The Semantics of Parsing with Semantic Actions

Robert Atkey
Department of Computer and Information Sciences
University of Strathclyde
Glasgow, UK
Robert.Atkey@cis.strath.ac.uk

***Abstract*—The recovery of structure from flat sequences of input data is a problem that almost all programs need to solve. Computer Science has developed a wide array of declarative languages for describing the structure of languages, usually based on the context-free grammar formalism, and there exist parser generators that produce efficient parsers for these descriptions. However, when faced with a problem involving parsing, most programmers opt for ad-hoc hand-coded solutions, or use parser combinator libraries to construct parsing functions.**

This paper develops a hybrid approach, treating grammars as collections of *active right-hand sides*, indexed by a set of non-terminals. Active right-hand sides are built using the standard monadic parser combinators and allow the consumed input to affect the language being parsed, thus allowing for the precise description of the realistic languages that arise in programming.

We carefully investigate the semantics of grammars with active right-hand sides, not just from the point of view of language acceptance but also in terms of the generation of parse results. Ambiguous grammars may generate exponentially, or even infinitely, many parse results and these must be efficiently represented using Shared Packed Parse Forests (SPPFs). A particular feature of our approach is the use of Reynolds-style parametricity to ensure that the language that grammars describe cannot be affected by the representation of parse results.

***Index Terms*—Parsing, parser combinators, monads, ambiguity, parametricity, context sensitivity.**

I. Introduction

Some form of parsing is employed by nearly every program. Parsing turns external input into a useful internal representation. Clear, concise and implementation independent formal descriptions of the languages recognised by parsers are an essential component of useful documentation of a program's interaction with its environment.

Computer Science, building on work from formal linguistics, has developed a dazzling array of formalisms for describing languages. Accompanying these formalisms are links to automata and computational complexity, yielding a deep and rich body of theory. This work has given rise to software tools such as regular expression engines and parser generators that may be used as building blocks for developing parsers.

Unfortunately, the use of theoretically-based formalisms for parsing is uncommon. Programmers often resort to hand-coded ad-hoc solutions to interpret external input. In the situations when theoretically-based tools are used, for example the Yacc tool and its descendants, they are augmented with additional programming to handle extra-grammatical constructs. An example is the so-called "lexer hack" to parse C programs with `typedefs`. Thus the specification of the language recognised by a parser is nothing more than the implementation of the parser, leaving future programmers to disentangle implementation artifacts from essential details of parser implementations.

The goal of this paper is to develop a grammar formalism and associated declarative parsing semantics that can handle the features required for practical parsing. A key requirement for practical parsing problems is the ability for the language being parsed to depend on prior input. We have already mentioned the need for context-sensitivity in the parsing of C. Other examples include the balancing of XML tags, the length-prefixed fields common in binary formats and languages with user-defined syntax, as is common in many proof assistants.

Noting that most parsers are constructed by hand in general-purpose programming languages, our formalism extends the existing formalism of context-free grammars (CFGs) by incorporating some of the power of general-purpose programming languages. We then use techniques from programming language theory, namely parametricity and Kripke logical relations, to ensure that our grammars are well-behaved.

We retain the organisational principle of CFGs as collections of *right-hand sides* (RHSs) indexed by non-terminals, but we enhance the RHSs to be active in the following two senses:

- RHSs may change their behaviour based on prior input. We accomplish this by organising RHSs as a monad, taking the approach of monadic parser combinator libraries [4]. We build RHSs using the basic monadic operators:

$$\begin{array}{rcl} \mathit{return} & : & A \to \mathsf{RHS}(A) \\ \gg\!= & : & \mathsf{RHS}(A) \to (A \to \mathsf{RHS}(B)) \to \mathsf{RHS}(B) \end{array}$$

  and combinators specific to parsing. For parsing tokens we have $\mathit{tok} : \mathcal{P}(\mathcal{T}) \to \mathsf{RHS}(\mathcal{T})$, where $\mathcal{T}$ is the set of tokens and an element of $\mathcal{P}(\mathcal{T})$ is a set of tokens. Note that $\mathit{tok}\ S$ is a monadic action that returns the actual token that was read; and via the $\gg\!=$ operator this is used to alter the language being parsed. Further combinators offer choice and failure:

$$\begin{array}{rcl} \oslash & : & \mathsf{RHS}(A) \\ \oplus & : & \mathsf{RHS}(A) \to \mathsf{RHS}(A) \to \mathsf{RHS}(A), \end{array}$$

  and reference to other non-terminals: $\mathit{nt} : \mathcal{NT} \to \mathsf{RHS}(1)$, where $\mathcal{NT}$ is the (not necessarily finite) set of non-terminals and $1 = \{*\}$ is a chosen set with one element. To keep our presentation clear, we do not treat the case when non-terminals may also return values that may affect future parsing. Nevertheless, this is a straightforward extension of the present framework.

- A feature common in treatments of parsing in the literature [1], [15], [7], [3] is the imposition of a fixed type for representing parse results. A feature common in most practical parsers is the definition of a custom abstract-syntax tree (AST) type to represent parse results. When parser generators are used, custom ASTs are built using *semantic actions* attached to the grammar. In Section V, we present a formalism that allows principled use of semantic actions to construct ASTs. The combinator for non-terminals now has a dependent type:

$$nt : \Pi_{X \in \mathcal{NT}}.\ \mathsf{RHS}(V(\mathrm{cat}(X)))$$

  where the function cat computes a syntactic category in a set $\mathcal{C}$ from a non-terminal, and the type is parameterised by a type $V : \mathcal{C} \to \mathsf{Set}$ for representing sub-parse results. We make use of Reynolds-style parametricity [13] to ensure that the RHS cannot be affected by the choice of V. We instantiate V to be full parse trees for the simple declarative semantics of parsing, and references to trees for efficient representation of ambiguity. In an implementation, other instantiations of V may be considered for different disambiguation and representation strategies.

Our formalism inherits from CFGs the possibility of ambiguity: multiple parses of a single input. The traditional approach, taken by most LL- and LR-based parser generators, as well as other formalisms such as Parsing Expression Grammars (PEGs) [2], is to regard ambiguity as erroneous. This negatively impacts on the compositionality of grammars, and also forces programmers to put all the logic for disambiguation into the parser. McPeak discusses ambiguity in the C++ grammar [12], and how it must be disambiguated using type information. To enable more flexible disambiguation, we must use an efficient representation of ambiguous parses. We give an introduction to efficient representation of ambiguity using *Shared Packed Parse Forests* (SPPFs) in Section II.

A. The Contribution of this Paper

The central contribution of this paper, building on the grammar formalism we define, is the definition of a semantics of grammars that takes into account the features we identified above: dependency on input, semantic actions for generating parse results and efficient representation of ambiguity.

This paper is in two halves. We first define grammars that do not have semantic actions (but do have input-dependence and ambiguity) in Section III. In Section IV, we present two semantics of grammars: a simple declarative semantics that relates input strings to parse results, and a more complex trace semantics that can represent all the (potentially infinitely many) ambiguous parses of a single input string in a finite derivation. We prove that the two semantics are sound and complete with respect to each other. The second half of the paper, Section V and Section VI, treats grammars with semantic actions, making use of Reynolds-style parametricity [13] to ensure that semantic actions do not interfere with parsing. We show how grammars with semantic actions can be factorised into grammars without semantic actions and a transformation of parse results. We state soundness and completeness theorems linking the two grammar formalisms. A key condition of *affineness* is required to link our formalisms.

B. Related Work

Jim, Mandelbaum and Walker [7] present a convincing case that current parser generator technology fails to accommodate the needs of the majority of programmers that have parsing problems to solve. Jim *et al.* advocate a solution that allows for regular right-hand sides, parameterised non-terminals, the binding of parsed strings to variables and the imposition of constraints on parse results. Their ideas are incorporated in the YAKKER tool. We characterise their approach as *refining* context-free grammars: each YAKKER grammar has an underlying context-free grammar with regular right-hand sides, and the constraints allow for sophisticated data-dependent filtering of parses. In contrast, we consider active right-hand sides that *generate* the grammar as the input is read. Jim and Mandelbaum's subsequent work [5] emphasises their refinement approach, showing that the YAKKER input language can be supported by multiple backends, including parser generators for context-free grammars that allow for threaded state. Jim and Mandelbaum [6] have also considered the factorisation of grammars into parsing and semantic action phases, in a similar fashion to our factorisation construction in Section V-C.

Hutton and Meijer [4] present the canonical example of monadic parser combinators in the functional programming language Haskell. We mention the UU-parsing combinators of Swierstra [17] and the Parsec library of Leijen and Meijer [9] as exemplars of parser combinator libraries. These libraries cannot deal with left recursive grammars and perform best when restricted to an LL(1) fragment. Frost *et al.*[3] present a system of parser combinators that can efficiently handle left recursion and ambiguity, but are limited to applicative parser combinators [11], where the grammar cannot depend on input.

