

Program Extraction Applied to Monadic Parsing (Extended Abstract)

Ulrich Berger, Alison Jones, Monika Seisenberger

May 2, 2014

Abstract

This paper outlines a proof-theoretic approach to developing correct and terminating monadic parsers. Using a modified realisability interpretation, we extract provably correct and terminating programs from formal proofs. If the proof system is proven to be correct, then any extracted program is guaranteed to be so. By extracting parsers, we can ensure that they are correct, complete and terminating for any input. The work is ongoing, and is being carried out in the interactive proof system MINLOG.

1 Introduction

Parsing any language can be problematic. With natural language, the problems are exacerbated by its inherent ambiguities and irregularities. In Linguistics, syntax can be analysed using several different formalisms, none of which is universally agreed upon. At the semantic level there is even less consensus. This sets natural languages apart from more formal languages encountered by parsers. Despite these differences, however, there are key properties that all parsers should ideally have; they should be complete and correct (with respect to a given grammar), and they should terminate with any input. The importance of termination is demonstrated by the well-known left-recursion problem.

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. Monadic parsers can be directly implemented in Haskell. This is the approach taken in much of the literature (see [HM], [HM96], [Wad92b], [Wad92a]). Testing these parsers could check (to a degree) that they work as expected. A more rigorous approach would be to verify them by formalising the implementations and proving that they work as expected. A third option, and the one explored here, is to extract correct parsers directly from the formal proofs. This is carried out using the interactive proof system MINLOG.

2 The Parsing Problem

In the rest of this paper, we use sets to formalise parser relations, which we call parsers for short. Let $\mathbb{P}(X)$ denote the classical powerset of X (i.e. all subsets of X including $\emptyset$ and X). Let $\mathbb{P}_{<\omega}(X) \subseteq \mathbb{P}(X)$ denote its finite subsets. This can be defined inductively as follows:

- $\emptyset \in \mathbb{P}_{<\omega}(X)$
- If $E \in \mathbb{P}_{<\omega}(X)$ and $x \in X$, then $\{x\} \cup E \in \mathbb{P}_{<\omega}(X)$

In the following, let the set of all parsers with result type A be defined as:

$$\mathbb{P}(S \times A)$$

where S is the type of input strings and A is the type of parsing results.

2.1 Formalising the Parsing Problem

For any parser $P \subseteq S \times A$ the parsing problem can be stated as follows: Given a string $s \in S$, 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 $s \in S$, find all pairs (a, r) such that s is equal to $s' \upharpoonright r$ and $(s', a) \in P$. Formally, this partial parsing problem can be represented by a *parsing function* $[P] : S \rightarrow \mathbb{P}(A \times S)$, defined by:

$$[P](s) := \{(a, r) \mid \exists s'. s = s' \upharpoonright r \wedge (s', a) \in P\} \quad (1)$$

I.e., the input string s is 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*:

$$\text{FB}_{S,A}(P) := \forall s \in S. [P](s) \in \mathbb{P}_{<\omega}(S \times A)$$

2.2 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 $\mathbf{p}$ that implements the parsing function $[P]$. Using Haskell notation, $\mathbf{p}$ will have the type:

$$\mathbf{p} :: \text{String} \rightarrow [(a, \text{String})]$$

where $\mathbf{a}$ is the type of representations of parsing results for P . Note that the type of $\mathbf{p}$ is the type of monadic parsers in [HM]. 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 *realisability* – will be

given in Section 4. Essentially, we extract monadic parsers in the style of [HM], but with the additional benefit of a formal proof of correctness, completeness and termination. Instead of building complex parsers from simple ones at the *program* level, we will do so at the *proof* level.

3 Formalising parsers and their properties

3.1 Basic Parsers

In this section, we formalise the return operator and two basic monadic parsers: the failure parser (which fails with any input) and the item parser (which parses one “unit” of input). This is done at the abstract level of sets.

The return operator takes an element and returns a subset of $\mathbb{P}(S \times A)$ without consuming any input:

$$\begin{aligned} \text{return} &: A \rightarrow \mathbb{P}(S \times A) \\ \text{return } x &= \{([], x)\} \end{aligned}$$

As we are defining parsers as sets, the type declarations for fail and item are self-explanatory. Failure essentially defines a lack of parsing results. In set-theoretic terms, this is represented by the empty set:

$$\begin{aligned} \text{fail} &: \mathbb{P}(S \times A) \\ \text{fail} &= \emptyset \end{aligned}$$

The item parser consists of all parsing results such that c is a character (the “unit” for strings) and $[c]$ is a list of such units.

$$\begin{aligned} \text{item} &: \mathbb{P}(S \times \text{Char}) \\ \text{item} &= \{([c], c) \mid c \in \text{Char}\} \end{aligned}$$

The corresponding parser functions are:

$$[\text{return}(x)](s) = \{(x, s)\}$$

$$[\text{fail}](s) = \emptyset$$

$$\begin{aligned} [\text{item}]([]) &= \emptyset \\ [\text{item}](c : cs) &= \{(c, cs)\} \end{aligned}$$

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. As in (1), we do this at the set-theoretic level.

