP st1)
    ([trst]
      (cFinLemmaOneb (rtree word cat@@list char)
             (rtree word cat@@list char))
      (cFinRecSent right trst)
      ([trst1]
        (cFinLemmaTwoa rtree word cat)
        ((RBranch cat word)Sent(tr::left trst::(left trst1):))
        right trst1))))
```

The program captures ambiguity by returning a list of derivation trees for an input string. As an example, if we run the program with the input sentence

Annie saw Beth with the telescope

it will return two complete derivation trees. For readability, we give corresponding diagrams in Figure 1 instead of the MINLOG output. By discarding the primed and empty categories in these derivation trees, we get the valid trees for the original left-recursive grammar.

We end this section by highlighting a problem with using weakly equivalent grammars for natural language. Whilst the derivations in the example above match those generated by the left-recursive grammar, this is not always the case (as is noted in [10]). For example, the noun phrase

Annie or Beth and the telescope

should be parsed in two ways according to Grammar A (see Figure 2). The parsed syntax matches the semantics intuitively here. In the first tree, we have

Annie or (Beth and the telescope)

In the second tree, we have the alternative interpretation of the phrase as

(Annie or Beth) and the telescope

With the weakly equivalent Grammar B used in our extracted program, we still get two derivation trees (see Figure 3). The program therefore still captures the ambiguity of the noun phrase. However, the two trees do not capture the two possible meanings. In *both* cases, "Annie" is parsed as a single noun phrase. The two trees then differ in the way they derive the category np' from "or Beth and the telescope":

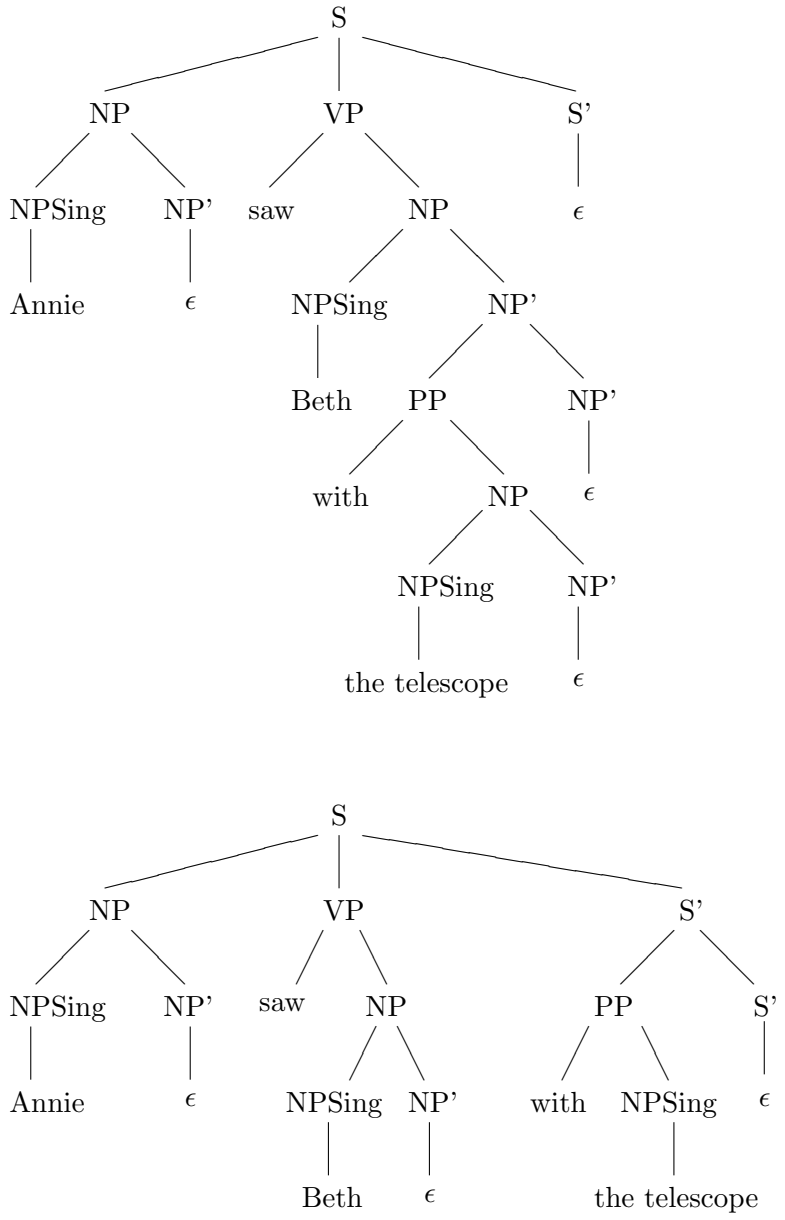

Figure 1: Derivation trees for "Annie saw Beth with the telescope" with Grammar B

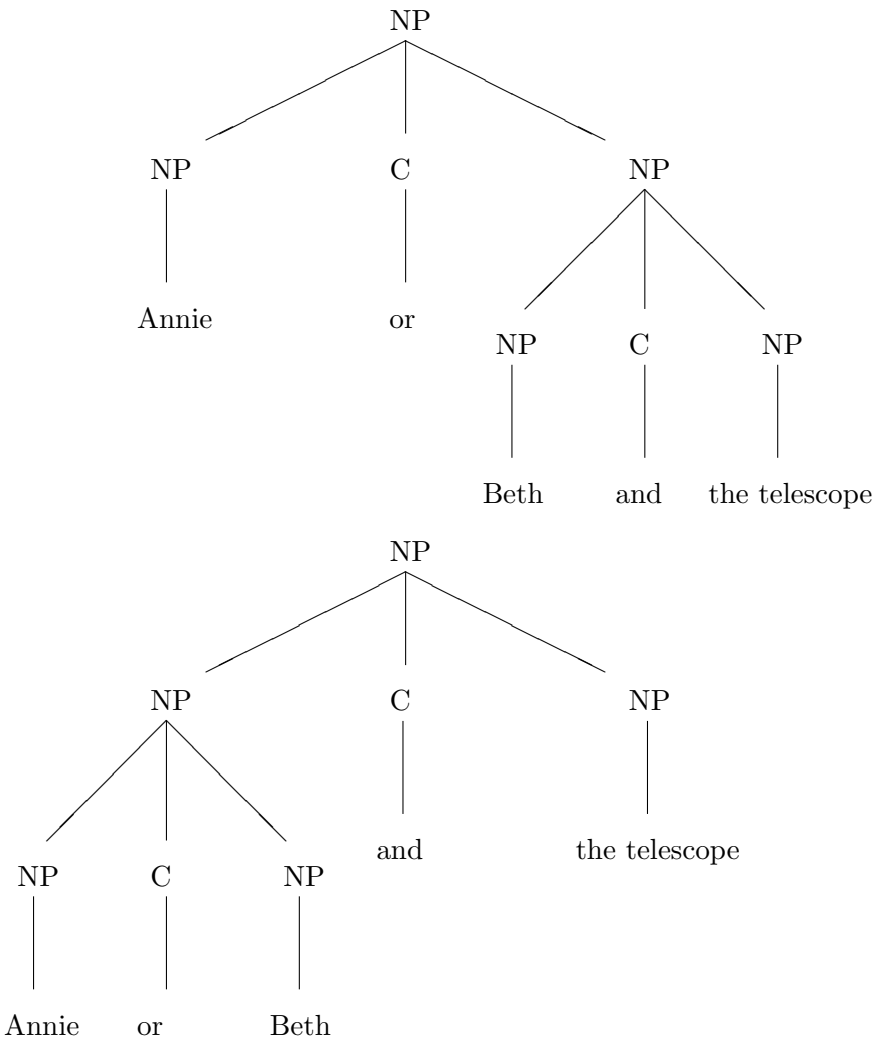

Figure 2: Derivation trees for "Annie or Beth and the telescope" with Grammar A

Annie (or Beth (and the telescope))
Annie (or Beth) (and the telescope)

These weakly equivalent derivations for Grammar B do not reflect the semantics of Grammar A.

