Well-typed Islands Parse Faster

Erik Silkensen and Jeremy Siek

University of Colorado
Boulder, CO, USA
{erik.silkensen,jeremy.siek}@colorado.edu

Abstract. This paper addresses the problem of specifying and parsing the syntax of domain-specific languages (DSLs) in a modular, user-friendly way. We want to enable the design of *composable* DSLs that combine the natural syntax of external DSLs with the easy implementation of internal DSLs. The challenge in parsing these DSLs is that the composition of several languages is likely to contain ambiguities. We present the design of a system that uses a type-oriented variant of island parsing to efficiently parse the syntax of composable DSLs. In particular, we argue that the running time of type-oriented island parsing doesn't depend on the number of DSLs imported. We also show how to use our tool to implement DSLs on top of a host language such as Typed Racket.

1 Introduction

Domain-specific languages (DSLs) provide high productivity for programmers in many domains, such as computer systems, linear algebra, and other sciences. However, a series of trade-offs face the prospective DSL designer today. On one hand, many general-purpose languages include a host of tricks for implementing *internal*, or embedded DSLs, e.g., templates in C++ [2], macros in Scheme [25], and type classes in Haskell [11]. These features allow DSL designers to take advantage of the underlying language and to enjoy an ease of implementation. However, the resulting DSLs are often leaky abstractions, with a syntax that is not quite right, compilation errors that expose the internals of the DSL, and a lack of diagnostic tools that are aware of the DSL [21]. On the other hand, one may choose to implement their language by hand or with parser generators *à la* YACC. The resulting *external* DSLs achieve a natural syntax and often provide more friendly diagnostics, but come at the cost of interoperability issues [3] and an implementation that requires computer science expertise.

In this paper, we make progress towards combining the best of both worlds into what we call *composable* DSLs. Since applications routinely use multiple DSLs, our goal is to enable fine-grained mixing of languages with the natural syntax of external DSLs and the interoperability of internal DSLs. At the core of this effort is a parsing problem: although the grammar for each DSL in use may be unambiguous, programs, such as the one in Figure 1, need to be parsed using the union of their grammars, which is likely to contain ambiguities [14]. Instead of relying on the grammar author to resolve them (as in the LALR tradition), the parser for such an application must efficiently deal with ambiguities.

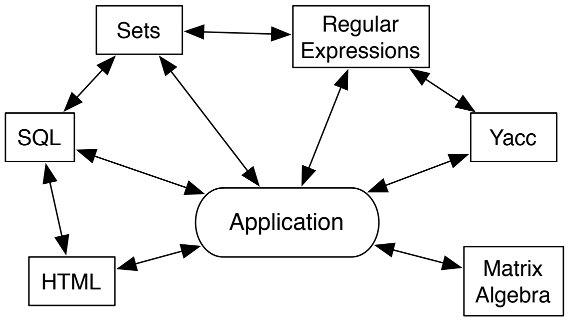


Fig. 1. Our common case: an application using many DSLs.

We should emphasize that our goal is to create a parsing system that provides much more syntactic flexibility than is currently offered through operator overloading in languages such as C++ and Haskell. However, we are not trying to build a general purpose parser; that is, we are willing to place restrictions on the allowable grammars, so long as those restrictions are easy to understand (for our users) and do not interfere with composability.

As a motivating example, we consider an application that imports DSLs for matrix algebra, sets, and regular expressions. Suppose the grammars for these languages are written in the traditional style, including the following rules, with associativity specified separately.

```
Expr ::= Expr "+" Expr | Expr "-" Expr
```
(Matrix DSL)

```
Expr ::= Expr "+" Expr | Expr "-" Expr    Expr ::= Expr "+"
```
(Set DSL) (Regexp DSL)

The union of these individually unambiguous grammars is greatly ambiguous, so importing them can increase the parse time by orders of magnitude without otherwise changing programs containing expressions such as `A + B + C`. An obvious fix is to merge the grammars and refactor to remove ambiguity. However, that would require coordination between the DSL authors which is not scalable.

1.1 Type-Oriented Grammars

To address the problem of parsing composed DSLs, we observe that different DSLs typically define different types. We suggest an alternate style of grammar organization inspired by Sandberg [20] that we call *type-oriented grammars*. In this style, a DSL author creates one nonterminal for each type in the DSL and uses the most specific nonterminal/type for each operand in a grammar rule. For example, the above `Expr` rules would instead be written

```
Matrix ::= Matrix "+" Matrix | Matrix "-" Matrix

Set ::= Set "+" Set | Set "-" Set    Regexp ::= Regexp "+"
```

1.2 Type-based Disambiguation

While we can rewrite the DSLs for matrix algebra, regular expressions, and sets to be type oriented, programs such as $\texttt{A} + \texttt{B} + \texttt{C} \cdots$ are still highly ambiguous if the variables `A`, `B`, and `C` can each be parsed as either `Matrix`, `Regexp`, or `Set`. Many prior systems [17, 5] use chart parsing [15] or GLR [26] to produce a parse forest and then type check to filter out the ill-typed trees. This solves the ambiguity problem, but these parsers are still inefficient on ambiguous grammars because of the large number of parse trees in the forest (see Section 4).

