A Verified Earley Parser

Martin Rau
Department of Computer Science, Technical University of Munich, Germany

Tobias Nipkow
Department of Computer Science, Technical University of Munich, Germany

Abstract

An Earley parser is a top-down parsing technique that is capable of parsing arbitrary context-free grammars. We present a functional implementation of an Earley parser verified using the interactive theorem prover Isabelle/HOL. Our formalization builds upon Cliff Jones' extensive, refinement-based paper proof. We implement and prove soundness and completeness of a functional recognizer modeling Jay Earley's original imperative implementation and extend it with the necessary data structures to enable the construction of parse trees following the work of Elizabeth Scott. Building upon this foundation, we develop a functional parser and prove its soundness. We round off the paper by providing an informal argument and empirical data regarding the running time and space complexity of our implementation.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Parsers

Keywords and phrases Verification, Parsers, Earley, Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.31

1 Introduction

Parsing is fundamental. Nearly every application interacts with its environment, and usually by means of parsing textual input into a structured data format. In the age of big data, applications handle enormous amounts of data and any parser bugs or vulnerabilities entail severe security risks. Although the semantics of a parser are relatively easy to specify, correctly implementing a parser is a difficult task. Attackers regularly exploit parsing bugs to obtain sensitive user data [1, 3, 2]. Hence, parsing algorithms are well-suited for formal verification which allows us to precisely specify the semantics of a parser and obtain strong correctness guarantees.

A zoo of parsing algorithms exists, and one of the core trade-offs one has to make when deciding on a parser is between performance and usability. Earley [12] parsing, originally conceived by Jay Earley in 1968, is an algorithm that allows the full range of context-free grammars while still being very performant for a large subset. In this paper, we present the, to our knowledge, first formalization of an Earley parser. Our formalization builds upon Cliff Jones' [20] extensive, refinement-based paper proof.

Section 2 shortly introduces Isabelle/HOL [29, 28]. Section 3 contains the formalization of context-free grammars and derivations. Section 4 defines and proves correct an inductive definition of an Earley recognizer. Section 5 refines this definition to an executable algorithm. Section 6 extends the recognizer to a parser. Section 7 contains an analysis of the running time. Sections 8 and 9 discuss related work and conclude.

The whole formalization, including all proofs, can be found online in the Archive of Formal Proofs [33]. The size of the formalization (more than 6000 lines) prohibits a detailed exposition in this paper, especially of the proofs. The interested reader is referred to the online material.

2 Isabelle/HOL

Isabelle [29, 28] is an interactive theorem prover based on a fragment of higher-order logic. It supports core concepts commonly found in functional programming languages.

The notation $t :: \tau$ indicates that term t has type τ. Basic types include *bool* and *nat*, while type variables are written $'a$, $'b$ etc. Pairs are expressed as (a, b), and triples as (a, b, c), and so forth. Functions *fst* and *snd* return the first and second components of a pair, while the operator $(\times)$ is used for pairs at the type level. Most type constructors are written postfix, such as $'a$ *set* and $'a$ *list*, and the function space arrow is $\Rightarrow$. Function *set* converts a list to a set.

Algebraic data types are defined using the **datatype** keyword. Non-recursive definitions are introduced with the **definition** keyword. Lists are constructed from the empty list [] using the infix cons-operator (#). The operator (@) appends two lists, $|xs|$ denotes the length of xs, $xs\ !\ n$ returns the n-th item of the list xs (starting with $n = 0$), and $xs[i := x]$ returns an updated list by setting the n-th item to the value x.

3 Context-free Grammars and Derivations

A symbol, either non-terminal or terminal, is represented as an arbitrary type $'a$. We use lowercase letters a, b, c to denote terminals and capital letters A, B, C to denote non-terminals. Additionally, we use the letters s, t to represent arbitrary symbols. A *sentence* is defined as a list of symbols and can be represented by either Greek letters α, β, γ or lowercase letters u, v, w.

The data type *cfg* represents context-free grammars. An instance $\mathcal{G}$ comprises a list of production rules $\mathfrak{R}\ \mathcal{G}$, where each rule is a pair consisting of a left-hand side, *lhs_rule*, a single symbol, and a right-hand side, *rhs_rule*, a list of symbols. Additionally, the instance $\mathcal{G}$ contains the start symbol $\mathfrak{S}\ \mathcal{G}$.

We formalize the set of non-terminals as the union of all left-hand sides of a grammar's production rules and its start symbol. A *word* is a sentence that consists only of terminal symbols, meaning $is\_word\ \mathcal{G}\ \omega = (nonterminals\ \mathcal{G} \cap set\ \omega = \emptyset)$. The empty word is denoted by [].

Given a grammar $\mathcal{G}$, the sentence β can be derived from the sentence α in a single step, denoted by $\mathcal{G} \vdash \alpha \Rightarrow \beta$, if $\mathcal{G}$ contains a production rule (A, γ) such that α is of the form $u\ @\ [A]\ @\ v$ and $\beta = u\ @\ \gamma\ @\ v$. Defining $derivations\ \mathcal{G} = \{(\alpha, \beta) \mid \mathcal{G} \vdash \alpha \Rightarrow \beta\}^*$ we abbreviate $(\alpha, \beta) \in derivations\ \mathcal{G}$ by $\mathcal{G} \vdash \alpha \Rightarrow^* \beta$.

Some of the core proofs of this work make use of an analogous formalization of derivations. The term $Derivation\ \mathcal{G}\ \alpha\ D\ \beta$ signifies that the grammar $\mathcal{G}$ allows the sentence β to be derived from the sentence α via the *derivation* D. In this context, D is a list containing pairs of production rules and indices, which constitute the specific rewriting steps. When applied in sequence to α, these steps lead to β. Both definitions of derivations are indeed equivalent, meaning $\mathcal{G} \vdash \alpha \Rightarrow^* \beta$, if and only if, there exists a derivation D such that the predicate $Derivation\ \mathcal{G}\ \alpha\ D\ \beta$ holds. We omit the proof.

4 Defining the Set of Earley Items

An Earley recognizer determines whether the input ω is in the language defined by the grammar $\mathcal{G}$ by following a two-step process: first, it generates a set of items, then it checks if there exists a *finished* item. In the following, we consider a fixed grammar $\mathcal{G}$ and input ω.

An item takes the form *Item r d i j*, which consists of four components: a production rule r from the grammar $\mathcal{G}$, referred to as the *rule_item*, a natural number d, or *dot_item*, marking how far the algorithm has processed the right-hand side of r, and two natural numbers i, j, *start_item* and *end_item*, representing the start index and the end index (exclusive) of the sublist of the input ω recognized by the item. Alternatively, an item with a production rule $A \to \alpha\beta$, which recognizes the subsequence of the input from index i up to but excluding j by processing α, is written $A \to \alpha \bullet \beta, i, j$.

The functions *lhs_item* and *rhs_item* project the *lhs_rule* and *rhs_rule* of an item. Functions *α_item* and *β_item* split the production rule body at the position of the dot. An item is complete, *is_complete*, when the dot is at the end of the production rule body, as in $A \to \alpha\bullet, i, j$. The *next_symbol* of an item can either be *None*, if it is complete, or *Some s*, where s is the symbol in the production rule body following the dot.

An item x is well-formed, *wf_item*, if the item's rule belongs to the grammar $\mathcal{G}$, the item dot must be within the length of the item's right-hand side, the item start does not exceed the item end, and finally, the item end must be at most the length of the input ω.

An item is finished, *is_finished*, if it is of the form $\mathfrak{S}\ \mathcal{G} \to \alpha\bullet, 0, |\omega|$, meaning the left-hand side of the item is the start symbol of the grammar $\mathcal{G}$, the item is complete, and the entire input ω has been recognized; or the item start is zero, and the item end is the length of ω.