Section 8 illustrates the way in which monadic parsers accommodate natural language ambiguity. Using the generalized many combinator, we have extracted a program that generates the expected number of derivation trees for input sentences. To extend this example further, we would like to extract a parser program whose grammar is *strongly equivalent* to the left-recursive Grammar A, i.e., it not only recognizes the

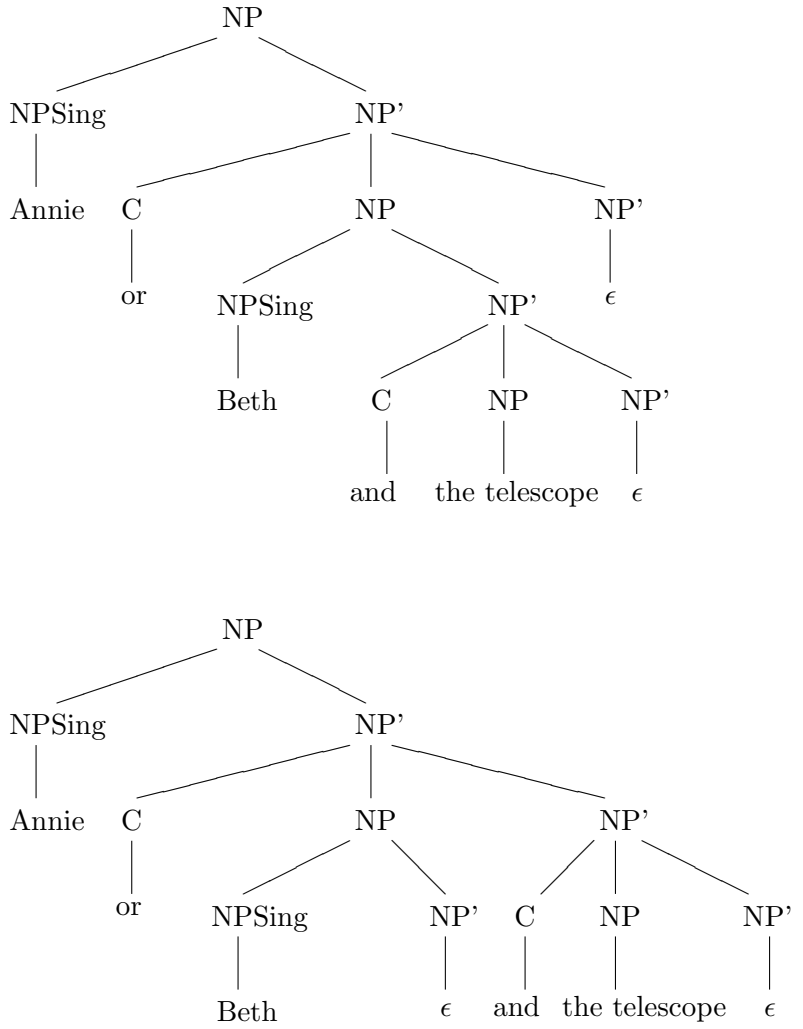

Figure 3: Derivation trees for "Annie or Beth and the telescope" with Grammar B

same set of sentences, but also generates the same set of parse results. This is part of the broader goal outlined in the next section.

9 Conclusion

Using Minlog, we are able to extract programs directly from proofs. Instead of implementing parsers and then, in a separate step, verifying that they are correct and terminating, we only need to concentrate on the proofs and get by a fully automated process the parser program and its formal correctness proof for free. The Minlog terms

that we have extracted run directly, as was shown in the examples. Alternatively, these terms can also be (automatically) translated into any higher-order functional programming language, such as Haskell or Scheme.

A fundamental benefit of treating monadic parsers at the proof level is that we can often work directly with parsers (that is, parser relations) as opposed to their parsing functions. Parsing functions are more complex objects, and verifying their correctness at the program level creates unnecessary work. A further advantage is the fact that for a recursive parser relation to be well-defined it suffices that all recursive calls are at positive positions. This is not the case for corresponding recursive parser programs since they may fail to terminate.

To our knowledge, this is the first time that program extraction has been applied to monadic parsing algorithms. The results in this paper suggest that our approach has good potential in the areas of parsing and NLP. In the two case studies, we worked with weakly equivalent versions of left-recursive grammars. Future work will include overcoming this limitation by working with the original (or strongly equivalent) grammars. Furthermore, one could optimize the data structure extracted from the finiteness property, as the concatenation of lists is not very efficient.

A proof of correctness, completeness and termination is advantageous in any parsing context. In terms of potential applications to natural language semantics in particular, there is a common concern in the literature for modular grammar engineering ([7], [6]), which is facilitated by monads. There is further potential for overlap in the area of discourse analysis and semantic inference ([8], [26]).

Minlog Sources

The source files can be found on the Swansea Minlog Repository [2]. The proofs are divided into several separate files. The first three should be loaded in sequential order. The case studies are independent of each other.