Earley presented an algorithm for context-free grammars [1], and this provides the basis for our trace derivation semantics. Scott [14] and Scott and Johnstone [15] describe how to correctly alter Earley's algorithm to produce the efficient shared packed representation we use in this paper. Tomita's Generalised LR [18] is the source of the SPPF representation.

C. Mathematical Background

We assume that the reader is familiar with the basic concepts of category, functor, natural transformation and monads (in Kleisli triple form) [10]. We make use of the category Set where objects are sets and morphisms are functions. We write 1 for a chosen set with one element $*$. We also use the categories Set^X, for some set X, where objects are X-indexed families of sets, and morphisms between $A, B \in X \to \mathsf{Set}$ are families of functions $\{f_x : A(x) \to B(x)\}_{x \in X}$.

To represent trees of parse results we make use of the notion of initial F-algebra for a functor F. Recall that an F-algebra is a pair of an object A and a morphism $h : FA \to A$. A homomorphism of F-algebras (A, h_1), (B, h_2) is a morphism $f : A \to B$ such that $h_2 \circ Ff = f \circ h_1$. An initial F-algebra is an initial object in the category of F-algebras and F-algebra homomorphisms. The (up to isomorphism) initial F-algebra is usually denoted $(\mu F, \mathsf{in})$. By Lambek's Lemma, $\mathsf{in} : F(\mu F) \to \mu F$ is an isomorphism.

II. Representing Parse Results

Ambiguous grammars may yield multiple derivation trees for the same input string. An obvious way to represent ambiguity is to return a list of parse results for a given input text. However, there are grammars that can produce exponentially many results in the size of the input. The grammar $E \mapsto E\texttt{+}E \ \mid\ \texttt{x}$ on the input x+x+x+x yields 5 results, corresponding to the associations of each + symbol:

$$\texttt{x+}(\texttt{x+}(\texttt{x+x})) \qquad \texttt{x+}((\texttt{x+x})\texttt{+x}) \qquad (\texttt{x+x})\texttt{+}(\texttt{x+x})$$

$$(\texttt{x+}(\texttt{x+x}))\texttt{+x} \qquad ((\texttt{x+x})\texttt{+x})\texttt{+x}$$

We have used non-typewriter parentheses to represent the possible ways of grouping the input symbols.

On an input string with 5 xs, there are 14 ways to group the sub-expressions, and with 6 xs there are 42 ways. In fact, for an input with n xs, the number of parse trees is equal to the nth Catalan number, a well-known sequence that commonly arises in combinatorial situations. This sequence has the closed form solution $C(n) = \frac{1}{n+1}\binom{2n}{n}$. Evidently, returning each individual parse tree for this grammar will result in an exponential runtime. This is disappointing when recognition of strings against any CFG can be done in polynomial time.

Since ambiguity is unavoidable when dealing with generalisations of context-free grammars, we must efficiently deal with the explosion of parse trees. In the rest of this section, we describe Shared Packed Parse Forests (SPPFs) [18], [15], [14], and how it relates to our grammar formalism. For conciseness, we present most of the examples in this section as CFGs rather than the formalism we introduce in Section III.

A. Shared Packed Parse Forests

In the example above, we can observe that there are trees yielded by parsing that share prefixes. We can represent these as trees with "packed" nodes containing all the possibilities:

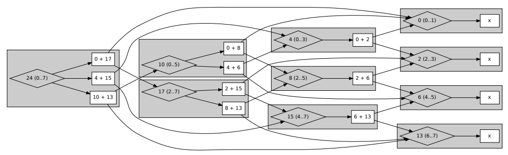
$$\texttt{x+}[\texttt{x+}(\texttt{x+x}), (\texttt{x+x})\texttt{+x}] \qquad (\texttt{x+x})\texttt{+}(\texttt{x+x})$$

$$[\texttt{x+}(\texttt{x+x}), (\texttt{x+x})\texttt{+x}]\texttt{+x}$$

We have used square brackets to represent the possibility of multiple choices at that point in the tree.

Examination of the forest of packed trees yielded by parsing x+x+x+x reveals that there are multiple parses of the non-terminal E for the same input range that have separate representations. For example, the parse (x+x) of the middle two occurrences of x appears twice in the packed representation, once when the entire input is parsed as E+x and once when the entire input is parsed as x+E. The result that the first x can be derived from the non-terminal E is represented 4 times, and likewise for each of the other occurrences of x.

The packed representation removes the redundancy of repeated prefixes of parse trees. We can compress trees further by sharing duplicated parse trees. Sharing all the parse results for a given non-terminal between two points in the input results in *Shared Packed Parse Forests* (SPPFs). The SPPF for the input x+x+x+x is:



The total number of nodes has been reduced from 33 in the packed representation to 10 in the shared packed representation. In this SPPF, for the entire input, there are three possibilities, corresponding to the three items in the packed parse forest presented above. At the next level down, when we are representing parses of three xs, there are two possibilities, corresponding to left or right association. In contrast to the packed representation we have now shared the parses of occurrences of two xs between the other nodes.

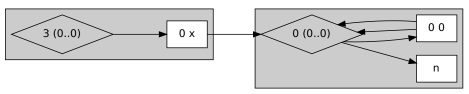
In general, the sharing achieved in the SPPF representation described above is not maximal. As pointed out by Scott, Johnstone and Economopolous [16], citing Johnson [8], grammars matching the schema $S \mapsto \texttt{x} \mid S^k$ require SPPFs of size $O(n^{k+1})$ to represent their results, where the input is of length n. Scott *et al.* [16], [14], [15] show that this unbounded polynomial complexity can be avoided by the use of *binarised* SPPFs, where results for intermediate parse results (i.e., part way through a right-hand side) are packed, rather than just at the level of non-terminals. In this case, Scott *et al.* show that the SPPFs have size at most $O(n^3)$ for any context-free grammar. Unfortunately, due to the way that our active right-hand sides are permitted to generate their own parse results, we cannot use this representation in our formalism.

B. Infinitely Many Parse Results and Cyclic SPPFs

It is not the case that SPPFs are necessarily acyclic. There are grammars that admit infinitely many parses due to loops between non-terminals that do not require progress through the input. A grammar that allows such behaviour is the following:

$$E \mapsto F\texttt{x} \qquad F \mapsto FF \mid \epsilon$$

where ϵ represents the empty string. The language of this grammar has a single element "x", but there are infinitely many ways of parsing the empty string before the x. This is represented by an SPPF with loops in the graph:



In this graph, the node 0 represents all the parses of the non-terminal F between the positions 0 and 0. Written out in full, some of the possible parses of the empty string against this grammar are: $\epsilon, \epsilon\epsilon, (\epsilon\epsilon)\epsilon, \epsilon(\epsilon\epsilon), \ldots$. The SPPF representation compresses all these to a single node.

In order to deal with cyclic SPPFs, and the infinite sets of parse results that they represent, we make use of a co-inductive definition of the relationship between SPPF graphs and individual parse results in Section IV-D.

C. Representation of the Results of Semantic Actions

Our discussion in this section has been limited to the case of parsing context-free grammars, with a carrier type of parse results induced by the grammar. We wish to give a semantics to grammars that generate their own parse results. However, this causes problems when attempting to prove a soundness property of an SPPF representation. Consider the following grammar with active right-hand sides and semantic actions[1]:

$$\begin{array}{rcl} X & \mapsto & \mathsf{do}\ x \leftarrow \mathit{nt}\ E; \mathit{return}\ (\mathsf{C}\ x\ x) \\ E & \mapsto & \mathit{return}\ \mathsf{C}_1 \oplus \mathit{return}\ \mathsf{C}_2 \end{array}$$

The non-terminal X references the non-terminal E, obtains a result and then duplicates this result. The language of this grammar contains only the empty string, but parses it ambiguously: the empty string yields the parse trees $\mathsf{C}\ \mathsf{C}_1\ \mathsf{C}_1$ and $\mathsf{C}\ \mathsf{C}_2\ \mathsf{C}_2$. A problem arises when considering the SPPF representation of the results of parsing against this grammar:

$$\begin{array}{rcl} [0, X, 0] & \mapsto & \{\mathsf{C}\ [0, E, 0]\ [0, E, 0]\} \\ [0, E, 0] & \mapsto & \{\mathsf{C}_1, \mathsf{C}_2\} \end{array}$$

We have represented an SPPF by listing the assignment of sets of parse results to [start position,non-terminal,end position] triples. Reading from this assignment there are now *four* possible results: we have included $\mathsf{C}\ \mathsf{C}_1\ \mathsf{C}_2$ and $\mathsf{C}\ \mathsf{C}_2\ \mathsf{C}_1$, which are not valid trees arising from parsing the empty string.

