Parsing Parses

A Pearl of (Dependently Typed) Programming and Proof

Jason Gross
MIT CSAIL
jgross@mit.edu

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

Abstract

We present a functional parser for arbitrary context-free grammars, together with soundness and completeness proofs, all inside Coq. By exposing the parser in the right way with parametric polymorphism and dependent types, we are able to use the parser to prove its own soundness, and, with a little help from relational parametricity, prove its own completeness, too. Of particular interest is one strange instantiation of the type and value parameters: by parsing *parse trees* instead of strings, we convince the parser to generate its own completeness proof. We conclude with highlights of our experiences iterating through several versions of the Coq development, and some general lessons about dependently typed programming.

1. Introduction

Parsing is one of the fundamental problems of computer science, and in this paper we present an unusual way of implementing and proving the correctness of a general parser for arbitrary context-free grammars. Our parser is implemented in Coq, making liberal use of dependent types; the adventurous reader is invited to browse the included source code. An especially surprising element of our parser is that we reuse the parsing algorithm to generate parts of its own soundness and completeness proofs. That algorithm is phrased with parametric polymorphism, so that we can instantiate it not just to parse strings into parse trees, but also to "parse" parse trees into minimized parse trees (with certain wasteful detours eliminated); the existence of such a minimization algorithm is a key part of our completeness proof.

We begin with an overview of the general setting, and a description of our approach to parsing.

The job of a parser is to decompose a flat list of characters, called a *string*, into a structured tree, called a *parse tree*, on which further operations can be performed. As a simple example, we can parse `"ab"` as an instance of the regular expression $\texttt{(ab)}^*$, giving this parse tree, where we write $\cdot$ for string concatenation.

$$\dfrac{\dfrac{}{\texttt{"a"} \in \texttt{'a'}} \qquad \dfrac{}{\texttt{"b"} \in \texttt{'b'}} \qquad \dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in \texttt{(ab)}^*}}{\dfrac{\texttt{"a"} \cdot \texttt{"b"} \cdot \texttt{""} \in \texttt{ab(ab)}^*}{\texttt{"ab"} \in \texttt{(ab)}^*}}$$

Our parse tree is implicitly constructed from a set of general inference rules for parsing. There is a naive approach to parsing a string s: run the inference rules as a logic program. Several execution orders work: we may proceed bottom-up, by generating all of the strings that are in the language and not longer than s, checking each one for equality with s; or top-down, by splitting s into smaller parts in a way that mirrors the inference rules. In this paper, we present an implementation based on the second strategy, parameterizing over a "splitting oracle" that provides a list of candidate ways to split the string, based on the available inference rules. To be sound, each "split" must be a genuine split of the string; (`"a"`, `"b"`) is not a split of the string `"abc"` nor of the string `"zz"`. To be complete, if any split of the string yields a valid parse, the oracle must give at least one split that also yields a valid parse. Different splitters yield different simple recursive-descent parsers.

We eventually plan to synthesize optimized parsers. We believe that parameterizing the parser over a splitter gives us enough expressiveness to implement essentially all optimizations of interest, while being a sufficiently simple language to make proofs relatively straightforward. For example, to achieve linear parse time on the $\texttt{(ab)}^*$ grammar, we could have a splitter that, when trying to parse $\texttt{'}c_1\texttt{'} \cdot \texttt{'}c_2\texttt{'} \cdot \texttt{s}$ as $\texttt{ab(ab)}^*$, splits the string into ($\texttt{'}c_1\texttt{'}$, $\texttt{'}c_2\texttt{'}$, `s`); and when trying to parse `s` as ϵ, does not split the string at all.

Proving completeness—that our parser succeeds whenever there is a valid parse tree—is conceptually straightforward: trace the algorithm, showing that if the parser returns `false` at a given point, then assuming a corresponding parse tree exists yields a contradiction. The one wrinkle in this approach is that the algorithm, the logic program, is not guaranteed to terminate.

1.1 Infinite regress

Since we have programmed our parser in Coq, our program must be terminating by construction. However, naive recursive-descent parsers do not always terminate!

To see how such parsers can diverge, consider the following example. When defining the grammar $\texttt{(ab)}^*$, perhaps we give the following production rules:

$$\dfrac{s \in \epsilon}{s \in \texttt{(ab)}^*}\ (\epsilon) \qquad \dfrac{s_0 \in \texttt{'a'} \qquad s_1 \in \texttt{'b'}}{s_0 s_1 \in \texttt{(ab)}^*}\ (\texttt{"ab"})$$

$$\dfrac{s_0 \in \texttt{(ab)}^* \qquad s_1 \in \texttt{(ab)}^*}{s_0 s_1 \in \texttt{(ab)}^*}\ (\texttt{(ab)}^*\texttt{(ab)}^*)$$

Now, let us try to parse the string `"ab"` as $\texttt{(ab)}^*$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in (\texttt{ab})^*} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in (\texttt{ab})^*} \qquad \dfrac{\iddots}{\texttt{"ab"} \in (\texttt{ab})^*}}{\texttt{""} \cdot \texttt{"ab"} \in (\texttt{ab})^*}}{\texttt{"ab"} \in (\texttt{ab})^*}}{\texttt{""} \cdot \texttt{"ab"} \in (\texttt{ab})^*}}{\texttt{"ab"} \in (\texttt{ab})^*}$$

Thus, by making a poor choice in how we split strings and choose productions, we can quickly hit an infinite regress.

Assuming we have a function `split` : `String` → `[String` × `String]` which is our splitting oracle, we may write out a potentially divergent parser specialized to this grammar.

```
any_parses : [String × String] → Bool
any_parses [] := false
any_parses (("a", "b") :: _) := true
any_parses ((s₁, s₂) :: rest_splits)
   := (parses s₁ && parses s₂) || any_parses rest_splits
```

```
parses : String → Bool
parses "" := true
parses str := any_parses (split str)
```

If `split` returns `("", "ab")` as the first item in its list when given `"ab"`, then the code given above will diverge in the way demonstrated above with the infinite derivation tree.

1.2 Aborting early

To work around this wrinkle, we keep track of what nonterminals we have not yet tried to parse the current string as, and we abort early if we see a repeat. Note that this strategy only works for grammars with finite sets of nonterminals, in line with most formalizations of context-free grammars. For our example grammar, since there is only one nonterminal, we only need to keep track of the current string. We refactor the above code to introduce a new parameter `prev_s`, recording the previous string we were parsing. We use `s` < `prev_s` to denote the test that `s` is strictly shorter than `prev_s`.

```
any_parses : String → [String × String] → Bool
any_parses _ [] := false
any_parses _ (("a", "b") :: _) := true
any_parses prev_s ((s₁, s₂) :: rest_splits)
   := (s₁ < prev_s && s₂ < prev_s
        && parses s₁ && parses s₂)
       || any_parses prev_s rest_splits
```

```
parses : String → Bool
parses ""  := true
parses str := any_parses str (split str)
```

We can convince Coq that this definition is total via well-founded recursion on the length of the string passed to `parses`. For a more-complicated grammar, we'd need to use a well-founded relation that also included the number of nonterminals not yet tried for this string; we do this in Figure 2 in Subsection 5.2.

With this refactoring, however, completeness is no longer straightforward. We must show that aborting early does not eliminate good parse trees.

We devote the rest of this paper to describing an elegant approach to proving completeness. Ridge [11] carried out a proof about essentially the same algorithm in HOL4, a proof assistant that does not support dependent types. We instead refine our parser to have a more general polymorphic type signature that takes advantage of dependent types, supporting a proof strategy with a different kind of aesthetic appeal. Relational parametricity frees us from worrying about different control flows with different instantiations of the arguments: when care is taken to ensure that the execution of the algorithm does not depend on the values of the arguments, we are guaranteed that all instantiations succeed or fail together. Freed from this worry, we convince our parser to prove its own soundness and completeness by instantiating its arguments correctly.

2. Standard Formal Definitions

Before proceeding, we pause to standardize on terminology and notation for context-free grammars and parsers. In service of clarity for some of our later explanations, we formalize grammars via natural-deduction inference rules, a slightly nonstandard choice.

