15-819K: Logic Programming

Lecture 13

Abstract Logic Programming

Frank Pfenning

October 10, 2006

In this lecture we discuss general criteria to judge whether a logic or a fragment of it can be considered a logic programming language. Taking such criteria as absolute is counterproductive, but they might nonetheless provide some useful insight in the design of richer languages. Three criteria emerge: the first two characterize the relationship to logic in that the language should be sound and (non-deterministically) complete, allowing us to interpret both success and finite failure. The third is operational: we would like to be able to interpret connectives in goals as search instructions, giving them a predictable operational semantics.

13.1 Logic and Logic Programming

For a language to claim to be a logic programming language, the first criterion seems to be soundness with respect to the logical interpretation of the program. I consider this non-negotiable: when the interpreter claims something is true, it should be true. Otherwise, it may be a programming language, but the connection to logic has been lost. Prolog, unfortunately, is unsound in this respect, due to the lack of occurs-check and the incorrect treatment of disequality. We either have to hope or verify that these features of Prolog did not interfere with the correctness of the answer. Other non-logical features such as meta-call, cut, or input and output are borderline with respect to this criterion: since these do not have a logical interpretation, it is difficult to assess soundness of such programs, except by reference to an operational semantics.

The second criterion is non-deterministic completeness. This means that if search fails finitely, no proof can exist. This does not seem quite

as fundamental, since we should be mostly interested in obtaining proofs when they exist, but from an abstract perspective it is certainly desirable. Again, this fails for full Prolog, but is satisfied by pure Prolog even with a depth-first interpreter.

Summary: if we would like to abstractly classify logics or logical fragments as suitable basis for logic programming languages, we would expect at least soundness and non-deterministic completeness so we can correctly interpret success and failure of goals.

13.2 Logic Programming as Goal-Directed Search

Soundness and completeness (in some form) establish a connection to logic, but by themselves they are clearly insufficient from a programming perspective. For example, a general purpose theorem prover for a logic is sound and complete, and yet not by itself useful for programming. We would like to ensure, for example, that our implementation of quicksort really is an implementation of quicksort, and similarly for mergesort. The programmer should be able to predict and control operational behavior well enough to cast algorithms into correct implementations.

As we have seen in the case of Prolog, if the language of goals is sufficiently rich, we can transform all the clauses defining a predicate into the form $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G$ through a process of residuation. Searching for a proof of a goal $p(\mathbf{t})$ then becomes a procedure call, solving instead $G(\mathbf{t}/\mathbf{x})$, which is another goal. In that way, all computational mechanisms are concentrated on the interpretation of goals. Logic programming, as conceived so far, is goal-directed search. Elevating these observations to a design principle we postulate:

> *An* abstract logic programming language *is defined by a subset of the propositions in a logic together with an operational semantics via proof search on that subset. The operational semantics should be sound, non-deterministically complete, and goal-directed.*

But what exactly does goal-directed search mean? If we consider a sequent $\ldots \Vdash G\ \mathit{true}$ where "$\ldots$" collects all hypotheses (linear, unrestricted, and whatever judgments may arise in other logics), then search is *goal-directed* if we can always break down the structure of the goal G first before considering the hypotheses, including the program. This leads us to the definition of asynchronous connectives.

13.3 Asynchronous Connectives

A logial constant or connective is *asynchronous* if its right rule can always be applied eagerly without loosing completeness. For example, $A \,\&\, B$ is asynchronous, because the rule

$$\frac{\Delta \Vdash A \ true \quad \Delta \Vdash B \ true}{\Delta \Vdash A \,\&\, B \ true}\ \&R$$

can always be used for a conjunctive goal $A \,\&\, B$, rather than first applying a left rule to an assumption Δ or the using a clause in the program Γ. This intuitively coincides with our reading of conjunction as a search instructions: to search for a proof of $A \,\&\, B$, first find a proof of A and then of B. This does not talk about possibly having to decompose the program. The property of being asynchronous turns out to be relatively easy to prove. For example, given $\Delta \Vdash A \,\&\, B \ true$, we can prove that $\Delta \Vdash A \ true$ and $\Delta \Vdash B \ true$ by induction on the structure of the given derivation.