The culprit is the duplication of the single result returned by the reference to the non-terminal E in the right-hand side for the non-terminal X. The reference to the sub-results has been duplicated, but the constraint that they must refer to the *same* result has been forgotten. To fix this problem, we have formulated a technical condition of *affineness* to identify grammars that do not duplicate results of sub-parses. We give this definition in Section VI-B, although to state it precisely we must first consider grammars without semantic actions. We do this in Section III and Section IV.

III. Grammars that Induce Parse Result Types

We split our technical development into grammars that induce a type of parse results, and then build on this to develop the theory of grammars that generate results as elements of a pre-existing type. In this section and the next, we develop the theory of grammars that induce a parse result type.

Fix a set $\mathcal{T}$ of input tokens, ranged over by t. Let $\mathcal{NT}$ be a set of non-terminals, ranged over by italic roman letters X, Y. A grammar Γ is defined as a mapping $\mathcal{NT} \to \mathsf{RHS}(1)$, where for a result set A, the set $\mathsf{RHS}(A)$ of right-hand sides that return values in A is defined inductively by the following rules. We use Greek letters α, β to range over right-hand sides.

$$\frac{x \in A}{\mathsf{acpt}(x) \in \mathsf{RHS}(A)} \qquad \frac{\alpha \in \mathsf{RHS}(A) \qquad \beta \in \mathsf{RHS}(A)}{\mathsf{or}(\alpha, \beta) \in \mathsf{RHS}(A)}$$

$$\frac{}{\mathsf{fail} \in \mathsf{RHS}(A)} \qquad \frac{S \subseteq \mathcal{T} \qquad f \in S \to \mathsf{RHS}(A)}{\mathsf{tok}(S, f) \in \mathsf{RHS}(A)}$$

$$\frac{X \in \mathcal{NT} \qquad \alpha \in \mathsf{RHS}(A)}{\mathsf{nt}(X, \alpha) \in \mathsf{RHS}(A)}$$

Each of these rules introduces a constructor that we will give a formal semantics to in Section IV. The key feature of monadic right-hand sides is that the language recognised can depend on the tokens that have been accepted by an instance of $\mathsf{tok}(S, f)$.

It is easy to see that the sets $\mathsf{RHS}(-)$ define a monad on the category Set of sets and functions, with the return operation defined as $\mathit{return}(x) = \mathsf{acpt}(x)$ and the bind ($\gg\!=$) operation:

$$\begin{array}{rcl} \mathsf{acpt}(x) \gg\!= k & = & k(x) \\ \mathsf{or}(\alpha, \beta) \gg\!= k & = & \mathsf{or}(\alpha \gg\!= k, \beta \gg\!= k)) \\ \mathsf{fail} \gg\!= k & = & \mathsf{fail} \\ \mathsf{tok}(S, f) \gg\!= k & = & \mathsf{tok}(S, \lambda t.\ f(t) \gg\!= k) \\ \mathsf{nt}(X, \alpha) \gg\!= k & = & \mathsf{nt}(X, \alpha \gg\!= k) \end{array}$$

We can also implement the other structure we described in the introduction, directly in terms of the constructors:

$$\mathit{tok}\ S = \mathsf{tok}(S, \lambda t.\ \mathsf{acpt}(t)) \qquad \mathit{nt}\ X = \mathsf{nt}(X, \mathsf{acpt}(*))$$

$$\oslash = \mathsf{fail} \qquad \alpha \oplus \beta = \mathsf{or}(\alpha, \beta)$$

Each right-hand side $\alpha \in \mathsf{RHS}(1)$ has an associated set of parse results for that right-hand side. This set is parameterised by the representation of sub-results. In the next section we will instantiate this representation either with full parse trees or with references to other parse results.

Definition 1: Given a right-hand side $\alpha \in \mathsf{RHS}(1)$, and $V \in \mathsf{Set}^{\mathcal{NT}}$ representing sub-results, the set of parse results $[\![\alpha]\!]$ associated with α is defined inductively as follows:

$$\begin{array}{rcl} [\![\mathsf{acpt}(*)]\!] & = & 1 \\ [\![\mathsf{or}(\alpha, \beta)]\!] & = & \{\mathsf{inl}(x) \mid x \in [\![\alpha]\!]\} \cup \{\mathsf{inr}(x) \mid x \in [\![\beta]\!]\} \\ [\![\mathsf{fail}]\!] & = & \emptyset \\ [\![\mathsf{tok(S,f)}]\!] & = & \{(t, x) \mid t \in S, x \in [\![f(t)]\!]\} \\ [\![\mathsf{nt}(X, \alpha)]\!] & = & \{(v, x) \mid v \in V(X), x \in [\![\alpha]\!]\} \end{array}$$

This definition is clearly functorial in V, so we have defined a functor $[\![\alpha]\!] : \mathsf{Set}^{\mathcal{NT}} \to \mathsf{Set}$. We extend this to associate a functor $\mathsf{Set}^{\mathcal{NT}} \to \mathsf{Set}^{\mathcal{NT}}$ to every grammar:

Definition 2: Given a grammar $\Gamma \in \mathcal{NT} \to \mathsf{RHS}(1)$ define a functor $[\![\Gamma]\!] \in \mathsf{Set}^{\mathcal{NT}} \to \mathsf{Set}^{\mathcal{NT}}$ by $[\![\Gamma]\!] V X = [\![\Gamma(X)]\!] V$.

Since the functor $[\![\Gamma]\!]$ derived from any grammar is strictly positive, the initial algebra (i.e., least fixpoint) $\mu[\![\Gamma]\!] \in \mathsf{Set}^{\mathcal{NT}}$ exists, with a family of isomorphisms $\mathsf{in}_X : [\![\Gamma]\!](\mu[\![\Gamma]\!])X \to$

[1]We have used Haskell's do-notation to concisely express terms built from the monadic operators.

$\mu[\![\Gamma]\!]X$. This initial algebra represents all the possible parse trees that can result from parsing against this grammar, indexed by non-terminals.

IV. Semantics of Grammars

We define two semantic interpretations for grammars against given inputs. The first provides a simple to understand semantics relating input strings to parse trees. For ambiguous grammars, there may be infinitely many parse results derivable from a single input, and hence infinitely many possible meanings in the tree-producing semantics. We therefore define a second semantics of parses for a given grammar against a given input that can represent infinitely many parse results in a finite derivation. We then relate the two interpretations.

For the rest of this section, we fix a grammar Γ and (except for Definition 3) an input string $input$. We write $input[i]$ to denote the ith token of $input$.

A. Tree-Producing Semantics

The tree-producing semantics is a simple declarative way of stating that a parse tree is derivable for a given grammar and input. The following collection of rules derives judgements of the form $(i, \alpha, j) \Rightarrow x$, where i and j are natural numbers with $i \leq j$, representing start and end positions in the input string, $\alpha \in \mathsf{RHS}(1)$ is a right-hand side that describes the language to be recognised, and $x \in [\![\alpha]\!](\mu[\![\Gamma]\!])$ is a parse result for α with full trees as sub-results.

Thus a derivation of $(i, \alpha, j) \Rightarrow x$ declares that the right-hand side α accepts the input between i and j and yields the result x. We write $(i, X, j) \Rightarrow x$ if $\Gamma(X) = \alpha$ and $(i, \alpha, j) \Rightarrow x$. In this case, $x \in [\![\Gamma]\!](\mu[\![\Gamma]\!])X \cong \mu[\![\Gamma]\!]X$.

$$\frac{}{(i, \mathsf{acpt}(*), i) \Rightarrow *}\ (\textsc{Accept})$$

$$\frac{(i, \alpha, j) \Rightarrow x}{(i, \mathsf{or}(\alpha, \beta), j) \Rightarrow \mathsf{inl}(x)}\ (\mathrm{L}) \qquad \frac{(i, \beta, j) \Rightarrow x}{(i, \mathsf{or}(\alpha, \beta), j) \Rightarrow \mathsf{inr}(x)}\ (\mathrm{R})$$

$$\frac{input[i] = t \qquad t \in S \qquad (i+1, f(t), j) \Rightarrow x}{(i, \mathsf{tok}(S, f), j) \Rightarrow (t, x)}\ (\textsc{Tok})$$

$$\frac{(i, X, j) \Rightarrow y \qquad (j, \alpha, k) \Rightarrow x}{(i, \mathsf{nt}(X, \alpha), k) \Rightarrow (\mathsf{in}(y), x)}\ (\mathrm{NT})$$

The rule Accept states that any right-hand side in the accepting state yields a trivial parse result for any zero-length span of the input. The L and R rules capture non-determinism: yielding results for an "or" node in a right-hand side if either side yields a result. There is no rule to derive results for the right-hand side fail. The Tok rule permits derivation of parse results for right-hand sides that demand input. The rule NT allows the derivation of parse results from a parse with the referenced non-terminal and a parse of the rest of the input.

