Bottom-up Grammar Analysis

— A Functional Formulation —

Johan Jeuring* and Doaitse Swierstra
Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
email: {johan,doaitse}@cs.ruu.nl

Abstract

This paper discusses bottom-up grammar analysis problems such as the EMPTY problem and the FIRST problem. It defines a general class of bottom-up grammar analysis problems, and from this definition it derives a functional program for performing bottom-up grammar analysis. The derivation is purely calculational, using theorems from lattice theory, the Bird-Meertens calculus, and laws for list-comprehensions. Sufficient conditions guaranteeing the existence of a solution emerge as a byproduct of the calculation. The resulting program is used to construct programs for the EMPTY problem and the FIRST problem.

1 Introduction

Grammar analysis is performed in many different situations: Yacc tests whether or not its input grammar is LALR(1), parser generators contain functions for determining whether or not a nonterminal can derive the empty string (EMPTY) as part of determining the set of all symbols that can appear as the first symbol of a derived string (FIRST), and for determining the set of symbols that can appear as the first symbol following upon a string derived by a given nonterminal (FOLLOW). Other, similar, problems arise when analysing attribute dependencies in attribute grammars: determine the inherited attributes upon which a synthesised attribute depends (IS), and, conversely, determine the synthesised attributes upon which an inherited attribute depends (SI). Such problems are called *grammar analysis problems*.

Grammar analysis problems can be divided into two classes: *bottom-up* and *top-down*. The difference between these classes is that the required information for a nonterminal in a top-down problem depends on the possible contexts for that nonterminal, whereas in a bottom-up problem the contexts of a nonterminal can be ignored. Often the output of a bottom-up problem is used in a top-down problem.

*address from January 1, 1994: Chalmers Tekniska Högskola, Institutionen för Datavetenskap, S-412 96 Göteborg, Sweden

The specification of a grammar analysis problem determines the class to which it belongs: `EMPTY`, `FIRST`, and `IS` are bottom-up grammar analysis problems, the `FOLLOW` and `SI` problems belong to the top-down class. This paper studies bottom-up grammar analysis.

Grammar analysis problems are described by sets of mutually recursive equations, and the solution of a grammar analysis problem is a fixed point of this equational system. Möncke and Wilhelm [11] observe this, and give several solutions, depending on the conditions that are satisfied, for such problems. The goal of this paper is to derive the solutions given by Möncke and Wilhelm. We start with a very general specification of a bottom-up grammar analysis problem, and we derive a function of which the fixed point gives the solution of the problem. This function is obtained by applying laws to components of the expressions occurring in the specification. The laws we apply are familiar laws for, for example, list-comprehensions [13], and maps [1, 9]. Sufficient conditions for guaranteeing the existence of a fixed point solution emerge as a byproduct of this derivation. An important advantage of a derivation of a program is that it is clear why and where conditions are imposed upon the components of the program. Finally we give the implementation of the derived algorithm in the functional language Gofer [7, 3]. Incorporating the functions for solving grammar analysis problems in parser generators such as a functional version of Yacc [12], Ratatosk [10], and Happy [4] would reduce the amount of code used in these parser generators.

This paper is organised as follows. Section 2 defines the datatypes that are used in manipulating grammars in Gofer. Section 3 introduces the concepts of lattice theory needed in the subsequent sections. Section 4 defines the class of bottom-up grammar analysis problems, and gives some examples. Section 5 sketches a derivation of an algorithm that can be used to solve bottom-up problems. The complete derivation is given in [6]. Section 6 gives an implementation in Gofer of the datatypes and functions constructed in the previous sections. Section 7 concludes the paper.

2 Datatypes for Grammars in Gofer

This section defines various datatypes in Gofer used in analysing and representing grammars. Most of the non-standard notation is taken from the Bird-Meertens calculus [1].

Functions

Projection exl (exr) selects the left (right) component of a pair. Given functions $f : A \rightarrow B$ and $g : A \rightarrow C$, function $f \triangle g : A \rightarrow B \times C$ (split) applies both f and g to an argument: $(f \triangle g) a = (f a, g a)$. The type $B \times C$ is the cartesian product of the sets B and C . Given functions $f : A \rightarrow B$ and $g : C \rightarrow D$, function $f \times g : A \times C \rightarrow B \times D$ (product) applies f to the first component, and g to the second component of its argument: $(f \times g) (a, c) = (f a, g c)$. The laws for compositions of projections, split, and product are omitted. Function application binds stronger than a binary operator; composition binds weakest.

Lists

The datatype (finite) *list* is a prominent datatype in the subsequent sections, and

we will use a number of properties that are satisfied by functions defined on the datatype *list*. The empty list is denoted by $[]$, and the concatenation of two lists x and y is denoted by $x \mathbin{++} y$. Prepending an element x to a list xs is denoted by $x : xs$. The datatype *list* over base type A is denoted by A^* . For $f : A \rightarrow B$, function $f^* : A^* \rightarrow B^*$, called a *map* function takes a list and applies function f to all elements in the list, so $f^* xs = [f x \mid x \leftarrow xs]$. For the map function we have

$$\begin{aligned} f^* [] &= [] \\ f^* (x \mathbin{++} y) &= f^* x \mathbin{++} f^* y \\ f^* (x : xs) &= f x : f^* xs \end{aligned} \tag{1}$$