2.1 Context Free Grammar

A *context-free grammar* consists of *items*, which may be either *terminals* (characters) or *nonterminals*; plus a set of *productions*, each mapping a nonterminal to a sequence of items.

2.1.1 Example: `(ab)*`

The inference rules of the regular-expression grammar $(\texttt{ab})^*$ are:
Terminals:

$$\dfrac{}{\texttt{"a"} \in \texttt{'a'}} \qquad \dfrac{}{\texttt{"b"} \in \texttt{'b'}}$$

Productions and nonterminals:

$$\dfrac{s \in \epsilon}{s \in (\texttt{ab})^*} \qquad \dfrac{}{\texttt{""} \in \epsilon}$$

$$\dfrac{s_0 \in \texttt{'a'} \qquad s_1 \in \texttt{'b'} \qquad s_2 \in (\texttt{ab})^*}{s_0 s_1 s_2 \in (\texttt{ab})^*}$$

2.2 Parse Trees

A string `s` *parses* as:

- a given terminal `ch` iff `s` = `'ch'`.
- a given sequence of items $\texttt{x}_i$ iff `s` splits into a sequence of strings $\texttt{s}_i$, each of which parses as the corresponding item $\texttt{x}_i$.
- a given nonterminal `nt` iff `s` parses as one of the item sequences that `nt` maps to under the set of productions.

We may define mutually inductive dependent type families of `ParseTreeOf`s and `ParseItemsTreeOf`s for a given grammar:

$$\texttt{ParseTreeOf} : \texttt{Item} \to \texttt{String} \to \textbf{Type}$$
$$\texttt{ParseItemsTreeOf} : [\texttt{Item}] \to \texttt{String} \to \textbf{Type}$$

For any terminal character `ch`, we have the constructor

$$(\texttt{'ch'}) : \texttt{ParseTreeOf}\ \texttt{'ch'}\ \texttt{"ch"}$$

For any production `rule` mapping a nonterminal `nt` to a sequence of items `its`, and any string `s`, we have this constructor:

$$(\texttt{rule}) : \texttt{ParseItemsTreeOf}\ \texttt{its}\ \texttt{s} \to \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s}$$

We have the following two constructors of `ParseItemsTree`. In writing the type of the latter constructor, we adopt a common space-saving convention where we assume that all free variables are quantified implicitly with dependent function (Π) types. We also write constructors in the form of schematic natural-deduction rules, since that notation will be convenient to use later on.

$$\overline{\texttt{""} \in \epsilon} : \texttt{ParseItemsTreeOf [] ""}$$

$$\frac{s_1 \in \texttt{it} \qquad s_2 \in \texttt{its}}{s_1 s_2 \in \texttt{it} :: \texttt{its}} : \texttt{ParseTreeOf it } s_1 \to \texttt{ParseItemsTreeOf its } s_2 \to \texttt{ParseItemsTreeOf (it :: its) } s_1 s_2$$

For brevity, we will sometimes use the notation $\overline{\texttt{s} \in \texttt{X}}$ to denote both `ParseTreeOf X s` and `ParseItemsTreeOf X s`, relying on context to disambiguate based on the type of `X`. Additionally, we will sometimes fold the constructors of `ParseItemsTreeOf` into the (`rule`) constructors of `ParseTreeOf`, to mimic the natural-deduction trees.

We also define a type of all parse trees, independent of the string and item, as this dependent-pair (Σ) type, using set-builder notation; we use `ParseTree` to denote the type

$$\{(\texttt{nt}, \texttt{s}) : \texttt{Nonterminal} \times \texttt{String} \mid \texttt{ParseTreeOf nt s}\}$$

2.3 Parsers, Soundness, and Completeness

Parsers come in a number of flavors. The simplest flavor is the *recognizer*, which simply says whether or not there exists a parse tree of a given string for a given nonterminal; it returns Booleans. There is also a richer flavor of parser that returns inhabitants of `option ParseTree`.

For any recognizer `has_parse` : `Nonterminal` $\to$ `String` $\to$ `Bool`, we may ask whether it is *sound*, meaning that when it returns `true`, there is always a parse tree; and *complete*, meaning that when there is a parse tree, it always returns `true`. We may express these properties as theorems (alternatively, dependently typed functions) with the following type signatures:

$$\begin{aligned}
\texttt{has\_parse\_sound} : &(\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{has\_parse nt s} = \texttt{true}\\
&\to \texttt{ParseTreeOf nt s}\\
\texttt{has\_parse\_complete} : &(\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{ParseTreeOf nt s}\\
&\to \texttt{has\_parse nt s} = \texttt{true}
\end{aligned}$$

For any parser

$$\texttt{parse} : \texttt{Nonterminal} \to \texttt{String} \to \texttt{option ParseTree},$$

we may also ask whether it is sound and complete, leading to theorems with the following type signatures, using $\texttt{p}_1$ to denote the first projection of `p`:

$$\begin{aligned}
\texttt{parse\_sound} : &(\texttt{nt} : \texttt{Nonterminal})\\
&\to (\texttt{s} : \texttt{String})\\
&\to (\texttt{p} : \texttt{ParseTree})\\
&\to \texttt{parse nt s} = \texttt{Some p}\\
&\to \texttt{p}_1 = (\texttt{nt}, \texttt{s})\\
\texttt{parse\_complete} : &(\texttt{nt} : \texttt{Nonterminal})\\
&\to (\texttt{s} : \texttt{String})\\
&\to \texttt{ParseTreeOf nt s}\\
&\to \texttt{parse nt s} \neq \texttt{None}
\end{aligned}$$

Since we are programming in Coq, this separation into code and proof actually makes for more awkward type assignments. We also have the option of folding the soundness and completeness conditions into the types of the code. For instance, the following type captures the idea of a sound and complete parser returning parse trees, using the type constructor $+$ for disjoint union (i.e., sum or variant type):

$$\begin{aligned}
\texttt{parse} : &(\texttt{nt} : \texttt{Nonterminal})\\
&\to (\texttt{s} : \texttt{String})\\
&\to \texttt{ParseTreeOf nt s} + (\texttt{ParseTreeOf nt s} \to \bot)
\end{aligned}$$

That is, given a nonterminal and a string, `parse` either returns a valid parse tree, or returns a *proof* that the existence of any parse tree is *contradictory* (i.e., implies $\bot$, the empty type). Our implementation follows this dependently typed style. Our main goal in the project was to arrive at a `parse` function of just this type, generic in an arbitrary choice of context-free grammar, implemented and proven correct in an elegant way.

3. Proving Completeness: Conceptual Approach

Recall from Subsection 1.2 that the essential difficulty with proving completeness is dealing with the cases where our parser aborts early; we must show that doing so does not eliminate good parse trees.

The key is to define an intermediate type, that of "minimal parse trees." A "minimal" parse tree is simply a parse tree in which the same (string, nonterminal) pair does not appear more than once in any path of the tree. Defining this type allows us to split the completeness problem in two; we can show separately that every parse tree gives rise to a minimal parse tree, and that having a minimal parse tree in hand implies that our parser succeeds (returns `true` or `Some _`).

Our dependently typed parsing algorithm subsumes the soundness theorem, the minimization of parse trees, and the proof that having a minimal parse tree implies that our parser succeeds. We write one parametrically polymorphic parsing function that supports all three modes, plus the several different sorts of parsers (recognizers, generating parse trees, running semantic actions). That level of genericity requires us to be flexible in which type represents "strings," or inputs to parsers. We introduce a parameter that is often just the normal `String` type, but which needs to be instantiated as the type of *parse trees themselves* to get a proof of parse tree minimizability. That is, we "parse" parse trees to minimize them, reusing the same logic that works for the normal parsing problem.

Before presenting our algorithm's interface, we will formally define and explain minimal parse trees, which will provide motivation for the type signatures of our parser's arguments.

4. Minimal Parse Trees: Formal Definition

In order to make tractable the second half of the completeness theorem, that having a minimal parse tree implies that parsing succeeds, it is essential to make the inductive structure of minimal parse trees mimic precisely the structure of the parsing algorithm. A minimal parse tree thus might better be thought of as a parallel trace of parser execution.