The set of *Earley* items, *Earley* $\mathcal{G}\ \omega$, is an inductive definition of Earley's recognizer, i.e. an inductively defined set. The four defining rules are: the initial set of items, and one rule for each of the core operations that expand the set of items: scanning, prediction, and completion.

$$\frac{(\mathfrak{S}\ \mathcal{G},\ \alpha) \in \mathit{set}\ (\mathfrak{R}\ \mathcal{G})}{\mathfrak{S}\ \mathcal{G} \to \bullet\alpha, 0, 0 \in \mathit{Earley}\ \mathcal{G}\ \omega}\ \textsc{Init}$$

$$\frac{A \to \alpha \bullet a\beta, i, j \in \mathit{Earley}\ \mathcal{G}\ \omega \qquad \omega\ !\ j = a \qquad j < |\omega|}{A \to \alpha a \bullet \beta, i, j+1 \in \mathit{Earley}\ \mathcal{G}\ \omega}\ \textsc{Scan}$$

$$\frac{A \to \alpha \bullet B\beta, i, j \in \mathit{Earley}\ \mathcal{G}\ \omega \qquad (B,\ \gamma) \in \mathit{set}\ (\mathfrak{R}\ \mathcal{G})}{B \to \bullet\gamma, j, j \in \mathit{Earley}\ \mathcal{G}\ \omega}\ \textsc{Predict}$$

$$\frac{A \to \alpha \bullet B\beta, i, j \in \mathit{Earley}\ \mathcal{G}\ \omega \qquad B \to \gamma\bullet, j, k \in \mathit{Earley}\ \mathcal{G}\ \omega}{A \to \alpha B \bullet \beta, i, k \in \mathit{Earley}\ \mathcal{G}\ \omega}\ \textsc{Complete}$$

The *Init* rule specifies all initial items $\mathfrak{S}\ \mathcal{G} \to \bullet\alpha, 0, 0$. There is one item for each grammar rule that begins with the grammar's start symbol. For these items, the dot, start, and end indices are all initialized to 0. This signifies that we haven't processed the right-hand side of the rule at all, started the recognition process at the beginning of the word, and still are at this initial position.

The *Scan* rule applies if there is a terminal symbol to the right of the dot: $A \to \alpha \bullet a\beta, i, j$. In this case, if the j-th symbol of ω is the *next_symbol* of the item, we add a new item $A \to \alpha a \bullet \beta, i, j+1$, moving the dot over the recognized terminal symbol.

The *Predict* rule is applicable to an item when there is a non-terminal symbol to the right of the dot: $A \to \alpha \bullet B\beta, i, j$. It adds a new item $B \to \bullet\gamma, j, j$ for each production rule $(B,\ \gamma)$ of the grammar. Similar to the initial items, the dot is set to 0, but the start and end indices are set to j to indicate that we are beginning recognition at position j in the input ω.

The *Complete* rule is applied to all complete items $B \to \gamma\bullet, j, k$. These items indicate successful recognition of a subsequence of ω starting at index j and ending at index k. Now, we consider any items where we already predicted the non-terminal symbol B. Specifically, we look for items $A \rightarrow \alpha \bullet B\beta, i, j$ with a matching end index j and a dot in front of the non-terminal B. Since we have successfully recognized the predicted non-terminal, we are allowed to move the dot, resulting in the addition of a new item $A \rightarrow \alpha B \bullet \beta, i, k$.

We will prove soundness and completeness of *Earley*:

1. Soundness: If $x \in$ *Earley* $\mathcal{G}\ \omega$ and *is_finished* $\mathcal{G}\ \omega\ x$, then $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$.
2. Completeness: If $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$, then there exists an item $x \in$ *Earley* $\mathcal{G}\ \omega$ such that *is_finished* $\mathcal{G}\ \omega\ x$.

Two further important properties that we will need:

1. Well-formedness: For all $x \in$ *Earley* $\mathcal{G}\ \omega$, *wf_item* $\mathcal{G}\ \omega\ x$ holds.
2. Finiteness: The set *Earley* $\mathcal{G}\ \omega$ is finite.

4.1 Proving Well-formedness and Finiteness

The proof of the well-formedness of the set of Earley items is straightforward by induction on the definition of *Earley*. We omit it.

Furthermore, there exist only a finite number of Earley items: Given that all Earley items are well-formed, it suffices to prove that there is only a finite number of well-formed Earley items. We define the set T as *set* $(\mathfrak{R}\ \mathcal{G}) \times \{0..m\} \times \{0..|\omega|\} \times \{0..|\omega|\}$, where m denotes the maximum length of all right-hand sides of production rules from the grammar $\mathcal{G}$. The set T is finite as there is only a finite number of production rules, and both the right-hand side of each production rule and the input ω are finite. Furthermore, T is an over-approximation of *Earley* $\mathcal{G}\ \omega$, as every well-formed Earley item is contained within T by definition of well-formedness.

4.2 Proving Soundness

An item $A \rightarrow \alpha \bullet \beta, i, j$ is considered sound, *sound_item*, if it satisfies $\mathcal{G} \vdash [A] \Rightarrow^* (\omega_{i/j}\ @\ \beta)$, where $\omega_{i/j}$ is the subsequence of ω from index i to (but excluding) j. Let x denote an arbitrary item in *Earley* $\mathcal{G}\ \omega$. We prove *sound_item* $\mathcal{G}\ \omega\ x$ by induction on the definition of *Earley*.

For the *Init* rule, we have $x = \mathfrak{S}\,\mathcal{G} \rightarrow \bullet\alpha, 0, 0$. Furthermore, we know that $\omega_{0/0}$ equals the empty list. Our goal is to show that there exists a derivation from $\mathfrak{S}\ \mathcal{G}$ to α. This is immediately evident as $(\mathfrak{S}\ \mathcal{G},\ \alpha)$ is a production rule from the grammar due to the well-formedness of the item.

For the *Scan* rule, we deal with an item $x = A \rightarrow \alpha \bullet a\beta, i, j$, and the induction hypothesis is that $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j}\ @\ (a\ \#\ \beta)$. Since we have that $\omega\ !\ j = a$, this is equivalent to $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j\,+\,1}\ @\ \beta$, which, in turn, implies the soundness of the new item $x = A \rightarrow \alpha a \bullet \beta, i, j+1$.

For the *Prediction* rule, the new item is $x = B \rightarrow \bullet\alpha, j, j$. Since $\omega_{j/j}$ equals the empty list, our task is to show that $\mathcal{G} \vdash [B] \Rightarrow^* \alpha$. This follows directly as $(B,\ \alpha)$ is a production rule of the grammar.

For the *Complete* rule, we have two items: $x = A \rightarrow \alpha \bullet B\beta, i, j$ and $y = B \rightarrow \gamma\bullet, j, k$. The two induction hypothesises are $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j}\ @\ (B\ \#\ \beta)$ and $\mathcal{G} \vdash [B] \Rightarrow^* \omega_{j/k}$. Combining these statements yields $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/j}\ @\ \omega_{j/k}\ @\ \beta$, which is equivalent to $\mathcal{G} \vdash [A] \Rightarrow^* \omega_{i/k}\ @\ \beta$, and thus, implies the soundness of the new item $A \rightarrow \alpha B \bullet \beta, i, k$, concluding the soundness proof.

theorem *soundness_Earley*:
  assumes $\exists\, x \in$ *Earley* $\mathcal{G}\ \omega$. *is_finished* $\mathcal{G}\ \omega\ x$
  shows $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

4.3 Proving Completeness

Completeness is the most intricate proof obligation, and we begin by providing some intuition about the fundamental proof idea. We call a set I of items *partially completed* if for every item $A \to \alpha \bullet s\beta, i, j$ in I and every derivation $\mathcal{G} \vdash [s] \Rightarrow^* \omega_{j/k}$, the set I also contains the item $A \to \alpha s \bullet \beta, i, k$.

Now, consider the item $A \to \bullet s_0 s_1 \ldots s_n, i, i_0$. If this item is present in a partially completed set of items I, and there exists a derivation $\mathcal{G} \vdash [s_0] \Rightarrow^* \omega_{i_0/i_1}$, then the item $A \to s_0 \bullet s_1 \ldots s_n, i, i_1$ is also included in the set. If there exists another derivation $\mathcal{G} \vdash [s_1] \Rightarrow^* \omega_{i_1/i_2}$, the statement holds again for the item $A \to s_0 s_1 \bullet \ldots s_n, i, i_2$, and so on. This continues until, for a derivation $\mathcal{G} \vdash [s_n] \Rightarrow^* \omega_{i_n/j}$, the completed item $A \to s_0 s_1 \ldots s_n \bullet, i, j$ is in I, provided that we have $i \le i_0 \le i_1 \le \cdots \le i_n \le j$.