Map-distributivity says that the composition of two maps is a map again, i.e., for all functions f and g :

$$f^* \cdot g^* = (f \cdot g)^* \tag{2}$$

Furthermore, the result of mapping the identity function over an argument is the argument itself, so $id_A^* = id_{A^*}$. These equalities say that $*$ is a *functor*. An important functional programming construct we use is *list-comprehension*. For example,

$$[(x, y) \mid x \leftarrow [1, 2], y \leftarrow [3, 4]] = [(1, 3), (1, 4), (2, 3), (2, 4)]$$

We will use the following laws for list-comprehensions [13] in some calculations.

$$[t \mid t \leftarrow ts] = ts \tag{3}$$

$$[f t \mid q] = f^* [t \mid q] \tag{4}$$

$$[t \mid p, q] = concat [[t \mid q] \mid p] \tag{5}$$

where function $concat : A^{**} \rightarrow A^*$ flattens a list of lists. Function $concat$ is defined in terms of the reduce operator. The *reduce* operator $/$ takes an associative operator $\oplus$ with unit $1_\oplus$, and a list, and places the operator in between the elements of a list, so $\oplus / [a, b, c] = a \oplus b \oplus c$. For operator $\oplus : A \times A \rightarrow A$ we have $\oplus / : A^* \rightarrow A$. It can be defined by

$$\begin{aligned} \oplus / [] &= 1_\oplus \\ \oplus / (x \mathbin{++} y) &= (\oplus / x) \oplus (\oplus / y) \\ \oplus / (x : xs) &= x \oplus (\oplus / xs) \end{aligned} \tag{6}$$

Function $concat$ is an example of a reduce: it is defined by $concat = \mathbin{++} /$. For the composition of a map and function $\mathbin{++} /$, and for the composition of a reduce, and function $\mathbin{++} /$ we have

$$f^* \cdot \mathbin{++} / = \mathbin{++} / \cdot f^{**} \tag{7}$$

$$\oplus / \cdot \mathbin{++} / = \oplus / \cdot (\oplus /)^* \tag{8}$$

Terminals and Nonterminals

Suppose a terminal is a value of type b , and a nonterminal is a value of type a . A

symbol that is either a nonterminal of type a or a terminal of type b is a value of the datatype *Symbol* defined as

$$\text{data Symbol } a \ b \ = \ N \ a \mid T \ b$$

An element $N \ x$ is considered to be a nonterminal, and an element $T \ y$ is considered to be a terminal.

Grammars

A context-free grammar consists of sets of nonterminals, terminals, productions, and a start-symbol. In Gofer, the sets of nonterminals and terminals correspond with the types a and b , respectively. These types are parameters of the definition of a context-free grammar. We represent a context-free grammar in Gofer by a pair, the first component of which denotes the start-symbol, and the second component of which denotes the productions of the grammar. The start-symbol is a nonterminal, i.e., a value of type a . The productions of a grammar are a set of pairs the left-component of which is a non-terminal, and the right component of which is a list of symbols. A context-free grammar is a value of the type *Grammar*, which is defined by

$$\text{type Grammar } a \ b \ = \ (a, [(a, [\text{Symbol } a \ b])])$$

Function *rhss* takes a grammar and a nonterminal nt and returns the right-hand sides of the productions of nt . It is defined by

$$\text{rhss } g \ nt \ = \ [rhs \mid (nt, rhs) \leftarrow \text{exr } g]$$

Function *nts* takes a grammar, and returns the list of nonterminals of the grammar. We assume that for each nonterminal there exists at least one production. Let function *rmdups* remove duplicates from a list, then function *nts* is defined by

$$\text{nts } g \ = \ \text{rmdups } (\text{exl} * (\text{exr } g))$$

Parse Trees

To determine whether or not the empty string can be derived from a nonterminal (the EMPTY problem), we have to refer to all sentences that are derivable from the given nonterminal in the given grammar. A derivation using productions of a context-free grammar corresponds to a *parse tree* or *derivation tree*, i.e., an element of the datatype *Rosetree*, which is defined by

$$\text{data Rosetree } a \ b \ = \ \text{Node } a \ [\text{Rosetree } a \ b] \mid \text{Leaf } b$$

Suppose function $\text{top} : \text{Rosetree } a \ b \rightarrow \text{Symbol } a \ b$ returns the top of a rose-tree. For each subtree of a derivation tree of the form $\text{Node } a \ x$ we have that $a \rightarrow \text{top} * x$ is a production of the grammar.

The function *sen* takes a rose-tree, and returns the sentence of which the rose-tree is a derivation. Function *sen* is defined by

$$\begin{aligned} \text{sen } (\text{Node } a \ x) &= \text{++} / (\text{sen} * x) \\ \text{sen } (\text{Leaf } b) &= [b] \end{aligned}$$

Catamorphisms on Rose-Trees

For many kinds of recursive datatypes we can define a class of functions, called catamorphisms, which recursively replace constructors by functions [8]. By definition, a *catamorphism* on the datatype *Rosetree* is a function $h : \text{Rosetree } a \ b \rightarrow c$ that is uniquely determined by functions f and g as follows.

$$\begin{aligned} h (\text{Node } a \ x) &= f \ a \ (h * x) \\ h (\text{Leaf } b) &= g \ b \end{aligned}$$

For such a function h we write $h = \llbracket f, g \rrbracket$. The function *sen* defined above is a catamorphism, i.e.,

$$\text{sen} = \llbracket \lambda a \ x \rightarrow \# / x, \lambda b \rightarrow [b] \rrbracket$$

Another example of a catamorphism on *Rosetree* is the *height* function, which returns the height of a rose-tree. The definition of function *height* is omitted. We will encounter several other catamorphisms on rose-trees in the following sections.