Definition 3: For a grammar Γ and right-hand side $\alpha \in \mathsf{RHS}(1)$, their *language* $\mathcal{L}(\Gamma, \alpha)$ is the set of $input$s such that $(0, \alpha, n) \Rightarrow x$ is derivable for some x.

Theorem 1: The following closure properties hold:

1) $\mathcal{L}(\Gamma, \mathsf{or}(\alpha, \beta)) = \mathcal{L}(\Gamma, \alpha) \cup \mathcal{L}(\Gamma, \beta)$
2) $\mathcal{L}(\Gamma, \alpha \ggg\!= \lambda x.\beta) = \mathcal{L}(\Gamma, \alpha) \cdot \mathcal{L}(\Gamma, \beta)$

where $S_1 \cdot S_2$ denotes the concatenation of all strings in S_1 with all strings in S_2.

B. Trace Semantics

Where the tree-producing semantics builds full parse trees, the trace semantics we define in this section builds partial parse trees, replacing sub-trees with references to whole collections of parse trees for a given non-terminal between two positions in the input. By replacing concrete results with references, the trace semantics can represent an infinite number of results in a finite derivation. SPPFs are generated by considering certain items within derivations.

We use the family of sets $\mathcal{V} \in \mathsf{Set}^{\mathcal{NT}}$, defined as $\mathcal{V}(X) = \{[i, X, j] \mid 0 \leq i \leq j \leq n\}$, to represent references to parse results for specific non-terminals between points in the input.

A derivation D in the trace semantics consists of a list of items of the form $(i, X \to \alpha, k, j)$. Such an item represents an attempt to parse the non-terminal X starting at position i which has reached position j in the input, with right-hand side α still to be parsed. The component $k \in [\![\alpha]\!]\mathcal{V} \to [\![\Gamma(X)]\!]\mathcal{V}$ represents the current stack of data that has been gathered and is used to generate the result of parsing X from position i.

A derivation starts with an axiom item of the form $(0, X \to \alpha, \lambda x.x, 0)$, where $\Gamma(X) = \alpha$. Items are added according to the rules listed below; items above the line must appear in the derivation before the item below the line may be added.

$$\frac{(i, X \to \mathsf{or}(\alpha, \beta), k, j)}{(i, X \to \alpha, k \circ \mathsf{inl}, j)}\ (\text{T-L})$$

$$\frac{(i, X \to \mathsf{or}(\alpha, \beta), k, j)}{(i, X \to \beta, k \circ \mathsf{inr}, j)}\ (\text{T-R})$$

$$\frac{input[j] = t \qquad t \in S \qquad (i, X \to \mathsf{tok}(S, f), k, j)}{(i, X \to f(t), \lambda x.\ k(t, x), j+1)}\ (\text{T-}\textsc{Tok})$$

$$\frac{\Gamma(Y) = \beta \qquad (i, X \to \mathsf{nt}(Y, f), k, j)}{(j, Y \to \beta, \lambda x.x, j)}\ (\text{T-}\textsc{Call})$$

$$\frac{\begin{array}{c}(j, Y \to \mathsf{acpt}(*), k, j') \\ (i, X \to \mathsf{nt}(Y, \alpha), k', j)\end{array}}{(i, X \to \alpha, \lambda x.k'([j, Y, j'], x), j')}\ (\text{T-}\textsc{Complete})$$

We think of the set of items in a derivation as a collection of processes all acting in parallel on the input. Each process contains a right-hand side α determining its evolution. When the right-hand side describes a non-deterministic choice, the process splits to explore the two possibilities: this is handled by the T-L and T-R rules. When the right-hand side demands a token from the input, the input at the current location is checked and if it matches, the process is allowed to proceed: this is handled by the T-Tok rule. When the right-hand side controlling a process references a non-terminal, a new process is spawned and the spawning process is suspended awaiting a reply: this is the T-CALL rule. When a process terminates—when its right-hand side becomes $\mathsf{acpt}(*)$—this fact is communicated back to suspended processes: this is the T-COMPLETE rule.

To get a full parse of the input for a non-terminal X, a derivation seeded with $(0, X \to \alpha, \lambda x.x, 0)$, where $\Gamma(X) = \alpha$, must contain the item $(0, X \to \mathsf{acpt}(*), k, n)$. Note that in the T-COMPLETE rule, the result derived from the parse of Y between j and j' is not transmitted back to the calling process, only a reference $[j, Y, j']$ to this result. A derivation may have multiple items of the form $(j, Y \to \mathsf{acpt}(*), k, j')$ indicating multiple ways of parsing this non-terminal between these points. The single reference $[j, Y, j']$ refers to all of these. It is easy to check that if $[j, Y, j']$ appears in some parse result, then there is at least one associated item $(j, Y \to \mathsf{acpt}(*), k, j')$ in the derivation, by the T-COMPLETE rule.

We are only really interested in the items of the form $(i, X \to \mathsf{acpt}(*), k, j)$ that represent complete parses of right-hand sides. Each new complete parse gives us more information about the possible parses of the input with respect to the grammar. We arrange derivations into a preorder based on this information content:

Definition 4: $D \sqsubseteq D'$ iff $(i, X \to \mathsf{acpt}(*), k, j) \in D$ implies $(i, X \to \mathsf{acpt}(*), k, j) \in D'$.

Note that $D \sqsubseteq D'$ does not imply that D' is longer than D. It may be the case that D' contains repeated sub-derivations.

C. An Example Trace Derivation

We demonstrate how a finite derivation may represent infinitely many parse results by showing how the rules operate on the grammar:

$$\begin{array}{rcl} E & \mapsto & \mathsf{nt}(F, \mathsf{tok}(\{\mathrm{x}\}, \lambda\_.\mathsf{acpt}(*))) \\ F & \mapsto & \mathsf{or}(\mathsf{acpt}(*), \mathsf{nt}(F, \mathsf{acpt}(*))) \end{array}$$

This is a variant on the grammar we used in Section II-B to illustrate cyclic SPPFs, with a single repetition of the non-terminal F, recast as a grammar with monadic right-hand sides. We have spelt out the right-hand sides using the constructors of RHS instead of the monadic combinators in order to show directly the construction of a derivation.

On the input "x", the following list of items is derivable. We use underscore notation to represent λ-expressions with a single variable. Each item is annotated by the rule and previous items used to justify it.

1) $(0, E \to \mathsf{nt}(F, \mathsf{tok}(\{\mathrm{x}\}, \lambda\_.\mathsf{acpt}(*))), \_, 0)$
2) $(0, F \to \mathsf{or}(\mathsf{acpt}(*), \mathsf{nt}(F, \mathsf{acpt}(*))), \_, 0)$ by T-CALL(1)
3) $(0, F \to \mathsf{acpt}(*), \mathsf{inl}(\_), 0)$ by T-L(2)
4) $(0, F \to \mathsf{nt}(F, \mathsf{acpt}(*)), \mathsf{inr}(\_), 0)$ by T-R(3)
5) $(0, E \to \mathsf{tok}(\{\mathrm{x}\}, \lambda\_.\mathsf{acpt}(*)), ([0, F, 0], \_), 0)$ by T-COMPLETE(3,1)
6) $(0, E \to \mathsf{acpt}(*), ([0, F, 0], \mathrm{x}, \_), 1)$ by T-TOK(5)
7) $(0, F \to \mathsf{acpt}(*), \mathsf{inr}([0, F, 0], \_), 0)$ by T-COMPLETE(3,4)

This derivation is maximal in the sense that all other justifiable extra items that may be added are already present. One could apply T-COMPLETE(7,4) to generate a new item, but this would be the same as item 7. Likewise, T-CALL(4) could generate a new item, but it would be the same as item 2.

From the items of the form $(i, X \to \mathsf{acpt}(*), k, j)$ in the derivation, we can get the following list of assignments of parse results to $[i, X, j]$ triples, where we derive a parse result with variables for sub-parses in $[\![\Gamma(X)]\!]\mathcal{V}$ by feeding the continuation k the value $*$.

$$\begin{array}{rcl} [0, E, 1] & \mapsto & \{([0, F, 0], \mathrm{x}, *)\} \\ [0, F, 0] & \mapsto & \{\mathsf{inl}(*), \mathsf{inr}([0, F, 0], *)\} \end{array}$$

This collection of assignments is the SPPF representing all possible parses of this input. By starting from $[0, E, 1]$ and following the links to other results, we can reconstruct full parse results in $\mu[\![\Gamma]\!](E)$. For example, we can obtain $(\mathsf{in}(\mathsf{inl}(*)), \mathrm{x}, *)$ and $(\mathsf{in}(\mathsf{inr}(\mathsf{in}(\mathsf{inl}(*)), *)), \mathrm{x}, *)$.