As in Subsection 2.2, we define mutually inductive type families of `MinParseTreeOf`s and `MinItemsTreeOf`s for a given grammar. Because our parser proceeds by well-founded recursion on the length of the string and the list of nonterminals not yet attempted for that string, we must include both of these in the types. Let us call the initial list of all nonterminals $\texttt{unseen}_0$.

$$\begin{aligned}
\texttt{MinParseTreeOf} : &\texttt{String} \to [\texttt{Nonterminal}]\\
&\to \texttt{Item} \to \texttt{String} \to \textbf{Type}\\
\texttt{MinItemsTreeOf} : &\texttt{String} \to [\texttt{Nonterminal}]\\
&\to [\texttt{Item}] \to \texttt{String} \to \textbf{Type}
\end{aligned}$$

Much as in the case of parse trees, for any terminal character `ch`, any string $\texttt{s}_0$, and any list of nonterminals `unseen`, we have the constructor

$$\texttt{min\_parse}_{\texttt{'ch'}} : \texttt{MinParseTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{'ch'}\ \texttt{"ch"}$$

For any production $\texttt{rule}$ mapping a nonterminal $\texttt{nt}$ to a sequence of items $\texttt{its}$, any string $\texttt{s}_0$, any list of nonterminals $\texttt{unseen}$, and any string $\texttt{s}$, we have two constructors, corresponding to the two ways of progressing with respect to the well-founded relation. Letting $\texttt{unseen}' := \texttt{unseen} - \{\texttt{nt}\}$, we have the following, where we interpret the $<$ relation on strings in terms of lengths.

$$\begin{aligned}
(\texttt{rule})_< : {} & \texttt{s} < \texttt{s}_0 \\
& \to \texttt{MinItemsTreeOf}\ \texttt{s}\ \texttt{unseen}_0\ \texttt{its}\ \texttt{s} \\
& \to \texttt{MinParseTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{nt}\ \texttt{s} \\
(\texttt{rule})_= : {} & \texttt{s} = \texttt{s}_0 \\
& \to \texttt{nt} \in \texttt{unseen} \\
& \to \texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{unseen}'\ \texttt{its}\ \texttt{s} \\
& \to \texttt{MinParseTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{nt}\ \texttt{s}
\end{aligned}$$

In the first case, the length of the string has decreased, so we may reset the list of not-yet-seen nonterminals, as long as we reset the base of well-founded recursion $\texttt{s}_0$ at the same time. In the second case, the length of the string has not decreased, so we require that we have not yet seen this nonterminal, and we then remove it from the list of not-yet-seen nonterminals.

Finally, for any string $\texttt{s}_0$ and any list of nonterminals $\texttt{unseen}$, we have the following two constructors of `MinItemsTreeOf`.

$$\begin{aligned}
\texttt{min\_parse}_{[]} : {} & \texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{[]}\ \texttt{""} \\
\texttt{min\_parse}_{::} : {} & \texttt{s}_1\texttt{s}_2 \le \texttt{s}_0 \\
& \to \texttt{MinParseTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{it}\ \texttt{s}_1 \\
& \to \texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{its}\ \texttt{s}_2 \\
& \to \texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{unseen}\ (\texttt{it} :: \texttt{its})\ \texttt{s}_1\texttt{s}_2
\end{aligned}$$

The requirement that $\texttt{s}_1\texttt{s}_2 \le \texttt{s}_0$ in the second case ensures that we are only making well-founded recursive calls.

Once again, for brevity, we will sometimes use the notation $\overline{\texttt{s} \in \texttt{X}}^{<(\texttt{s}_0,\texttt{v})}$ to denote both $\texttt{MinParseTreeOf}\ \texttt{s}_0\ \texttt{v}\ \texttt{X}\ \texttt{s}$ and $\texttt{MinItemsTreeOf}\ \texttt{s}_0\ \texttt{v}\ \texttt{X}\ \texttt{s}$, relying on context to disambiguate based on the type of $\texttt{X}$. Additionally, we will sometimes fold the constructors of `MinItemsTreeOf` into the two $(\texttt{rule})$ constructors of `MinParseTreeOf`, to mimic the natural-deduction trees.

5. Parser Interface

Roughly speaking, we read the interface of our general parser off from the types of the constructors for minimal parse trees. Every constructor leads to one parameter passed to the parser, much as one derives the types of general "fold" functions for arbitrary inductive datatypes. For instance, lists have constructors `nil` and `cons`, so a fold function for lists has arguments corresponding to `nil` (initial accumulator) and `cons` (step function). The situation for the type of our parser is similar, though we need parallel success (managed to parse the string) and failure (could prove that no parse is possible) parameters for each constructor of minimal parse trees.

The type signatures in the interface are presented in Figure 1. We explain each type one by one, presenting various instantiations as examples. Note that the interface we actually implemented is also parameterized over a type of `String`s, which we will instantiate with parse trees later in this paper. The interface we present here fixes `String`, for conciseness.

Since we want to be able to specialize our parser to return either `Bool` or `option ParseTree`, we want to be able to reuse our soundness and completeness proofs for both. Our strategy for generalization is to parameterize on dependent type families for "success" and "failure", so we can use relational parametricity to ensure that all instantiations of the parser succeed or fail together. The parser has the rough type signature

$$\texttt{parse} : \texttt{Nonterminal} \to \texttt{String} \to \texttt{T}_{\texttt{success}} + \texttt{T}_{\texttt{failure}}.$$

To instantiate the parser as a Boolean recognizer, we instantiate everything trivially; we use the fact that $\top + \top \cong \texttt{Bool}$. Just to show how trivial everything is, here is a precise instantiation of the parser, still parameterized over the initial list of nonterminals and the splitter, where $\top$ is the one constructor of the one-element type $\top$:

$$\begin{aligned}
&\texttt{T}_{\texttt{success}}\ \_\ \_\ \_ := \top \\
&\texttt{T}_{\texttt{failure}}\ \_\ \_\ \_ := \top
\end{aligned}$$

$$\begin{aligned}
&\texttt{terminal\_success}\ \_\ \_\ \_ := () \\
&\texttt{terminal\_failure}\ \_\ \_\ \_\ \_\ \_ := () \\
&\texttt{nil\_success}\ \_\ \_ := () \\
&\texttt{nil\_failure}\ \_\ \_\ \_\ \_ := () \\
&\texttt{cons\_success}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := () \\
&\texttt{cons\_failure}\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := () \\
&\texttt{production\_success}_<\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := () \\
&\texttt{production\_success}_=\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := () \\
&\texttt{production\_failure}_<\ \_\ \_\ \_\ \_\ \_\ \_ := () \\
&\texttt{production\_failure}_=\ \_\ \_\ \_\ \_\ \_\ \_ := () \\
&\texttt{production\_failure}_{\notin}\ \_\ \_\ \_\ \_\ \_\ \_ := ()
\end{aligned}$$

To instantiate our parser so that it returns `option ParseTree` (rather, the dependently typed flavor, `ParseTreeOf`), we take advantage of the isomorphism $T + \top \cong \texttt{option}\ T$. We show only the `success` instantiations, as the `failure` ones are identical with the Boolean recognizer. For readability of the code, we write schematic natural-deduction proof trees inline.

$$\texttt{T}_{\texttt{success}}\ \_\ \_\ (\overline{\texttt{s} \in \texttt{X}}) := \overline{\texttt{s} \in \texttt{X}}$$

$$\begin{aligned}
&\texttt{terminal\_success}\ \_\ \_\ \texttt{ch} := (\texttt{'ch'}) \\
&\texttt{nil\_success}\ \_\ \_ := \overline{\texttt{""} \in \epsilon} \\
&\texttt{cons\_success}\ \_\ \_\ \texttt{it}\ \texttt{its}\ \texttt{s}_1\ \texttt{s}_2\ \_\ \texttt{d}_1\ \texttt{d}_2 := \dfrac{\dfrac{\texttt{d}_1}{\texttt{s}_1 \in \texttt{it}} \qquad \dfrac{\texttt{d}_2}{\texttt{s}_2 \in \texttt{its}}}{\texttt{s}_1\texttt{s}_2 \in \texttt{it} :: \texttt{its}} \\
&\texttt{production\_success}_<\ \_\ \_\ \texttt{it}\ \texttt{nt}\ \texttt{s}\ \_\ \texttt{p}\ \texttt{d} := \dfrac{\dfrac{\texttt{d}}{\texttt{s} \in \texttt{its}}}{\texttt{s} \in \texttt{nt}}\ (\texttt{p}) \\
&\texttt{production\_success}_=\ \_\ \_\ \texttt{it}\ \texttt{nt}\ \texttt{s}\ \_\ \texttt{p}\ \texttt{d} := \dfrac{\dfrac{\texttt{d}}{\texttt{s} \in \texttt{its}}}{\texttt{s} \in \texttt{nt}}\ (\texttt{p})
\end{aligned}$$