3.2 Union

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

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

The corresponding function is: From this equation, one can easily see that the this is the point-wise union of the two functions:

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

3.3 Composition

The sequencing of two parsers can be defined as follows. Suppose that parser P has the parsing result type A , and that Q_a is a family of parsers with parsing result type B , parametrised over a value of type A . Their composition results in a parser, called the monadic bind, with parsing result type B :

$$P \gg= Q = \{(s \mapsto r, b) \mid \exists a. (s, a) \in P \wedge (r, b) \in Q_a\}$$

The corresponding parsing function is:

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

3.4 The Basic Parsers and Parser Combinators are Finitely Branching

To illustrate that these basic parsers and parser combinators are finitely branching, one must prove the following statements:

1. The return operation is finitely branching.
This follows trivially from the parser function for $\text{return}(x)$.
2. All basic parsers are finitely branching.
This follows trivially from the parser function definitions.
3. If parser P and parser Q (of the same parsing result type) are finitely branching, then so is $P \cup Q$:

$$\text{FB}_{S,A}(P) \rightarrow \text{FB}_{S,A}(Q) \rightarrow \text{FB}_{S,A}(P \cup Q) \quad (2)$$

To prove this, assume that P and Q are finitely branching, and carry out induction on P being finitely branching.

4. If P is finitely branching and has the parsing result type A , and Q_a is a family of finitely branching parsers with the result type B , parametrised over A , then their composed set is finitely branching.

$$\text{FB}_{S,A}(P) \rightarrow \forall a \in A \text{FB}_{S,B}(Q_a) \rightarrow \text{FB}_{S,B}(P \gg= Q) \quad (3)$$

To prove this, one needs to prove that the union of finitely many finite sets is also finite.

3.5 The Parsing Monad

The operations return and $>>=$ endow the type operator $A \mapsto \mathbb{P}(S \times A)$ with the structure of a monad. Monads were connected with computation by [Mog91] and are now widely used in programming. The monad $\mathbb{P}(S \times A)$ is different though related to the parser monad $\mathbf{p} :: \text{String} \rightarrow [(\mathbf{a}, \text{String})]$ of [HM98] or, set-theoretically, the type of parser functions $S \rightarrow \mathbb{P}(A \times S)$ (as a functor of A) which can be given essentially the same monadic structure as the parser monad in [HM98]. More precisely, the mapping sending a parser (relation) P to its parser function $[P]$ is a monad homomorphism.

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

$$\begin{array}{ll} \text{do } \{ a \leftarrow P; Q_a \} & \text{stands for } P >>= \lambda a. Q_a \\ \text{do } \{ P; Q \} & \text{stands for } P >>= \lambda a. Q \text{ (} a \text{ not in } Q \text{)} \end{array}$$

3.6 Recursive parsers

Nearly all non-trivial parsers use some form of recursion. In our setting recursive parsers are relations that are defined inductively, that is, as least fixed points monotone operators. Hence, in order to prove that recursively defined parsers are finitely branching (and hence give rise to a parsing programs) we need to identify conditions on monotone operators $\Phi : \mathbb{P}(S \times A) \rightarrow \mathbb{P}(S \times A)$ that guarantee that their least fixed points are finitely branching.

In the following, we assume that all operators under discussion are *monotone*, that is $P \subseteq Q \rightarrow \Phi(P) \subseteq \Phi(Q)$. This assumption is harmless since in all constructions below it is clear that the involved operators are monotone.

For a parser $P \in \mathbb{P}(S \times A)$ and a natural number n we define the restricted parsers

$$\begin{aligned} P_{\leq n} &:= \{(s, a) \in P \mid |s| \leq n\} \\ P_{< n} &:= \{(s, a) \in P \mid |s| < n\} \end{aligned}$$

We call an operator $\Phi : \mathbb{P}(S \times A) \rightarrow \mathbb{P}(S \times A)$

- *regressive* if $\Phi(P)_{\leq n} \subseteq \Phi(P_{< n})$,
- *finitely branching* if $\Phi(P)$ is finitely branching for all finitely branching $P \in \mathbb{P}(S \times A)$.