On the other hand, the disjunction $A \oplus B$ (which corresponds to $A \vee B$ in ordinary logic programming) is not asynchronous. As a countexample, consider

$$C, C \multimap (B \oplus A) \Vdash A \oplus B.$$

We can use neither the $\oplus R_1$ nor the $\oplus R_2$ rule, because neither

$$C, C \multimap (B \oplus A) \Vdash A$$

nor

$$C, C \multimap (B \oplus A) \Vdash B$$

are provable. Instead we can prove it as follows in the sequent calculus:

$$\dfrac{\dfrac{}{C \Vdash C}\ \text{id} \quad \dfrac{\dfrac{\dfrac{}{B \Vdash B}\ \text{id}}{B \Vdash A \oplus B}\ \oplus R_2 \quad \dfrac{\dfrac{}{A \Vdash A}\ \text{id}}{A \Vdash A \oplus B}\ \oplus R_1}{B \oplus A \Vdash A \oplus B}\ \oplus L}{C, C \multimap (B \oplus A) \Vdash A \oplus B}\ \multimap L.$$

Observe that we took two steps on the left ($\multimap L$ and $\oplus L$) before decomposing the right.

This counterexample shows that we could not decompose $A \oplus B$ eagerly in goal-directed search, unless we are willing to sacrifice completeness. But what, then, would the program $C, C \multimap (B \oplus A)$ mean?

We have already seen that disjunction is useful in Prolog programs, and the same is true for linear logic programs, so this would seem unfortunate. Before we rescue disjunction, let us analyze which connectives are asynchronous.

We postpone the proofs that the asynchronous connectives are indeed asynchronous, and just give counterexamples for those that are not asynchronous.

$$\begin{array}{lcl} C, C \multimap (B \otimes A) & \Vdash & A \otimes B \\ C, C \multimap \mathbf{1} & \Vdash & \mathbf{1} \\ C, C \multimap (B \oplus A) & \Vdash & A \oplus B \\ C, C \multimap \mathbf{0} & \Vdash & \mathbf{0} \\ C, C \multimap\, !A & \Vdash & !A \end{array}$$

In each case we have to apply two left rules first, before any right rule can be applied.

This leaves $A \multimap B$, $A \,\&\, B$, and $\top$ as asynchronous. Atomic propositions have a somewhat special status in that we cannot decompose them, but we have to switch to the left-hand side and focus on an assumption (which models procedure call).

We have not discussed the rules for linear quantifiers (see below) but it turns out that $\forall x.\, A$ is asynchronous, while $\exists x.\, A$ is not asynchronous.

13.4 Asynchronous Connectives vs. Invertibility of Rules

We call an inference rules *invertible* if the premisses are provable whenever the conclusion is. It is tempting to think that a connective is asynchronous if and only if its right rule is invertible. Not so. Please consider the question and see a counterexample at the end of the lecture notes only in desparation.

13.5 Unrestricted Implication

The analysis so far would suggest that the fragment of our logic has the form

$$A \quad ::= \quad P \mid A_1 \multimap A_2 \mid A_1 \,\&\, A_2 \mid \top \mid \forall x.\, A$$

However, this is insufficient, because it is purely linear. Usually we integrate non-linear reasoning into linear logic using the "*of course*" operator '!', but this is not asynchronous. Instead we can use ordinary implication to complete the picture. The proposition $A \supset B$ is true if assuming A as an

unrestricted resource we can prove B.

$$\frac{\Gamma, A\ \mathit{ures}; \Delta \Vdash B\ \mathit{true}}{\Gamma; \Delta \Vdash A \supset B\ \mathit{true}}\ \supset R \qquad \frac{\Gamma; \cdot \Vdash A\ \mathit{true} \quad \Gamma; \Delta \Vdash C\ \mathit{true}}{\Gamma; \Delta, A \supset B\ \mathit{res} \Vdash C\ \mathit{true}}\ \supset L$$

In the left rule for implication, we cannot use any linear resources to prove A, because A may be used in an unrestricted way when proving B. We would need those resources potentially many times in the proof of B, violating linearity of the overall system.