What remains is to instantiate the parser in such a way that proving completeness is trivial. The simpler of our two tasks is to show that when the parser fails, no minimal parse tree exists. Hence we instantiate the types as follows, where $\bot$ is the empty type (equivalently, the false proposition).

$$\begin{aligned}
&\texttt{T}_{\texttt{success}}\ \_\ \_\ \_ := \top \\
&\texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{X}}) := \left(\overline{\texttt{s} \in \texttt{X}}^{<(\texttt{s}_0,\texttt{unseen})}\right) \to \bot
\end{aligned}$$

We use `ParseQuery` to denote the type of all propositions like "`"a" ∈ 'a'`"; a query consists of a string and a grammar rule the string might be parsed into. We use the same notation for `ParseQuery` and `ParseTree` inhabitants. All `*_success` and `*_failure` type signatures are implicitly parameterized over a string $\texttt{s}_0$ and a list of nonterminals `unseen`. We assume we are given $\texttt{unseen}_0 : \texttt{[Nonterminal]}$.

$$
\begin{aligned}
\texttt{T}_{\texttt{success}},\ \texttt{T}_{\texttt{failure}} &: \texttt{String} \to \texttt{[Nonterminal]} \to \texttt{ParseQuery} \to \textbf{Type}\\
\texttt{split} &: \texttt{String} \to \texttt{[Nonterminal]} \to \texttt{ParseQuery} \to \texttt{[String} \times \texttt{String]}\\
\texttt{split\_sound} &: \forall\ \texttt{s}_0\ \texttt{unseen}\ \texttt{query}\ \texttt{s}_1\ \texttt{s}_2,\ (\texttt{s}_1, \texttt{s}_2) \in \texttt{split}\ \texttt{s}_0\ \texttt{unseen}\ \texttt{query} \to \texttt{s}_1\texttt{s}_2 = \texttt{string\_part}\ \texttt{query}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{terminal\_success} &: (\texttt{ch} : \texttt{Char}) \to \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{"ch"} \in \texttt{'ch'}})\\
\texttt{terminal\_failure} &: (\texttt{ch} : \texttt{Char}) \to (\texttt{s} : \texttt{String}) \to \texttt{s} \neq \texttt{"ch"} \to \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{'ch'}})\\
\texttt{nil\_success} &: \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{""} \in \epsilon})\\
\texttt{nil\_failure} &: (\texttt{s} : \texttt{String}) \to \texttt{s} \neq \texttt{""} \to \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \epsilon})
\end{aligned}
$$

$$
\begin{aligned}
\texttt{cons\_success} &: (\texttt{it} : \texttt{Item}) \to (\texttt{its} : \texttt{[Item]}) \to (\texttt{s}_1 : \texttt{String}) \to (\texttt{s}_2 : \texttt{String})\\
&\to \texttt{s}_1\texttt{s}_2 \le \texttt{s}_0\\
&\to \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s}_1 \in \texttt{it}})\\
&\to \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s}_2 \in \texttt{its}})\\
&\to \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s}_1\texttt{s}_2 \in \texttt{it} :: \texttt{its}})\\
\texttt{cons\_failure} &: (\texttt{it} : \texttt{Item}) \to (\texttt{its} : \texttt{[Item]}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{s} \le \texttt{s}_0\\
&\to \big(\forall\ (\texttt{s}_1, \texttt{s}_2) \in \texttt{split}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{it} :: \texttt{its}}),\\
&\qquad \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s}_1 \in \texttt{it}}) + \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s}_2 \in \texttt{its}})\big)\\
&\to \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{it} :: \texttt{its}})
\end{aligned}
$$

$$
\begin{aligned}
\texttt{production\_success}_{<} &: (\texttt{its} : \texttt{[Item]}) \to (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{s} < \texttt{s}_0\\
&\to (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its})\\
&\to \texttt{T}_{\texttt{success}}\ \texttt{s}\ \texttt{unseen}_0\ (\overline{\texttt{s} \in \texttt{its}})\\
&\to \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{nt}})\\
\texttt{production\_success}_{=} &: (\texttt{its} : \texttt{[Item]}) \to (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{nt} \in \texttt{unseen}\\
&\to (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its})\\
&\to \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ (\texttt{unseen} - \{\texttt{nt}\})\ (\overline{\texttt{s} \in \texttt{its}})\\
&\to \texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{nt}})\\
\texttt{production\_failure}_{<} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{s} < \texttt{s}_0\\
&\to \big(\forall\ (\texttt{its} : \texttt{[Item]})\ (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}),\ \texttt{T}_{\texttt{failure}}\ \texttt{s}\ \texttt{unseen}_0\ (\overline{\texttt{s} \in \texttt{its}})\big)\\
&\to \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{nt}})\\
\texttt{production\_failure}_{=} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{s} = \texttt{s}_0\\
&\to \big(\forall\ (\texttt{its} : \texttt{[Item]})\ (\texttt{p} : \text{a production mapping } \texttt{nt} \text{ to } \texttt{its}),\ \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ (\texttt{unseen} - \{\texttt{nt}\})\ (\overline{\texttt{s} \in \texttt{its}})\big)\\
&\to \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{nt}})\\
\texttt{production\_failure}_{\notin} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String})\\
&\to \texttt{s} = \texttt{s}_0\\
&\to \texttt{nt} \notin \texttt{unseen}\\
&\to \texttt{T}_{\texttt{failure}}\ \texttt{s}_0\ \texttt{unseen}\ (\overline{\texttt{s} \in \texttt{nt}})
\end{aligned}
$$

Figure 1: The dependently typed interface of our parser

Using $\lightning$ to denote deriving a contradiction, we can unenlighteningly instantiate the arguments as

$$
\begin{aligned}
&\texttt{terminal\_success}\ \_\ \_\ \_ := ()\\
&\texttt{terminal\_failure}\ \_\ \_\ \_\ \_\ \_ := \lightning\\
&\texttt{nil\_success}\ \_\ \_ := ()\\
&\texttt{nil\_failure}\ \_\ \_\ \_\ \_ := \lightning\\
&\texttt{cons\_success}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := ()\\
&\texttt{cons\_failure}\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := \lightning\\
&\texttt{production\_success}_{<}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := ()\\
&\texttt{production\_success}_{=}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_ := ()\\
&\texttt{production\_failure}_{<}\ \_\ \_\ \_\ \_\ \_\ \_ := \lightning\\
&\texttt{production\_failure}_{=}\ \_\ \_\ \_\ \_\ \_\ \_ := \lightning\\
&\texttt{production\_failure}_{\not\in}\ \_\ \_\ \_\ \_\ \_\ \_ := \lightning
\end{aligned}
$$

A careful inspection of the proofy arguments to each `failure` case will reveal that there is enough evidence to derive the appropriate contradiction. For example, the $\texttt{s} \neq \texttt{""}$ hypothesis of `nil_failure` contradicts the equalities implied by the type signature of $\texttt{min\_parse}_{[]}$, and the use of `[]` contradicts the equality implied by the use of `it::its` in the type signature of $\texttt{min\_parse}_{[]}$. Similarly, the $\texttt{s} \neq \texttt{"ch"}$ hypothesis of `terminal_failure` contradicts the equality implied by the usage of the single identifier `ch` in two different places in the type signature of $\texttt{min\_parse}_{\texttt{'ch'}}$.

