Lecture Notes on Forward Computing

15-317: Constructive Logic

Frank Pfenning André Platzer

Lecture 22
November 19, 2015

1 Introduction

In this lecture we continue forward logic programming with a nice application to understand parsing as proving. We also explore arithmetic and related forward and backward reasoning.

2 Context-Free Grammars

Grammars are designed to describe languages, where in our context a *language* is just a set of strings. Abstractly, we think of strings as a sequence of so-called *terminal symbols*. Inside a compiler, these terminal symbols are most likely *lexical tokens*, produced from a bare character string by lexical analysis that already groups substrings into tokens of appropriate type and skips over whitespace.

A *context-free grammar* consists of a set of productions of the form $X \Rightarrow \gamma$, where X is a *non-terminal symbol* and γ is a potentially mixed sequence of terminal and non-terminal symbols. It is also sometimes convenient to distinguish a *start symbol* traditionally named S, for *sentence*. We will use the word *string* to refer to any sequence of terminal and non-terminal symbols. We denote strings by $\alpha, \beta, \gamma, \ldots$. non-terminals are generally denoted by X, Y, Z and terminals by $\mathtt{a}, \mathtt{b}, \mathtt{c}$

For example, consider the following grammar of linear arithmetic.

$$
\begin{array}{llll}
[\mathsf{zero}] & T & \Rightarrow & \mathtt{0} \\
[\mathsf{one}] & T & \Rightarrow & \mathtt{1} \\
[\mathsf{id}] & T & \Rightarrow & \mathtt{id} \\
[\mathsf{plus}] & T & \Rightarrow & T\,\mathtt{+}\,T \\
[\mathsf{neg}] & T & \Rightarrow & \mathtt{-}\,T \\
[\mathsf{pars}] & T & \Rightarrow & \mathtt{(}T\mathtt{)}
\end{array}
$$

We usually label the productions in the grammar so that we can refer to them by name. The production $T \Rightarrow \mathtt{id}$ says that any identifier is accepted as a term, and can be thought of as a shorthand for a whole list of productions $T \Rightarrow \mathtt{x}$, $T \Rightarrow \mathtt{y}$, $T \Rightarrow \mathtt{z}$ and so on.

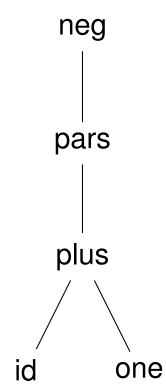
A *derivation* of a sentence w from start symbol T is a sequence $T = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_n = w$, where w consists only of terminal symbols. In each step we choose exactly one occurrence of a non-terminal X in α_i and one production $X \Rightarrow \gamma$ and replace this occurrence of X in α_i by γ.

Then the following is a derivation of the string `-(x+1)`, where each transition is labeled with the production that has been applied.

$$
\begin{array}{llll}
T & \Rightarrow & \mathtt{-}T & [\mathsf{neg}] \\
 & \Rightarrow & \mathtt{-(}T\mathtt{)} & [\mathsf{pars}] \\
 & \Rightarrow & \mathtt{-(}T\mathtt{+}T\mathtt{)} & [\mathsf{plus}] \\
 & \Rightarrow & \mathtt{-(x+}T\mathtt{)} & [\mathsf{id}] \\
 & \Rightarrow & \mathtt{-(x+1)} & [\mathsf{one}]
\end{array}
$$

We have labeled each derivation step with the corresponding grammar production that was used. The same principle can be used to describe the grammar for formulas in logic itself. We focus on arithmetic here.

Derivations are clearly not unique, because when there is more than one non-terminal, then we can replace it in any order in the string. In order to avoid this kind of harmless ambiguity in rule order, we like to construct a *parse tree* in which the nodes represents the non-terminals in a string, with the root being S. In the example above we obtain the following tree:



While the parse tree removes some ambiguity, it turns out the sample grammar is ambiguous in another way. There are two different parse trees of the string `- x + 1` and the above grammar does not specify which one it means.

Whether a grammar is ambiguous in the sense that there are sentences permitting multiple different parse trees is an important question for the use of grammars for the specification of programming languages. The basic problem is that it becomes ambiguous in which grammatical function a specific terminal occurs in the source program. This could lead to misinterpretations. In the above example, it is crucial whether unary negation only affects the `x` as in `(- x ) + 1` or whether it also affects the subsequent addition as in `- (x + 1 )`. Additional precedence information or grammar transformations are needed to make that unambiguous.