One can show that every regressive operator has a unique fixed point, which is finitely branching if in addition the operator is finitely branching.

It is usually easy to prove that a given operator is finitely branching. For example, the operator $\lambda P. P >>= Q$ is finitely branching provided Q_a is for all $a \in A$.

In order to prove regressiveness, parsers that reject the empty string are useful. We call a parser $P \in \mathbb{P}(S \times A)$ *hungry* if $([], a) \notin P$ for any $a \in A$. Similarly a family of parsers $Q : A \rightarrow \mathbb{P}(S \times B)$ is called hungry if Q_a is hungry for all $a \in A$. Now, one can prove, for example, that the operator $\lambda P. P >>= Q$ is regressive if Q_a is hungry for all $a \in A$.

Of course, this leads to the question of how to prove that a parser is hungry. For example, a parser defined by a chain of compositions of parsers is hungry provided one parser in the chain is hungry, while for the union of two parsers to be hungry both parser need to be hungry. Regarding recursively defined parsers one can prove that a regressive operator that preserves hungriness has a hungry fixed point.

The considerations above should be viewed as initial steps towards a library of proof tactics supporting a combinatorial style of proving that a parser is finitely branching which can be automatized.

We illustrate the approach by showing that the finite iteration of a hungry and finitely branching parser is finitely branching. For a parser $P \in \mathbb{P}(S \times A)$ we define inductively the parser $\text{many}(P) \in \mathbb{P}(S \times A^*)$:

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

This definition can be rewritten as the least fixed point of the operator $\Phi : \mathbb{P}(S \times A^*) \rightarrow \mathbb{P}(S \times A^*)$ defined by

$$\Phi(Q) := \{([], [])\} \cup (P \gg= \lambda a. Q \gg= \lambda as. \text{return}(a : as))$$

or, using do-notation,

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

Now, if P is hungry and finitely branching, then Φ is regressive and finitely branching. Hence, its least fixed point, $\text{many}(P)$, is finitely branching. Clearly, $\text{many}(P)$ is not hungry, but its variant

$$\text{many}1(P) := \text{do} \{ a \leftarrow P ; as \leftarrow \text{many}(P) ; \text{return}(a : as) \}$$

is (and still finitely branching).

4 Formalisation and Program Extraction in Minlog

MINLOG([Min], [BBS⁺98]) is a proof system based on a first order natural deduction calculus. It is interactive, using a backwards reasoning style. It 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. The theoretical background of MINLOG is explained in [SW11].

4.1 Formalisation

In the following, we discuss some relevant aspects of the formalisation in MINLOG. So far, the development of the theory was independent from the representation of sets. In the MINLOG formalisation, we decided to treat sets as boolean functions. A set of elements of type α therefore has the type $\alpha \rightarrow \text{Boole}$. If P is a set and a is an element of type α , $P a$ represents the statement $a \in P$. (In our example, α will be instantiated by the type $A \times S$, i.e., pairs of parsing results and strings.)

We define the operations empty set (**Em**), insertion (**Ins**) and union (**Un**) as program constants in MINLOG with appropriate term-rewriting rules. For example, the empty set is a function that returns false for all elements.

With these constants, we can define an inductive predicate **Fin** of arity **alpha=>boole** (where **alpha** is a type variable), containing all finite sets (cf. $\mathbb{P}_{<\omega}(X)$ in Section 2) as follows:

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

Here **x** is a variable of type **alpha** and **p** is a variable of type **alpha=>boole**. It consists of two closure axioms: one for the base case of the empty set, and one for the step case of a finite set with an additional inserted element. 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: $\forall E. E \in \mathbb{P}_{<\omega}(X) \rightarrow \forall x. x \in X \rightarrow \{x\} \cup E \in \mathbb{P}_{<\omega}(X)$

As an example, we can now discuss the proof of a first property of parsers, which essentially states that the union of two finitely branching parsing functions is also finite (see (2)). In MINLOG syntax, we write our initial goal as:

```
(set-goal (parse-formula "allnc q (Fin q ->
                           allnc p (Fin p ->
                               Fin ((Un alpha) p q)))"))
```

We then proceed by carrying out induction on the inductive predicate **Fin p**.

Noteworthy 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).

4.2 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. A consequence of this correspondence is that 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 these proofs, the system uses a modified realisability interpretation (see [Kre59], [SW11] 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 realise inductive definitions.