D. Relating the Tree-Producing and Trace Semantics

To relate the tree-producing semantics and the trace semantics, we must precisely define what it means for a derivation D to realise a parse tree $x \in \mu[\![\Gamma]\!]X$ for some non-terminal X. To this end, we first define what it means for two single-layer parse results over different sets of representations of sub-results to be related, given a relation that relates sub-results.

Given two ways of representing sub-results, $V, V' \in \mathsf{Set}^{\mathcal{NT}}$, and $R(X) \subseteq V(X) \times V'(X)$ defining how they are related for all $X \in \mathcal{NT}$, we extend this to relate parse results: $\mathcal{R}[\![\alpha]\!]R \subseteq [\![\alpha]\!]V \times [\![\alpha]\!]V'$, by induction on the structure of α:

$$\begin{array}{rcl} \mathcal{R}[\![\mathsf{acpt}(*)]\!]R & = & \{(*, *)\} \\ \mathcal{R}[\![\mathsf{or}(\alpha, \beta)]\!]R & = & \begin{array}{l} \{(\mathsf{inl}(x), \mathsf{inl}(x')) \mid (x, x') \in \mathcal{R}[\![\alpha]\!]R\} \\ \cup \{(\mathsf{inr}(y), \mathsf{inr}(y')) \mid (y, y') \in \mathcal{R}[\![\beta]\!]R\} \end{array} \\ \mathcal{R}[\![\mathsf{fail}]\!]R & = & \emptyset \\ \mathcal{R}[\![\mathsf{tok}(S, f)]\!]R & = & \{((t, x), (t, x')) \mid (x, x') \in \mathcal{R}[\![f(t)]\!]R\} \\ \mathcal{R}[\![\mathsf{nt}(X, \alpha)]\!]R & = & \left\{ ((v, x), (v', x')) \mid \begin{array}{l} (v, v') \in R(X), \\ (x, x') \in \mathcal{R}[\![\alpha]\!]R \end{array} \right\} \end{array}$$

Simply put, a value in $[\![\alpha]\!]V$ is related to a value in $[\![\alpha]\!]V'$ when all their sub-result components are related by R and everything else is equal. The next lemma states that relatedness of parse results is monotonic in relatedness of sub-results. This will be used to handle the accumulation of related parse results as a trace derivation is extended.

Lemma 1: If $\forall X.\ R(X) \subseteq R'(X)$, then $\mathcal{R}[\![\alpha]\!]R \subseteq \mathcal{R}[\![\alpha]\!]R'$.

We can now state what it means for a parse result reference $[i, X, j]$ in some derivation D to be related to some parse tree:

Definition 5: An $\mathcal{NT}$-indexed relation $R(X) \subseteq \mathcal{V}(X) \times \mu[\![\Gamma]\!](X)$ is a D-relation for a derivation D if for all $([i, X, j], x) \in R(X)$, there exists an k such that $(i, X \to \mathsf{acpt}(*), k, j)$ appears in D and $(k(*), \mathsf{in}^{-1}x) \in \mathcal{R}[\![\Gamma(X)]\!]R$.

For any derivation D, the $\mathcal{NT}$-indexed relation $\hat{D}(X) \subseteq \mathcal{V}(X) \times \mu[\![\Gamma]\!]X$ is defined to relate $[i, X, j]$ and x if there exists a D-relation R that relates them (so $\hat{D}$ is the union of all D-relations).

For the example derivation in Section IV-C, the following are two possible D-relations, demonstrating how two different elements of $\mu[\![\Gamma]\!](E)$ can be represented:

$$R_1 = \left\{ \begin{array}{l} ([0,E,1], (\mathsf{in}(\mathsf{inl}(*)), \mathtt{x}, *)), \\ ([0,F,0], \mathsf{inl}(*)) \end{array} \right\}$$

and

$$R_2 = \left\{ \begin{array}{l} ([0,E,1], (\mathsf{in}(\mathsf{inr}(\mathsf{in}(\mathsf{inl}(*)),*)), \mathtt{x}, *)), \\ ([0,F,0], \mathsf{inr}(\mathsf{in}(\mathsf{inl}(*)),*)), \\ ([0,F,0], \mathsf{inl}(*)) \end{array} \right\}$$

Clearly, we can represent any of the infinitely many parse results derivable from the tree-producing semantics for this grammar and input by selecting the appropriate D-relation R. The definition $\hat{D}$ is the greatest fixpoint of a family of monotonic endofunctions on the power set of $\mathcal{V}(X) \times \mu[\![\Gamma]\!](X)$. This coinductive definition allows us to represent infinitely many possible parse results.

E. Soundness and Completeness

To prove soundness of the tree-producing semantics with respect to the trace semantics, we isolate sub-derivations of a trace semantics derivation that generate single layers in parse trees. We define the value of a sub-derivation as the single-layer of a parse tree that is built by that sub-derivation. The notion of value of a sub-derivation flips from the forwards continuation-passing style of the rules to a backwards value-accumulating style.

Definition 6: Given a derivation D a sub-derivation $(i, X \to \alpha, k_0, j') \cdots (i, X \to \mathsf{acpt}(*), k, j)$ is a sub-sequence of items in D linking the two items via applications of T-L, T-R, T-TOK and the second premise of T-COMPLETE.

The *value* of a sub-derivation $(i, X \to \alpha, k_0, j') \cdots (i, X \to \mathsf{acpt}(*), k, j)$ is a value $z \in [\![\alpha]\!]\mathcal{V}$, defined by induction on the sequence of items in the sub-derivation:

- For the sequence $(i, X \to \mathsf{acpt}(*), k, j)$, the value is $*$;
- When the first rule applied in the sequence is T-L and the rest of the sequence has value x, the value of the sequence is $\mathsf{inl}(x)$; the case for T-R is similar;
- When the first rule applied in the sequence is T-TOK with token t and the value of the rest of the sequence is x, the value is (t, x);
- If the first rule is T-COMPLETE with $[j, Y, j']$ and the value of the rest of the sequence is x, the value is $([j, Y, j'], x)$.

Lemma 2: If $(i, X \to \alpha, k_0, j') \cdots (i, X \to \mathsf{acpt}(*), k, j)$ is a sub-derivation with value z, then $k_0(z) = k(*)$.

Theorem 2 (Soundness): If $([i, X, j], \mathsf{in}(x)) \in \hat{D}(X)$ for some derivation D, then $(i, X, j) \Rightarrow x$.

Before the completeness property, we first state some properties of the relations $\hat{D}$ derived from derivations D. The next lemma states that adding results to a derivation does not invalidate the representability of existing results:

Lemma 3: If $D \sqsubseteq D'$ then for all X, $\hat{D}(X) \subseteq \hat{D}'(X)$.

Lemma 4 holds because D-relations are closed under union.

Lemma 4: If $(i, X \to \mathsf{acpt}(*), k, j) \in D$ and $(k(*), x) \in \mathcal{R}[\![\Gamma]\!]\hat{D}$ then $([i, X, j], \mathsf{in}(x)) \in \hat{D}(X)$.

We must also define what it means for two functions taking results from one right-hand side α to another right-hand side β to be related. This is nothing more than the definition of relatedness for function types with Kripke logical relations.

Definition 7: For a pair of right-hand sides α and β, a pair of functions $k : [\![\alpha]\!]\mathcal{V} \to [\![\beta]\!]\mathcal{V}$ and $k' : [\![\alpha]\!](\mu[\![\Gamma]\!]) \to [\![\beta]\!](\mu[\![\Gamma]\!])$ are related in a derivation D if for all D' such that $D \sqsubseteq D'$, and for all $(x, x') \in \mathcal{R}[\![\alpha]\!]\hat{D}'$, then $(k(x), k'(x')) \in \mathcal{R}[\![\beta]\!]\hat{D}'$.

We can now state the main lemma for completeness.

Lemma 5: Suppose $(j, \alpha, j') \Rightarrow x$ is derivable. Then for any derivation D that contains $(i, X \to \alpha, k, j)$ and $k' : [\![\alpha]\!](\mu[\![\Gamma]\!]) \to [\![\Gamma(X)]\!](\mu[\![\Gamma]\!])$ such that k and k' are related in D, then there exists a derivation D' extending D that includes $(i, X \to \mathsf{acpt}(*), k^\dagger, j')$ such that $(k^\dagger(*), k'(x))$ are related in $\mathcal{R}[\![\Gamma(X)]\!]\hat{D}'$.