3 Parse Trees are Deduction Trees

We now present a formal definition of when a terminal string w matches a string γ. We write:

$[r]X \Rightarrow \gamma$	production r maps non-terminal X to string γ
$w : \gamma$	terminal string w matches string γ

The second judgment is defined by the following four simple rules. Here we use string concatenation, denoted by juxtaposing to strings. Note that the empty string ϵ satisfies $\gamma\,\epsilon = \epsilon\,\gamma = \gamma$ and that concatenation is associative (mathematically speaking, strings form a monoid, which is like a group that does not have inverse elements).

$$\frac{}{\epsilon : \epsilon}\,P_1 \qquad \frac{w_1 : \gamma_1 \quad w_2 : \gamma_2}{w_1\, w_2 : \gamma_1\, \gamma_2}\,P_2 \qquad \frac{}{\mathtt{a} : \mathtt{a}}\,P_3 \qquad \frac{\begin{array}{l}[r]X \Rightarrow \gamma \\ w : \gamma\end{array}}{w : X}\,P_4(r)$$

We have labeled the fourth rule by the name of the grammar production, while the others remain unlabeled. This allows us to omit the actual grammar rule from the premises since it can be looked up in the grammar directly by its name. Then the earlier derivation of `-(x+1)` becomes the following deduction, where we just write neg instead of $P_4(\mathsf{neg})$ for brevity. We also write P_2, P_2 if we use P_2 twice in one step.

$$\dfrac{\dfrac{}{\mathtt{-} : \mathtt{-}}\,P_3 \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{(} : \mathtt{(}}\,P_3 \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{x} : T}\,\mathsf{id} \quad \dfrac{}{\mathtt{+} : \mathtt{+}}\,P_3 \quad \dfrac{}{\mathtt{1} : T}\,\mathsf{one}}{\mathtt{x+1} : T\mathtt{+}T}\,P_2, P_2}{\mathtt{x+1} : T}\,\mathsf{plus} \qquad \dfrac{}{\mathtt{)} : \mathtt{)}}\,P_3}{\mathtt{(x+1)} : \mathtt{(}T\mathtt{)}}\,P_2, P_2}{\mathtt{(x+1)} : T}\,\mathsf{pars}}{\dfrac{\mathtt{-(x+1)} : \mathtt{-}T}{\mathtt{-(x+1)} : T}\,\mathsf{neg}}\,P_2$$

We observe that the P_4 labels have the same structure as the parse tree, except that it is written upside-down. *Parse trees are therefore just deduction trees*.

4 CYK Parsing

The rules above that formally define when a terminal string matches an arbitrary string can be used to immediately give an algorithm for parsing.

Assume we are given a grammar with start symbol S and a terminal string w_0. Start with a databased of assertions $\epsilon : \epsilon$ and $a : a$ for any terminal symbol occurring in w_0. Now arbitrarily apply the given rules in the following way: if the premises of the rules can be matched against the database, and the conclusion $w : \gamma$ is such that w is a substring of the input w_0 and γ is a string occurring in the grammar, then add $w : \gamma$ to the database. The side conditions are used to focus the parsing process to the facts that may matter during the parsing (i.e., that talk about the actual input string w_0 being parsed and that fit to the actual grammatical productions in the grammar).

We repeat this process until we reach *saturation*: any further application of any rule leads to conclusion are already in the database. We stop at this point and check if we see $w_0 : S$ in the database. If we see $w_0 : S$, we succeed parsing w_0; if not we fail.

This process must always terminate, since there are only a fixed number of substrings of the grammar, and only a fixed number of substrings of the query string w_0. In fact, only $O(n^2)$ terms can ever be derived if the grammar is fixed and $n = |w|$. Using a meta-complexity result by Ganzinger and McAllester [McA02, GM02] we can obtain the complexity of this algorithm as the maximum of the size of the saturated database (which is $O(n^2)$) and the number of so-called *prefix firings* of the rule. We count this by bounding the number of ways the premises of each rule can be instantiated, when working from left to right. The crucial rule is the splitting rule

$$\frac{w_1 : \gamma_1 \quad w_2 : \gamma_2}{w_1\, w_2 : \gamma_1\, \gamma_2}\ P_2$$

There are $O(n^2)$ substrings, so there are $O(n^2)$ ways to match the first premise against the database. Since $w_1\, w_2$ is also constrained to be a substring of w_0, there are only $O(n)$ ways to instantiate the second premise, since the left end of w_2 in the input string is determined, but not its right end. This yields a complexity of $O(n^2 * n) = O(n^3)$.