The definitions of *partially_completed* and the subsequent theorem that captures the outlined proof idea are more intricate in their details. They also encompass the necessary bounds for the indices, and make use of the analogous definition of derivations through the predicate *Derivation*, which contains an actual derivation D. They also incorporate an additional predicate on D. Its purpose is to limit the length of the derivation D. This is crucial because the proof of the partial completeness of the set of Earley items is by induction on the length of the derivation.

partially_completed $l\ \mathcal{G}\ \omega\ I\ P = (\forall\, r\ d\ i\ j\ k\ x\ s\ D.$
  $j \le k \wedge k \le l \wedge l \le |\omega| \wedge$
  $x = \mathit{Item}\ r\ d\ i\ j \wedge x \in I \wedge \mathit{next\_symbol}\ x = \mathit{Some}\ s \wedge$
  $\mathit{Derivation}\ \mathcal{G}\ [s]\ D\ (\omega_{j/k}) \wedge P\ D \longrightarrow \mathit{Item}\ r\ (d{+}1)\ i\ k \in I)$

theorem *partially_completed_upto*:
  assumes $j \le k$ **and** $k \le |\omega|$
  assumes $x = \mathit{Item}\ (A,\alpha)\ d\ i\ j$ **and** $x \in I$ **and** $\forall x \in I.\ \mathit{wf\_item}\ \mathcal{G}\ \omega\ x$
  assumes $\mathit{Derivation}\ \mathcal{G}\ (\beta\_\mathit{item}\ x)\ D\ (\omega_{j/k})$
  assumes $\mathit{partially\_completed}\ k\ \mathcal{G}\ \omega\ I\ (\ \lambda D'.\ |D'| \le |D|\ )$
  shows $\mathit{Item}\ (A,\alpha)\ |\alpha|\ i\ k \in I$

theorem *partially_completed_Earley*:
  shows $\mathit{partially\_completed}\ |\omega|\ \mathcal{G}\ \omega\ (\mathit{Earley}\ \mathcal{G}\ \omega)\ (\lambda\_.\ \mathit{True})$

To establish the completeness of the inductive definition for the set of Earley items, we apply both of the preceding theorems. By assumption, there exists a derivation of the input ω from the grammar's start symbol. We can decompose this derivation into a single initial production rule $(\mathfrak{S}\ \mathcal{G},\ \alpha)$ and a subsequent derivation *Derivation* $\mathcal{G}\ \alpha\ D\ \omega$. Additionally, we know, by definition of the *Init* rule, that the item $\mathfrak{S}\ \mathcal{G} \to \bullet\alpha, 0, 0$ is in *Earley* $\mathcal{G}\ \omega$. Moreover, considering that each Earley item is well-formed and the set of Earley items is partially completed, as proved by the theorem *partially_completed_Earley*, we can consequently discharge the assumptions of the theorem *partially_completed_upto*. As a result, we know that the finished item $\mathfrak{S}\ \mathcal{G} \to \alpha\bullet, 0, |\omega|$ is indeed present in *Earley* $\mathcal{G}\ \omega$.

theorem *completeness_Earley*:
  assumes $\mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$ **and** $\mathit{is\_word}\ \mathcal{G}\ \omega$
  shows $\exists x \in \mathit{Earley}\ \mathcal{G}\ \omega.\ \mathit{is\_finished}\ \mathcal{G}\ \omega\ x$

theorem *correctness_Earley*:
  assumes $\mathit{is\_word}\ \mathcal{G}\ \omega$
  shows $(\exists x \in \mathit{Earley}\ \mathcal{G}\ \omega.\ \mathit{is\_finished}\ \mathcal{G}\ \omega\ x) \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

5 An Executable Earley Recognizer

We refine the inductive Earley definition of the previous section to an executable algorithm, a *recognizer* that tells us if the input ω is in the language specified by the grammar $\mathcal{G}$. Our Earley recognizer is a functional algorithm modeled after Earley's original imperative implementation. We start with an informal explanation. The algorithm processes the input $\omega = a_0, \ldots, a_{n-1}$ while maintaining a list of $n+1$ *bins*. An initial bin B_0 and one bin B_{i+1} for each symbol a_i in the input. Each bin is a variable length list of Earley *entries*. Each entry is a pair consisting of an Earley item and "pointers", i.e. indices, indicating the originating entry needed for the construction of parse trees. These pointers are elements of a data type with three alternatives:

A pointer can either be a *null* pointer, denoted by $\perp$, a predecessor pointer representing a single index i, or a nonempty list of reduction pointers containing triples of indices, $(a, b, c), (d, e, f), \ldots$. We define the exact semantics in the following paragraphs. To improve readability we omit showing any constructors of the entry and pointer data types and only use the shorthand notation. For example, an entry comprising the item $A \to \alpha \bullet \beta, i, j$ and the reduction pointer (a, b, c) is written $A \to \alpha \bullet \beta, i, j; (a, b, c)$.

The algorithm generates the bins in ascending order, starting at bin B_0. Each bin serves a dual purpose: as a worklist of entries to be processed, and as a set of items that are already present, ensuring that no two entries with identical items are present within the same bin. An entry with the item $A \to \alpha \bullet \beta, i, j$ is always in bin B_j, in other words, the end index of the item equals the index of the bin.

Initially, the algorithm populates bin B_0 with the items corresponding to the *Init* rule of the inductive Earley definition. Each initial item is accompanied by a null pointer. Table 1 illustrates the executable algorithm by example for the toy grammar $\mathcal{G} : \ S \to x \mid S + S$ and input: $\omega = x + x + x$, showcasing the complete bins after a run of the algorithm. In the example, the bins contain the two initial entries $S \to \bullet x, 0, 0; \perp$ and $S \to \bullet S + S, 0, 0; \perp$ in bin B_0. The algorithm proceeds to process the worklist, from top to bottom, until the bin stabilizes. Then, it moves on to the next bin.

For each item x of the current entry at index l in the k-th bin, the algorithm applies operations corresponding to the three rules *Scan*, *Predict*, *Complete*.

- Case $x = A \to \alpha \bullet a\beta, j, k$: if the symbol at position k in ω is the terminal symbol a, the entry $A \to \alpha a \bullet \beta, j, k+1; l$ is inserted into the next bin B_{k+1}. The index l indicates the predecessor index, signifying that the originating entry of this new entry resides in the previous bin at index l. Table 1 contains the entry $S \to x\bullet, 0, 1; 0$ at index 0 in bin B_1, and its predecessor is the entry $S \to \bullet x, 0, 0; \perp$ at index 0 in bin B_0.
- Case $x = A \to \alpha \bullet B\beta, j, k$: for each production rule (B, γ) of the grammar $\mathcal{G}$, an entry $B \to \bullet\gamma, k, k; \perp$ is inserted into the current bin B_k. A null pointer is added to the entry, as no origin information is required for constructing parse trees. Table 1 contains the entries $S \to \bullet x, 2, 2; \perp$ and $S \to \bullet S + S, 2, 2; \perp$ in bin B_2, both predicted by the entry $S \to S + \bullet S, 0, 2; 1$ in the same bin.
- Case $x = B \to \gamma\bullet, j, k$: if an item is complete, the algorithm searches the origin bin B_j for any entries with items of the form $A \to \alpha \bullet B\beta, i, j$. If it finds such an entry at index l', it inserts one new entry $A \to \alpha B \bullet \beta, i, k; (j, l', l)$ into the current bin. The origin information (j, l', l) is a reduction pointer. The first two indices, j and l', indicate that the predecessor entry resides in bin B_j at index l'. The last index, l, describes the position of the reduction entry at index l in the current bin B_k. An entry may contain more than one reduction pointer in cases where the grammar is ambiguous and there are

multiple ways to derive the input corresponding to the item. Table 1 contains the entry $S \to S + S\bullet, 0, 5; (4,1,0), (2,0,1)$, capturing the two possible derivations of ω: $(x + x) + x$ and $x + (x + x)$. The entry, with a single reduction pointer $(4, 1, 0)$, was initially created due to the reduction entry $S \to x\bullet, 4, 5; 2$ at index 0 in bin B_5 and the predecessor entry $S \to S + \bullet S, 0, 4; 3$ at index 1 in bin B_4. However, the second reduction pointer $(2, 0, 1)$ was later added due to the reduction entry $S \to S + S\bullet, 2, 5; (4, 0, 0)$ at index 1 in bin B_5 and the predecessor entry $S \to S + \bullet S, 0, 2; 1$ at index 0 in bin B_2.

The algorithm inserts an entry into a bin as follows: Iterate through the bin, and, for each entry, check if its item matches the item of the entry to be inserted. If a match is found, and the pointer of the entry to be inserted is a reduction pointer, merge the items by adding the reduction pointer to the already present entry. Otherwise, if there is no match or the pointer is not a reduction pointer, do not make any additions. In both cases, terminate the insertion process. If there are no entries with matching items, append the entry to the end of the bin.

Table 1 Earley bins for $\mathcal{G}$: $S \to x \mid S + S$, and $\omega = x + x + x$.