This is where our contribution comes in: *island parsing with eager, type-based disambiguation* is able to efficiently parse programs that simultaneously use many DSLs. We use a chart parsing strategy, called island parsing [23] (or bidirectional bottom-up parsing [19]), that enables our algorithm to grow parse trees outwards from what we call *well-typed terminals*. The statement

declare `A:Matrix, B:Matrix, C:Matrix` { ... }

gives the variables `A`, `B`, and `C` the type `Matrix` and brings them into scope for the region of code within the braces. We integrate type checking into the parsing process to prune ill-typed parse trees before they have a chance to grow, drawing inspiration from the field of natural language processing, where *selection restriction* uses types to resolve ambiguity [13].

Our approach does not altogether prohibit grammar ambiguities; it strives to remove ambiguities from the common case when composing DSLs so as to enable efficient parsing.

1.3 Contributions

1. We present the first parsing algorithm, *type-oriented island parsing* (Section 3), whose time complexity is *constant* with respect to (i.e., independent of) the number of DSLs in use, so long as the nonterminals of each DSL are largely disjoint (Section 4).
2. We present our extensible parsing system that adds several features to the parsing algorithm to make it convenient to develop DSLs on top of a host language such as Typed Racket [24] (Section 5).
3. We demonstrate the utility of our parsing system with examples (included along with the implementation available on Racket's PLaneT package repository) in which we embed syntax for DSLs in Typed Racket.

Section 2 introduces the basic definitions and notation used in the rest of the paper. We discuss our contributions in relation to the prior literature and conclude in Section 6.

2 Background

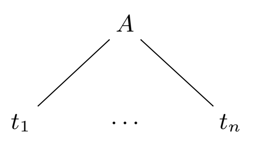
We review the definition of a grammar and parse tree and present a framework for comparing parsing algorithms based on the parsing schemata of Sikkel [22].

2.1 Grammars and Parse Trees

A *context-free grammar* (CFG) is a 4-tuple $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ where Σ is a finite set of terminals, Δ is a finite set of nonterminals, $\mathcal{P}$ is finite set of grammar rules, and S is the start symbol. We use a, b, c, and d to range over terminals and A, B, C, and D to range over nonterminals. The variables X, Y, Z range over symbols, that is, terminals and nonterminals, and $\alpha, \beta, \gamma, \delta$ range over sequences of symbols. Grammar rules have the form $A \to \alpha$. We write $\mathcal{G} \cup (A \to \alpha)$ as an abbreviation for $(\Sigma, \Delta, \mathcal{P} \cup (A \to \alpha), S)$.

We are ultimately interested in parsing programs, that is, converting token sequences into abstract syntax trees. So we are less concerned with the recognition problem and more concerned with determining the parse trees for a given grammar and token sequence. The *parse trees* for a grammar $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$, written $\mathcal{T}(\mathcal{G})$, are trees built according to the following rules.

1. If $a \in \Sigma$, then a is a parse tree labeled with a.
2. If $t_1, \ldots, t_n$ are parse trees labeled $X_1, \ldots, X_n$ respectively, $A \in \Delta$, and $A \to X_1 \ldots X_n \in \mathcal{P}$, then the following is a parse tree labeled with A.



We sometimes use a horizontal notation $A \to t_1 \ldots t_n$ for parse trees and we often subscript parse trees with their labels, so t_A is parse tree t whose root is labeled with A. We use an overline to represent a sequence: $\bar{t} = t_1 \ldots t_n$.

The *yield* of a parse tree is the concatenation of the labels on its leaves:

$$\begin{aligned} \mathit{yield}(a) &= a \\ \mathit{yield}([A \to t_1 \ldots t_n]) &= \mathit{yield}(t_1) \ldots \mathit{yield}(t_n) \end{aligned}$$

Definition 2.1. The set of parse trees for a CFG $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ and input w, written $\mathcal{T}(\mathcal{G}, w)$, is defined as follows

$$\mathcal{T}(\mathcal{G}, w) = \{t_S \mid t_S \in \mathcal{T}(\mathcal{G}) \text{ and } \mathit{yield}(t_S) = w\}$$

2.2 Parsing Algorithms

We wish to compare the essential characteristics of several parsing algorithms without getting distracted by implementation details. Sikkel [22] introduces a high-level formalism for presenting and comparing parsing algorithms, called *parsing schemata*, that presents each algorithm as a deductive system. We loosely follow his approach, but make some minor changes to better suit our needs.

Each parsing algorithm corresponds to a deductive system with judgments of the form $H \vdash \xi$, where ξ is an *item* and H is a set of items. An item has the form $[p, i, j]$ where p is either a parse tree or a partial parse tree and the integers i and j mark the left and right extents of what has been parsed so far.

$$(\mathrm{BU})\,\dfrac{(\textsc{Fnsh})\,\dfrac{(\mathrm{BU})\,\dfrac{(\textsc{Hyp})\,\dfrac{[\mathtt{A},0,1]\in H}{H\vdash[\mathtt{A},0,1]}\qquad \mathtt{E}\to\mathtt{A}\in\mathcal{P}}{H\vdash[\mathtt{E}\to\boldsymbol{.}\mathtt{A}\boldsymbol{.},0,1]}}{H\vdash[\mathtt{E}\to\mathtt{A},0,1]}\qquad \mathtt{E}\to\mathtt{E+E}\in\mathcal{P}}{H\vdash[\mathtt{E}\to\boldsymbol{.}[\mathtt{E}\to\mathtt{A}]\boldsymbol{.}\ \mathtt{+}\ \mathtt{E},0,1]}$$

Fig. 2. A partial (bottom-up Earley) derivation of the parse tree for `"A + B"`, having parsed `"A "` but not yet `"+ B"`.