The algorithm we have just presented is an abstract form of the Cocke-Younger-Kasami (CYK) parsing algorithm invented in the 1960s. It originally assumes the grammar is in a normal form, and represents substring by their indices in the input rather than directly as strings. However, its general running time is still $O(n^3)$.

As an example, we apply this algorithm using an n-ary concatenation rule as a short-hand. We try to parse `-x+1` with our grammar of matching parentheses. We start with three facts that derive from rules P_1 and P_3. When working forward it is important to keep in mind that we only infer facts $w : \gamma$ where w is a substring of $w_0 =$ `-x+1` and γ is a substring of the grammar.

1	`(`	:	(	
2	`)`	:	)	
3	`+`	:	+	
4	`-`	:	−	
5	`1`	:	T	$P_4(\mathsf{one})$
6	`x`	:	T	$P_4(\mathsf{id})$
8	`x+1`	:	T+T	$P_2, P_2(6, 3, 5)$
9	`x+1`	:	T	$P_4(\mathsf{plus})$ 8
11	`(x+1)`	:	(T)	$P_2, P_2(1, 9, 2)$
12	`(x+1)`	:	T	$P_4(\mathsf{pars})$ 11
13	`-(x+1)`	:	−T	$P_2(4, 12)$
14	`-(x+1)`	:	T	$P_4(\mathsf{neg})$ 13

A few more redundant facts might have been generated, but otherwise parsing is reasonably focused in this case. From the justifications in the right-hand column is it easy to generate the same parse tree we saw earlier.

5 CKY Parsing in Chomsky-normal Form

The above algorithm is very general. A minor nuisance is that its conclusions assume there is a way of composing strings, which is not what we could, for example, write in Prolog directly. So in this section we investigate a direct way. This algorithm assumes that the grammar is in Chomsky-normal form, where productions all have the form

$$\begin{array}{lcl} x & \Rightarrow & y\,z \\ x & \Rightarrow & a \end{array}$$

where x, y, and z stand for non-terminals and a for terminal symbols. The idea of the algorithm is to use the grammar production rules from right to left as reductions to compute which sections of the input string can be parsed as which non-terminals.

We initialize the database with facts $\mathsf{rule}(x, \mathsf{char}(a))$ for every grammar production $x \Rightarrow a$ and $\mathsf{rule}(x, \mathsf{cat}(y, z))$ for every production $x \Rightarrow y\,z$. We further represent the input string $a_1 \ldots a_n$ by assumptions $\mathsf{string}(i, a_i)$. For simplicity, we represent numbers in unary form such as $\mathsf{s}(\mathsf{s}(0))$.

Our rules will infer propositions $\mathsf{parse}(x, i, j)$ which we will deduce if the substring $a_i \ldots a_j$ can be parsed as an x. Then the program is represented by the following two rules, to be read in the forward direction:

$$\frac{\begin{array}{l}\mathsf{rule}(X, \mathsf{char}(A)) \\ \mathsf{string}(I, A)\end{array}}{\mathsf{parse}(X, I, I)} \qquad \frac{\begin{array}{l}\mathsf{rule}(X, \mathsf{cat}(Y, Z)) \\ \mathsf{parse}(Y, I, J) \\ \mathsf{parse}(Z, \mathsf{s}(J), K)\end{array}}{\mathsf{parse}(X, I, K)}$$

After saturating the database with these rules we can see if the whole string is in the language generated by the start symbol ℓ by checking if the fact $\mathsf{parse}(\ell, \mathsf{s}(0), n)$ is in the database.

Let g be the number of grammar productions and n the length of the input string. In the completed database we have g grammar rules, n facts $\mathsf{string}(i, a)$, and at most $O(g \cdot n^2)$ facts $\mathsf{parse}(x, i, j)$.