5.1 Parsing Parses

We finally come to the most twisty part of the parser: parsing parse trees. Recall that our parser definition is polymorphic in a choice of `String` type. We proceed with the straw-man solution of literally passing in parse trees as strings to be parsed, such that parsing generates *minimal* parse trees, as introduced in Section 3 and defined formally in Section 4. Intuitively, we run a top-down traversal of the tree, pausing at each node before descending to its children. During that pause, we *eliminate one level of wastefulness*: if the parse tree is proving $\overline{\texttt{s} \in \texttt{X}}$, we look for any subtrees also proving $\overline{\texttt{s} \in \texttt{X}}$. If we find any, we replace the original tree with *the smallest duplicative subtree*. If we do not find any, we leave the tree unchanged. In either case, we then descend into "parsing" each subtree.

We define a function `deloop` to perform the one step of eliminating waste:

$$\texttt{deloop} : \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} \rightarrow \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s}$$

This transformation is straightforward to define by structural recursion.

To implement all of the generic parameters of the parser, we must actually augment the result type of `deloop` with stronger types. Define the predicate $\texttt{Unloopy}(t)$ on parse trees t to mean that, where the root node of t proves $\overline{\texttt{s} \in \texttt{nt}}$, for every subtree proving $\overline{\texttt{s} \in \texttt{nt}'}$ (same string, possibly different nonterminal), (1) $\texttt{nt}'$ is in the set of allowed nonterminals, `unseen`, associated to the overall tree with dependent types, and (2) if this is not the root node, then $\texttt{nt}' \neq \texttt{nt}$.

We augment the return type of `deloop`, writing:

$$\{t : \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} \mid \texttt{Unloopy}(t)\}.$$

We instantiate the generic "string" type parameter of the general parser with this type family, so that, in implementing the different parameters to pass to the parser, we have the property available to us.

Another key ingredient is the "string" splitter, which naturally breaks a parse tree into its child trees. We define it like so:

$$
\begin{aligned}
&\texttt{split}\ \_\ \_\ (\overline{\texttt{s} \in \texttt{it} :: \texttt{its}}) :=\\
&\quad \textbf{case}\ \texttt{parse\_tree\_data}\ \texttt{s}\ \textbf{of}\\
&\quad\ \ \Big|\ \dfrac{\dfrac{\texttt{p}_1}{\texttt{s}_1 \in \texttt{it}} \qquad \dfrac{\texttt{p}_2}{\texttt{s}_2 \in \texttt{its}}}{\texttt{s}_1\texttt{s}_2 \in \texttt{it} :: \texttt{its}} \rightarrow \big[(\texttt{deloop}\ \texttt{p}_1, \texttt{deloop}\ \texttt{p}_2)\big]\\
&\quad\ \ \Big|\ \_ \rightarrow \lightning\\
&\texttt{split}\ \_\ \_\ \_ := []
\end{aligned}
$$

Note that we use `it` and `its` nonlinearly; the pattern only binds if its `it` and `its` match those passed as arguments to `split`. We thus return a nonempty list only if the query is about a nonempty sequence of items. Because we use dependent types to enforce the requirement that the parse tree associated with a string match the query we are considering, we can derive contradictions in the non-matching cases.

This splitter satisfies two important properties. First, it never returns the empty list on a parse tree whose list of productions is nonempty; call this property *nonempty preservation*. Second, it preserves `Unloopy`. We use both facts in the other parameters to the generic parser (and we leave their proofs as exercises for the reader—Coq solutions may be found in our source code).

Now recall that our general parser always returns a type of the form $\texttt{T}_{\texttt{success}} + \texttt{T}_{\texttt{failure}}$, for some $\texttt{T}_{\texttt{success}}$ and $\texttt{T}_{\texttt{failure}}$. We want our tree minimizer to return just the type of minimal trees. However, we can take advantage of the type isomorphism $T + \bot \cong T$ and instantiate $\texttt{T}_{\texttt{failure}}$ with $\bot$, the uninhabited type; and then apply a simple fix-up wrapper on top. Thus, we instantiate the general parser like so:

$$
\begin{aligned}
&\texttt{T}_{\texttt{success}}\ \texttt{s}_0\ \texttt{unseen}\ \ (\texttt{d} : \overline{\texttt{s} \in \texttt{X}}) := \overline{\texttt{s} \in \texttt{X}}^{\,<(\texttt{s}_0, \texttt{unseen})}\\
&\texttt{T}_{\texttt{failure}}\ \_\ \_\ \_ := \bot
\end{aligned}
$$

The `success` cases are instantiated in an essentially identical way to the instantiation we used to get `option ParseTree`. The `terminal_failure` and `nil_failure` cases provide enough information ($\texttt{s} \neq \texttt{"ch"}$ and $\texttt{s} \neq \texttt{""}$, respectively) to derive $\bot$ from the existence of the appropriately typed parse tree. In the `cons_failure` case, we make use of the splitter's *nonempty preservation* behavior, after which all that remains is $\bot + \bot \rightarrow \bot$, which is trivial. In the $\texttt{production\_failure}_{<}$ and $\texttt{production\_failure}_{=}$ cases, it is sufficient to note that every nonterminal is mapped by some production to some sequence of items. Finally, to instantiate the $\texttt{production\_failure}_{\not\in}$ case, we need to appeal to the `Unloopy`-ness of the tree to deduce that $\texttt{nt} \in \texttt{unseen}$. Then we can derive $\bot$ from the hypothesis that $\texttt{nt} \notin \texttt{unseen}$, and we are done.

We instantiate the general parser with an input type that requires `Unloopy`, so our final tree minimizer is really the composition of the instantiated parser with `deloop`, ensuring that invariant as we kick off the recursion.

5.2 Example

In Subsection 1.1, we defined an ambiguous grammar for $\texttt{(ab)}^*$ which led our naive parser to diverge. We will walk through the minimization of the following parse tree of `"abab"` into this grammar. For reference, Figure 2 contains the fully general implementation of our parser, modulo type signatures.

For reasons of space, define $\overline{T}$ to be the parse tree

$$
\dfrac{\dfrac{\dfrac{}{\texttt{""} \in \epsilon}}{\texttt{""} \in \texttt{(ab)}^*} \qquad \dfrac{\dfrac{\dfrac{}{\texttt{"a"} \in \texttt{'a'}} \qquad \dfrac{}{\texttt{"b"} \in \texttt{'b'}}}{\texttt{"a"} \cdot \texttt{"b"} \in \texttt{(ab)}^*}}{\texttt{"ab"} \in \texttt{(ab)}^*}}{\texttt{""} \cdot \texttt{"ab"} \in \texttt{(ab)}^*}\ (\texttt{(ab)}^*\texttt{(ab)}^*)
$$

```
parse nt s := parse′ ($s_0$ := s) (unseen := $\text{unseen}_0$) ($\overline{s \in nt}$)

parse′ ($\overline{\texttt{"ch"} \in \texttt{'ch'}}$) := inl terminal_success
parse′ ($\overline{\_ \in \texttt{'ch'}}$) := inr (terminal_failure $\not\downarrow$)
parse′ ($\overline{\texttt{""} \in \epsilon}$) := inl nil_success
parse′ ($\overline{\_ \in \epsilon}$) := inr (nil_failure $\not\downarrow$)
parse′ ($\overline{s \in it :: its}$) :=
    case any_parse it its (split ($\overline{s \in it :: its}$)) of
     | inl ret  →  inl ret
     | inr ret  →  inr (cons_failure _ ret)
parse′ ($\overline{s \in nt}$) :=
    if s < $s_0$
    then if (parse′ ($s_0$ := s) (unseen := $\text{unseen}_0$) ($\overline{s \in its}$)) succeeds returning d
                              for any production p mapping nt to its
         then inl (production_success$_<$ _ p d)
         else inr (production_failure$_<$ _ _)
    else if nt ∈ unseen
         then if (parse′ (unseen := unseen − {nt}) ($\overline{s \in its}$)) succeeds returning d
                              for any production p mapping nt to its
              then inl (production_success$_=$ _ p d)
              else inr (production_failure$_=$ _ _)
         else inr (production_failure$_{\not\in}$ _ _)

any_parse it its [] := inr ($\lambda$ _ : (_ ∈ []). $\not\downarrow$)
any_parse it its (($s_1$, $s_2$) :: xs) :=
    case parse′ ($\overline{s_1 \in it}$), parse′ ($\overline{s_2 \in its}$), any_parse it its xs of
     | inl $\text{ret}_1$, inl $\text{ret}_2$, _        →  inl (cons_success _ $\text{ret}_1$ $\text{ret}_2$)
     | _      , _      , inl ret′  →  inl ret′
     | $\text{ret}_1$    , $\text{ret}_2$    , inr ret′  →  inr _
```

where the hole on the last line constructs a proof of

$$\forall\, (s_1', s_2') \in ((s_1, s_2) :: xs)\,,\;\; \mathrm{T}_{\texttt{failure}}\ \_\ \_\ (\overline{s_1' \in it}) + \mathrm{T}_{\texttt{failure}}\ \_\ \_\ (\overline{s_2' \in its})$$

by using ret′ directly when $(s_1', s_2') \in xs$, and using whichever one of ret_1 and ret_2 is on the right when $(s_1', s_2') = (s_1, s_2)$. While straightforward, the use of sum types makes it painfully verbose without actually adding any insight; we prefer to elide the actual term.