The set of *partial parse trees* is defined by the following rule: if $A \to \alpha\beta\gamma \in \mathcal{P}$, then $A \to \alpha\boldsymbol{.}\bar{t}_\beta\boldsymbol{.}\gamma$ is a partial parse tree labeled with A, where markers surround a sequence of parse trees for β, while α and γ remain to be parsed. We reserve the variables s and t for parse trees, not partial parse trees. A *complete parse* of an input w of length n is a derivation of $H_0(w) \vdash [t_S, 0, n]$, where $H_0(w)$ is the initial set of items that represent the result of tokenizing the input w.

$$H_0(w) = \{[w_i, i, i+1] \mid 0 \le i < |w|\}$$

Definition 2.2. A bottom-up variation [22] of the Earley algorithm [8] applied to a grammar $\mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S)$ is defined by the following deductive rules.

$$(\textsc{Hyp})\,\dfrac{\xi \in H}{H \vdash \xi} \qquad (\textsc{Fnsh})\,\dfrac{H \vdash [A \to \boldsymbol{.}\bar{t}_\alpha\boldsymbol{.}, i, j]}{H \vdash [A \to \bar{t}_\alpha, i, j]}$$

$$(\mathrm{BU})\,\dfrac{H \vdash [t_X, i, j] \qquad A \to X\beta \in \mathcal{P}}{H \vdash [A \to \boldsymbol{.}t_X\boldsymbol{.}\beta, i, j]}$$

$$(\textsc{Compl})\,\dfrac{H \vdash [A \to \boldsymbol{.}\bar{s}_\alpha\boldsymbol{.}X\beta, i, j] \qquad H \vdash [t_X, j, k]}{H \vdash [A \to \boldsymbol{.}\bar{s}_\alpha t_X\boldsymbol{.}\beta, i, k]\}}$$

Example 2.1. Figure 2 shows the beginning of the bottom-up Earley derivation of a parse tree for `A + B` with the grammar:

```
E ::= E "+" E | "A" | "B"
```

3 Type-Oriented Island Parsing

The essential ingredients of our parsing algorithm are type-based disambiguation and island parsing. In Section 4, we show that an algorithm based on these two ideas parses with time complexity that is independent of the number of DSLs in use, so long as the nonterminals of the DSLs are largely disjoint. (We also make this claim more precise.) But first, in this section we introduce our type-oriented island parsing algorithm (TIP) as an extension of the bottom-up Earley algorithm (Definition 2.2).

Island parsing [23] is a bidirectional, bottom-up parsing algorithm that was developed in the context of speech recognition. In that domain, some tokens can be identified with a higher confidence than others. The idea of island parsing is to begin the parsing process at the high confidence tokens, the so-called islands, and expand the parse trees outward from there.

Our main insight is that if our parser can be made aware of variable declarations, and if a variable's type corresponds to a nonterminal in the grammar, then each occurrence of a variable can be treated as an island. We introduce the following special form for declaring a variable a of type A that may be referred to inside the curly brackets.

$$\textbf{declare}\ a : A\ \{\ldots\}$$

Specifically, if $t_X \in \mathcal{T}(\mathcal{G} \cup \{A \to a\})$, then the following is a parse tree in $\mathcal{T}(\mathcal{G})$.

$$X \to \textbf{declare}\ a : A\ \{t_X\}$$

To enable temporarily extending the grammar during parsing, we augment the judgments of our deductive system with an explicit parameter for the grammar. So judgments now have the form

$$\mathcal{G}; H \vdash \xi$$

This adjustment also enables the import of grammars from different modules.

We define the parsing rule for the **declare** form as follows.

$$(\textsc{Decl})\frac{\mathcal{G} \cup (A \to a); H \vdash [t_X, i+5, j]}{\mathcal{G}; H \vdash [X \to \textbf{declare}\ a : A\ \{t_X\}, i, j+1]}$$

Note the $i+5$ accounts for "**declare** $a : A$ {" and $j+1$ for "}".

