Lecture Notes on Backward Chaining

15-816: Linear Logic
Frank Pfenning

Lecture 17
March 26, 2012

In the last lecture we saw that it is difficult to identify subcomputations in pure forward chaining. By comparison, in functional programming calling a function naturally gives rise to a subcomputation that is given some arguments and returns a result. In logic programming, the same is true if we compute using *backward chaining* instead of forward chaining. As we will see, it has other problems, so sometimes we will want to use backward chaining and sometimes forward chaining, and some problems benefit from a combination of these. Today, we will discuss pure backward chaining. The idea of backward chaining for the Horn fragment of logic goes back to Kowalski and Colmerauer (see [Kow88] for a recounting of its beginning). A more general notion was introduced as *uniform proofs* [MNPS91], which was based on the negative fragment of intuitionistic higher-order logic. A proof-theoretic introduction and many further discussions and references can be found in a set of lecture notes [Pfe06].

1 Negative Atoms and Goal-Directed Search

Backward chaining characteristically performs goal-directed search. We recall an earlier example:

$$\mathsf{a}, \mathsf{a} \multimap \mathsf{b}, \mathsf{b} \multimap \mathsf{c} \to \mathsf{c}$$

When all atoms are *positive*, we can only focus on $\mathsf{a} \multimap \mathsf{b}$. When all atoms are *negative* we can only focus on $\mathsf{b} \multimap \mathsf{c}$. In other words, the only applicable clause is one whose conclusion matches the right-hand side of the sequent.

One of the first questions should be which fragment of linear logic maintains this property. When we are *focused* on the left on a *clause* D, we should eventually come to a negative atom (which must match the succedent of the sequent), and not a positive proposition. In this process we spawn *goals* G, in right focus, which should decompose far enough that we eventually reach a negative atom, rather than stopping at a positive proposition which would not provide at atom for goal-directed search.[1]

$$\begin{array}{llll} \text{Clauses} & D & ::= & P^- \mid G \multimap D \mid D_1 \mathbin{\&} D_2 \mid \top \mid \forall x.\, D \\ \text{Goals} & G & ::= & P^+ \mid G_1 \otimes G_2 \mid \mathbf{1} \mid G_1 \oplus G_2 \mid \mathbf{0} \mid \exists x.\, G \mid\, !D \mid D \end{array}$$

2 Binary Addition

As an example, we revisit binary addition, this time as a backward-chaining program. This means all atoms will be considered negative. The main application we have in mind is a clause such as

$$\mathsf{retn}^+(n) \bullet \mathsf{cont}^+(m + \_) \bullet \mathsf{plus}^-(m, n, r) \twoheadrightarrow \mathsf{retn}^+(r)$$

where we compute r from m and n. Declaratively, $\mathsf{plus}(m, n, r)$ should be true if $m + n = r$.

We write addition as inference rules, but as we previously observed we can go back and forth between such clauses. Recall that binary numbers follow the structure

$$\begin{array}{llll} n & ::= & \epsilon & (0) \\ & \mid & n\mathsf{0} & (2n) \\ & \mid & n\mathsf{1} & (2n+1) \end{array}$$

First, adding 0 leaves a number unchanged.

$$\frac{}{\mathsf{plus}(\epsilon, n, n)} \qquad \frac{}{\mathsf{plus}(m, \epsilon, m)}$$

Then the cases that add the least significant digits. In each case, except the last, it reduces to just adding the subterms representing the higher bits. In

[1] In many prior presentation of pure backward chaining (and in lecture), all atoms were considered negative. However, their presence in goals does not appear to upset the goal-directed nature of search.

the last case we have to appeal to an increment operation, yet to be defined.

$$\frac{\mathsf{plus}(m, n, r)}{\mathsf{plus}(m0, n0, r0)} \qquad \frac{\mathsf{plus}(m, n, r)}{\mathsf{plus}(m0, n1, r1)}$$

$$\frac{\mathsf{plus}(m, n, r)}{\mathsf{plus}(m1, n0, r1)} \qquad \frac{\mathsf{plus}(m, n, k) \quad \mathsf{inc}(k, r)}{\mathsf{plus}(m1, n1, r0)}$$