	B_0	B_1	B_2
0	$S \to \bullet x, 0, 0; \perp$	$S \to x\bullet, 0, 1; 0$	$S \to S + \bullet S, 0, 2; 1$
1	$S \to \bullet S + S, 0, 0; \perp$	$S \to S \bullet + S, 0, 1; (0, 1, 0)$	$S \to \bullet x, 2, 2; \perp$
2			$S \to \bullet S + S, 2, 2; \perp$

	B_3	B_4	B_5
0	$S \to x\bullet, 2, 3; 1$	$S \to S + \bullet S, 2, 4; 2$	$S \to x\bullet, 4, 5; 2$
1	$S \to S + S\bullet, 0, 3; (2, 0, 0)$	$S \to S + \bullet S, 0, 4; 3$	$S \to S + S\bullet, 2, 5; (4, 0, 0)$
2	$S \to S \bullet + S, 2, 3; (2, 2, 0)$	$S \to \bullet x, 4, 4; \perp$	$S \to S + S\bullet, 0, 5; (4, 1, 0), (2, 0, 1)$
3	$S \to S \bullet + S, 0, 3; (0, 1, 1)$	$S \to \bullet S + S, 4, 4; \perp$	$S \to S \bullet + S, 4, 5; (4, 3, 0)$
4			$S \to S \bullet + S, 2, 5; (2, 2, 1)$
5			$S \to S \bullet + S, 0, 5; (0, 1, 2)$

5.1 Recognizer Implementation

We now examine the formal definition of the recognizer. There are four functions Init_L, Scan_L, $\mathit{Predict}_L$, and $\mathit{Complete}_L$ implementing *list*-based versions of the four corresponding rules of the inductive Earley definition. Due to space restrictions we only show the function Init_L, constructing the initial bins, and the function $\mathit{Predict}_L$ that returns a list of new entries to be inserted into the bins. Functions Scan_L and $\mathit{Complete}_L$ have the same return type.

```
$Init_L\ \mathcal{G}\ \omega = ($
  $let\ rs = filter\ (\lambda r.\ lhs\_rule\ r = \mathfrak{S}\ \mathcal{G})\ (remdups\ (\mathfrak{R}\ \mathcal{G}))\ in$
  $let\ b0 = map\ (\lambda r.\ (Item\ r\ 0\ 0\ 0,\ Null))\ rs\ in$
  $let\ bs = replicate\ (\ |\omega| + 1\ )\ ([])\ in\ bs[0 := b0])$

$Predict_L\ k\ \mathcal{G}\ A = ($
  $let\ rs = filter\ (\lambda r.\ lhs\_rule\ r = A)\ (\mathfrak{R}\ \mathcal{G})\ in$
  $map\ (\lambda r.\ (Item\ r\ 0\ k\ k,\ Null))\ rs)$
```

The central piece of the implementation is the function Earley_L_bin'. The function computes the entries of the k-th bin starting at the entry at index i. It examines the symbol following the dot of the item of the entry and, depending on the type of the symbol or whether such a symbol exists at all, applies one of the three executable operations, obtaining a list of potentially new entries. These entries are subsequently inserted into the bins using the function *upd_bins* (definition omitted). Function $Earley_L$*_bin* starts this process at the beginning of the bin at index 0.

```
$Earley_L\_bin'\ k\ \mathcal{G}\ \omega\ bs\ i = ($
  $if\ i \geq |(items\ (bs!k))|\ then\ bs$
  $else$
    $let\ x = items\ (bs!k)\ !\ i\ in$
    $let\ bs' =$
      $case\ next\_symbol\ x\ of$
        $Some\ s \Rightarrow ($
          $if\ s \notin nonterminals\ \mathcal{G}\ then$
            $if\ k < |\omega|\ then\ upd\_bins\ bs\ (k{+}1)\ (Scan_L\ k\ \omega\ s\ x\ i)\ else\ bs$
          $else\ upd\_bins\ bs\ k\ (Predict_L\ k\ \mathcal{G}\ s))$
      $|\ None \Rightarrow upd\_bins\ bs\ k\ (Complete_L\ k\ x\ bs\ i)$
    $in\ Earley_L\_bin'\ k\ \mathcal{G}\ \omega\ bs'\ (i{+}1))$

$Earley_L\_bin\ k\ \mathcal{G}\ \omega\ bs = Earley_L\_bin'\ k\ \mathcal{G}\ \omega\ bs\ 0$
```

The function $Earley_L\_bin'$ is defined as a partial function as it might not terminate if it keeps inserting newly generated entries forever into the bin it currently operates on. However, we know that the newly generated entries do not contain arbitrary but only well-formed bin items. In other words, each bin B_k contains only entries with items that are well-formed and additionally have the end index k. We have already proved that the number of well-formed Earley items is finite, and the implementation ensures that a new entry is added to the bin only if its item is not already present in one of the bin's entries. Therefore, the function will eventually run out of new entries to insert into the bin it currently operates on and terminate.

Although HOL is a logic of total functions, Isabelle supports the definition of potentially non-terminating functions provided they are tail-recursive (like $Earley_L\_bin'$) or their result is an optional value (like function *build_tree'* below). The underlying domain-theoretic definitional constructions are due to Krauss [24]. However, we cannot prove anything about such a function because Isabelle does not know for which inputs it terminates, or if it terminates at all. As a result, Isabelle does not generate an appropriate induction schema for it. Such a schema must be proved by hand by specifying a suitable type and measure for which the function terminates. For the function $Earley_L\_bin'$ we define the measure $earley\_measure\ (k, \mathcal{G}, \omega, bs)\ i = |\{x \mid wf\_item\ \mathcal{G}\ \omega\ x \wedge end\_item\ x = k\}| - i$ and prove that it is strictly decreasing for every tail-recursive function call.

The function $Earley_L\_bins$ computes the bins upto a specific index starting at bin zero. And finally, function $Earley_L$ computes the complete bins.

$$Earley_L\_bins\ 0\ \mathcal{G}\ \omega = Earley_L\_bin\ 0\ \mathcal{G}\ \omega\ (Init_L\ \mathcal{G}\ \omega)$$
$$Earley_L\_bins\ (Suc\ n)\ \mathcal{G}\ \omega = Earley_L\_bin\ (Suc\ n)\ \mathcal{G}\ \omega\ (Earley_L\_bins\ n\ \mathcal{G}\ \omega)$$

$$Earley_L\ \mathcal{G}\ \omega = Earley_L\_bins\ |\omega|\ \mathcal{G}\ \omega$$

5.2 Recognizer Correctness Proof

We follow Jones' [20] refinement approach, proving that the set of items formed by the implementation's bins is exactly the inductive set of Earley items, thereby establishing soundness and completeness. The main complications arise since the deterministic implementation necessarily generates the set of Earley items in a particular order. It starts with the initial items in bin zero and constructs the subsequent bins in a horizontal ascending order. But each bin is computed top to bottom, introducing a second vertical order. Our refinement approach reflects these two orders. We first refine the inductive definition to an intermediate fixpoint algorithm, and then refine this algorithm further to the actual list-based implementation.

Let *bin I k* denote the subset of the set of items I that end with index k. Furthermore, let *base* ω *I k* denote the subset of I that forms the k-th *base* of a bin, meaning the subset of I containing only items of the form $A \to \alpha a \bullet \beta, i, j$, where a is a terminal symbol preceding the dot. If k is zero, *base* ω *I 0* consists of all initial items $\mathfrak{S}\ \mathcal{G} \to \bullet\alpha, 0, 0$.

For the intermediate fixpoint algorithm we define the set of initial items Init_F and three functions $\mathit{Predict}_F$, Scan_F, and $\mathit{Complete}_F$ mirroring the rules of the inductive definition. Using $\mathit{Earley}_F\_bin\_step\ k\ \mathcal{G}\ \omega\ I = I \cup \mathit{Scan}_F\ k\ \omega\ I \cup \mathit{Complete}_F\ k\ I \cup \mathit{Predict}_F\ k\ \mathcal{G}\ I$ we define the computation of a single bin as a fixpoint computation. The remaining functions Earley_F_bins and Earley_F are defined analogously to the list-based implementation. The following lemma states the completeness argument for the first refinement step.

lemma $\mathit{Earley\_bin\_base\_sub\_Earley}_F\mathit{\_bin}$:
  assumes $\mathit{Init}_F\ \mathcal{G} \subseteq I$ **and** $\forall k' < k.\ \mathit{bin}\ (\mathit{Earley}\ \mathcal{G}\ \omega)\ k' \subseteq I$
  assumes $\mathit{base}\ \omega\ (\mathit{Earley}\ \mathcal{G}\ \omega)\ k \subseteq I$ **and** $\mathit{is\_word}\ \mathcal{G}\ \omega$
  shows $\mathit{bin}\ (\mathit{Earley}\ \mathcal{G}\ \omega)\ k \subseteq \mathit{bin}\ (\mathit{Earley}_F\mathit{\_bin}\ k\ \mathcal{G}\ \omega\ I)\ k\ \wedge$
    $\mathit{base}\ \omega\ (\mathit{Earley}\ \mathcal{G}\ \omega)\ (k{+}1) \subseteq \mathit{bin}\ (\mathit{Earley}_F\mathit{\_bin}\ k\ \mathcal{G}\ \omega\ I)\ (k{+}1)$

The fixpoint computation of the k-th bin yields a superset of the k-th bin and base $k{+}1$ of the inductive definition. We omit the proof, and the analogous but much simpler soundness lemma. As both the inductive and fixpoint definition commence with the same items, (*base*) ω $(\mathit{Earley}\ \mathcal{G}\ \omega)\ 0 = \mathit{Init}_F\ \mathcal{G}$, we apply this argument $|\omega|$ *times* (*i.e. by induction*), yielding the correctness proposition $\mathit{Earley}\ \mathcal{G}\ \omega = \mathit{Earley}_F\ \mathcal{G}\ \omega$.