3 Lattice Theory

This section only gives the definitions of notions from lattice theory that are used in the subsequent sections. For a more extensive introduction to lattice theory the reader is referred to e.g. [2].

Lattices and CPO's

A *partial order* on a set A is a reflexive, antisymmetrical, and transitive binary relation on A . A *partially ordered set* or *poset* is a pair $(D, \sqsubseteq)$ consisting of a set D together with a partial order $\sqsubseteq$ on D . If it exists, the least or *bottom* element of a poset is usually denoted by $\perp$. Given $d, d' \in D$, their *join*, denoted by $d \sqcup d'$, is the least element in D that is greater than both d and d' . It is fully characterised by the following equation:

$$c = d \sqcup d' \equiv (\forall e :: c \sqsubseteq e \equiv d \sqsubseteq e \wedge d' \sqsubseteq e)$$

Note that the join of two elements in D is uniquely defined when it exists. The *least upperbound* or *lub* of a subset $X \subseteq D$ is denoted by $\sqcup / X$. It is defined by

$$c = \sqcup / X \equiv (\forall e :: c \sqsubseteq e \equiv (\forall x : x \in X : x \sqsubseteq e))$$

Not every $X \subseteq D$ needs to have a lub. Let $(D, \sqsubseteq)$ be a poset. If for all elements d and d' their join $d \sqcup d'$ exists, then $(D, \sqsubseteq)$ is called a *join semilattice*. Let S be a subset of a poset. S is said to be *directed* if every finite subset of S has an upper bound. A poset D is a *complete partial order* or *CPO* if it contains a bottom element, and if each directed subset of D has a lub, so $\sqcup / X$ exists for all directed subsets $X \subseteq D$.

Fixed Points

An element $d \in D$ is a *fixed point* of function $f : D \rightarrow D$ if $f \ d = d$. It is a *least fixed point* if for any other fixed point d' of f we have $d \sqsubseteq d'$. A function $f : D \rightarrow E$

is *monotonic* if it respects the ordering on D , i.e., $d \sqsubseteq d' \Rightarrow f d \sqsubseteq f d'$. A function $f : D \rightarrow E$ is *continuous* if it respects lubs of directed subsets, i.e., if $X \subseteq D$ is a directed subset, then $(f \cdot \sqcup/) X = (\sqcup/ \cdot f*) X$.

Let D be a finite set, $(D, \sqsubseteq)$ a CPO with bottom $\perp$, and $g : D \rightarrow D$ a continuous function. It follows from the CPO Fixed Point Theorem I [2] that function g has a least fixed point μg , defined by $\mu g = \sqcup/ [g^n \perp \mid n \leftarrow [0..]]$. Since $g^i \perp \sqsubseteq g^{i+1} \perp$, we have that the least fixed point of g equals the first element in $[g^n \perp \mid n \leftarrow [0..]]$ that occurs twice, i.e., $\sqcup/ [g^n \perp \mid n \leftarrow [0..]] = lfp g \perp$ where function lfp is defined by

$$lfp f x = \begin{cases} x & \text{if } f x = x \\ lfp f (f x) & \text{otherwise} \end{cases}$$

The *Fixed Point Fusion Theorem* (or Plotkin's Lemma) is used to reason about fixed points. This theorem reads as follows.

$$f \perp = \perp \wedge f \cdot h = g \cdot f \Rightarrow f \mu h = \mu g$$

We use the Fixed Point Fusion Theorem and the CPO Fixed Point Theorem I as follows. Consider the function $(+1)$. Define $\infty = \mu(+1)$. Taking $h = (+1)$ and writing 0 for the bottom $\perp$ of natural numbers, we get, applying the Fixed Point Fusion Theorem,

$$f 0 = \perp \wedge f (n+1) = g (f n) \Rightarrow f \infty = \mu g$$

4 Grammar Analysis Problems

Although in some grammar analysis problems only a property of the start-symbol of the grammar is sought, we define a grammar analysis problem to be a problem which requires finding information about all nonterminals of the grammar. This section defines bottom-up grammar analysis problems. The first subsection gives some examples of grammar analysis problems. The second subsection discusses functions for generating derivation trees. The third subsection defines bottom-up grammar analysis problems.

4.1 Examples of Grammar Analysis Problems

Part of determining whether or not a grammar is LL(1) consists of solving the grammar analysis problems EMPTY, FIRST, and FOLLOW.

EMPTY

Given a grammar g and a nonterminal nt from g , the expression *Empty g nt* is a boolean expressing whether or not it is possible to derive the empty string from nt , using the productions from g . *Empty g nt* is defined by

$$Empty g nt = [] \neq [x \mid nt \xRightarrow{*} x, x = []]$$

where $\xRightarrow{*}$ denotes a derivation with productions from g .