Moving on to the rules, in the first rule there are $O(g)$ ways to match the grammar rule (which fixes A) and then n ways to match $\mathsf{string}(I, A)$, so we have $O(g \cdot n)$. The second rule, again we have $O(g)$ ways to match the grammar rule (which fixes X, Y, and Z) and then $O(n^2)$ ways to match $\mathsf{parse}(Y, I, J)$. In the third premiss now only K is unknown, giving us $O(n)$ way to match it, which means $O(g \cdot n^3)$ prefix firings for the second rule.

The size of the saturated database is $O(g \cdot n^2)$ and the two rules have $O(g \cdot n^3)$ prefix firings. These considerations give us an overall complexity of $O(g \cdot n^3)$, which is also the traditional complexity bound for CKY parsing: for fixed grammar, cubic in the input size. That gives an algorithm for parsing context-free grammars that is efficient in terms of its worst-case complexity. Linear complexity parsing algorithms for simpler classes of context-free grammars can be obtained in the same way.

6 Logical Arithmetic

There is a gigantic difference between unification ($s \doteq t$, written `s=t` in Prolog) and arithmetic ($s = t$ written `s is t` in Prolog[1]). Unification would consider the terms $x + (y + 1)$ and $(x + y) + 1$ different, because both have different and ununifiable syntactic structure. General arithmetic should consider $x + (y + 1)$ and $(x + y) + 1$ equal, because both will always result in the same value no matter what value x, y happen to have. Unification

[1]In fact, Prolog limits the use of `s is t` to cases where `t` is ground and only involves numerical literals such as in `2+(4+6) is (2+4)+6`, where equality is easily decided by evaluation and comparison, or `X is (2+4)+6`, where evaluation on the right and unification with the left easily yields the unifier $\sigma = (12/X)$.

would consider the terms $x + y$ and $x + z$ equal in the sense of being unifiable by the unifier $\sigma = (y/z)$. Arithmetic would *not* consider $x + y$ and $x + z$ equal since y and z are different variables (unless they happen to both evaluate to the same value in the context of Prolog).

Arithmetic can be understood more generally, though, whether with concrete numerical data or with variables as symbolic parameters. While the approach we discuss here is not the most general approach (for example we omit multiplication, which, of course, has its own challenges), it suffices to illustrate the challenges and solutions.

It is straight-forward to develop (sequent) proof rules that understand addition as being an associative ($+a$) commutative ($+c$) operation with 0 as neutral element ($+n$) and with $-t$ as the inverse element of t for each number ($+i$).

$$\frac{\Gamma \Longrightarrow s = t}{\Gamma \Longrightarrow s = t + 0}\,+n \qquad \frac{\Gamma \Longrightarrow s = r + (t + u)}{\Gamma \Longrightarrow s = (t + r) + u}\,+a \qquad \frac{\Gamma \Longrightarrow s = r + t}{\Gamma \Longrightarrow s = t + r}\,+c$$

$$\frac{\Gamma \Longrightarrow s = 0}{\Gamma \Longrightarrow s = t + (-t)}\,+i$$

Note that these rules are all phrased as algebraic transformations on the right-hand side of an equation, here. That alone can clearly never lead to a proof, nor can the same transformations be applied on the left-hand side of an equation. Duplicating each rule would help, but there is a conceptually easier way.

7 Equality

Proof rules to capture (some) important aspects of equality express that equality is reflexive ($=r$), symmetric ($=s$), and transitive ($=t$):

$$\frac{}{\Gamma \Longrightarrow s = s}\,=r \qquad \frac{\Gamma \Longrightarrow t = s}{\Gamma \Longrightarrow s = t}\,=s \qquad \frac{\Gamma \Longrightarrow s = r \quad \Gamma \Longrightarrow r = t}{\Gamma \Longrightarrow s = t}\,=t$$

The symmetry rule $=s$ can be used to use the algebraic transformations $+a$, $+c$, $+n$ and $+i$ also on the left-hand side of an equation by commuting it with $=s$, applying the transformation, and commuting it back. The reflexivity rule $=r$ can close some arithmetic proofs such as:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Longrightarrow 0 = 0}\;=r}{\Longrightarrow 0 = 0 + 0}\;+n}{\Longrightarrow 0 + 0 = 0}\;=s}{\Longrightarrow 0 + 0 = (x + y) + (-(x + y))}\;+i}{\Longrightarrow 0 + 0 = x + (y + (-(x + y)))}\;+a$$