Figure 2: Pseudo-Implementation of our parser. We take the convention that dependent indices to functions (e.g., unseen) are implicit.

Then we consider minimizing the parse tree:

$$\dfrac{\dfrac{\dfrac{\overline{T}}{\texttt{"ab"} \in \texttt{(ab)}^*} \qquad \dfrac{\overline{T}}{\texttt{"ab"} \in \texttt{(ab)}^*}}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*}\ (\texttt{(ab)}^*\texttt{(ab)}^*)}{\texttt{"abab"} \in \texttt{(ab)}^*}$$

Letting $\overline{T'_m}$ denote the same tree as $\overline{T'}$, but constructed as a `MinParseTree` rather than a `ParseTree`, the tree we will end up with is:

$$\dfrac{\dfrac{\dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]}) \qquad \dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]})}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[]})}{\texttt{"abab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[(ab)}^*\texttt{]})$$

To begin, we call `parse`, passing in the entire tree as the string, and $\texttt{(ab)}^*$ as the nonterminal. To transform the tree into one that satisfies `Unloopy`, the first thing `parse` does is call `deloop` on our tree. In this case, `deloop` is a no-op; it promotes the deepest non-root nodes labeled with ($\texttt{"abab"} \in \texttt{(ab)}^*$), of which there are none.

We then take the following execution steps, starting with $\texttt{unseen} := \texttt{unseen}_0 := \texttt{[(ab)}^*\texttt{]}$, the singleton list containing the only nonterminal, and $\texttt{s}_0 := \texttt{"abab"}$.

1. We first ensure that we are not in an infinite loop. We check if $\texttt{s} < \texttt{s}_0$ (it is not, for they are both equal to `"abab"`), and then check if our current nonterminal, $\texttt{(ab)}^*$, is in `unseen`. Since the second check succeeds, we remove $\texttt{(ab)}^*$ from `unseen`; calls made by this stack frame will pass `[]` for `unseen`.
2. We may consider only the productions for which the parse tree associated to the string is well-typed; we will describe the headaches this seemingly innocuous simplification caused us in Subsection 7.2. The only such production in this case is the one that lines up with the production used in the parse tree, labeled $\texttt{(ab)}^*\texttt{(ab)}^*$.
3. We invoke `split` on our parse tree.
   (a) The `split` that we defined then invokes `deloop` on the two copies of the parse tree

   $$\dfrac{\overline{T}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

   Since there are non-root nodes labeled with ($\texttt{"ab"} \in \texttt{(ab)}^*$), the label of the root node, we promote the deepest one. Letting T' denote the tree

   $$\dfrac{\overline{\texttt{"a"} \in \texttt{'a'}} \qquad \overline{\texttt{"b"} \in \texttt{'b'}}}{\texttt{"a"} \cdot \texttt{"b"} \in \texttt{(ab)}^*}\ (\texttt{"ab"})$$

   the result of calling `deloop` is the tree

   $$\dfrac{\overline{T'}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

   (b) The return of `split` is thus the singleton list containing a single pair of two parse trees; each element of the pair is the parse tree for $\texttt{"ab"} \in \texttt{(ab)}^*$ that was returned by `deloop`.
4. We invoke `parse` on each of the items in the sequence of items associated to $\texttt{(ab)}^*$ via the rule ($\texttt{(ab)}^*\texttt{(ab)}^*$). The two items are identical, and their associated elements of the pair returned by `split` are identical, so we only describe the execution once, on

   $$\dfrac{\overline{T'}}{\texttt{"ab"} \in \texttt{(ab)}^*}$$

   (a) We first ensure that we are not in an infinite loop. We check if $\texttt{s} < \texttt{s}_0$. This check succeeds, for `"ab"` is shorter than `"abab"`. We thus reset `unseen` and $\texttt{s}_0$; calls made by this stack frame will pass $\texttt{unseen}_0 \equiv \texttt{[(ab)}^*\texttt{]}$ for `unseen`, and $\texttt{s} \equiv \texttt{"ab"}$ for $\texttt{s}_0$.
   (b) We may again consider only the productions for which the parse tree associated to the string is well-typed. The only such production in this case is the one that lines up with the production used in the parse tree T', labeled (`"ab"`).
   (c) We invoke `split` on our parse tree.
      i. The `split` that we defined then invokes `deloop` on the trees $\overline{\texttt{"a"} \in \texttt{'a'}}$ and $\overline{\texttt{"b"} \in \texttt{'b'}}$. Since these trees have no non-root nodes (let alone non-root nodes sharing a label with the root), `deloop` is a no-op.
      ii. The return of `split` is thus the singleton list containing a single pair of two parse trees; the first is the parse tree $\overline{\texttt{"a"} \in \texttt{'a'}}$, and the second is the parse tree $\overline{\texttt{"b"} \in \texttt{'b'}}$.
   (d) We invoke `parse` on each of the items in the sequence of items associated to $\texttt{(ab)}^*$ via the rule (`"ab"`). Since both of these items are terminals, and the relevant equality check (that `"a"` is equal to `"a"`, and similarly for `"b"`) succeeds, `parse` returns `terminal_success`. We thus have the two `MinParseTree`s: $\overline{\texttt{"a"} \in \texttt{'a'}}$ and $\overline{\texttt{"b"} \in \texttt{'b'}}$.
   (e) We combine these using `cons_success` (and `nil_success`, to tie up the base case of the list). We thus have the tree $\overline{T'_m}$.
   (f) We apply $\texttt{production\_success}_<$ to this tree, and return the tree

   $$\dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]})$$

5. We now combine the two identical trees returned by `parse` using `cons_success` (and `nil_success`, to tie up the base case of the list). We thus have the tree

   $$\dfrac{\dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]}) \qquad \dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]})}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[]})$$

6. We apply $\texttt{production\_success}_=$ to this tree, and return the tree we claimed we would end up with,

   $$\dfrac{\dfrac{\dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]}) \qquad \dfrac{\overline{T'_m}}{\texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"ab"}, \texttt{[(ab)}^*\texttt{]})}{\texttt{"ab"} \cdot \texttt{"ab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[]})}{\texttt{"abab"} \in \texttt{(ab)}^*} < (\texttt{"abab"}, \texttt{[(ab)}^*\texttt{]})$$

5.3 Parametricity

Before we can combine different instantiations of this interface, we need to know that they behave similarly. Inspection of the code, together with relational parametricity, validates assuming the following axiom, which should also be internally provable by straightforward induction (though we have not bothered to prove it).

The *parser extensionality axiom* states that, for any fixed instantiation of `split`, and any arbitrary instantiations of the rest of the interface, giving rise to two different functions $\texttt{parse}_1$ and $\texttt{parse}_2$, we have

$$\forall\, (\texttt{nt} : \texttt{Nonterminal})\ (\texttt{s} : \texttt{String}),$$
$$\texttt{bool\_of\_sum}\ (\texttt{parse}_1\ \texttt{nt}\ \texttt{s}) = \texttt{bool\_of\_sum}\ (\texttt{parse}_2\ \texttt{nt}\ \texttt{s})$$

where `bool_of_sum` is, for any types A and B, the function of type $A + B \to \texttt{Bool}$ obtained by sending everything in the left component to `true`, and everything in the right component to `false`.