Next we replace the bottom-up rule (BU) with the following (BU-Islnd) rule. The (BU-Islnd) rule is no different than the (BU) rule when $X \in \Delta$, except that X can now appear anywhere on the right-hand side. When $X \in \Sigma$, however, we require that α and β are both sequences of terminals.

$$(\text{BU-}\textsc{Islnd})\frac{\begin{array}{c}\mathcal{G}; H \vdash [t_X, i, j] \\ A \to \alpha X \beta \in \mathcal{P} \quad \mathcal{G} = (\Sigma, \Delta, \mathcal{P}, S) \\ X \in \Sigma \implies \neg\exists k.\ \alpha_k \in \Delta \vee \beta_k \in \Delta\end{array}}{\mathcal{G}; H \vdash [A \to \alpha . t_X . \beta, i, j]}$$

This restriction ensures that when $X \in \Sigma$, the (BU-Islnd) rule only triggers the formation of an island using grammar rules that arise from variable declarations and literals (constants) defined in a DSL; by allowing α and β to be nonempty, we support literals defined by more than one token. For example, the (BU-Islnd) rule doesn't apply when $X =$ `"+"` in `E ::= E "+" E`. In this case, the grammar rule is not defining a variable declaration or constant, and only the `E`'s on either side of the `"+"` give *type* information, so we shouldn't start parsing from `"+"`. We motivate and discuss this rule further in Section 4.

Finally, because islands appear in the middle of the input string, we need both left and right-facing versions of the (Compl) rule.

$$(\textsc{RCompl})\frac{\mathcal{G}; H \vdash [A \to \alpha.\overline{s}_\beta.X\gamma, i, j] \qquad \mathcal{G}; H \vdash [t_X, j, k]}{\mathcal{G}; H \vdash [A \to \alpha.\overline{s}_\beta t_X.\gamma, i, k]\}}$$

$$(\textsc{LCompl})\frac{\mathcal{G}; H \vdash [t_X, i, j] \qquad \mathcal{G}; H \vdash [A \to \alpha X.\overline{s}_\beta.\gamma, j, k]}{\mathcal{G}; H \vdash [A \to \alpha.t_X\overline{s}_\beta.\gamma, i, k]\}}$$

Definition 3.1. The *type-oriented island parsing* algorithm is defined as the deductive system comprised of the rules (Hyp), (Fnsh), (Decl), (BU-Islnd), (RCompl), and (LCompl).

The TIP algorithm is correct in that it can derive a tree for an input string if and only if there is a valid parse tree whose yield is the input string.

Theorem 3.1. For some i and j, $\mathcal{G}; H_0(\mathit{yield}(t_X)) \vdash [t_X, i, j]$ iff $t_X \in \mathcal{T}(\mathcal{G})$.

Proof. By induction on derivations (soundness) and trees (completeness).

The implementation of our algorithm explores derivations in order of *most specific first*, which enables parsing of languages with overloading (and parameterized rules, as in Section 5.2). For example, consider the following rules with an overloaded `+` operator.

```
Float ::= Float "+" Float | Int    Int ::= Int "+" Int
```

The program `1 + 2` can be parsed at least three different ways: with zero, one, or two coercions from `Int` to `Float`. Our algorithm returns the parse with no coercions, which we call most specific: $\texttt{Int} \to [\texttt{Int} \to 1] \ + \ [\texttt{Int} \to 2]$

Definition 3.2. If $B \to A \in \mathcal{P}$, then we say A *is at least as specific as* B, written $A \geq B$, where $\geq$ is the reflexive and transitive closure of this relation. We extend this ordering to terminals and sequences by defining $a \geq b$ iff $a = b$, $\alpha \geq \beta$ iff $|\alpha| = |\beta|$, and $\alpha_i \geq \beta_i$ for $i \in \{1, \ldots, |\alpha|\}$. A parse tree $A \to \overline{s}_\alpha$ is at least as specific as another parse tree $B \to \overline{t}_\beta$ iff $A \geq B$ and $\overline{s}_\alpha \geq \overline{t}_\beta$.

We implement this strategy by comparing the parse trees for a part of the input (e.g., from i to j) and pursuing only the most-specific tree. We save the others on a stack, instead of discarding them as we would for associativity or precedence conflicts (Section 5.1); if the current most-specific parse eventually fails, we pop the stack and resume parsing one of the earlier attempts.

4 Experimental Evaluation

In this section we evaluate the performance of type-oriented island parsing. Specifically, we are interested in the performance of the algorithm for programs that are held constant while the size of the grammar increases.

Chart parsing algorithms have a general worst-case running time of $O(|\mathcal{G}|n^3)$ for a grammar $\mathcal{G}$ and string of length n. In our setting, $\mathcal{G}$ is the union of the grammars for the k DSLs that are in use within a given scope, that is $\mathcal{G} = \bigcup_{i=1}^{k} \mathcal{G}_i$, where $\mathcal{G}_i$ is the grammar for DSL i. We claim that the total size of the grammar $\mathcal{G}$ is not a factor for type-oriented island parsing, and instead the time complexity is $O(mn^3)$ where $m = \max\{|\mathcal{G}_i| \mid 1 \leq i \leq k\}$. This claim deserves considerable explanation to be made precise.

Technically, we assume that $\mathcal{G}$ is *sparse*, which we define as follows.

Definition 4.1. Form a Boolean matrix with a row for each nonterminal and a column for each production rule in a grammar $\mathcal{G}$. A matrix element (i, j) is true if the nonterminal i appears on the right-hand side of the rule j, and it is false otherwise. We say that $\mathcal{G}$ is *sparse* if its corresponding matrix is sparse, that is, if the number of nonzero elements is on the order of $m + n$ for an $m \times n$ matrix.