References

[1] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolfgang Zuber. Proof Theory at Work: Program Development in the Minlog System. In *Automated Deduction*, volume II of *Applied Logic Series*, pages 41–71. Kluwer, 1998.

[2] Ulrich Berger, Alison Jones, and Monika Seisenberger. Program Extraction Applied to Monadic Parsing. In Swansea Minlog Repository. `http://cs.swansea.ac.uk/minlog/monadicparsing/`.

[3] Ulrich Berger, Alison Jones, and Monika Seisenberger. Program Extraction Applied to Monadic Parsing. In Valeria de Paiva, Walther Neuper, Pedro Quaresma, Christian Retoré, Lawrence S. Moss, and Jordi Saludes, editors, *Joint Proceedings of the Second Workshop*

on Natural Language and Computer Science (NLCS'14) and 1st International Workshop on Natural Language Services for Reasoners (NLSR 2014), pages 143–154, 2014.

[4] Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. Minlog - a Tool for Program Extraction Supporting Algebras and Coalgebras. In Andrea Corradini, Bartek Klin, and Corina Cirstea, editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 393–399. Springer, 2011.

[5] Ulrich Berger and Monika Seisenberger. Applications of Inductive Definitions and Choice Principles to Program Synthesis. In *From Sets and Types to Topology and Analysis. Towards practicable foundations for constructive mathematics*, volume 48 of *Oxford Logic Guides*, pages 137–148. Oxford University Press, 2005.

[6] Patrick Blackburn and Johan Bos. Representation and Inference for Natural Language: A First Course in Computational Semantics (Volume II), 1999. `http://www.let.rug.nl/bos/comsem/`.

[7] Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language: A First Course in Computational Semantics*. Centre for the Study of Language & Information, 2004.

[8] Johan Bos. Applying Automated Deduction to Natural Language Understanding. *Journal of Applied Logic*, 7(1):100–112, 2009.

[9] Nils Anders Danielsson. Total Parser Combinators. *SIGPLAN Not.*, 45(9):285–296, September 2010.

[10] Richard A. Frost, Rahmatullah Hafiz, and Paul C. Callaghan. Modular and Efficient Top-down Parsing for Ambiguous Left-recursive Grammars. In *Proceedings of the 10th International Conference on Parsing Technologies*, IWPT '07, pages 109–120, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.

[11] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-Verlag, 2008.

[12] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

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

[14] Stephen C. Johnson. YACC: Yet Another Compiler–Compiler. `http://dinosaur.compilertools.net/yacc/index.html`.

[15] Adam Koprowski and Henri Binsztok. TRX: A Formally Verified Parser Interpreter. In *Proceedings of the 19th European Symposium on Programming (ESOP '10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 345–365, 2010.

[16] Georg Kreisel. Interpretation of Analysis by Means of Constructive Functionals of Finite Types. *Constructivity in Mathematics*, pages 101–128, 1959.

[17] Paul Lickman. Parsing With Fixed Points. MSc Thesis, Oxford University, 1995. `https://sites.google.com/a/lickman.com/paul-lickman/`.

[18] Simon Marlow. Happy User Guide, 2009. `https://www.haskell.org/happy/doc/html/index.html`.

[19] The Minlog System. `http://www.minlog-system.de`.

[20] Eugenio Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

[21] Robert Moore. Removing Left Recursion from Context-Free Grammars. In *6th Applied Natural Language Processing Conference*, pages 249–255, 2000.

[22] Lutz Schröder and Till Mossakowski. Monad-Independent Dynamic Logic in HasCasl. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques*, volume 2755 of *Lecture Notes in Computer Science*, pages 425–441. Springer Berlin Heidelberg, 2003.

[23] Lutz Schröder and Till Mossakowski. Monad-Independent Hoare Logic in HasCasl. In *Fundamental Aspects of Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2003.

[24] Helmut Schwichtenberg and Stanley S. Wainer. *Proofs and Computations: Perspectives in Logic*. Cambridge University Press, 2011.

[25] Wouter Swierstra. A Hoare Logic for the State Monad. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 2009.

[26] Jan van Eijck and Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.

[27] Philip Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:4:461–493, 1992.

[28] Philip Wadler. Monads for Functional Programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming: Proceedings of the Bàstad Spring School, May 1995*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.