5.4 Putting it all together

Now we have parsers returning the following types:

$$\begin{aligned}
\texttt{has\_parse} &: \texttt{Nonterminal} \to \texttt{String} \to \texttt{Bool} \\
\texttt{parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\quad \to \texttt{option}\ (\texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s}) \\
\texttt{has\_no\_parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\quad \to \top + (\texttt{MinParseTreeOf}\ \texttt{nt}\ \texttt{s} \to \bot) \\
\texttt{min\_parse} &: (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\quad \to \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} \\
&\quad \to \texttt{MinParseTreeOf}\ \texttt{nt}\ \texttt{s}
\end{aligned}$$

Note that we have taken advantage of the isomorphism $\top + \top \cong$ `Bool` for `has_parse`, the isomorphism $A + \top \cong$ `option` A for `parse`, and the isomorphism $A + \bot \cong A$ for `min_parse`.

We can compose these functions to obtain our desired correct-by-construction parser:

$$\begin{aligned}
&\texttt{parse\_full} : (\texttt{nt} : \texttt{Nonterminal}) \to (\texttt{s} : \texttt{String}) \\
&\qquad \to \texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} + (\texttt{ParseTreeOf}\ \texttt{nt}\ \texttt{s} \to \bot)
\end{aligned}$$

```
parse_full nt s :=
  case parse nt s, has_no_parse nt s  of
  | Some d, _        →  inl d
  | _     , inr nd   →  inr (nd ∘ min_parse)
  | _     , _        →  ↯
```

In the final case, we derive a contradiction by applying the parser extensionality axiom, which says that `parse` and `has_no_parse` must agree on whether or not `s` parses as `nt`.

6. Semantic Actions

Our parsing algorithm can also be specialized to handle the common use case of semantic actions. Consider, for example, the following simultaneous specification of a grammar and some semantic actions:

$$\begin{aligned}
e ::= {} & n && \{\texttt{int\_of\_string}(n)\} \\
\mid {} & e_1\ \texttt{"+"}\ e_2 && \{e_1 + e_2\} \\
\mid {} & \texttt{"("}\ e\ \texttt{")"} && \{e\}
\end{aligned}$$

Supposing we have defined this grammar separately for our parser, we can instantiate the interface as follows to implement these semantic actions:

$$\begin{aligned}
&\texttt{T}_{\texttt{success}}\ \_\ \_\ \ (\underline{\_ \in e}) && := \mathbb{Z} \\
&\texttt{T}_{\texttt{success}}\ \_\ \_\ \ (\underline{\_ \in \texttt{'ch'}}) && := \top \\
&\texttt{T}_{\texttt{success}}\ \_\ \_\ \ (\underline{\_ \in (\texttt{its} : [\texttt{Item}])}) && := \prod_{\texttt{it} \in \texttt{its}} \texttt{T}_{\texttt{success}}\ \_\ \_\ (\underline{\_ \in \texttt{it}}) \\
&\texttt{T}_{\texttt{failure}}\ \_\ \_\ \_ && := \top
\end{aligned}$$

As all `failure` cases are instantiated with `()`, we elide them.

The `terminal` case is trivial:

$$\texttt{terminal\_success}\ \_\ \_\ \_ := ()$$

The `nil` and `cons` cases are similarly straightforward; we have defined $\texttt{T}_{\texttt{success}}$ on item sequences to be the corresponding tuple type.

$$\begin{aligned}
&\texttt{nil\_success}\ \_\ \_ := () \\
&\texttt{cons\_success}\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \_\ \texttt{x}\ \texttt{xs} := (\texttt{x}, \texttt{xs})
\end{aligned}$$

We will use a single definition definition `production_success` to combine $\texttt{production\_success}_<$ and $\texttt{production\_success}_=$ here, as the definition does not depend on any of the arguments that vary between them. This is where the semantic actions take place. We deal first with the case of a number:

$$\texttt{production\_success}\ \_\ n\ e\ \texttt{s}\ \_\ \_\ \_ := \texttt{int\_of\_string}\ \texttt{s}$$

In the case of e_1 `"+"` e_2, we get a tuple of three values: the value corresponding to e_1, the value corresponding to `"+"` (which in this case must just be `()`), and the value corresponding to e_2:

$$\begin{aligned}
&\texttt{production\_success}\ \_\ [e, \texttt{'+'}, e]\ e\ \_\ \_\ \_\ (\texttt{v}_1, \_, \texttt{v}_2) \\
&\quad := \texttt{v}_1 + \texttt{v}_2
\end{aligned}$$

Finally, we deal with the case of `"("` e `")"`. We again get a tuple of three values: the value corresponding to `"("`, the value corresponding to e, and the value corresponding to `")"`. As above, the character literals are mapped to dummy $\top$ semantic values, so we ignore them.

$$\begin{aligned}
&\texttt{production\_success}\ \_\ [\texttt{'('}, e, \texttt{')'}]\ e\ \_\ \_\ \_\ (\_, \texttt{v}, \_) \\
&\quad := \texttt{v}
\end{aligned}$$

7. Missteps, Insights, and Dependently Typed Lessons

We will now take a step back from the parser itself, and briefly talk about the process of coding it. We encountered a few pitfalls that we think highlight some key aspects of dependently typed programming, and our successes suggest benefits to be reaped from using dependent types.

7.1 The trouble of choosing the right types

Although we began by attempting to write the type-signature of our parser, we found that trying to write down the correct interface, without any code to implement it, was essentially intractable. Giving your functions dependent types requires performing a nimble balancing act between being uselessly general on the one hand, and too overly specific on the other, all without falling from the highropes of well-typedness onto the unforgiving floor of type errors.

We have found what we believe to be the worst sin the type-checker will let you get away with: having different levels of generality in different parts of your code base, which are supposed to interface with each other without a thoroughly vetted abstraction barrier between them. Like setting your highropes at different tensions, every trip across the interface will be costly, and if the abstraction levels get too far away, recovering your balance will require Herculean effort.

We eventually gave up on writing a dependently typed interface from the start, and decided instead to implement a simply typed Boolean recognizer, together with proofs of soundness and completeness. Once we had in hand these proofs, and the data types required to carry them out, we found that it was mostly straightforward to write down the interface and refine our parser to inhabit its newly generalized type.

7.2 Misordered splitters

One of our goals in this presentation was to hide most of the abstraction-level mismatch that ended up in our actual implementation, often through clever use of notation overloading. One of the most significant mismatches we managed to overcome was the way to represent the set of productions. In this paper, we left the type as an abstract mathematical set, allowing us to forgo concerns about ordering, quantification, and occasionally well-typedness.

In our Coq implementation, we fixed the type of productions to be a list very early on, and paid the price when we implemented our parse-tree parser. As mentioned in the execution of the example in Subsection 5.2, we wanted to restrict our attention to certain productions, and rule out the other ones using dependent types. This should be possible if we parameterize over not just a splitter, but a production-selector, and only require that our string type be well-typed for productions given by the production-selector. However, the implementation that we currently have requires a well-typed string type for all productions; furthermore, it does not allow the order in which productions are considered to depend on the augmented string data. We paid for this with the extra 300 lines of code we had to write to interleave two different splitters, so that we could handle the cases that we dismissed above as being ill-typed and therefore not necessary to consider. That is, because our types were not formulated in a way that actually made these cases ill-typed, we had to deal with them, much to our displeasure.

7.3 Minimal Parse Trees vs. Parallel Traces

Taking another step back, our biggest misstep actually came before we finished the completeness proof for our simply typed Boolean recognizer.

When first constructing the type `MinParseTree`, we thought of them genuinely as minimal parse trees (ones without a duplicate label in any single path). After much head-banging, of knowledge that a theorem was obviously true, against proof goals that were obviously impossible, we discovered the single biggest insight—albeit a technical one—of the project. The type of "minimal parse trees" we had originally formulated did not match the parse trees produced by our algorithm. A careful examination of the algorithm execution in Subsection 5.2 should reveal the difference.[1] Our insight, thus, was to conceptualize the data type as the type of traces of parallel executions of our particular parser, rather than as truly minimal parse trees.