We conjecture that, in the common case, the union of many type-oriented grammars (or DSLs) is sparse.

To verify that both the type-oriented style of grammar and the island parsing algorithm are necessary for this result, we show that removing either of these ingredients results in parse times that are dependent on the size of the entire grammar. We consider the performance of the top-down and bottom-up Earley algorithms, in addition to island parsing, with respect to untyped, semi-typed, and type-oriented grammars, which we explain in the following subsections.

We implemented all three algorithms using a chart parsing algorithm, which efficiently memoizes duplicate items. The chart parser continues until it has generated all items that can be derived from the input string. (It does not stop at the first complete parse because it needs to continue to check whether the input string is ambiguous, which means the input would be in error.[1]) Also, we should note that our system currently employs a fixed tokenizer, but that we plan to look into scannerless parsing.

To capture the essential, asymptotic behavior of the parsing algorithms, we count the number of items generated during the parsing of a program with untyped, semi-typed, and typed grammars. For this experiment, the program is the expression `--A`.

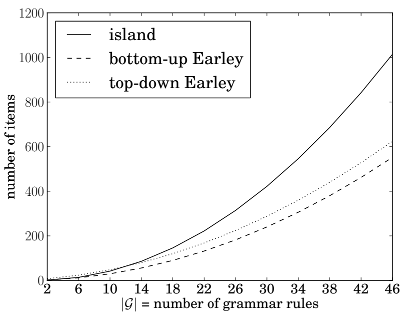
4.1 Untyped Grammar Scaling

In the untyped scenario, all grammar rules are defined in terms of an expression nonterminal (`E`), and variables are simply parsed as identifiers (`Id`).

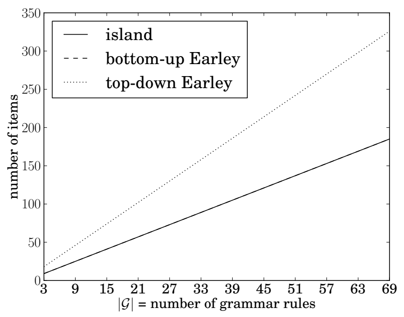
```
E ::= "-" E | Id
```

The results for parsing `--A` after importing k copies of the grammar, for increasing k, are shown in Figure 3(a). The y-axis is the number of items generated

[1] While ambiguous input is allowed if there is a single most-specific parse tree, there may be more than one since the $\geq$ relation is not necessarily a total order.



(a) Untyped Grammar



(b) Semi-typed Grammar

Fig. 3. Comparison of Earley and island parsing with two styles of grammars.

by each parsing algorithm and the x-axis is the total number of grammar rules at each k. In the untyped scenario, the size of the grammar affects the performance of each algorithm, with each generating $O(k^2)$ items.

We note that the two Earley algorithms generate about half as many items as the island parser because they are unidirectional (left-to-right) instead of bidirectional.

4.2 Semi-typed Grammar Scaling

In the semi-typed scenario, the grammars are nearly type-oriented: rules are defined in terms of `V` (for vector) and `M` (for matrix); however, variables are again parsed as identifiers. We call this scenario *semi-typed* because it doesn't use variable declarations to provide type-based disambiguation.

```
E ::= V | Mi    V ::= "-" V | Id    Mi ::= "-" Mi | Id
```

The results for parsing `--A` after importing the `V` rules followed by k copies of the `M` rules (i.e., `M1 ::= "-" M1`, `M2 ::= "-" M2`, ...) are shown in Figure 3(b). The lines for bottom-up Earley and island parsing coincide. Each algorithm generates $O(k)$ items, so we see that type-oriented grammars are not, by themselves, enough to achieve constant scaling with respect to grammar size.

We note that the top-down Earley algorithm generates almost twice as many items as the bottom-up algorithms: the alternatives for the start symbol `E` grow with the input length n, which affects the top-down strategy more than the bottom-up strategy.

4.3 Typed Grammar Scaling

The typed scenario is identical to semi-typed except that it no longer includes the `Id` nonterminal. Instead, programs must declare their own typed variables.

```
E ::= V | Mi    V ::= "-" V    Mi ::= "-" Mi
```

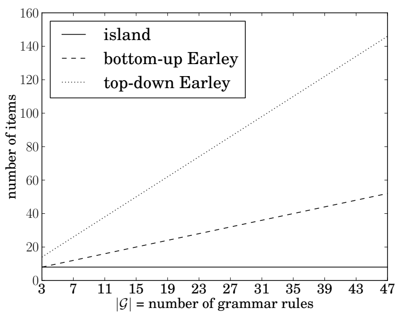


Fig. 4. Comparison of Earley and island parsing with type-oriented grammars.

In this scenario, the grammars are sparse and the terminal `V` is well-typed. The results for parsing `--A`, after declaring `A:V` and importing the `V` rules followed by k copies of the `M` rules, are shown in Figure 4. The island parsing algorithm generates a constant number of items as the size of the grammar increases, while the Earley algorithms remain linear. Thus, the *combination* of type-based disambiguation, type-oriented grammars, and island parsing provides a scalable approach to parsing programs that use many DSLs.