Refining the algorithm further to the list-based implementation uncovers a well-known problem concerning the computation of a single bin. Consider an item $A \to \bullet, j, k$ for an epsilon rule $(A, [])$. Since the item is by definition complete the algorithm applies the $\mathit{Complete}_L$ operation. It identifies the origin bin j of the item. Due to the epsilon rule this is the k-th bin, meaning the bin that the algorithm is currently computing. It then searches this bin for any items $B \to \alpha \bullet A\beta, i, j$. However, the bin might not be fully constructed at this point, and some of these items could be missing. Consequently, the algorithm may not generate all items $B \to \alpha A \bullet \beta, i, j$, when applying the completion operation for the item $A \to \bullet, j, k$. Moreover, there could be transitively dependent items that the algorithm fails to compute. Various solutions have been proposed:

- Earley [12] suggests that the implementation keeps track of items with epsilon rules and considers this information in the subsequent execution of the algorithm.
- Grune and Jacobs [18] and Aho and Ullman [4] propose to interleave the prediction and completion operations until the algorithm stabilizes.
- Kegler [22] addresses the problem by internally rewriting the grammar into epsilon-free form.
- Aycock and Horspool [6] precompute nullable non-terminals and modify the prediction operation.
- Polat et al. [32] roughly follow the work of Aycock and Horspool.

We follow Jones [20], define $\varepsilon\_free\ \mathcal{G} = (\forall r{\in}set\ (\mathfrak{R}\ \mathcal{G}).\ rhs\_rule\ r \neq [])$, and consequently restrict the grammar to be epsilon free. If we disallow any production rules of the form $(A, [])$, then the function Earley_L_bin is idempotent and in particular the result of the completion operation is invariant of state of the current bin.

On paper this argument is straightforward, but the formalization is surprisingly tricky in the details. The function $\mathit{Earley}_L\_bin\ k\ \mathcal{G}\ \omega\ bs = \mathit{Earley}_L\_bin'\ k\ \mathcal{G}\ \omega\ bs\ 0$ is defined in terms of Earley_L_bin' which may start its computation at an arbitrary index i instead of *0*.

We need the following two generalized lemmas for the completeness proof.

lemma $\mathit{Earley_F\_bin\_step\_sub\_Earley_L\_bin'}$:
  assumes $(k, \mathcal{G}, \omega, \mathit{bs}) \in \mathit{wf\_earley\_input}$ **and** $\mathit{is\_word}\ \mathcal{G}\ \omega$
  assumes $\forall x \in \mathit{bins}\ \mathit{bs}.\ \mathit{sound\_item}\ \mathcal{G}\ \omega\ x$ **and** $\varepsilon\mathit{\_free}\ \mathcal{G}$
  assumes $\mathit{Earley_F\_bin\_step}\ k\ \mathcal{G}\ \omega\ (\mathit{bins\_upto}\ \mathit{bs}\ k\ i) \subseteq \mathit{bins}\ \mathit{bs}$
  shows $\mathit{Earley_F\_bin\_step}\ k\ \mathcal{G}\ \omega\ (\mathit{bins}\ \mathit{bs}) \subseteq \mathit{bins}\ (\mathit{Earley_L\_bin'}\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i)$

If applying a single step step of the fixpoint computation, $\mathit{Earley_F\_bin\_step}$, to the bins including the items of the first k bins but only up (but not including) the i-th item of the k-th bin doesn't change the content of the bins, or, in other words, those items are already correctly processed, then the list-based implementation computes at least the same items as applying one step of the fixpoint computation.

lemma $\mathit{Earley_L\_bin'\_idem}$:
  assumes $(k, \mathcal{G}, \omega, \mathit{bs}) \in \mathit{wf\_earley\_input}$
  assumes $i \leq j\ \forall x \in \mathit{bins}\ \mathit{bs}.\ \mathit{sound\_item}\ \mathcal{G}\ \omega\ x$ **and** $\varepsilon\mathit{\_free}\ \mathcal{G}$
  shows $\mathit{bins}\ (\mathit{Earley_L\_bin'}\ k\ \mathcal{G}\ \omega\ (\mathit{Earley_L\_bin'}\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i)\ j) = \mathit{bins}\ (\mathit{Earley_L\_bin'}\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i)$

Using those two lemmas we can prove completeness of the list-based algorithm for a single bin. Since the list-based algorithm follows the same horizontal order as the fixpoint algorithm the completeness proof for all bins is then straightforward. The soundness proof is again similar in structure, but once more much simpler.

We then define $\mathit{recognizer}\ \mathcal{G}\ \omega = (\exists\, x{\in}\mathit{set}\ (\mathit{items}\ (\mathit{Earley_L}\ \mathcal{G}\ \omega\ !\ |\omega|)).\ \mathit{is\_finished}\ \mathcal{G}\ \omega\ x)$, prove the equivalence of Earley and $\mathit{Earley_L}$ and obtain a corollary stating the correctness of the recognizer under the assumption of an epsilon-free grammar.

theorem $\mathit{Earley\_eq\_Earley_L}$:
  assumes $\mathit{is\_word}\ \mathcal{G}\ \omega$ **and** $\varepsilon\mathit{\_free}\ \mathcal{G}$
  shows $\mathit{Earley}\ \mathcal{G}\ \omega = \mathit{bins}\ (\mathit{Earley_L}\ \mathcal{G}\ \omega)$

corollary $\mathit{correctness\_recognizer}$:
  assumes $\mathit{is\_word}\ \mathcal{G}\ \omega$ **and** $\varepsilon\mathit{\_free}\ \mathcal{G}$
  shows $\mathit{recognizer}\ \mathcal{G}\ \omega \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

6 An Earley Parser

We now upgrade our recognizer to a parser. Extending an Earley recognizer to a parser is no simple task. Tomita [36] even pointed out a bug in Earley's original implementation that may lead to erroneous derivations.

A major complication is that Earley's parser allows for ambiguous grammars, which may lead to exponentially many or even infinitely many parse trees. For the ambiguous grammar $S \to SS \,|\, a$, the number of possible parse trees corresponds to the Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$ for an input of length $n-1$. For the cyclic grammar $A \to B \,|\, a, B \to A$ the input a has infinitely many parse trees because of the by cycle of non-terminals A and B.

An Earley recognizer can be made to run in at most quadratic space and cubic time. Any extension of the recognizer to a parser, especially for ambiguous or cyclic grammars, must choose a suitable data representation and be implemented carefully, in order not to degrade those time and space bounds too much.

Probably the most well-known data representation is the *shared packed parse forest* (SPPF), as described and used by Tomita [37]. However, Johnson [19] showed that these forests are of unbounded polynomial size in the worst case. On the other hand, Scott [35] introduced a slightly different version of SPPFs, and proved that her Earley parser implementation runs in cubic time and space.

Following Aho and Ullman [4], we choose to construct only a single parse tree, showing correctness but not completeness. Our formalization implements the parser as a separate algorithm that extracts the parse trees from the bins via the pointers that our recognizer maintains but does not utilize. The pointer implementation follows the work of Scott [35].