FIRST

Given a grammar g and a nonterminal nt from g , the expression *First g nt* is the set

of terminals (the set of terminals is the set X) that can appear as the first element of a string of terminals derivable from nt . It is defined by

$$\text{First } g \text{ } nt = \text{rmdups } [a \mid nt \xRightarrow{*} [a] \uparrow x, x \in X^*]$$

FOLLOW

Given a grammar g and a nonterminal nt from g , the expression $\text{Follow } g \text{ } nt$ is the set of terminals that can follow on nt in a derivation starting with the start-symbol S from g . It is defined by

$$\text{Follow } g \text{ } nt = \text{rmdups } [a \in X \mid S \xRightarrow{*} u \uparrow [nt, a] \uparrow v]$$

Bottom-up versus Top-down

The definitions in the first two examples given above require finding information about a nonterminal, and do not refer to the context in which such a nonterminal appears. These two examples are bottom-up grammar analysis problems. The definition in the last example explicitly refers to the context in which the nonterminal appears, namely $u \uparrow [_, a] \uparrow v$. This example is a top-down grammar analysis problem. In the rest of the paper we limit ourselves to bottom-up problems.

4.2 Generating Trees

The definitions in the examples of grammar analysis problems given in the previous subsection typically refer somehow to all sentences derivable from a nonterminal. The sentences derivable from a nonterminal can be obtained from the derivation trees of the grammar with the given nonterminal in the root. In this subsection we define a function returning all possible derivation trees of a grammar.

Function *generate* takes a grammar, and returns a list of lists, in which each list contains all derivation trees with the same nonterminal in the root. Before we give the definition, we discuss the function *cp* (cartesian product), which is used in the definition of function *generate*.

Function *cp*

Function *cp* returns the cartesian product of a list of lists. It is defined as a map followed by a reduce by

$$\begin{aligned} cp &= \lambda / \cdot [\cdot]** \\ xs \lambda ys &= [x \uparrow y \mid x \leftarrow xs, y \leftarrow ys] \end{aligned}$$

where $[\cdot]$ takes an element a , and returns the singleton list containing that element: $[a]$. Note that $[[\]]$ is the unit of operator λ . Function *cp* commutes with function *f*** for all functions f , i.e., for all functions f we have

$$f** \cdot cp = cp \cdot f** \tag{9}$$

Function *generate*

Function *gh* is defined in the context of a grammar *g*, which from now on is considered a constant. It takes a natural number *n*, and a symbol *s*, and returns the collection of all derivation trees, of height at most *n*, derivable with the productions of *g* with symbol *s* in the root, so

$$gh\ n\ s = [y \mid s \xrightarrow{*} y \wedge height\ y \leq n]$$

where we suppose that $\xrightarrow{*}$ derives derivation trees instead of strings with productions from grammar *g*. Function *generate* is defined in terms of function *gh* as follows.

$$generate\ g = (gh \ \infty \cdot N)*\ (nts\ g)$$

Function *gh* can be defined recursively in various ways; we have chosen the following definition which is easily manipulated in calculations. Function *gh* is defined by pattern matching on its arguments. There are no trees of height zero, so $gh\ 0\ symbol = []$. There is just one derivation tree of height at most *n*+1 that can be built from a terminal: $gh\ (n+1)\ (T\ b) = [Leaf\ b]$. The list of derivation trees of height at most *n*+1 derivable from a nonterminal *nt* contains the list of the derivation trees of height at most *n* derivable from *nt*. Furthermore, for each production for *nt* we add the cartesian product of the derivation trees of height at most *n* of the symbols of the right-hand side of a production for *nt*; each element of the cartesian product is turned into a derivation tree using function *Node nt*.

$$\begin{aligned} gh\ (n+1)\ (N\ a) &= (gh\ n\ (N\ a)) \uplus \\ &\quad [Node\ a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)*\ rhs)] \end{aligned} \quad (10)$$

We do not bother about duplicate elements in $gh\ n\ s$; applying function *rmdups* to the right-hand expression of the last equation would have removed them. The right-hand side argument of $\uplus$ in the last equation of the definition of function *gh* can be rewritten using laws for list-comprehensions. The resulting equality will be used in the calculation in Section 5.

$$\begin{aligned} &[Node\ a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)*\ rhs)] \\ &= \text{equations (5), (3), and (4) for list-comprehensions} \\ &((Node\ a)* \cdot \uplus / \cdot cp* \cdot (gh\ n)**)\ (rhss\ g\ a) \end{aligned}$$

4.3 Bottom-up Problems

We formalise the notion of a grammar analysis problem. In case of the **EMPTY** problem, we want to determine for all nonterminals *nt* from a grammar *g* whether or not it is possible to derive the empty string from nonterminal *nt*. A non-executable specification for this problem reads as follows. Given a nonterminal *nt* we apply a function *p* to each derivation tree with *nt* in the root. Function *p* determines whether or not the string represented by the derivation tree is empty, i.e., $p = ([\] =) \cdot sen$. Note that function *p* corresponds with the two expressions $nt \xrightarrow{*} x, x = [\]$ occurring in the list-comprehension in the definition of *Empty g nt*. To determine whether or not it is possible to derive the empty string from nonterminal *nt*, we apply the function *combine* to the list of results obtained by applying function *p* to

all derivation trees with nt in the root. Function *combine* is defined by $combine = ([] \neq)$, which equals the reduction $\vee/$. It corresponds with the expression in front of the list-comprehension in the definition of *Empty g nt*. Generalising this pattern, we now define the class of bottom-up grammar analysis problems.