4.4 Discussion

The reason type-oriented island parsing scales is that it is more conservative with respect to prediction than either top-down or bottom up, so grammar rules from other DSLs that are irrelevant to the program fragment being parsed are never used to generate items.

In top-down Earley parsing, any grammar rule that produces the nonterminal B, regardless of which DSL it resides in, will be entered into the chart via a top-down prediction rule. Such items have a zero-length extent which indicates that the algorithm does not yet have a reason to believe that this item will be able to complete.

Looking at the (BU) rule of bottom-up Earley parsing, we see that all it takes for a rule (from any DSL) to be used is that it starts with a terminal that occurs in the program. However, it is quite likely that different DSLs will have rules with some terminals in common. Thus, the bottom-up algorithm also introduces items from irrelevant DSLs.

Finally, consider the (BU-ISLND) rule of our island parser. The difference between this rule and (BU) is that it doesn't apply to a terminal on the right-hand side of a grammar rule when it could apply to some other nonterminal (which corresponds to a type) instead. For example, by avoiding the `"-"` in the above grammars, the (BU-ISLND) rule proceeds directly to the rule for `V` without introducing items from DSLs with only `M`*i* terms that could not complete.

5 A System for Extensible Syntax

In this section we describe the parsing system that we have built as a front end to the Racket programming language. In particular, we describe how we implement four features that are needed in a practical extensible parsing system: associativity and precedence, parameterized grammar rules, grammar rules with variable binders and scope, and rule-action pairs, which combine the notions of semantic actions, function definitions, and macros. We also extend type-oriented grammars so that nonterminals can represent structural types.

Users may define DSLs by writing grammar rules inside a **module** block; the input to our tool consists of programs written in the language of the DSLs that they **import**. For example, one might write

```
module MatrixAlgebra {
  Matrix ::= Matrix "+" Matrix
         ⋮           ⋮
}
```

and then **import** `MatrixAlgebra` in a program using the matrix algebra DSL.

An implementation containing all the features described below is available on Racket's PLaneT package repository. To install, start the Racket interpreter and enter `(require (planet esilkensen/esc))`.

5.1 Associativity and Precedence

Our treatment of associativity and precedence is largely based on that of Visser [27], although we treat this as a semantic issue instead of an optimization issue. From the user perspective, we extend rules (and similarly parse trees) to have the form $A \rightarrow_{\ell,p} \alpha$ where $\ell \in \{\mathsf{left}, \mathsf{right}, \mathsf{non}, \bot\}$ indicates the associativity and $p \in \mathbb{N}_\bot$ indicates the precedence. Concretely, we annotate rules with associativity and precedence inside square brackets in the following way.

```
Matrix ::= Matrix "+" Matrix [left,1]
```

The change to the island parsing algorithm to handle precedence and associativity is straightforward. We simply make sure that a partial parse tree does not violate an associativity or precedence rule before converting it into a (complete) parse tree. We replace the (FNSH) rule with the following.

$$(\textsc{FnshP})\frac{\mathcal{G}; H \vdash [A \rightarrow_{\ell,p} .\overline{t}_\alpha., i, j] \qquad \neg \mathit{conflict}(A \rightarrow_{\ell,p} \overline{t}_\alpha)}{\mathcal{G}; H \vdash [A \rightarrow_{\ell,p} \overline{t}_\alpha, i, j]}$$

Definition 5.1. We say that a parse tree t has a *root priority conflict*, written *conflict*(t), if one of the following holds.

1. It violates the right, left or non-associativity rules, that is, t has the form
   - $A \rightarrow_{\ell,p} (A \rightarrow_{\ell,p} \overline{t}_{A\alpha})\overline{s}_\alpha$ where $\ell = \mathsf{right}$ or $\ell = \mathsf{non}$; or
   - $A \rightarrow_{\ell,p} \overline{s}_\alpha(A \rightarrow_{\ell,p} \overline{t}_{\alpha A})$ where $\ell = \mathsf{left}$ or $\ell = \mathsf{non}$.
2. It violates the precedence rule, that is, t has the form:

$$t = A \rightarrow_{\ell,p} \overline{s}(B \rightarrow_{\ell',p'} \overline{t})\overline{s'} \text{ where } p' < p.$$

5.2 Parameterized Rules

With the move to type-oriented grammars, the need for parameterized rules immediately arises. For example, consider how one might translate the following grammar rule for conditional expressions into a type-oriented rule.

```
Expr ::= "if" Expr "then" Expr "else" Expr
```

By extending grammar rules to enable the parameterization of nonterminals, we can write the following, where `T` stands for any type/nonterminal.

forall `T. T ::= "if" Bool "then" T "else" T`

Parameterized rules have the form $\forall\overline{x}.\, A \to \alpha$, where $\overline{x}$ is a sequence of *variables* (containing no duplicates). We wish to implicitly instantiate parameterized rules, that is, automatically determine which nonterminals to substitute for the parameters. Towards this end, we define a partial function named *match* that compares two symbols with respect to a substitution σ and a sequence of variables and produces a new substitution σ' (if the match is successful). We augment partial parse trees with substitutions to incrementally accumulate the matches, giving them the form $\forall\overline{x}.\, A \to^{\sigma} \alpha.\overline{t}_{\beta}.\gamma$.