At the program level, the inductively defined predicate **Fin** corresponds to an algebra **algFin**. 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 realised by the constructor called **cInitFin** of type **(algFin alpha)**. **GenFin** is realised by the constructor:

```
cGenFin:  alpha -> (algFin alpha) -> (algFin alpha)
```

It is worth noting that this corresponds exactly to the standard **cons** constructor for lists. I.e., in general, any proof that a parsing function is finitely branching yields, via program extraction, a program that lists all parsing results.

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

4.3 Extracted Program

From an inductive proof that the union of two finite sets is finite, we can automatically extract the following recursive program in MINLOG, where we have replaced automatically generated variables by more meaningful ones:

```
[q-fin,p-fin]
(Rec algFin alpha=>algFin alpha)p-fin q-fin
([p'-fin,un-p'q-fin,a]
 (CGenFin alpha)un-p'q-fin a)
```

Here, the corner brackets denote lambda abstraction. **p-fin** and **q-fin** are variables of type **(algFin alpha)** representing the finiteness of **p** and **q**. The program runs by recursion on the finiteness of **p**. In the base case, when **p** is empty, the finiteness of the union follows from the finiteness of **q** (**q-fin**). In the step case, where **p** equals **Ins a p'**, **CGenFin** produces a realiser for **Un (Ins a p')q** using **un-p'q-fin** and **a**. Essentially, the program implements the concatenation of lists.

Note that the sets **p** and **q** do not occur in the program. This is due to our use of the non-computational quantifier **allnc** in the proof.

MINLOG will automatically generate a soundness proof stating the correctness of the extracted program. In this case, it makes use of the inductively

defined predicate `FinMR` with two arguments: a term `t` and a set `p`, with the intended meaning that `t` is a realiser for the finiteness of `p`. The soundness theorem states that if we have realisers for the finiteness of `p` and `q`, then the extracted program applied to these realisers realises that the union of `p` and `q` is finite. We show the original output:

```
allnc q,(algFin alpha)^2354(FinMR(algFin alpha)^2354 q -->
allnc p,(algFin alpha)^2355(FinMR(algFin alpha)^2355 p -->
  FinMR((Rec algFin alpha=>algFin alpha)
    (algFin alpha)^2355(algFin alpha)^2354
    ([ (algFin alpha)^2364,(algFin alpha)^2365,a]
      (CGenFin alpha)(algFin alpha)^2365 a))
    ((Un alpha)p q)))
```

5 Conclusion

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.

To our knowledge, this is the first time that program extraction has been applied to parsing algorithms. We are still at the early stages, so it remains to be seen how far we can get with this approach. However, we think that it has good potential.

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 ([BB04], [BB99]), which is facilitated by monads. We also share an interest in the functional programming framework and the lambda calculus [vEU10]. There is further potential for overlap in the area of discourse analysis and semantic inference [BBKdN98]. We welcome suggestions of other possible applications to natural language semantics.

References

- [BB99] 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/>.
- [BB04] 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.
- [BBKdN98] Patrick Blackburn, Johan Bos, Michael Kohlbase, and Hans de Nivelle. Automated Theorem Proving for Natural Language Under-

- standing. In *Proceedings of CADE-15 workshop Problem-solving methodologies with Automated Deduction*, 1998.
- [BBS⁺98] 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.
 - [BMSS11] 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 Crstea, editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 393–399. Springer Berlin Heidelberg, 2011.
 - [BS05] Ulrich Berger and Monika Seisenberger. Applications of inductive definitions and choice principles to program synthesis. In Laura Crosilla and Peter Schuster, editors, *Applications of inductive definitions and choice principles to program synthesis*. Oxford University Press, 2005.
 - [HM] Graham Hutton and Erik Meijer. Functional Pearls: Monadic Parsing in Haskell. <http://eprints.nottingham.ac.uk/223/1/pearl.pdf>.
 - [HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators, 1996. <http://www.cs.nott.ac.uk/~gmh/monparsing.pdf>.
 - [HM98] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.
 - [Kre59] Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics*, pages 101–128, 1959.
 - [Min] The Minlog System. <http://www.minlog-system.de>.
 - [Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
 - [SW11] Helmut Schwichtenberg and Stanley S. Wainer. *Proofs and Computations: Perspectives in Logic*. Cambridge University Press, 2011.
 - [vEU10] Jan van Eijck and Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.
 - [Wad92a] Philip Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:4, 1992.
 - [Wad92b] Philip Wadler. The Essence of Functional Programming, 1992. www.math.pku.edu.cn/teachers/qiuzy/plan/lits/essence.pdf.