(11) Definition A bottom-up grammar analysis problem, which analyses a grammar g with respect to a function $p : \text{Rosetree } a \ b \rightarrow c$, and an operator $\oplus : c \times c \rightarrow c$ with unit $1_\oplus$, is an expression of the form $ag \ g \ p \ \oplus$, where function ag is defined as follows.

$$\begin{aligned} ag \ g \ p \ \oplus &= (id \ \Delta \ (af \cdot gh \ \infty \cdot N)) * (nts \ g) \\ &\text{where } af = combine \cdot properties \\ &\quad properties = p * \\ &\quad combine = \oplus / \end{aligned}$$

In case of the bottom-up grammar analysis problems **EMPTY** and **FIRST** we write

$$\begin{aligned} empties \ g &= ag \ g \ (([] =) \cdot sen) \ \vee \\ firsts \ g &= ag \ g \ (take \ 1 \cdot sen) \ \cup \end{aligned}$$

where operator $\cup$ is defined by $x \cup y = rmdups \ (x \ ++ \ y)$.

5 The Derivation of an Algorithm

Function ag can be implemented in a functional language, but executing $ag \ g \ p \ \oplus$ will result in a nonterminating computation because of the occurrence of ∞ in the definition of function ag . This section derives an algorithm that can be implemented as an always terminating program that returns the value of $ag \ g \ p \ \oplus$. To obtain this algorithm we use the lattice theory given in Section 3.

Replacing the constant ∞ by a variable n in the definition of function ag (Definition (11)) results in the following equality.

$$\begin{aligned} ag \ g \ p \ \oplus &= agn \ \infty \\ &\text{where } agn \ n = (id \ \Delta \ (af \cdot gh \ n \cdot N)) * (nts \ g) \end{aligned}$$

We use the CPO fixed point theorems to find the value of $agn \ \infty$ in finite time. Suppose there exists a function K such that for $n \geq 0$

$$agn \ (n+1) = K \ (agn \ n) \tag{12}$$

If we suppose furthermore that there exists a CPO $(E, \sqsubseteq_E)$ with bottom $agn \ 0$, then the results in Section 3 show that if function $K : E \rightarrow E$ is continuous, then it has a least fixed point μK and $agn \ \infty = \mu K$.

The domains used in the grammar analysis problems are finite, that is, the target type E of function ag is a finite type. Since every finite join semilattice is a CPO, and since each monotonic function on a finite domain is continuous, it suffices to find a join semilattice with bottom $agn \ 0$, and a monotonic function K satisfying (12).

This section consists of three subsections. The first subsection constructs a join semilattice with bottom $agn\ 0$ for bottom-up grammar analysis problems. The second subsection derives a definition of a monotonic function K that satisfies equation (12). The third subsection discusses the conditions imposed upon the components of the bottom-up grammar analysis problem during the derivation of function K .

5.1 Constructing a Join Semilattice with Bottom $agn\ 0$

We want to construct a join semilattice $(E, \sqsubseteq_E)$ with bottom $agn\ 0$ and join $\sqcup_E$. For that purpose, we impose our first condition on bottom-up grammar analysis problems.

We omit the calculation of the following equality for value $agn\ 0$.

$$agn\ 0 = (id \triangle 1_{\oplus}^*) * (nts\ g)$$

where $a \star b = a$ for all a and b . It follows that $agn\ 0$ is a list of length equal to the number of nonterminals of g , of which the second components are all equal to $1_{\oplus}$. This suggests to construct the following join semilattice. Let E be the set of lists x of length equal to the number of nonterminals of g of which $exl \star x = nts\ g$, and of which the second component of each element is an element of c , the result type of operator $\oplus$.

For the definition of the relation $\sqsubseteq_E$ and the join $\sqcup_E$, we suppose that there exists a relation $\sqsubseteq_c$ such that $(c, \sqsubseteq_c)$ with join $\sqcup_c$ is a join semilattice, and such that the unit $1_{\oplus}$ of operator $\oplus$ occurring in the definition of a bottom-up grammar analysis problem is the bottom of c . Both the relation $\sqsubseteq_E$ and the join $\sqcup_E$ are now straightforward extensions of $\sqsubseteq_c$ and $\sqcup_c$, respectively. Relation $\sqsubseteq_E$ is defined by pairwise comparing elements with $\sqsubseteq_c$.

$$x \sqsubseteq_E y \equiv and\ (exr \star x\ \Upsilon_{\sqsubseteq_c}\ exr \star y)$$

where function and is the reduce $\wedge$, and where $\Upsilon_{\oplus}$, with $\oplus$ a binary function, zips two lists of equal length to a list of pairs, and then applies operator $\oplus$ to all pairs in the list. The join of two elements is defined by pairwise joining the second components of the pairs.

$$x \sqcup_E y = exl \star x\ \Upsilon\ (exr \star x\ \Upsilon_{\sqcup_c}\ exr \star y)$$

It is easy to prove that $agn\ 0$ is the bottom of E , using the fact that $1_{\oplus}$ is the bottom of c , and that $(E, \sqsubseteq_E)$ is a join semilattice.

5.2 Finding Function K

In this subsection we derive a definition of a monotonic function K satisfying (12). It follows that $agn\ \infty$ is the least fixed point of function K , i.e., $agn\ \infty = \mu K$.