Using these definitions, we can implement parameterized rules with a few changes to the base island parser, such as (PRCompl) below.

$$\text{(PRCompl)}\frac{\begin{array}{c}\mathcal{G}; H \vdash [\forall\overline{x}.\, A \to^{\sigma_1} \alpha.\overline{s}_{\beta}.X'\gamma, i, j] \\ \mathcal{G}; H \vdash [t_X, j, k] \qquad \mathit{match}(X', X, \sigma_1, \overline{x}) = \sigma_2\end{array}}{\mathcal{G}; H \vdash [\forall\overline{x}.\, A \to^{\sigma_2} \alpha.\overline{s}_{\beta} t_X.\gamma, i, k]\}}$$

5.3 Grammar Rules with Variable Binders

Consider what would be needed to define a type-oriented grammar rule to parse a `let` expression such as the following, with `n` in scope between the curly brackets.

```
let n = 7 { n * n }
```

We need a way for the rule to refer to the parse tree for `Id` and to say that the identifier has type `T1` inside the brackets. To facilitate such binding forms, we add labeled symbols [12] and a scoping construct [7] to our system.

For example, the `let` rule below binds the variable `x` to the identifier with `x:Id`; the unquoted brackets mark the scoping construct, and `x:T1` says `x` should have type `T1` inside the brackets (any nonempty sequence of bindings may appear before the semicolon):

forall `T1 T2. T2 ::= "let" x:Id "=" T1 { x:T1; T2 }`

We implement these rules by parsing in phases, where initially, all regions enclosed in curly brackets are ignored. Once enough surrounding text has been parsed, a region is "opened" and the next phase of parsing begins with an extended grammar. In the `let` example, the grammar is extended with the rule `T1` $\to$ `x` (with `T1` instantiated and `x` replaced by its associated string).

5.4 Rule-Action Pairs

Sandberg [20] introduces the notion of a *rule-action pair*, which pairs a grammar rule with a semantic action that provides code to give semantics to the syntax. In his paper, rule-action pairs behave like macros; we provide ones that behave like functions as well (with call-by-value semantics). Thus users of our system can embed their DSLs in Typed Racket with two kinds of rule-action pairs.

The ⇒ operator defines a *rule-function*: we compile these rules to functions whose parameters are the variables bound on the right-hand side, and whose body is the Typed Racket code after the arrow. Below is an example adapted from Sandberg's paper giving syntax for computing the absolute value of an integer.

```
Integer ::= "|" i:Integer "|" ⇒ (abs i)
```

Similarly, the = operator defines a *rule-macro*: we simply compile a rule-macro to a macro instead of a function. Macros are necessary in some situations. For example, we need a macro to embed the `let` rule, which we can do as follows.

```
forall T1 T2. T2 ::= "let" x:Id "=" e1:T1 { x:T1; e2:T2 } =
  (let: ([x : T1 e1]) e2)
```

We translate DSLs to Typed Racket by generating the appropriate function or macro call for parsed instances of rule-action pairs.

5.5 Structural Nonterminals

We support representations of structural types in type-oriented grammars by enabling the definition of *structural nonterminals*. In our system, the reserved symbol `Type` gives the syntax of types/nonterminals, and the ≡ operator maps parse trees to Typed Racket types.

Users may define structural nonterminals, as long as they are mapped to Typed Racket types, by writing new rules for `Type` inside a **types** block. For example, consider the following rule for a product type.

```
Type ::= T1:Type "×" T2:Type ≡ (Pairof T1 T2)
```

We can then use this syntax in any grammar rules inside the module; for example, we could write the rule below for accessing the first element of a pair.

```
forall T1 T2. T1 ::= p:(T1 × T2) "." "fst" ⇒ (car p)
```

5.6 Examples

Our implementation includes concrete examples of using the features from this section to embed DSLs in the host language Typed Racket. We show how to give ML-like syntax to several operators and forms of Typed Racket, and how to combine this DSL with literal syntax for set and vector operations.[2]

[2] To access the examples, enter `raco docs` at the command line and look under "Parsing Libraries" for the documentation.

6 Related Work and Conclusions

There have been numerous approaches to extensible syntax for programming languages. In this section, we summarize the approaches and discuss how they relate to our work. We organize this discussion in a roughly chronological order.

Aasa et al. [1] augments the ML language with extensible syntax for dealing with algebraic data types. They develop a generalization of the Earley algorithm that performs Hindley-Milner type inference during parsing. However, Pettersson and Fritzson [18] report that the algorithm was too inefficient in practice. Pettersson and Fritzson [18] build a more efficient system based on LR(1) parsing. Of course, LR(1) parsing is not suitable for our purposes because LR(1) is not closed under union, which we need to compose DSLs. Several later works also integrate type inference into the Earley algorithm [16, 28]. It may be possible to integrate these ideas with our approach to handle languages with type inference.

Several extensible parsing systems use Ford's Parsing Expression Grammar (PEG) formalism [9]. PEGs are stylistically similar to CFGs; however, PEGs avoid ambiguity by introducing a prioritized choice operator for rule alternatives and PEGs disallow left-recursive rules. We claim that these two restrictions are not appropriate for composing DSLs. The order in which DSLs are imported should not matter and DSL authors should be free to use left recursion if that is the most natural way to express their grammar.