In the last case, adding $m + n = k$, an intermediate value, which have to increment with the carry before tacking on the last bit 0.

$$\frac{}{\mathsf{inc}(\epsilon, \epsilon 1)} \qquad \frac{}{\mathsf{inc}(n0, n1)} \qquad \frac{\mathsf{inc}(n, r)}{\mathsf{inc}(n1, r0)}$$

These can be written in propositional form as follows:

$$\begin{array}{l}
\forall n.\, \mathsf{plus}(\epsilon, n, n) \\
\forall m.\, \mathsf{plus}(m, \epsilon, m) \\
\forall m.\, \forall n.\, \forall r.\, \mathsf{plus}(m, n, r) \twoheadrightarrow \mathsf{plus}(m0, n0, r0) \\
\forall m.\, \forall n.\, \forall r.\, \mathsf{plus}(m, n, r) \twoheadrightarrow \mathsf{plus}(m0, n1, r1) \\
\forall m.\, \forall n.\, \forall r.\, \mathsf{plus}(m, n, r) \twoheadrightarrow \mathsf{plus}(m1, n0, r1) \\
\forall m.\, \forall n.\, \forall r.\, \mathsf{plus}(m, n, k) \bullet \mathsf{inc}(k, r) \twoheadrightarrow \mathsf{plus}(m1, n1, r0)
\end{array}$$

$$\begin{array}{l}
\mathsf{inc}(\epsilon, \epsilon 1) \\
\forall n.\, \mathsf{inc}(n0, n1) \\
\forall n.\, \forall r.\, \mathsf{inc}(n, r) \twoheadrightarrow \mathsf{inc}(n1, r0)
\end{array}$$

Note that focusing on these clauses leads to the previously presented rules, in a slightly modified form. For example,

$$\frac{\Gamma\ ;\ \Delta\ ;\ \Omega_L \to \mathsf{plus}(m, n, k) \quad \Gamma\ ;\ \Delta'\ ;\ \Omega_R \to \mathsf{inc}(k, r)}{\Gamma\ ;\ \Delta, \Delta'\ ;\ \Omega_L, \Omega_R \to \mathsf{plus}(m1, n1, r0)}$$

We know that, ultimately, Δ, Δ', Ω, and Ω' will all be empty in this particular program and signature, but this is not a consequence of focusing itself.

3 An Operational Reading

Next we examine what proof search is actually like, using the rules in a goal-directed way. This is isomorphic to what we would obtain by forward

chaining on the corresponding formulas. The complication we want to explore here is that in our example, when $\mathsf{plus}(m, n, r)$ comes up as a goal, we actually know m and n, but r is unknown. To clearly distinguish these, we write uppercase letters for unknowns. Unknown terms are also called *logic variables* (in logic programming), *metavariables* (in logical frameworks) or *essentially existential variables* (in theorem proving).

We start with

$$\begin{array}{c}\vdots\\ \mathsf{plus}(\epsilon 10, \epsilon 11, R)\end{array}$$

which represents $2 + 3$ and should yield 5. There is only one applicable rule, so the shape of the proof must be

$$\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(\epsilon 1, \epsilon 1, R')\end{array}}{\mathsf{plus}(\epsilon 10, \epsilon 11, R)}\ R = R'1$$

where we have noted the constraint that must be satisfied by R. Again, only one rule applies

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \mathsf{plus}(\epsilon, \epsilon, K)\end{array}\quad\begin{array}{c}\vdots\\ \mathsf{inc}(K, R'')\end{array}}{\mathsf{plus}(\epsilon 1, \epsilon 1, R')}\ R' = R''0}{\mathsf{plus}(\epsilon 10, \epsilon 11, R)}\ R = R'1$$

In this step we have introduced two new variables, K and R''.

At this point there is a decision to make: we could try to find a proof of $\mathsf{plus}(\epsilon, \epsilon, K)$ or a proof of $\mathsf{inc}(K, R'')$. Because we want to use backward chaining as the basis for a programming language, we always solve the subgoals in left-to-right order. In logic programming terminology, this is called *left-to-right subgoal selection*.