If we assume that there exists a J such that $af \cdot gh\ (n+1) \cdot N = J\ (agn\ n)$, then we easily calculate the following equality for $agn\ (n+1)$.

$$agn\ (n+1) = (id \triangle (J\ (agn\ n))) * (nts\ g)$$

It follows by abstracting from $agn\ n$ in the last expression that a function K satisfying (12) is defined by

$$K\ x = (id \triangle J\ x)*\ (nts\ g) \quad (13)$$

It remains to find a function J such that $af \cdot gh\ (n+1) \cdot N = J\ (agn\ n)$ holds.

Function J is obtained by manipulating the expression $af\ (gh\ (n+1)\ (N\ a))$, where a is an element of $nts\ g$. Applying the definitions of map and reduce, we have

$$\begin{aligned} af\ (gh\ (n+1)\ (N\ a)) &= af\ (gh\ n\ (N\ a)) \oplus \\ &\quad af\ [Node\ a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)*\ rhs)] \end{aligned}$$

We express the arguments of operator $\oplus$ in the last expression above in terms of $agn\ n$ separately. By definition of $agn\ n$ we have

$$af \cdot gh\ n \cdot N = r\ (agn\ n) \quad (14)$$

if function r is defined by $r\ x\ a = at\ x\ a$, where function at returns the right component of the pair in x of which the left component equals a . This equation is used to express the left-hand argument of operator $\oplus$ in terms of $agn\ n$. It remains to express the right-hand argument of operator $\oplus$ in terms of $agn\ n$.

$$\begin{aligned} &af\ [Node\ a\ c \mid rhs \leftarrow rhss\ g\ a, c \leftarrow cp\ ((gh\ n)*\ rhs)] \\ &= \text{equality from Section 4} \\ &\quad (af \cdot (Node\ a)* \cdot \# / \cdot cp* \cdot (gh\ n)**)\ (rhss\ g\ a) \end{aligned}$$

If we can push af to the right within the map $(gh\ n)**$ in the composition of functions of the last expression in the above calculation, then we can use equation (14) again to obtain an expression of the desired form. Aiming at pushing af to the right then, we proceed with the composition of functions $af \cdot (Node\ a)* \cdot \# / \cdot cp* \cdot (gh\ n)**$. Abbreviate function $Node\ a$ to mt . Applying the definition of af , and equations (2), (7), and (8) we obtain

$$af \cdot mt* \cdot \# / \cdot cp* \cdot (gh\ n)** = \oplus / \cdot (\oplus / \cdot (p \cdot mt)* \cdot cp \cdot (gh\ n))*$$

At this point of the calculation we assume that there exists a function pn such that

$$p\ (Node\ nt\ x) = pn\ nt\ ((top \triangle p)*\ x) \quad (15)$$

$$p\ (Leaf\ x) = pl\ x \quad (16)$$

This condition is not unreasonable: for all *Rosetree* catamorphisms there exists such a function pn . We proceed the calculation with the expression within the map in the last expression of the above calculation.

$$\begin{aligned} &\oplus / \cdot (p \cdot mt)* \cdot cp \cdot (gh\ n)* \\ &= \text{assumption (15), equations (2), (9)} \\ &\quad \oplus / \cdot (pn\ nt)* \cdot cp \cdot ((top \triangle p)*\ (gh\ n))* \\ &= \text{assume equation (17) below} \\ &\quad H\ nt \cdot (id \triangle (af \cdot gh\ n))* \\ &= \text{introduction of function } r' \text{ below} \\ &\quad H\ nt \cdot (r' \cdot (agn\ n))* \end{aligned}$$

In this calculation we have assumed the existence of two functions: H , and r' such that a number of properties is satisfied. We have assumed the existence of a function H such that the following equality is satisfied.

$$\oplus / \cdot (pn \ nt) * \cdot cp \cdot ((top \ \Delta \ p) * \cdot (gh \ n)) * = H \ nt \cdot (id \ \Delta \ (af \cdot gh \ n)) * \quad (17)$$

Finally, function r' is defined by

$$\begin{aligned} r' \ x \ (N \ a) &= (N \ a, r \ x \ a) \\ r' \ x \ (T \ b) &= (T \ b, pl \ b) \end{aligned}$$

The above derivation shows that if there exists a function H satisfying (17), then there exists a function J such that $af \cdot gh \ (n+1) \cdot N = J \ (agn \ n)$ holds. Function J is defined by

$$J \ x \ a = (r \ x \ a) \oplus ((\oplus / \cdot (H \ a \cdot (r' \ x) *) *) (rhss \ g \ a)) \quad (18)$$

It remains to prove that there exists a function H such that equation (17) is satisfied, and that function K defined in equation (13) is monotonic. The former condition is discussed in the next subsection. The latter condition is satisfied if operator $\oplus$ is monotonic in both its arguments (which is for example true for the join $\sqcup_c$ of the semilattice c by means of which the semilattice E is defined), and if H is monotonic in its second argument. From now on we assume $\oplus = \sqcup_c$.

We give an operational interpretation of the functions we have derived. Given a grammar g and a CPO $(E, \sqsubseteq_E)$, we compute the least fixed point of function K , starting with $K \perp$, where $\perp$ is the bottom of E , and repeatedly applying K until we find a value x such that $K \ x = x$. Function K applies function J to all nonterminals of g . Function J takes the old value of K and a nonterminal nt , and returns the new value for nt by applying the function $H \ nt \cdot (r' \ x) *$ to all right-hand sides of the productions of nonterminal nt . The results are combined by taking the join $\oplus /$ of the values thus obtained, and, finally, by joining the result with the old value for nt .