Theorem 3 (Completeness): If $(0, X, n) \Rightarrow x$, then there is a derivation D with $([0, X, n], \mathsf{in}(x)) \in \hat{D}(X)$.

V. GRAMMARS WITH SEMANTIC ACTIONS

We now build on the results of the previous sections to give a semantics of grammars that use semantic actions to generate parse results in an independently defined type, rather than a type derived from the grammar definition. As we discussed in Section II-C, we must ensure that grammars are *affine* in order to get a strong soundness result relating the tree-producing and trace semantics.

For the rest of the paper, we assume a set $\mathcal{C}$ of syntactic categories and a functor $F \in \mathsf{Set}^{\mathcal{C}} \to \mathsf{Set}^{\mathcal{C}}$. We require that this functor have a least fixpoint $\mu F \in \mathsf{Set}^{\mathcal{C}}$, so it also has a $\mathcal{C}$-indexed family of isomorphisms $\mathsf{in}_c : F(\mu F)c \to \mu F c$. The $\mathcal{C}$-indexed family of sets μF is the type of ASTs that our grammars will produce.

A. Relational Interpretations

In Section IV-D, we defined a relational interpretation of the types induced by grammars. We also require a relational interpretation of the functor F used to represent parse results.

Definition 8: A relational lifting $\hat{F}$ of F is a mapping that takes $\mathcal{C}$-indexed relations $R(c) \subseteq A(c) \times B(c)$ to $\mathcal{C}$-indexed relations $\hat{F}R(c) \subseteq FA(c) \times FB(c)$.

We leave the exact relation interpretation $\hat{F}$ open, and we do not require any particular properties. Note that the exact meaning of Definition 15, which relates parse trees in μF with SPPFs constructed from derivations, depends on the relational interpretation that is chosen. Likewise, the definition of affineness (Definition 16) depends on the exact definition of $\hat{F}$. For many F, the choice of $\hat{F}$ is straightforward. When F is constructed as a $\mathcal{C}$-indexed collection of polynomial functors—built from constants, identity, products and sums—then a canonical $\hat{F}$ can be defined by induction on the structure.

B. Grammars with Semantic Actions

A set $\mathcal{NT}$ is a set of non-terminals for a grammar with semantic actions if each $X \in \mathcal{NT}$ has an associated $\mathrm{cat}(X) \in \mathcal{C}$, denoting the syntactic category of the abstract syntax trees produced by this non-terminal. Fix a set $\mathcal{NT}$ of non-terminals.

For a result set A and a $\mathcal{C}$-indexed set $V \in \mathsf{Set}^{\mathcal{C}}$ for representing sub-results, the set $\mathsf{RHS}_V(A)$ of right-hand sides that accept sub-results in V and return values in A is defined inductively by the following rules, where Greek letters α, β are used to range over right-hand sides.

$$\frac{x \in A}{\mathsf{acpt}(x) \in \mathsf{RHS}_V(A)} \qquad \frac{\alpha \in \mathsf{RHS}_V(A) \qquad \beta \in \mathsf{RHS}_V(A)}{\mathsf{or}(\alpha,\beta) \in \mathsf{RHS}_V(A)}$$

$$\frac{}{\mathsf{fail} \in \mathsf{RHS}_V(A)} \qquad \frac{S \subseteq \mathcal{T} \qquad f \in S \to \mathsf{RHS}_V(A)}{\mathsf{tok}(S,f) \in \mathsf{RHS}_V(A)}$$

$$\frac{X \in \mathcal{NT} \qquad f \in V(\mathrm{cat}(X)) \to \mathsf{RHS}_V(A)}{\mathsf{nt}(X,f) \in \mathsf{RHS}_V(A)}$$

In terms of language acceptance, the constructors have exactly the same meaning as in Section III. The only difference is that now the rest of the right-hand side in the nt case takes a representation of a sub-result as an argument. This enables the right-hand side itself to construct the parse result.

It is easy to see that the sets $\mathsf{RHS}_V(-)$ also define a monad on the category Set, with almost identical definitions for $return$ and $\gg\!=$. The additional structure tok, nt, $\oslash$ and $\oplus$ is again defined directly in terms of the constructors.

Right-hand sides are parameterised over the $\mathcal{C}$-indexed set V in order to handle both the construction of concrete results in μF and the construction of SPPFs over F. Since we do not want the results of parsing to depend on the choice of representation of sub-parses, we will only be interested in right-hand sides that are uniform with respect to the choice of V. We state this property as preservation of relations between representations of sub-results, following Reynolds [13].

Definition 9: For relations $R_{V,V'}(c) \subseteq V(c) \times V'(c)$ and $R_A \subseteq A \times A'$ the relation $\widehat{\mathsf{RHS}}_{R_V}(R_A) \subseteq \mathsf{RHS}_V(A) \times \mathsf{RHS}_{V'}(A')$ is defined inductively as follows:

$$\frac{(a,a') \in R_A}{(\mathsf{acpt}(a), \mathsf{acpt}(a')) \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}$$

$$\frac{(\alpha,\alpha') \in \widehat{\mathsf{RHS}}_{R_V}(R_A) \qquad (\beta,\beta') \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}{(\mathsf{or}(\alpha,\beta), \mathsf{or}(\alpha',\beta')) \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}$$

$$\frac{}{(\mathsf{fail},\mathsf{fail}) \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}$$

$$\frac{\forall t \in S.\ (f(t), f'(t)) \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}{(\mathsf{tok}(S,f), \mathsf{tok}(S,f')) \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}$$

$$\frac{\forall v\ v'.\ (v,v') \in R_V(\mathrm{cat}(X)) \Rightarrow (f(v), f'(v')) \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}{(\mathsf{nt}(X,f), \mathsf{nt}(X,f')) \in \widehat{\mathsf{RHS}}_{R_V}(R_A)}$$

Definition 10: A *uniform right-hand side* for a syntactic category $c \in \mathcal{C}$ is a family $\alpha_V \in \mathsf{RHS}_V(FVc)$ of right-hand sides indexed by $V \in \mathsf{Set}^{\mathcal{C}}$ such that the following uniformity property holds: For all V, V' and $R_{V,V'}(c) \subseteq V(c) \times V'(c)$, it is the case that $(\alpha_V, \alpha_{V'}) \in \widehat{\mathsf{RHS}}_R(\hat{F}Rc)$. The collection of all uniform full right-hand sides for a syntactic category $c \in \mathcal{C}$ is denoted $\mathsf{URHS}(c)$.

In the definition of URHS we have quantified over all sets V, V' to generate a set, a construction not permitted in ZF set theory. We could observe that we only ever instantiate with either the one-element set or the $\mathcal{C}$-indexed families of sets μF and $\mathcal{V}_{\mathcal{C}}$ as defined below. Therefore we can restrict to sets that are smaller than the cardinality of the largest of these. Alternatively, we could work in a meta-theory that allows for the construction of large sets that live in some universe Set_1. The type theories implemented by Coq and Agda permit this.

In a language like Haskell, we can use higher-kinded polymorphism to define the type of uniform right-hand sides and—modulo non-termination and selective strictness—the uniformity property holds automatically.

Definition 11: A *grammar with semantic actions* Γ is a tuple $(\mathcal{NT}, \mathrm{cat}, \Gamma)$ where $(\mathcal{NT}, \mathrm{cat})$ is a set of non-terminals with associated syntactic categories as described above, and Γ is a dependently typed function $\Gamma \in \Pi_{X \in \mathrm{NT}}.\mathsf{URHS}(\mathrm{cat}(X))$ that maps non-terminals X to uniform right-hand sides for the syntactic category $\mathrm{cat}(X)$.

For a given representation of sub-results, $V \in \mathsf{Set}^{\mathcal{C}}$, and non-terminal X, we write $\Gamma(X)_V$ for the right-hand side of Γ associated with X specialised to V.

C. Factoring Semantic Actions

In the next section we give grammars with semantic actions a pair of semantics in the same style as Section IV. In order to relate the semantics of grammars with semantic actions with the semantics of grammars, we define, for every grammar with semantic actions Γ, an associated grammar Γ° and function $(\!|\Gamma|\!) : \mu[\![\Gamma^\circ]\!] \to \mu F \circ \mathrm{cat}$ that we can think of as a factorisaton of the grammar Γ.