Proceeding with the first subgoal, we see two rules are applicable, but both give the same result ($K = \epsilon$). Generally speaking, a backward-chaining logic program is considered in poor taste if there are multiple ways to prove the same conclusion because it can lead to unnecessary backtracking. We'll fix our program at the end of this section in the obvious way. After solving

the first subgoal, we arrive at:

$$\dfrac{\dfrac{}{\mathsf{plus}(\epsilon,\epsilon,K)}\ K=\epsilon \quad \begin{array}{c}\vdots\\ \mathsf{inc}(K,R'')\end{array}}{\dfrac{\mathsf{plus}(\epsilon 1,\epsilon 1,R')}{\mathsf{plus}(\epsilon 10,\epsilon 11,R)}\ R=R'1}\ R'=R''0$$

Under the constraint that $K = \epsilon$, only the first rule for increment applies. This yields $R'' = \epsilon 1$.

$$\dfrac{\dfrac{}{\mathsf{plus}(\epsilon,\epsilon,K)}\ K=\epsilon \quad \dfrac{}{\mathsf{inc}(K,R'')}\ R''=\epsilon}{\dfrac{\mathsf{plus}(\epsilon 1,\epsilon 1,R')}{\mathsf{plus}(\epsilon 10,\epsilon 11,R)}\ R=R'1}\ R'=R''0$$

At this point, there are no open subgoals. We can read off the substitution

$$R = \epsilon 101$$

and apply the solution to all the constraints to write down the final proof

$$\dfrac{\dfrac{\dfrac{}{\mathsf{plus}(\epsilon,\epsilon,\epsilon)} \quad \dfrac{}{\mathsf{inc}(\epsilon,\epsilon 1)}}{\mathsf{plus}(\epsilon 1,\epsilon 1,\epsilon 10)}}{\mathsf{plus}(\epsilon 10,\epsilon 11,\epsilon 101)}$$

4 Left-Pointing Arrow Notation

When writing backward chaining logic programs, it is visually preferable to write the succedent of the top-level implication first. This is so we can easily examine the program and discern which rules are applicable to which goals. Most clauses will then be written in one of the forms

$$P \circ\!\!- G_1 \otimes \cdots \otimes G_n$$
$$P \circ\!\!- G_1 \circ\!\!- \cdots \circ\!\!- G_n$$

which behave identically because $\circ\!\!-$ is considered a left-associative operator, so that $A \circ\!\!- B \circ\!\!- C$ is the same as $(A \circ\!\!- B) \circ\!\!- C$ which is the same as $C \multimap (B \multimap A)$. Nested subgoals will thus be solved from the inside-out.

We rewrite our program in linear logic to eliminate the unnecessary and potentially undesirable nondeterminism in our first program. We further use the convention that free variables are implicitly universally quantified.

$$
\begin{array}{l}
\mathsf{plus}(\epsilon, \epsilon, \epsilon). \\
\mathsf{plus}(\epsilon, n0, n0). \\
\mathsf{plus}(\epsilon, n1, n1). \\
\\
\mathsf{plus}(m0, \epsilon, m0). \\
\mathsf{plus}(m0, n0, r0) \multimap \mathsf{plus}(m, n, r). \\
\mathsf{plus}(m0, n1, r1) \multimap \mathsf{plus}(m, n, r). \\
\\
\mathsf{plus}(m1, \epsilon, m1). \\
\mathsf{plus}(m1, n0, r1) \multimap \mathsf{plus}(m, n, r). \\
\mathsf{plus}(m1, n1, r0) \multimap \mathsf{plus}(m, n, k) \otimes \mathsf{inc}(k, r). \\
\\
\mathsf{inc}(\epsilon, \epsilon 1). \\
\mathsf{inc}(n0, n1). \\
\mathsf{inc}(n1, r0) \multimap \mathsf{inc}(n, r).
\end{array}
$$

5 Modes

An important criteria when reasoning about the operational behavior of logic programs is to reason about which variables are *known* (have *ground* values containing variables) or *unknown* (potentially contain variables). Let's return to a point in the proof search in the earlier example.