5.3 The Conditions

In the previous subsections we have derived a function K by means of which a bottom-up grammar analysis problem can be solved. In the derivation we have imposed a number of conditions upon the components of the grammar analysis problems. This subsection discusses these conditions.

The first condition we imposed upon bottom-up grammar analysis problems is the following. We suppose there exists a join semilattice $(c, \sqsubseteq_c)$ such that $1_\oplus$ is the bottom of c , and $\oplus$ is the join $\sqcup_c$ of c .

For the second condition we suppose that there exists a monotonic function H , such that equality (17) holds. Such a function H always exists, but the definition of a general function satisfying equality (17) is rather useless (and omitted): it recomputes the required information from scratch instead of using the available information, and is therefore highly inefficient. To obtain a practical solution for a bottom-up grammar analysis problem we discuss a special case in which we can find a monotonic function H that can be implemented as an efficient program.

Suppose the property function p is a catamorphism on *Rosetree*. Then we have that function $pn\ nt$ defined by $pn\ nt = qn\ nt \cdot err*$, where function qn is the function of the *Rosetree* catamorphism for p , satisfies assumption (15). For the left-hand expression of equation (17) we have the following equality

$$\oplus / \cdot (pn\ nt)* \cdot cp \cdot ((top \Delta p)* \cdot (gh\ n))* = qn\ nt \cdot err* \cdot (id \Delta (af \cdot gh\ n))*$$

provided there exists a function qn such that

$$p \cdot (Node\ nt) = qn\ nt \cdot p* \quad (19)$$

$$\oplus / \cdot (qn\ nt)* \cdot cp = qn\ nt \cdot (\oplus /)* \quad (20)$$

It follows that function H can be defined by $H\ nt = qn\ nt \cdot err*$. The second assumption (20) is still rather unwieldy, and can be simplified. To obtain a simpler condition we apply the theory for cp developed in [5]. For that purpose, we first assume that function $qn\ nt$ is a reduction, that is, there exists an operator $\otimes$ with unit $1_\otimes$ such that $qn\ nt = \otimes /$. Now we apply a theorem from [5], which states that (20) holds, provided the sections $(a \otimes)$ and $(\otimes a)$ distribute over operator $\oplus$, and provided for all y , $1_\oplus \otimes y = y \otimes 1_\oplus = 1_\oplus$. Function $H\ nt$ is monotonic provided function $qn\ nt$ is monotonic, and function $qn\ nt$ is monotonic provided operator $\otimes$ is monotonic in both arguments.

5.4 Examples

This section shows how we apply the theory derived in the previous section to the examples of bottom-up grammar analysis problems given in Section 4. The algorithm derived in the previous section can be used to solve a bottom-up grammar analysis problem provided the components of the grammar analysis problem satisfy the conditions given in the previous section.

EMPTY

We verify the conditions the components of the definition of the bottom-up grammar analysis problem EMPTY have to satisfy.

First, the join semilattice $(c, \sqsubseteq_c)$ upon which the join semilattice $(E, \sqsubseteq_E)$ is built is the join semilattice of booleans.

For the second assumption, we have to construct a function H such that equation (17) holds. To obtain a definition of function H that can be implemented as an efficient program, we verify the conditions listed in the previous subsection. We have to show that function p defined by $p = ([] =) \cdot sen$ is a *Rosetree* catamorphism, i.e., there should exist a function qn such that $p (Node\ nt\ x) = qn\ nt (p* x)$ holds. Function qn is defined by $qn\ nt\ x = and\ x$. Furthermore, we have to show that $\wedge$, the operator of the reduction for *and*, distributes backwards and forwards over $\vee$, and that *false* is a zero of $\wedge$. These equalities hold for *false*, $\vee$ and $\wedge$. Finally, $\wedge$ is monotonic in both arguments.

FIRST

We verify the conditions the components of the definition of the bottom-up grammar analysis problem FIRST have to satisfy.

First the join semilattice $(c, \sqsubseteq_c)$ upon which the join semilattice $(E, \sqsubseteq_E)$ is built is the join semilattice of terminals, where c is the set of terminals, the relation $\sqsubseteq_c$ is

the subset relation, $[]$ is the bottom of c , and the join $\sqcup_c$ is set union, or $rmdups \cdot \#$. Clearly, set union is associative, and $[]$ is the unit of set union.

For the second assumption, we have to construct a function H that can be implemented as an efficient program, such that equation (17) holds. The condition (20) given in the previous subsection does not hold for function p defined by $p = take\ 1 \cdot sen$. It is not difficult to find a *Rosetree* catamorphism for p , so (19) is satisfied, but the second requirement (20) does not hold. It follows that we have to find another way to construct function H . Function H is defined by $H\ nt = foldr\ t\ 1_\oplus$ where function t is defined as follows. If the current symbol in the right-hand side of a production is a terminal, then the symbols that can appear as the first symbol of a string are the symbols found until then, and no more: $t\ (T\ b, y)\ x = y$. If the current symbol in the right-hand side of a production is a nonterminal $N\ a$, then we distinguish two cases depending on whether or not $N\ a$ can derive the empty string. If $N\ a$ can derive the empty string, then the symbols that can appear as the first symbol of a string are the symbols found until then together with the first symbols of the remaining part of the production. If $N\ a$ cannot derive the empty string then the symbols that can appear as the first symbol of a string are the symbols found until then, and no more.

$$t\ (N\ a, y)\ x = \begin{cases} rmdups\ (y \# x) & \text{if } at\ empties\ a \\ y & \text{otherwise} \end{cases}$$

We can prove equation (17) for function H thus defined by induction to the structure of lists: apply both sides to $[]$ and $[a] \# x$, and show that the resulting expressions have the same recursive structure. It can be shown that H is monotonic.