The implication $A \supset B$ is equivalent to $(!A) \multimap B$ (see Exercise 13.1), but they have different proof theoretic properties since $!A$ is not asynchronous.

13.6 Focusing and Synchronous Connectives

If all goal connectives are asynchronous, then we will hit eventually hit an atomic predicate without even looking at the program. What happens then? Recall from Lecture 8 that we now *focus* on a particular program clause and decompose this in a focusing phase. In the setting here this just means that the left rules are applied in sequence to the proposition in focus until an atomic formula is reached, and this formula then must match the conclusion.

In general, we call a connective *synchronous* if we can continue to focus on its components when it is in focus without losing completeness. Note that in logic programming we focus on assumptions (that is, program clauses or part of the state), so the status of a connective as synchronous or asynchronous has be considered separately depending on whether it occurs as a goal (on the right-hand side) or as an assumption (on the left-hand side). A remarkable fact is that all the connectives that were asynchronous as goals are synchronous as assumptions.

We now write $\Delta; A \ll P$ for a focus on A where we just wrote $\Delta; A \vdash P$ earlier, to avoid potential confusion with other hypothetical judgment forms.

Unlike the decomposition of asynchronous connectives (which is completely mechanical), the decomposition of propositions in focus in the synchronous phase of search involves choices. For example the pair of rules

$$\frac{\Delta; A_1\ \mathit{res} \ll P\ \mathit{true}}{\Delta; A_1 \wedge A_2\ \mathit{res} \ll P\ \mathit{true}}\ \&L_1 \qquad \frac{\Delta; A_2\ \mathit{res} \ll P\ \mathit{true}}{\Delta; A_1 \wedge A_2\ \mathit{res} \ll P\ \mathit{true}}\ \&L_2$$

requires a choice between A_1 and A_2 in the focusing phase of search, and similarly for other connectives.

To restate the focusing property again: it allows us to continue to make choices on the proposition in focus, without reconsidering other assumptions, until we reach an atomic proposition. If that matches the conclusion that branch of the proof succeeds, otherwise it fails.

In the next lecture we will get a hint on how to prove this, although we will not do this in full detail.

We close this section by giving the focusing rules for the asynchronous fragment of linear logic, which is at the heart of our logic programming language. The rules can be found in Figure 1. Unfortunately our motivating example from the earlier lecture does not fall into this fragment. For example, we used both simultaneous conjunction $A \otimes B$ and disjunction $A \oplus B$ as goals which are so far prohibit. We will resurrect them via residuation, not because they truly add expressive power, but they are convenient both for program expression and for compilation.

13.7 Historical Notes

The notion that a logic programming language should be characterized as a fragment of logic with complete goal-directed search originated with Miller, Nadathur, and Scedrov [11] who explored logic programming based on higher-order logic. A revised and expanded version appeared a couple of years later [10]. These papers introduced the term *uniform proofs* for those proofs that work asynchronously on the goal until it is atomic.

Some time later this was generalized by Andreoli and Pareschi who first recognized the potential of linear logic for logic programming [3]. They used a fragment of *classical* linear logic (rather than the intuitionistic linear logic we use here), which does not have a distinguished notion of goal. The language LO was therefore more suited to concurrent and object-oriented programming [1, 5, 4].

Andreoli also generalized the earlier notion of uniform proofs to *focusing proofs* [2], capturing now both the asynchronous as well as the synchronous behavior of connectives in a proof-theoretic manner. This seminal work subsequently had many important applications in logic, concurrency, and functional programming, and not just in logic programming.

The thread of research on intuitionistic, goal-directed logic programming resumed with the work by Hodas and Miller [8, 9] who proposed essentially what we presented in this lecture, with some additional extralogical features borrowed from Prolog. In honor of its central new connective the language was called Lolli. These ideas were later picked up in the design of a linear logical framework [6, 7] which augments Lolli with a

richer quantification and explicit proof terms.