Our parse trees and two basic operations are defined like this:

datatype $'a\ tree = Leaf\ 'a \mid Branch\ 'a\ ('a\ tree\ list)$

$yield\ (Leaf\ a) = [a]$ $\qquad$ $root\ (Leaf\ a) = a$

$yield\ (Branch\ \_\ ts) = concat\ (map\ yield\ ts)$ $\qquad$ $root\ (Branch\ N\ \_) = N$

We introduce three notions of well-formedness for parse trees:

- A parse tree must represent a valid derivation tree according to the rules of the grammar:

  $wf\_rule\_tree\ \_\ (Leaf\ a) = True$
  $wf\_rule\_tree\ \mathcal{G}\ (Branch\ N\ ts) =$
  $((\exists\, r{\in}set\ (\mathfrak{R}\ \mathcal{G}).\ N = lhs\_rule\ r \wedge map\ root\ ts = rhs\_rule\ r) \wedge$
  $(\forall\, t{\in}set\ ts.\ wf\_rule\_tree\ \mathcal{G}\ t))$

- A tree corresponds to an Earley item in structure and in yield:

  $wf\_item\_tree\ \_\ uu\ (Leaf\ a) = True$
  $wf\_item\_tree\ \mathcal{G}\ x\ (Branch\ N\ ts) =$
  $((N = lhs\_item\ x \wedge map\ root\ ts = take\ (dot\_item\ x)\ (rhs\_item\ x)) \wedge$
  $(\forall\, t{\in}set\ ts.\ wf\_rule\_tree\ \mathcal{G}\ t))$

  $wf\_yield\ \omega\ x\ t = (yield\ t = \omega_{start\_item\ x/end\_item\ x})$

6.1 Parser Implementation

After executing the $Earley_L$ function, we obtain bins that represent the complete set of Earley items. The null, predecessor and reduction pointers of the entries serve as a means to navigate through these bins.

The semantics of the pointers is as follows: a null pointer accompanies an item if that item was predicted. An entry contains a predecessor pointer, *Pre pre*, if its item was obtained by applying the *Scan* operation to the item of the entry in the previous bin at index *pre*. An entry contains reduction pointers, *PreRed p ps*, if its item was computed by the completion operation. For any $(k', pre, red) \in set\ (p\ \#\ ps)$, there exists a predecessor item in bin k' at index *pre* and a complete reduction item in the same bin at index *red*.

The function *build_tree'* constructs a single parse tree corresponding to the item x in entry e at index i of the k-th bin:

```
$build\_tree'\ bs\ \omega\ k\ i = (let\ e = bs\ !\ k\ !\ i\ in$
  $case\ snd\ e\ of$
    $Null \Rightarrow Some\ (Branch\ (lhs\_item\ (fst\ e))\ [])$
  $\mid Pre\ pre \Rightarrow$
      $do\ \{\ t \leftarrow build\_tree'\ bs\ \omega\ (k{-}1)\ pre;$
            $case\ t\ of\ Branch\ A\ ts \Rightarrow Some\ (Branch\ A\ (ts\ @\ [Leaf\ (\omega!(k{-}1))]))\ \}$
  $\mid PreRed\ (k', pre, red)\ \_ \Rightarrow$
      $do\ \{\ t \leftarrow build\_tree'\ bs\ \omega\ k'\ pre;$
            $case\ t\ of\ Branch\ A\ ts \Rightarrow$
              $do\ \{\ t \leftarrow build\_tree'\ bs\ \omega\ k\ red;$
                  $Some\ (Branch\ A\ (ts\ @\ [t]))\ \}\})$
```

If the pointer of e is a null pointer, the algorithm begins building a new branch. Specifically, it constructs a branch *Branch A* [], where A is the left-hand side symbol of the production rule of the item x. If the algorithm encounters a predecessor pointer *Pre pre*, it recursively calls itself for the previous bin, $k-1$, at index *pre*. This recursive call results in a partially completed parse branch. Following the semantics of the predecessor pointer, the algorithm appends a new Leaf containing the terminal symbol at index $k-1$ of the input ω to the list of subtrees of this branch. In the case of a reduction pointer, the algorithm considers only the first triple (k', *pre*, *red*). It calls itself recursively for the predecessor entry in bin k' at index *pre* and the completed entry in the same bin at index *red*. These recursive calls yield respectively a partially completed parse branch *Branch A ts* and a complete parse tree t. Following the semantics of the reduction pointer, the complete branch t is appended to the list of subtrees *ts*.

The final algorithm *build_tree* (definition omitted) first computes the complete bins using the function $Earley_L$. It then searches the last bin for any finished item x, and calls the function *build_tree'* at the index of x in the final bin, returning the resulting parse tree as an optional value, if such a tree exists, or *None* in case of non-termination.

6.2 Proving Termination

The function *build_tree'* is a partial function. It calls itself recursively, following the information provided by the pointers. Intuitively, it terminates because predecessor pointers lead to earlier bins, and reduction pointers point upwards within a bin. Consequently, we define a measure *build_tree'_measure* (*bs*, ω, *k*, *i*) = *foldl* (+) *0* (*map length* (*take k bs*)) + *i*, counting the number of entries in the first k bins up to the i-th entry in bin *k+1*. But in the case of malformed input, the pointers might result in a cycle of recursive calls and thus the measure is not strictly decreasing. And even for well-formed input, complications arise.

Consider an entry at index i in the k-th bin. If the entry contains a reduction triple (k', *pre*, *red*), the algorithm calls itself recursively for the reduction entry at index *red* in bin k. Now consider the cyclic grammar $A \to B \mid a, B \to A$ and the input $\omega = a$. In this case, the last bin contains a cycle of reductions: there is an entry $B \to A\bullet, 0, 1; (0,2,0), (0,2,2)$ at index 1, and its *second* reduction triple $(0,2,2)$ leads to index 2 of the same bin. There, we find the entry $A \to B\bullet, 0, 1; (0,0,1)$ with a reduction triple to index 1, completing the cycle, and leading to potential non-termination of the algorithm.

We constrain the input of the function *build_tree'* to *wf_tree_input* = {(*bs*, ω, *k*, *i*) | *sound_ptrs* ω *bs* $\wedge$ *mono_red_ptr bs* $\wedge$ $k < |bs| \wedge k \le |\omega| \wedge i < |bs\ !\ k|$} where the definition of *sound_ptrs* and *mono_red_ptr* is the following:

$$
\begin{aligned}
&\mathit{sound\_ptrs}\ \omega\ \mathit{bs} = (\forall k < |\mathit{bs}|.\ \forall e \in \mathit{set}\ (\mathit{bs}!k).\\
&\quad (\mathit{snd}\ e = \mathit{Null} \longrightarrow \mathit{predicts}\ (\mathit{fst}\ e)) \wedge\\
&\quad (\forall \mathit{pre}.\ \mathit{snd}\ e = \mathit{Pre}\ \mathit{pre} \longrightarrow\\
&\qquad k > 0 \wedge \mathit{pre} < |\mathit{bs}!(k{-}1)| \wedge \mathit{scans}\ \omega\ k\ (\mathit{fst}\ (\mathit{bs}!(k{-}1)!\mathit{pre}))\ (\mathit{fst}\ e)) \wedge\\
&\quad (\forall p\ \mathit{ps}\ k'\ \mathit{pre}\ \mathit{red}.\ \mathit{snd}\ e = \mathit{PreRed}\ p\ \mathit{ps} \wedge (k', \mathit{pre}, \mathit{red}) \in \mathit{set}\ (p\#\mathit{ps}) \longrightarrow\\
&\qquad k' < k \wedge \mathit{pre} < |\mathit{bs}!k'| \wedge \mathit{red} < |\mathit{bs}!k| \wedge \mathit{completes}\ k\ (\mathit{fst}\ (\mathit{bs}!k'!\mathit{pre}))\ (\mathit{fst}\ e)\ (\mathit{fst}\ (\mathit{bs}!k!\mathit{red}))))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mono\_red\_ptr}\ \mathit{bs} = (\forall k < |\mathit{bs}|.\ \forall i < |\mathit{bs}!k|.\\
&\quad \forall k'\ \mathit{pre}\ \mathit{red}\ \mathit{ps}.\ \mathit{snd}\ (\mathit{bs}!k!i) = \mathit{PreRed}\ (k', \mathit{pre}, \mathit{red})\ \mathit{ps} \longrightarrow \mathit{red} < i)
\end{aligned}
$$

The predicate *sound_ptrs* defines the semantics for the pointer datatype, ensuring that the pointers do not exceed the bounds of the bins and that related items follow the semantics of the respective operation. The function *build_tree'* always follows the first reduction triple (k', *pre*, *red*) $\in$ *set* (*p* # *ps*) for a reduction pointer *PreRed p ps*, and the predicate *mono_red_ptr* guarantees that the reduction pointer *red* of this reduction triple always points strictly upwards within the bin, even for cyclic grammars as in the example above, enabling us to prove termination of the algorithm.

lemma *build_tree'_termination*:
  assumes $(bs,\ \omega,\ k,\ i) \in \mathit{wf\_tree\_input}$
  shows $\exists\, A\ ts.\ \mathit{build\_tree'}\ bs\ \omega\ k\ i = \mathit{Some}\ (\mathit{Branch}\ A\ ts)$

6.3 Proving Correctness

To prove the correctness of the parse tree algorithm, we first show that the resulting tree corresponds in derivation and yield to the Earley item where the construction originated.

theorem *wf_item_yield_build_tree'*:
  assumes $(bs,\ \omega,\ k,\ i) \in \mathit{wf\_tree\_input}$ **and** $\mathit{wf\_bins}\ \mathcal{G}\ \omega\ bs$
  assumes $\mathit{build\_tree'}\ bs\ \omega\ k\ i = \mathit{Some}\ t$ **and** $x = \mathit{fst}\ (bs!k!i)$
  shows $\mathit{wf\_item\_tree}\ \mathcal{G}\ x\ t \wedge \mathit{wf\_yield}\ \omega\ x\ t$

The predicate *wf_bins* states that each bin only contains entries with distinct items, all items are well-formed and their end index equals the index of the bin they reside in. The proof is by induction on $\mathit{build\_tree'\_measure}\ (bs,\ \omega,\ k,\ i)$.