$$
\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \mathsf{plus}(\epsilon, \epsilon, K) & \mathsf{inc}(K, R'')\end{array}}{\mathsf{plus}(\epsilon 1, \epsilon 1, R')}\ R' = R''0}{\mathsf{plus}(\epsilon 10, \epsilon 11, R)}\ R = R'1
$$

If we reversed the two subgoals

$$
\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \mathsf{inc}(K, R'') & \mathsf{plus}(\epsilon, \epsilon, K)\end{array}}{\mathsf{plus}(\epsilon 1, \epsilon 1, R')}\ R' = R''0}{\mathsf{plus}(\epsilon 10, \epsilon 11, R)}\ R = R'1
$$

and tried to solve the first subgoal, we would now essentially guess K and R'' blindly and then filter our choices in the second subgoal. This can lead to non-termination (for example, if the clause $\mathsf{inc}(n1, r0) \circ\!\!- \mathsf{inc}(n, r)$ would be selected continuously) or a lot of backtracking.

In order to avoid such problems, it is important to check that predicates in backward chaining are used consistently with respect to what is known and what is generated. This is called *mode checking*, which is complementary to type checking which have have largely assumed, but not explicated. Actually, problematic situations could already arise in forward chaining, so mode checking is generally important in logic programming and not just in backward chaining.

We begin with dividing the arguments to predicates into *input* arguments, denoted by $(+)$, and *output* arguments, denoted by $(-)$.[2] These notion are fundamentally defined by the following:

1. When a negative atomic proposition P first appears as a goal in the succedent, all its input $(+)$ arguments must be ground, that is, contain no metavariables.

2. When a negative atomic proposition P succeeds as a goal in the succedent, all its output $(-)$ arguments must be ground.

We reason separately about each clause to determine if the property holds for. For example, consider the mode

$$\mathsf{plus}(+, +, -)$$

and the clause

$$\mathsf{plus}(\epsilon, n0, n0).$$

When $\mathsf{plus}(m_1, n_2, r_3)$ appears as a goal, the first and second argument will be ground. We match this goal against the given clause, which succeeds if $\epsilon = m_1$ and $N0 = n_2$ for a new metavariable N. But since n_2 is ground, this will ground N. Then $r_3 = N0$ will be ground as well.

A slightly more complex reasoning applies to the clause

$$\mathsf{plus}(m0, n0, r0) \circ\!\!- \mathsf{plus}(m, n, r).$$

When a well-moded goal matches the clause head $\mathsf{plus}(M0, N0, R0)$, both M and N will be ground. This means that the recursive call $\mathsf{plus}(M, N, R)$ is

[2] This should not be confused with positive or negative propositions, which is an entirely different notion.

well-moded. If it suceeds, R will be ground since it is an output argument. Hence, the output argument $R0$ in the head will be ground as well.

As a last example, consider

$$\mathsf{plus}(m1, n1, r0) \multimap \mathsf{plus}(m, n, k) \otimes \mathsf{inc}(k, r).$$

where $\mathsf{inc}(+, -)$. M and N in the head of the clause will be ground, so the first subgoal $\mathsf{plus}(M, N, K)$ is well-moded and, if it succeeds, will ground K. Therefore the call to $\mathsf{inc}(K, R)$ is also well-moded, and R will be ground if successful. Hence, $R0$, the output argument of the clause head, will also be ground upon success.

If we had reversed the subgoals,

$$\mathsf{plus}(m1, n1, r0) \multimap \mathsf{inc}(k, r) \otimes \mathsf{plus}(m, n, k).$$

it would not have been well-moded, since the input argument K in the call to $\mathsf{inc}(K, R)$ is unknown at the time that subgoal would be invoked. From this example it should be clear that the order in which subgoals are solved is crucial.

References

[Kow88] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Pfe06] Frank Pfenning. Logic programming. `http://www.cs.cmu.edu/~fp/courses/lp/`, December 2006. Lecture notes for a graduate course, Carnegie Mellon University.