6 Implementation

The definitions of some of the functions and datatypes given above are translated into Gofer as follows (the Gofer text is set in two columns).

```
split f g x = (f x , g x)                                ,Eq [(a,[b])]
                                                         ,Eq [[b]]
data Symbol a b = N a | T b                               ,Semilattice [b]
                                                         ) =>
type Grammar a b =                                         Grammar a b -> [(a,[b])]
  (a,[(a,[Symbol a b])])    firsts g =
                             ag
                             g
                             (\nt x -> foldr t bottom x)
                             (\b -> [b])
                             where
                             t (N a,y) x
                             | eg 'at' a = nub (y ++ x)
                             | otherwise = y
                             t (T b,y) x = y
                             eg = empties g
rhss :: Eq a => Grammar a b ->
  Grammar a b ->
  a ->
  [[Symbol a b]]
rhss g nt =
  [rhs | (z,rhs) <- snd g, z==nt]
nts :: Eq a => Grammar a b -> [a]
nts g = nub (map fst (snd g))
at :: Eq a => [(a,b)] -> a -> b
```

```

class Semilattice a where
  join  :: a -> a -> a
  bottom :: a

instance Semilattice Bool where
  join = (||)
  bottom = False

instance Eq a => Semilattice [a]
  where
    join = \a b -> nub (a ++ b)
    bottom = []

lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x = x
        | otherwise = lfp f (f x)

lub :: Semilattice a => [a] -> a
lub = foldl join bottom

empties :: Eq [(a,Bool)] =>
  Grammar a b ->
  [(a,Bool)]

empties g =
  ag
  g
  (\nt x -> and (map snd x))
  (\a -> False)

firsts :: (Eq [(a,Bool)])

at xys x =
  head [y | (z,y) <- xys, z==x]

ag :: (Eq a
      ,Eq [(a,c)]
      ,Eq [c]
      ,Semilattice c
      ) =>
  Grammar a b ->
  (a -> [(Symbol a b,c)] -> c) ->
  (b -> c) ->
  [(a,c)]

ag g pn pl =
  lfp
  k
  (map
    (split id (\x -> bottom))
    nt's)
  where
    nt's = nts g
    k x = map (split id (j x)) nt's
    j x nt = (r x nt) 'join'
            ((lub
              .map (pn nt
                    . map (r' x)
                    )
              .rhss g)
             nt
            )
    r x nt = at x nt
    r' x (N a) = (N a, r x a)
    r' x (T b) = (T b, pl b)

```

There are a number of ways in which a more efficient program can be obtained, for example by changing the definition of function `lfp`. For a discussion on this subject the reader is referred to [6].

7 Conclusions

This paper discusses bottom-up grammar analysis problems. We give a very general specification of bottom-up grammar analysis problems, and from this specification we derive, by means of program transformation applying laws to the components of the intermediate expressions, an algorithm for performing bottom-up grammar analysis. The driving force in the derivation of the algorithm is the construction of a fixed point. To obtain such a fixed point a number of conditions have to be imposed upon the components of the bottom-up grammar analysis problem. Thus we derive both the algorithm and the conditions under which the fixed point exists in one go. The derivation is an example of a derivation of a real-world program. The research reported on in this paper is still in progress: in the next version we want to split the calculation in two parts. The first part of the derivation assumes that

the function that computes the property of a parse tree is a *Rosetree* catamorphism and the second part of the derivation adds, if necessary, the extra information (for example in the case of *firsts*, where we use information about the *empties*). This simplifies the derivation. Future research will be directed towards the derivation of an algorithm for top-down grammar analysis.

Acknowledgements

Erik Meijer and Graham Hutton meticulously read an earlier version of this paper.

References

- [1] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.
- [2] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- [3] J.H. Fasel, P. Hudak, S. Peyton Jones, and P. Wadler. Sigplan Notices Special Issue on the Functional Programming Language Haskell. *ACM SIGPLAN notices*, 27(5), 1992.
- [4] Andy Gill and Simon Marlow. Happy manual. Published on `comp.lang.functional`, 1993.
- [5] J. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993.
- [6] J. Jeuring and S.D. Swierstra. Bottom-up grammar analysis —a functional formulation—. Technical Report UU-CS-1994-01, Utrecht University, 1994.
- [7] M.P. Jones. Introduction to Gofer 2.20. Programming Research Group, Oxford University, 1992.
- [8] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
- [9] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North-Holland, 1986.
- [10] Torben Mogensen. Ratatosk – a parser generator and scanner generator for Gofer. Published on `comp.lang.functional`, 1993.
- [11] Ulrich Möncke and Reinhard Wilhelm. Grammar flow analysis. In *Attribute Grammars, Applications and Systems*, SAGA '91, pages 151–186. Springer-Verlag, New York, 1991. LNCS 545.
- [12] Simon L. Peyton Jones. Yacc in Sasl – an exercise in functional programming. *Software-Practice and Experience*, 15(8):807–820, 1985.
- [13] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.