As a corollary, we obtain the first correctness statement for epsilon-free grammars: any constructed parse tree adheres to the rules of the grammar, is rooted at its start symbol and yields the complete input:

corollary *wf_rule_root_yield_build_tree_Earley*$_L$:
  assumes $\varepsilon\mathit{\_free}\ \mathcal{G}$ **and** $\mathit{build\_tree}\ \mathcal{G}\ \omega\ (\mathit{Earley}_L\ \mathcal{G}\ \omega) = \mathit{Some}\ t$
  shows $\mathit{wf\_rule\_tree}\ \mathcal{G}\ t \wedge \mathit{root}\ t = \mathfrak{S}\ \mathcal{G} \wedge \mathit{yield}\ t = \omega$

The *build_tree* function scans the last bin for any finished item x, and calls the function *build_tree'*. Given that the function Earley_L guarantees well-formed tree inputs, the resulting tree conforms both in derivation and yield to the item x, as proved in the preceding theorem. Since x is a finished item, the root of the tree corresponds to the start symbol of the grammar, the tree's yield encompasses the complete input, and the definition of *wf_item_tree* aligns with the definition of *wf_rule_tree*.

The second and final correctness theorem follows: the algorithm returns a parse tree, if and only if, there exists a derivation of ω from the grammar's start symbol.

theorem *correctness_build_tree_Earley*$_L$:
  assumes $\mathit{is\_word}\ \mathcal{G}\ \omega$ **and** $\varepsilon\mathit{\_free}\ \mathcal{G}$
  shows $(\exists\, t.\ \mathit{build\_tree}\ \mathcal{G}\ \omega\ (\mathit{Earley}_L\ \mathcal{G}\ \omega) = \mathit{Some}\ t) \longleftrightarrow \mathcal{G} \vdash [\mathfrak{S}\ \mathcal{G}] \Rightarrow^* \omega$

The function *build_tree* finds a finished item in the last bin and returns a parse tree, if and only if, the bins generated by the function Earley_L contain a finished item. These items align precisely with those items specified by the inductive definition. Lastly, the inductive set of Earley items contains a finished item, if and only if, there exists a derivation of the input from the grammar's start symbol.

7 Evaluation

We present an informal argument for the algorithm's running time, and empirically access its efficiency. Given that we construct solely a single parse tree, the running time is dominated by the function Earley_L.

Let n denote the length of the input ω. Each bin B_j $(0 \leq j \leq n)$ exclusively contains well-formed items *Item r d i j*. The number of possible production rules r, and possible dot positions d, are both constants independent of n, although they may be rather large. The index i is bounded by $0 \leq i \leq j$, and thus depends on j, which is bounded by n. The end index j always equals the index of the bin. Thus, the number of items in each bin B_j is $\mathcal{O}(n)$. The initial bin construction takes linear time. Scanning is also a linear operation, producing at most one new entry. Prediction runs in time proportional to the grammar, or constant time, adding a constant amount of new entries. Lastly, completion takes linear time as it scans the entire origin bin for applicable items. However, as the size of the origin bin is linear, this operation adds $\mathcal{O}(n)$ new entries to the current bin. The algorithm implements a bin as a list, so inserting e new entries into any bin takes time $e \times \mathcal{O}(n)$. The time for one execution of the body of the function $\mathit{Earley}_L\_\mathit{bin}'$ is dominated by the time it takes to update the bins with e new items. In the worst case, this is completion, resulting in a running time of $e \times \mathcal{O}(n)$, for $e \in \mathcal{O}(n)$. Moreover, the function calls itself at most n times due to the size of the bin it operates on, resulting in cubic time complexity. Finally, the algorithm iterates over all n bins, leading to a final upper bound on the running time of $\mathcal{O}(n^4)$.

The space complexity for an Earley recognizer is quadratic: n bins of linear size. However, as each entry may have n reduction pointers in the worst case, the space required to represent all Earley entries with pointers becomes cubic in n.

It is worth noting that the running time is not optimal. Earley's [12] original implementation achieves an optimal running time of $\mathcal{O}(n^3)$ by implementing a bin as an imperative singly-linked list and additionally maintaining a cache of already inserted items. This cache reduces the insertion time of a new entry into a bin from linear to constant time such that computing a single bin takes in the worst case quadratic time. Further refinement of our functional implementation to an imperative algorithm with cache is future work.

Additionally, we evaluate the running time of the exported recognizer Isabelle code in comparison to a hand-written imperative implementation of Earley's recognizer. Our target language is Scala (verified as well as handwritten) and the verified code can be integrated easily into existing code bases. Alternatively, Isabelle also supports Haskell, ML, and OCaml. We then conducted tests on five different grammars, averaging the execution of ten runs for each data point. The grammars and running times for Earley's [12] original implementation are:

1. An ambiguous grammar $S \to SS \mid a$, cubic running time.
2. A right-recursive grammar $S \to aS \mid a$, quadratic running time.
3. A palindrome grammar $S \to aSa \mid a$, quadratic running time.
4. A left recursive grammar $S \to Sa \mid a$, linear running time.
5. A grammar with bounded-ambiguity $S \to SX \mid a, X \to Y \mid Z, Y \to a, Z \to a$, linear running time.

Figure 1 depicts the running times in milliseconds for the exported Isabelle code on all five grammars. Figure 2 compares the verified code against the handwritten recognizer implementation for grammars one, two, and four. In both cases, the input is $\omega = a^n$. For sufficiently large inputs the hand-written implementation exhibits optimal running time, while the verified code exhibits a linear slowdown in the size of the input . This is also confirmed by a regression analysis. It is possible to fit polynomial models of the respective order, capturing the expected running time, to the data set, depicted as solid lines in the figures.

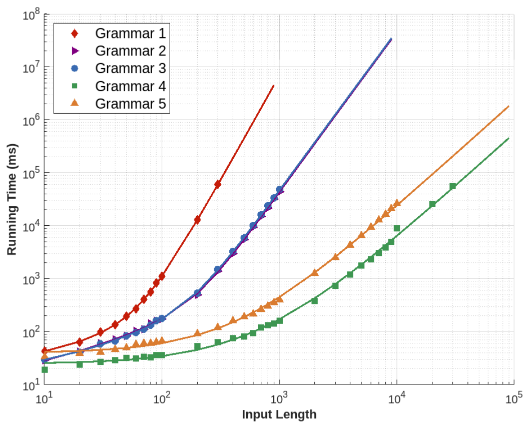

Figure 1 Isabelle: all grammars.

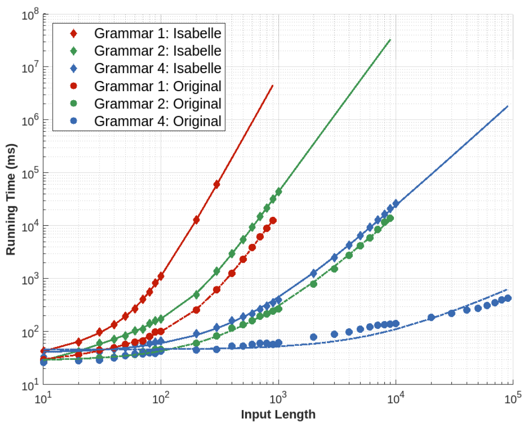

Figure 2 Isabelle vs Handwritten.

8 Related Work

We highlight related work on formalization of parsing algorithms, starting with LL-based parsing: Lasser et al. [25] verify an LL(1) parser generator using the Coq proof assistant. Edelmann et al. [13] formalize a derivative-based LL(1) parsing algorithm, proving correctness using the Coq proof assistant.

There also exist verified LR-based algorithms: Barthwal et al. [7] formalize background theory about context-free languages and grammars, and subsequently verify an SLR automaton and parser produced by a parser generator with the HOL4 proof assistant. Jourdan et al. [21] present a validator which checks if a context-free grammar and an LR(1) parser agree, producing correctness guarantees required by verified compilers.