This may be an instance of a more general phenomenon present when programming with dependent types: subtle friction between what you think you are doing and what you are actually doing often manifests as impossible proof goals.

8. Related Work

The field of parsing is one of the most venerable in computer-science. Still with us are a variety of parsing approaches born in times of much more severe constraints on memory and processor speed, including various flavors of LR parsers, which apply only to strict subsets of the context-free grammars, to guarantee ability to predict which production applies based on finite look-ahead into a string. However, despite rumors to the contrary, the field of parsing is far from dead. In the twentieth century, the functional-programming world experimented with a variety of approaches to *parser combinators* [5], where parsers are higher-order functions built from a small set of typed combinators. In the twenty-first century alone, a number of new parsing approaches have been proposed or popularized, including parsing expression grammars (PEGs) [4], derivative-based parsing [8], and GLL parsers [12].

However, our approach is essentially the same, algorithmically, as the one that Ridge demonstrated with a verified parser-combinator system [11], taking naive recursive-descent parsing and adding a layer to prune duplicative calls to the parser. His proof was carried out in HOL4, necessarily without using dependent types. Our new work may be interesting for the aesthetic appeal of our unusual application of dependent types to get the parser to generate some of its own soundness proof. Ridge's parser also has worst-case $O(n^5)$ running time in the input-string length. In the context of our verified implementation, we plan to explore a variety of optimizations based on clever, grammar-specific choices of string-splitter functions, which should have a substantial impact on the run-time cost of parsing some relevant grammars, and which we conjecture will not require any changes to the development presented in this paper.

A few other past projects have verified parsers with proof assistants, applying to derivative-based parsing [1] and SLR [2] and LR(1) [6] parsers. Several projects have used proof assistants to apply verified parsers within larger programming-language tools. RockSalt [9] does run-time memory-safety enforcement for x86 binaries, relying on a verified machine-code parser that applies derivative-based parsing for regular expressions. The verified Jitawa [10] and CakeML [7] language implementations include verified parsers, handling Lisp and ML languages, respectively.

Our final parser derivation relies on a relational parametricity property for polymorphic functions in Coq's type theory Gallina. With Coq as it is today, we need to prove this property manually for each eligible function, even though we can prove metatheoretically that it holds for them all. Bernardy and Guilhem [3] have shown how to extend type theories with support for materializing "free theorem" parametricity facts internally, and we might be able to simplify our implementation using such a feature.

9. Future Work and Conclusion

Dependent types have allowed us to refine our parsing algorithm to prove its own soundness and completeness.

However, we still have some work left to do to clean up the implementation of our parser.

Formal extensionality/parametricity proof To completely finish the formal proof of completeness, as described in this paper, we need to prove the parser extensionality axiom from Subsection 5.3. We need to prove that the parser does not make any decisions based on any arguments to its interface other than `split`, internalizing the obvious parametricity proof. (Alternatively, as mentioned above, we could hope to use an extension of Coq with internalized parametricity [3].)

Even more self-reference We might also consider reusing the same generic parser to generate the extensionality proofs, by instantiating the type families for success and failure with families of propositions saying that all instantiations of the parser, when called with the same parsing problem, always return values that are equivalent when converted to Booleans. A more specialized approach could show just that `has_parse` agrees with `parse` on successes and with `has_no_parse` on failures:

$$\begin{aligned} &\mathtt{T}_{\text{success}}\ \_\ \_\ (\overline{\mathtt{s} \in \mathtt{nt}}) \\ &\quad := \mathtt{has\_parse}\ \mathtt{nt}\ \mathtt{s} = \mathtt{true} \land \mathtt{parse}\ \mathtt{nt}\ \mathtt{s} \neq \mathtt{None} \\ &\mathtt{T}_{\text{failure}}\ \_\ \_\ (\overline{\mathtt{s} \in \mathtt{nt}}) \\ &\quad := \mathtt{has\_parse}\ \mathtt{nt}\ \mathtt{s} = \mathtt{false} \land \mathtt{has\_no\_parse} \neq \mathtt{inl}\ () \end{aligned}$$

More splitters In order to synthesize efficient parsers, we plan to construct other splitters that reduce the complexity of the algorithm to well-known bounds for various common classes of grammars.

Synthesizing dependent types automatically? Although finding sufficiently general (dependent) type signatures was a Herculean

[1] For readers wanting to skip that examination: the algorithm we described allows a label ($\overline{\mathtt{s} \in \mathtt{nt}}$) to appear one extra time along a path if, the first time it appears, its parent node's label, ($\overline{\mathtt{s}' \in \mathtt{nt}'}$), satisfies $\mathtt{s} < \mathtt{s}'$. That is, whenever the string being parsed shrinks, the first nonterminal the shrunken string is parsed as may be duplicated once before shrinking the string again.

task before we finished the completeness proof and discovered the idea of using parallel parse traces, it was mostly straightforward once we had proofs of soundness and completeness of the simply typed parser in hand; most of the issues we faced involving having to figure out how to thread additional hypotheses, which showed up primarily at the very end of the proof, through the entire parsing process. Subsequently instantiating the types was also mostly straightforward, with most of our time and effort being spent writing transformations between nearly identical types that had slightly different hypotheses, e.g., converting a `Foo` involving strings shorter than s_1 into another analogous `Foo`, but allowing strings shorter than s_2, where s_1 is not longer than s_2. Our experience raises the question of whether it might be possible to automatically infer dependently typed generalizations of an algorithm, which subsume already-completed proofs about it, and perhaps allow additional proofs to be written more easily.

Further generalization Finally, we believe our parser could be generalized even further; the algorithm we have implemented is essentially an algorithm for inhabiting arbitrary inductive type families, subject to some well-foundedness, enumerability, and finiteness restrictions on the arguments to the type family. The interface we described is, conceptually, a composition of this inhabitation algorithm with recursion and inversion principles for the type family we are inhabiting (`ParseTreeOf` in this paper). Our techniques for refining this algorithm so that it could prove itself sound and complete should therefore generalize to this viewpoint.

Concluding remarks Though there remain many useful extensions to investigate, we believe our approach to parsing highlights some of the benefits and pitfalls of dependently typed programming. Dependent types allow more code reuse—but require more type annotations and more thought about type signatures—and, in our experience, make it easier to think about proofs, and perhaps easier to maintain them altogether.

References

[1] J. B. Almeida, N. Moreira, D. Pereira, and S. a. M. de Sousa. Partial derivative automata formalized in Coq. In *Proceedings of the 15th International Conference on Implementation and Application of Automata*, CIAA'10, pages 59–68, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-18097-2. URL `http://dl.acm.org/citation.cfm?id=1964285.1964292`.

[2] A. Barthwal and M. Norrish. Verified, executable parsing. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 160–174, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3. . URL `http://dx.doi.org/10.1007/978-3-642-00590-9_12`.

[3] J.-P. Bernardy and M. Guilhem. Type-theory in color. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 61–72, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. . URL `http://doi.acm.org/10.1145/2500365.2500577`.

[4] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. . URL `http://doi.acm.org/10.1145/964001.964011`.

[5] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[6] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28868-5. . URL `http://dx.doi.org/10.1007/978-3-642-28869-2_20`.

[7] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. . URL `http://doi.acm.org/10.1145/2535838.2535841`.

[8] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 189–195, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. . URL `http://doi.acm.org/10.1145/2034773.2034801`.

[9] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. . URL `http://doi.acm.org/10.1145/2254064.2254111`.

[10] M. O. Myreen and J. Davis. A verified runtime for a verified theorem prover. In *Proceedings of the Second International Conference on Interactive Theorem Proving*, ITP'11, pages 265–280, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22862-9. URL `http://dl.acm.org/citation.cfm?id=2033939.2033961`.

[11] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In *Proceedings of the First International Conference on Certified Programs and Proofs*, CPP'11, pages 103–118, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25378-2. . URL `http://dx.doi.org/10.1007/978-3-642-25379-9_10`.

[12] E. Scott and A. Johnstone. GLL parsing. In *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications*, LDTA '09, pages 177–189, 2009.