Definition 12: Let $K_1(c) = 1$. Given $\alpha \in \mathsf{RHS}_{K_1}(c)$, define $\alpha^\circ \in \mathsf{RHS}(1)$ by induction on α.

$$(\mathsf{acpt}(x))^\circ = \mathsf{acpt}(*) \qquad (\mathsf{or}(\alpha,\beta))^\circ = \mathsf{or}(\alpha^\circ, \beta^\circ)$$

$$\mathsf{fail}^\circ = \mathsf{fail} \qquad (\mathsf{tok}(S,f))^\circ = \mathsf{tok}(S, \lambda t.(f(t))^\circ)$$

$$(\mathsf{nt}(X,f))^\circ = \mathsf{nt}(X, (f(*))^\circ)$$

Given $\alpha \in \mathsf{URHS}(c)$, we now wish to show that there is a family of functions $\overline{\alpha}_V : [\![\alpha^\circ_{K_1}]\!](V \circ \mathrm{cat}) \to FVc$ that maps the results of $\alpha^\circ_{K_1}$ to the results of α_V. This is delicate because we have defined $\alpha^\circ_{K_1}$ by induction on α_{K_1}, but we will need to define the function by induction on α_V, where V is the given carrier for sub-results. Fortunately, this is possible due to the uniformity of α: formally, $\bar{\cdot}_V$ will have type:

$$\begin{aligned}\bar{\cdot}_V :\ & \Pi_{\alpha \in \mathsf{RHS}_V(FVc)}.\Pi_{\alpha' \in \mathsf{RHS}_{K_1}(FK_1c)}.\\ & (\alpha,\alpha') \in \widehat{\mathsf{RHS}}_R(\hat{F}Rc) \to [\![\alpha'^\circ]\!](V \circ \mathrm{cat}) \to FVc\end{aligned}$$

where $R(c) \subseteq V(c) \times K_1(c)$ is defined as $R(c) = \{(v,*) \mid v \in V(c)\}$. We use uniformity here to ensure that the right-hand side we are using to obtain data from (α) has the same structure as the right-hand side that we are using to define the shape of the domain of the function (α'). Having satisfied ourselves that it is possible to define a function in this way, we elide the details in the following definition:

Definition 13: For $\alpha \in \mathsf{URHS}(c)$ and $V \in \mathsf{Set}^{\mathcal{C}}$, define the function $\overline{\alpha}_V : [\![\alpha^\circ]\!](V \circ \mathrm{cat}) \to FVc$ by induction on α_V:

$$\begin{aligned}
\overline{\mathsf{acpt}(x)}_V(*) &= x \\
\overline{\mathsf{or}(\alpha,\beta)}_V(\mathsf{inl}(x)) &= \overline{\alpha}_V(x) \\
\overline{\mathsf{or}(\alpha,\beta)}_V(\mathsf{inr}(x)) &= \overline{\beta}_V(x) \\
\overline{\mathsf{tok}(S,f)}_V(t,x) &= \overline{f(t)}_V(x) \\
\overline{\mathsf{nt}(X,f)}_V(v,x) &= \overline{f(v)}_V(x)
\end{aligned}$$

There is no clause for fail because $[\![\mathsf{fail}]\!]V = \emptyset$ has no elements.

Definition 14: Given a grammar with semantic actions Γ, define the grammar Γ° on the same set of non-terminals as $\Gamma^\circ(X) = (\Gamma(X))^\circ$. Define a family of functions, parameterised by $V \in \mathsf{Set}^{\mathcal{C}}$,

$$\begin{aligned}
\overline{\Gamma}_V(X) &: [\![\Gamma^\circ]\!](V \circ \mathrm{cat})X \to FV(\mathrm{cat}(X)) \\
\overline{\Gamma}_V(X) &= \overline{\Gamma(X)}_V
\end{aligned}$$

This family is natural in V.
From the natural transformation $\overline{\Gamma}$ we can define an $\mathcal{NT}$-indexed function $(\!|\Gamma|\!)X : \mu[\![\Gamma^\circ]\!]X \to \mu F(\mathrm{cat}(X))$, by structural recursion on $\mu[\![\Gamma]\!]$.

VI. Semantics of Semantic Actions

As above, we define a tree-producing semantics that generates specific abstract syntax trees, and a trace semantics that represents many parses in a single derivation.

A. Tree-Producing Semantics

For the tree-producing semantics, we instantiate the right-hand sides with $V(c) = \mu Fc$. Consequently, the trees generated will be ordinary abstract syntax trees.

The following collection of rules derives judgements of the form $(i, \alpha, j) \Rightarrow x$, where i and j are natural numbers with $i \leq j$, representing positions in the input string, $\alpha \in \mathsf{RHS}_{\mu F}(F(\mu F)c)$ is a right-hand side that will yield a parse tree, and $x \in F(\mu F)c$ is a parse tree. The syntactic category $c \in \mathcal{C}$ is left implicit. A derivation of $(i, \alpha, j) \Rightarrow x$ expresses that the right-hand side α accepts the input between i and j and yields the result x. We write $(i, X, j) \Rightarrow x$ if $\Gamma(X)_{\mu F} = \alpha$ and $(i, \alpha, j) \Rightarrow x$.

$$\frac{}{(i, \mathsf{acpt}(x), i) \Rightarrow x}\ (\text{P-ACCEPT})$$

$$\frac{(i, \alpha, j) \Rightarrow x}{(i, \mathsf{or}(\alpha,\beta), j) \Rightarrow x}\ (\text{P-L}) \qquad \frac{(i, \beta, j) \Rightarrow x}{(i, \mathsf{or}(\alpha,\beta), j) \Rightarrow x}\ (\text{P-R})$$

$$\frac{\mathit{input}[i] = t \qquad t \in S \qquad (i+1, f(t), j) \Rightarrow x}{(i, \mathsf{tok}(S,f), j) \Rightarrow x}\ (\text{P-TOK})$$

$$\frac{\begin{array}{c}\Gamma(X)_{\mu F} = \alpha \\ (i, \alpha, j) \Rightarrow y \qquad (j, f(\mathsf{in}\ y), k) \Rightarrow x\end{array}}{(i, \mathsf{nt}(X,f), k) \Rightarrow x}\ (\text{P-NT})$$

These rules have the same intended meaning as the corresponding rules in Section IV-A. The main difference is that now the right-hand side is responsible for constructing the parse result: when $\mathsf{acpt}(x)$ is reached, the whole parse result x has been constructed and this is what is returned. The NT rule is further different because the function f is fed the sub-result from parsing the non-terminal X. By the uniformity of f, this cannot be used to affect the language being parsed.

The following two theorems are proved by induction on the derivations against the grammar Γ and the derived grammar Γ° respectively. A similar generalisation of the induction hypothesis is required as in the definition of $\overline{\alpha}_V$ above, due to the need to deal with related instantiations of a uniform right-hand side. We use the notation $\Gamma \vdash (i, \alpha, j) \Rightarrow x$ to indicate derivations against particular grammars.

Theorem 4 (Soundness): Let $\alpha \in \mathsf{URHS}(c)$ be a uniform right-hand side. If $\Gamma \vdash (i, \alpha_{\mu F}, j) \Rightarrow x$ there there exists an $x' \in [\![\alpha^\circ]\!](\mu[\![\Gamma^\circ]\!])$ such that $\Gamma^\circ \vdash (i, \alpha^\circ, j) \Rightarrow x'$ and $\overline{\alpha}_{\mu F}([\![\alpha^\circ]\!](\!|\Gamma|\!)x') = x$.

Theorem 5 (Completeness): Let $\alpha \in \mathsf{URHS}(c)$ be a uniform right-hand side. If $\Gamma^\circ \vdash (i, \alpha^\circ, j) \Rightarrow x$ then $\Gamma \vdash (i, \alpha_{\mu F}, j) \Rightarrow \overline{\alpha}_{\mu F}([\![\alpha^\circ]\!](\!|\Gamma|\!)x)$.

B. Trace Semantics

The trace semantics for grammars with semantic actions incorporates the same changes as the tree-producing semantics. Right-hand sides are responsible for building parse results, so we do not carry around functions k for constructing results. To represent sub-results, we use the family of sets $\mathcal{V}_{\mathcal{C}} \in \mathsf{Set}^{\mathcal{C}}$ defined as $\mathcal{V}_{\mathcal{C}}(c) = \{[i, X, j] \mid 0 \leq i \leq j \leq n \wedge \mathrm{cat}(X) = c\}$.