Furthermore, there is relevant work on the verification of PEGs [17, 16], an alternative representation to CFGs. Blaudeau et al. [8] formalize the meta theory of PEGs. They build a verified parser interpreter based on higher-order parsing combinators for expression grammars using the PVS specification language and verification system. Koprowski et al. [23] present TRX: a PEG interpreter formally developed in Coq which also parses expression grammars.

Lastly, there exist a variety of verified parsers for general context-free grammars. Ridge [34] constructs a generic parser generator based-on combinator parsing. His approach has a worst case complexity of $\mathcal{O}(n^5)$ and is verified using the HOL4 theorem prover. Obua formalizes Local Lexing [31, 30] in Isabelle, a parsing concept that interleaves lexing and parsing allowing the lexing phase to be dependent on the parsing process. Firsov and Uustalu [14, 15] rewrite a context-free grammar into an equivalent one in Chomsky normal form and implement the CYK parsing algorithm. They verify their work in Agda. The CYK algorithm had already been verified by Bortin [9]. Danielsson [11] develops and verifies a monadic parser combinator library in Agda.

9 Conclusions

We formalized and verified a functional implementation of an Earley recognizer and parser based on Earley's [12] original imperative implementation, the refinement-based paper proof of Jones [20], and the work of Scott [35]. Initially, we defined an Earley recognizer inductively and proved soundness and completeness. We refined the inductive definition to a functional recognizer implementation (proving equivalence between the two levels). We also enhanced the implementation with "pointers", following the work of Scott [35]. Following Aho and Ullman [4], we implemented a functional algorithm that constructs a single parse tree, and proved its termination and correctness. Finally, we argued informally about the running time of our functional implementation, comparing it to an asymptotically optimal, hand-written, imperative implementation and providing empirical evidence supporting our claims.

Future work is mainly centered around improving the algorithm's efficiency. A first step is a refinement to an imperative implementation that incorporates a cache to achieve optimal cubic time and space bounds. Further performance optimizations include improving the representation of the grammar for faster prediction [32] and grouping the items of a bin based on their next symbol [12]. This avoids searching the complete origin bin during the completion operation. Leo [26] describes an extension applicable to an Earley recognizer and parser that improves the complexity for grammars containing right recursion from quadratic to linear time and space. Earley [12] suggested using lookhead for the completion operation to improve the performance of his algorithm. However, Bouckaert et al. [10] argued that lookahead is better suited for the prediction operation. McLean and Horsool [27] claimed that lookahead actually slowed down an Earley parser, and Aycock and Horspool [5] concluded that the necessity of lookahead is at least controversial. Lastly, we would like to incorporate the work of Aycock and Horspool [6] and Polat et al. [32] to lift the minor restriction to epsilon-free grammars.

References

1 Failure to patch two-month-old bug led to massive equifax breach. `https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug`. Accessed: 2024-03-16.

2 Stagefright: Scary code in the heart of android. `https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf`. Accessed: 2024-03-16.

3 Windows media parsing remote code execution vulnerability. `https://nvd.nist.gov/vuln/detail/CVE-2016-0101`. Accessed: 2024-03-16.

4 Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., USA, 1972.

5 John Aycock and R. Nigel Horspool. Directly-executable Earley parsing. In Reinhard Wilhelm, editor, *Compiler Construction, CC 2001*, volume 2027 of *LNCS*, pages 229–243. Springer, 2001. `doi:10.1007/3-540-45306-7_16`.

6 John Aycock and R. Nigel Horspool. Practical Earley parsing. *Comput. J.*, 45(6):620–630, 2002. `doi:10.1093/comjnl/45.6.620`.

7 Aditi Barthwal and Michael Norrish. Verified, executable parsing. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 160–174, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

8 Clement Blaudeau and Natarajan Shankar. A verified packrat parser interpreter for parsing expression grammars. In *Certified Programs and Proofs*, CPP 2020, pages 3–17. ACM, 2020. `doi:10.1145/3372885.3373836`.

9 Maksym Bortin. A formalisation of the Cocke-Younger-Kasami algorithm. *Archive of Formal Proofs*, April 2016. , Formal proof development. URL: `https://isa-afp.org/entries/CYK.html`.

10 M. Bouckaert, A. Pirotte, and M. Snelling. Efficient parsing algorithms for general context-free parsers. *Information Sciences*, 8(1):1–26, 1975. `doi:10.1016/0020-0255(75)90002-X`.

11 Nils Anders Danielsson. Total parser combinators. In Paul Hudak and Stephanie Weirich, editors, *International Conference on Functional Programming, ICFP 2010*, pages 285–296. ACM, 2010. doi:10.1145/1863543.1863585.

12 Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.

13 Romain Edelmann, Jad Hamza, and Viktor Kunčak. Zippy LL(1) parsing with derivatives. In *Programming Language Design and Implementation*, PLDI 2020, pages 1036–1051. ACM, 2020. doi:10.1145/3385412.3385992.

14 Denis Firsov and Tarmo Uustalu. Certified CYK parsing of context-free languages. *Journal of Logical and Algebraic Methods in Programming*, 83(5):459–468, 2014. Nordic Workshop on Programming Theory (NWPT 2012). doi:10.1016/j.jlamp.2014.09.002.

15 Denis Firsov and Tarmo Uustalu. Certified normalization of context-free grammars. In *Certified Programs and Proofs*, CPP '15, pages 167–174. ACM, 2015. doi:10.1145/2676724.2693177.

16 Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *International Conference on Functional Programming*, ICFP '02, pages 36–47. ACM, 2002. doi:10.1145/581478.581483.

17 Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Principles of Programming Languages*, POPL '04, pages 111–122. ACM, 2004. doi:10.1145/964001.964011.

18 Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: a Practical Guide*. Ellis Horwood, 1990.

19 Mark Johnson. *The Computational Complexity of GLR Parsing*, pages 35–42. Springer US, Boston, MA, 1991. doi:10.1007/978-1-4615-4034-2_3.

20 C B Jones. Formal development of correct algorithms: An example based on Earley's recogniser. In *Proceedings of ACM Conference on Proving Assertions about Programs*, pages 150–169. ACM, 1972. doi:10.1145/800235.807083.

21 Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In Helmut Seidl, editor, *European Symposium on Programming, ESOP 2012*, volume 7211 of *LNCS*, pages 397–416. Springer, 2012. doi:10.1007/978-3-642-28869-2_20.

22 Jeffrey Kegler. Marpa, a practical general parser: the recognizer, 2023. arXiv:1910.08129.

23 Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. *Log. Methods Comput. Sci.*, 7(2), 2011. doi:10.2168/LMCS-7(2:18)2011.

24 Alexander Krauss. Recursive definitions of monadic functions. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010*, volume 5 of *EPiC Series*, pages 1–13. EasyChair, 2010. doi:10.29007/1mdt.

25 Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. A Verified LL(1) Parser Generator. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2019.24.

26 Joop M.I.M. Leo. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science*, 82(1):165–176, 1991. doi:10.1016/0304-3975(91)90180-A.

27 Philippe McLean and R. Nigel Horspool. A faster Earley parser. In Tibor Gyimóthy, editor, *Compiler Construction, CC'96*, volume 1060 of *LNCS*, pages 281–293. Springer, 1996. doi:10.1007/3-540-61053-7_68.

28 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. URL: http://concrete-semantics.org.

29 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

30 Steven Obua. Local lexing. *Archive of Formal Proofs*, 2017. , Formal proof development. URL: https://isa-afp.org/entries/LocalLexing.html.

31 Steven Obua, Phil Scott, and Jacques Fleuriot. Local lexing, 2017. `arXiv:1702.03277`.

32 Sinan Polat, Merve Selcuk-Simsek, and Ilyas Cicekli. A modified Earley parser for huge natural language grammars. *Res. Comput. Sci.*, 117:23–35, 2016. URL: `https://rcs.cic.ipn.mx/2016_117/A%20Modified%20Earley%20Parser%20for%20Huge%20Natural%20Language%20Grammars.pdf`.

33 Martin Rau. Earley parser. *Archive of Formal Proofs*, July 2023. , Formal proof development. URL: `https://devel.isa-afp.org/entries/Earley_Parser.html`.

34 Tom Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs, CPP 2011*, volume 7086 of *LNCS*, pages 103–118. Springer, 2011. `doi:10.1007/978-3-642-25379-9_10`.

35 Elizabeth Scott. SPPF-style parsing from Earley recognisers. *Electronic Notes in Theoretical Computer Science*, 203(2):53–67, 2008. Workshop on Language Descriptions, Tools, and Applications (LDTA 2007). `doi:10.1016/j.entcs.2008.03.044`.

36 Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, USA, 1985.

37 Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1–2):31–46, January 1987.