8 Using Arithmetic / Clever Cuts

The above rules for arithmetic are quite canonical, but how should they be used to prove $x + (y + z) = x + (z + y)$? Commutativity $+c$ would be the rule to apply but deeper within the term, not on the top-level. The following rules do that by using equalities in the middle of a more complex term:

$$\dfrac{\Gamma \Longrightarrow r = f(t) \quad \Longrightarrow s = t}{\Gamma \Longrightarrow r = f(s)}\;=R \qquad \dfrac{\Gamma, r = f(t) \Longrightarrow C \quad \Longrightarrow s = t}{\Gamma, r = f(s) \Longrightarrow C}\;=L$$

Here, $f(\cdot)$ denotes a term with a reserved free variable $\cdot$ and $f(s)$ denotes the result of substituting s for $\cdot$ to yield $f(\cdot)\,(s/\cdot)$ and likewise for $f(t)$. This makes it possible to prove:

$$\dfrac{\dfrac{}{\Longrightarrow x + (y + z) = x + (y + z)}\;=r \quad \dfrac{\dfrac{}{\Longrightarrow z + y = z + y}\;=r}{\Longrightarrow z + y = y + z}\;+c}{\Longrightarrow x + (y + z) = x + (z + y)}\;= R$$

9 Forward and Backward

With one use of $=s$ and $+n$, the above proof can be continued to a proof of

$$\Longrightarrow x + (y + (-(x + y))) = 0$$

Once we have that, it is straightforward to conduct the following proof

$$\dfrac{\Gamma \Longrightarrow a = z * 0 \quad \dfrac{}{\Longrightarrow x + (y + (-(x + y))) = 0}\;\text{above}}{\Gamma \Longrightarrow a = z * (x + (y + (-(x + y))))}\;= R$$

and continue the proof search with the, substantially simplified, left premiss.

The problem is that rule $=R$ works somewhat like a cut and is only useful if we guess exactly the right t for which a proof of $s = t$ will succeed. Guessing the right t to use for $=R$ is as hard as solving the original proof problem, however.

The trick is to mix reasoning modes and combine forward reasoning with backward reasoning to obtain an efficient integration of proving and computation. There is a proof that will always succeed on the right premiss of $=R$ even if it is somewhat uninformative: that of $s = s$. From that fact, forward reasoning can conclude more interesting facts by which it ultimately constructs a term t that already has a proof of being equal to the s we started out with:

$$\dfrac{\Gamma \Longrightarrow r = f(t) \qquad \begin{matrix}\dfrac{}{\Longrightarrow s = s}{=r}\\ \vdots \\ \Longrightarrow s = t\end{matrix}}{\Gamma \Longrightarrow r = f(s)}{=R}$$

This should be somewhat reminiscent of the (overly) flexible bi-directional way we had of constructing proofs in the original two-dimensional natural deduction calculus. For example:

$$\dfrac{\begin{matrix}\vdots \\ \Gamma \Longrightarrow a = z * 0\end{matrix} \qquad \dfrac{\dfrac{\dfrac{}{\Longrightarrow x + (y + (-(x + y))) = x + (y + (-(x + y)))}{= r}}{\Longrightarrow x + (y + (-(x + y))) = (x + y) + (-(x + y))}{+a}}{\Longrightarrow x + (y + (-(x + y))) = 0}{+i}}{\Gamma \Longrightarrow a = z * (x + (y + (-(x + y))))}{= R}$$

In order to make sure the rules simplify appropriately, we need to turn the rules for neutral and inverse elements around to have the simpler equivalent as the conclusion, which, in this case, is still sound:

$$\dfrac{\Gamma \Longrightarrow s = t + 0}{\Gamma \Longrightarrow s = t}{+n} \qquad \dfrac{\Gamma \Longrightarrow s = t + (-t)}{\Gamma \Longrightarrow s = 0}{+i}$$

The associative and commutative rules are unchanged, but, in practice, need to be restricted to ensure the algebraic simplification actually makes things easier. Which is nontrivial, because, e.g., commutativity could be applied repeatedly without making any progress at all. The combination of associative and commutative is challenging, because some simplifications may only become possible after suitably mixing associativity and commutativity reasoning leading to the notion of simplification orders.

References

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.