Items in trace semantics derivations for grammars with semantic actions have the form $(i, X \to \alpha, j)$ where $i \leq j$ are start and end positions in the input, X is a non-terminal and $\alpha \in \mathsf{RHS}_{\mathcal{V}_{\mathcal{C}}}(\mathrm{cat}(X))$ describes the future input that is required. As before, derivations D are justified lists of items built from the following rules, starting from the axiom $(0, X \to \Gamma(X)_{\mathcal{V}_{\mathcal{C}}}, 0)$:

$$\frac{(i, X \to \mathsf{or}(\alpha,\beta), j)}{(i, X \to \alpha, j)}\ (\text{P-T-L})$$

$$\frac{(i, X \to \mathsf{or}(\alpha,\beta), j)}{(i, X \to \beta, j)}\ (\text{P-T-R})$$

$$\frac{\mathit{input}[j] = t \qquad t \in S \qquad (i, X \to \mathsf{tok}(S,f), j)}{(i, X \to f(t), j+1)}\ (\text{P-T-TOK})$$

$$\frac{(i, X \to \mathsf{nt}(Y,f), j) \qquad \Gamma(Y)_{\mathcal{V}_{\mathcal{C}}} = \beta}{(j, Y \to \beta, j)}\ (\text{P-T-CALL})$$

$$\frac{\begin{array}{c}(j, Y \to \mathsf{acpt}(x), j') \\ (i, X \to \mathsf{nt}(Y,f), j)\end{array}}{(i, X \to f([j, Y, j']), j')}\ (\text{P-T-COMPLETE})$$

These rules have the same intended meaning as in Section IV-B, but now the results are constructed by the right-hand sides themselves. In the P-T-COMPLETE rule, we pass the reference to all results for $[j, Y, j']$ to the right-hand side.

As in Section IV-B, a derivation D induces a relation $\hat{D}$ between members of $\mathcal{V}_{\mathcal{C}}$ and abstract syntax trees in μF:

Definition 15: A $\mathcal{C}$-indexed relation $R(c) \subseteq \mathcal{V}_{\mathcal{C}}(c) \times \mu F(c)$ is a D-relation for a derivation D if for all $([i, X, j], x) \in R(c)$ there exists an x' such that $(i, X \rightarrow \mathsf{acpt}(x'), j) \in D$ and $(x', x) \in \hat{F}Rc$. For any derivation D, the $\mathcal{C}$-indexed relation $\hat{D}(c) \subseteq \mathcal{V}_{\mathcal{C}}(c) \times \mu Fc$ is defined to relate $[i, X, j]$ and x if there exists a D-relation that relates them.

As we have done for the tree-producing semantics, we relate the trace semantics of grammars with semantic actions to the semantics of Section IV-B via the factorisation construction. The completeness direction is by induction on x:

Theorem 6 (Completeness): Let Γ be a grammar with semantic actions. Given a derivation D° against Γ° such that $([i, X, j], x) \in \widehat{D^\circ}$, then there is a derivation D against Γ such that $([i, X, j], (\!|\Gamma|\!)(\mathrm{cat}(X))x) \in \hat{D}$.

The soundness direction is more subtle. As we discussed in Section II-C, an arbitrary grammar with semantic actions may duplicate the information required to reconstruct the AST, causing confusion over which result was intended.

Definition 16: A grammar with semantic actions Γ is *affine* if for all derivations D, the following holds. If $X \in \mathcal{NT}$ is a non-terminal with $c = \mathrm{cat}(X)$, and R is a D-relation and $x \in F\mathcal{V}_{\mathcal{C}}c$ and $x' \in F(\mu F)c$ with $(x, x') \in \hat{F}Rc$, and there is a $y \in [\![\Gamma(X)^\circ]\!](\mathcal{V}_{\mathcal{C}} \circ \mathrm{cat})$ such that $\overline{\Gamma(X)}_{\mathcal{V}_{\mathcal{C}} \circ \mathrm{cat}}\, y = x$, then there exists a $y' \in [\![\Gamma(X)^\circ]\!](\mu F \circ \mathrm{cat})$ such that $(y, y') \in \mathcal{R}[\![\Gamma(X)^\circ]\!](R \circ \mathrm{cat})$ and $\overline{\Gamma(X)}_{\mu F \circ \mathrm{cat}}\, y' = x'$. Diagrammatically:

$$\begin{tikzcd}
y \in [\![\Gamma(X)^\circ]\!](\mathcal{V}_{\mathcal{C}} \circ \mathrm{cat}) \arrow[r, mapsto, "\overline{\Gamma(X)}_{\mathcal{V}_{\mathcal{C}} \circ \mathrm{cat}}"] \arrow[d, dash, "{\mathcal{R}[\![\Gamma(X)^\circ]\!](R \circ \mathrm{cat})}"'] & x \in F\mathcal{V}_{\mathcal{C}}c \arrow[d, dash, "\hat{F}Rc"] \\
\exists y' \in [\![\Gamma(X)^\circ]\!](\mu F \circ \mathrm{cat}) \arrow[r, mapsto, "\overline{\Gamma(X)}_{\mu F \circ \mathrm{cat}}"] & x' \in F(\mu F)c
\end{tikzcd}$$

The grammar is *linear* if there is a unique choice of y'.

We illustrate this definition with an example. Consider the grammar in Section II-C and the non-terminal X whose right-hand side duplicated a sub-result. In this case $\overline{\Gamma(X)}_{V}(v, *) = \mathsf{C}\ v\ v$. Now consider a derivation yielding the SPPF in Section II-C. In this derivation, the values $x = \mathsf{C}\ [0, E, 0]\ [0, E, 0]$ and $x' = \mathsf{C}\ \mathsf{C}_1\ \mathsf{C}_2$ are related as in the definition above, and with $y = ([0, E, 0], *)$, we have that $\overline{\Gamma(X)}_{\mathcal{V}_{\mathcal{C}} \circ \mathrm{cat}}\, y = x$. However, there is no $y' \in [\![\Gamma(X)^\circ]\!](\mu F \circ \mathrm{cat})$ such that $\overline{\Gamma(X)}_{\mu F \circ \mathrm{cat}}\, y' = x$: for any $y' = (z, *)$, we would have to have $z = \mathsf{C}_1$ and $z = \mathsf{C}_2$. Note that this definition permits the *discarding* of parse results: in this case we are free to pick any appropriate parse result in the derivation D to fill in the hole. If we had required the choice of y to always be unique then we would also be disallowing the discarding of sub-results: grammars would be linear.

Using affineness, the following soundness result is provable, using the same structure as the completeness property above, but exploiting the affineness to proceed with induction on x.

Theorem 7 (Soundness): Let Γ be an affine grammar with semantic actions. Given a derivation D against Γ such that $([i, X, j], x) \in \hat{D}$, then there is a derivation D° against Γ° and an $x' \in \mu[\![\Gamma]\!]X$ such that $([i, X, j], x^\circ) \in \widehat{D^\circ}$ and $(\!|\Gamma|\!)x^\circ = x$.

VII. CONCLUSIONS

We have presented a general class of grammars and developed a semantic theory to interpret these grammars. We intend to extend the formalism described here to be yet more expressive, including returned values from non-terminals and tail-called non-terminals. It also remains to perform empirical evaluation of semantically-based algorithms for parsing with our grammars, both with respect to efficiency and expressivity.

ACKNOWLEDGEMENTS

Thanks to the anonymous reviewers for their constructive and helpful comments. This work was funded by EPSRC grant EP/G068917/1.

REFERENCES

[1] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[2] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pages 111–122. ACM, 2004.

[3] R. A. Frost, R. Hafiz, and P. Callaghan. Parser combinators for ambiguous left-recursive grammars. In *Practical Aspects of Declarative Languages, 10th Int. Symp., PADL 2008*, volume 4902 of *LNCS*, pages 167–181. Springer, 2008.

[4] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.

[5] T. Jim and Y. Mandelbaum. A New Method for Dependent Parsing. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011*, volume 6602 of *LNCS*, pages 378–397. Springer, 2011.

[6] T. Jim and Y. Mandelbaum. Delayed semantic actions in Yakker. In *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceedings*. ACM, 2011.

[7] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 417–430. ACM, 2010.

[8] M. Johnson. The computational complexity of GLR parsing. In *Generalized LR Parsing*. Kluwer Academic Publishers, 1991.

[9] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[10] S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 2nd edition, 1998.

[11] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[12] S. McPeak. Elkhound: A Fast, Practical GLR Parser Generator. Technical Report UCB/CSD-2-1214, University of California, Berkeley, December 2002.

[13] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[14] E. Scott. SPPF-Style Parsing from Earley Recognisers. *Electronic Notes in Theoretical Computer Science*, 203:53–67, 2008.

[15] E. Scott and A. Johnstone. Recognition is not Parsing – SPPF-style parsing from cubic recognisers. *Science of Computer Programming*, 75:55–70, 2010.

[16] E. Scott, A. Johnstone, and G. Economopolous. BRN-table based GLR Parsers (Draft). Technical Report CSD-TR-03-06, Royal Holloway, University of London, 2003.

[17] S. D. Swierstra. Combinator parsing: A short tutorial. In *LerNet ALFA Summer School*, volume 5520 of *LNCS*, pages 252–300. Springer, 2008.

[18] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.