13.8 Exercises

Exercise 13.1 *Prove that $A \supset B$ is equivalent to $(!A) \multimap B$ in the sense that each entails the other, that is, $A \supset B \Vdash (!A) \multimap B$ and vice versa.*

Exercise 13.2 *At one point we defined pure Prolog with the connectives $A \wedge B$, $\top$, $A \vee B$, $\bot$, $A \supset B$, and $\forall x.\, A$, suitably restricted into legal goals and programs. Show how to translate such programs and goals into linear logic so that focusing proofs for (non-linear) logic are mapped isomorphically to focusing proofs in linear logic, and prove that your translation is correct in that sense.*

13.9 Answer

Consider the right rule for $!A$.

$$\frac{\Gamma; \cdot \Vdash A\ true}{\Gamma; \cdot \Vdash !A\ true}\ !R$$

This rule is invertible: whenever the conclusion is provable, then so is the premiss. However, $!A$ is not asynchronous (see the counterexample in this lecture). If we had formulated the rule as

$$\frac{\Delta = (\cdot) \quad \Gamma; \cdot \Vdash A\ true}{\Gamma; \Delta \Vdash !A\ true}\ !R$$

we would have recognized it as not being invertible, because Δ is not necessarily empty.

13.10 References

[1] Jean-Marc Andreoli. *Proposal for a Synthesis of Logic and Object-Oriented Programming Paradigms*. PhD thesis, University of Paris VI, 1990.

[2] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[3] Jean-Marc Andreoli and Remo Pareschi. LO and behold! Concurrent structured processes. In *Proceedings of OOPSLA'90*, pages 44–56, Ottawa, Canada, October 1990. Published as ACM SIGPLAN Notices, vol.25, no.10.

[4] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[5] Jean-Marc Andreoli and Remo Pareschi. Logic programming with sequent systems: A linear logic approach. In P. Schröder-Heister, editor, *Proceedings of Workshop to Extensions of Logic Programming, Tübingen, 1989*, pages 1–30. Springer-Verlag LNAI 475, 1991.

[6] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[7] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002.

[8] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS'91)*, pages 32–42, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[9] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[10] 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.

[11] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In David Gries, editor, *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

Judgmental Rules

$$\frac{}{P\ res \ll P\ true}\ \text{id}$$

$$\frac{A\ ures \in \Gamma \quad \Gamma; \Delta; A\ res \ll P\ true}{\Gamma; \Delta \Vdash P\ true}\ \text{copy} \qquad \frac{\Delta; A\ res \ll P\ true}{\Delta, A\ res \Vdash P\ true}\ \text{focus}$$

Multiplicative Connective

$$\frac{\Delta, A\ res \Vdash B\ true}{\Delta \Vdash A \multimap B\ true} \multimap R \qquad \frac{\Delta_A \Vdash A\ true \quad \Delta_B; B\ res \ll P\ true}{\Delta_A, \Delta_B; A \multimap B\ res \ll P\ true} \multimap L$$

Additive Connectives

$$\frac{\Delta \Vdash A\ true \quad \Delta \Vdash B\ true}{\Delta \Vdash A \,\&\, B\ true}\ \&R \qquad \frac{\Delta; A\ res \ll P\ true}{\Delta; A \,\&\, B\ res \ll P\ true}\ \&L_1 \qquad \frac{\Delta; B\ res \ll P\ true}{\Delta; A \,\&\, B\ res \ll P\ true}\ \&L_2$$

$$\frac{}{\Delta \Vdash \top\ true}\ \top R \qquad \text{no } \top L \text{ rule}$$

Exponential Connective

$$\frac{(\Gamma, A\ ures); \Delta \Vdash B\ true}{\Gamma; \Delta \Vdash A \supset B\ true} \supset R \qquad \frac{\Gamma; \cdot \Vdash A\ true \quad \Gamma; B\ res \ll P\ true}{\Gamma; \Delta; A \supset B\ res \ll P\ true} \supset L$$

Quantifier

$$\frac{\Delta \Vdash A \quad x \notin \mathrm{FV}(\Gamma; \Delta)}{\Delta \Vdash \forall x.\, A}\ \forall R \qquad \frac{\Delta; A(t/x)\ res \ll P\ true}{\Delta; \forall x.\, A\ res \ll P\ true}\ \forall L$$

Figure 1: Focused Intuitionistic Linear Logic; Asynchronous Fragment