The MetaBorg [5] system provides extensible syntax in support of embedding DSLs in general purpose languages. MetaBorg is built on the Stratego/XT toolset which in turn used the syntax definition framework SDF [10]. SDF uses scannerless GLR to parse arbitrary CFGs. The MetaBorg system performs type-based disambiguation after parsing to prune ill-typed parse trees from the resulting parse forest. Thus, the performance of MetaBorg degrades where there is considerable ambiguity. Our treatment of precedence and associativity is based on their notion of disambiguation filter [4]. We plan to explore the scannerless approach in the future. Bravenboer and Visser [6] look into the problem of composing DSLs and investigate methods for composing parse tables. We currently do not create parse tables, but we may use these ideas in the future to further optimize the efficiency of our algorithm.

In this paper we presented a new parsing algorithm, *type-oriented island parsing*, that is the first parsing algorithm to be constant time with respect to the size of the grammar under the assumption that the grammar is sparse. (Most parsing algorithms are linear with respect to the size of the grammar.)

We have developed an extensible parsing system that provides a front-end to Typed Racket, enabling the definition of macros and functions together with grammar rules that provide syntactic sugar. Our implementation provides features such as parameterized grammar rules and grammar rules with variable binders and scope.

In the future we plan to both analytically evaluate the performance of our algorithm and to continue testing our hypothesis about the sparsity of the union of several type-oriented DSLs in practice, with larger and more real-world grammars. In our implementation we plan to provide diagnostics for helping programmers resolve remaining ambiguities that are not addressed by typed-based disambiguation.

References

1. Aasa, A., Petersson, K., Synek, D.: Concrete syntax for data objects in functional languages. In: ACM Conference on LISP and Functional Programming. pp. 96–105. LFP, ACM (1988)
2. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
3. Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Fourth Annual USENIX Tcl/Tk Workshop (1996)
4. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: International Conference on Compiler Construction. pp. 143–158. CC, Springer-Verlag (2002)
5. Bravenboer, M., Vermaas, R., Vinju, J., Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax. In: Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE). pp. 157–172. Springer-Verlag (2005)
6. Bravenboer, M., Visser, E.: Software language engineering. chap. Parse Table Composition, pp. 74–94. Springer-Verlag, Berlin, Heidelberg (2009)
7. Cardelli, L., Matthes, F., Abadi, M.: Extensible syntax with lexical scoping. Tech. Rep. 121, DEC SRC (2 1994)
8. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13, 94–102 (1970)
9. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 111–122. POPL, ACM (2004)
10. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism sdf—reference manual—. SIGPLAN Not. 24(11), 43–75 (1989)
11. Hudak, P.: Modular domain specific languages and tools. In: ICSR '98: Proceedings of the 5th International Conference on Software Reuse. p. 134. IEEE Computer Society, Washington, DC, USA (1998)
12. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and algorithms for data-dependent grammars. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 417–430. POPL '10, ACM, New York, NY, USA (2010)
13. Jurafsky, D., Martin, J.: Speech and Language Processing. Pearson Prentice Hall (2009)
14. Kats, L.C., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. pp. 918–932. OOPSLA '10, ACM, New York, NY, USA (2010)

15. Kay, M.: Algorithm schemata and data structures in syntactic processing, pp. 35–70. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1986)
16. Missura, S.: Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing. Ph.D. thesis, ETH Zurich (1997)
17. Paulson, L.C.: Isabelle: A Generic Theorem Prover, LNCS, vol. 828. Springer (1994)
18. Pettersson, M., Fritzson, P.: A general and practical approach to concrete syntax objects within ml. In: ACM SIGPLAN Workshop on ML and its Applications (June 1992)
19. Quesada, J.F.: The scp parsing algorithm : computational framework and formal properties. In: Procesamiento del lenguaje natural. No. 23 (1998)
20. Sandberg, D.: Lithe: a language combining a flexible syntax and classes. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 142–145. POPL '82, ACM, New York, NY, USA (1982)
21. Siek, J.G.: General purpose languages should be metalanguages. In: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation. pp. 3–4. PEPM '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1706356.1706358`
22. Sikkel, K.: Parsing schemata and correctness of parsing algorithms. Theoretical Computer Science 199(1-2), 87–103 (1998)
23. Stock, O., Falcone, R., Insinnamo, P.: Island parsing and bidirectional charts. In: Conference on Computational Linguistics. pp. 636–641. COLING, Association for Computational Linguistics (1988)
24. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 395–406. POPL '08, ACM, New York, NY, USA (2008)
25. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. pp. 132–141. PLDI '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1993498.1993514`
26. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2. pp. 756–764. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1985)
27. Visser, E.: A case study in optimizing parsing schemata by disambiguation filters. In: International Workshop on Parsing Technology (IWPT'97). pp. 210–224. Massachusetts Institute of Technology, Boston, USA (September 1997)
28. Wieland, J.: Parsing Mixfix Expressions. Ph.D. thesis, Technische Universitat Berlin (2009)