15-819K: Logic Programming

Lecture 19

Verifying Progress

Frank Pfenning

November 2, 2006

In this lecture we discuss another logic program analysis, namely verifying the progress property. Progress guarantees that a predicate can never fail for arbitrary values of its input arguments. Together with termination this guarantees that a predicate is total in its given input arguments. As sketched in the previous lecture, this an important piece in the general technique of verifying properties of logic programs by reasoning about proof terms.

19.1 The Progress Property

Progress in general just says that during the execution of a program we have either finished computation with a value, or we can make a further step. In particular, computation can never "get stuck". In logic programming this translates to saying the computation can never fail. This requires an understanding of the intended input and output arguments of a predicate, as well as the domain on which it is to be applied.

Returning to the well-worn example of addition,

$$\begin{array}{l}\mathsf{plus}(\mathsf{z}, N, N).\\ \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P).\end{array}$$

the plus predicate is total in the first two arguments, assuming they are natural numbers. It is not total in the first and third argument, because a query such as $\mathsf{plus}(\mathsf{s}(\mathsf{z}), N, \mathsf{z})$ will fail. Totality decomposes into two subquestions, namely progress and termination, since we always assume the program is well-typed and well-moded. Termination is easy to see here because the

first argument decreases strictly in each recursive call. Progress is also easy to see because the first argument must be a natural number, and therefore be either of the form z or $\mathsf{s}(m)$ for some m, and the second argument can be anything because both clause heads have a variable in that position.

Even though the principal application of progress is probably the verification of proof of metatheorems presented in relational form, progress can also be used to check that some given predicates are total functions (although ruling out multi-valued functions requires another step). This may provide the programmer with additional confidence that no cases in the definition of a logic program were missed.

19.2 The Right Semantic Starting Point

As repeatedly emphasized, finding the right semantic starting point for an analysis is the key to obtaining a simple, predictable system and the easiest proof of correctness. For progress, the residuated form of the program is somewhat difficult to deal with. Consider the simple form of $\mathsf{plus}(+,+,-)$ above (easily seen to satisfy progress) and the residuated form

$$\begin{aligned}\mathsf{plus}(x_1, x_2, x_3) \leftarrow\ & (\exists N.\, x_1 \doteq \mathsf{z} \wedge x_2 \doteq N \wedge x_2 \doteq N) \\ \vee\ & (\exists M.\, \exists N.\, \exists P.\, x_1 \doteq \mathsf{s}(M) \wedge x_2 \doteq N \wedge x_3 \doteq \mathsf{s}(P) \wedge \mathsf{plus}(M, N, P))\end{aligned}$$

on which it is more difficult to discern the same property. Moreover, failure plays no role in the progress property because, in fact, it is never permitted to occur, so the semantics should not need to carry a failure continuation.

Hence we return to a fairly early semantics, in which the subgoal stack is explicit, but not the failure continuation. On the other hand, the substitution for the variables is crucial, so we make that explicit. Recall that there is a fixed program Γ with a set of closed clauses.

$$\frac{\tau \vdash G_1 \mathbin{/} G_2 \wedge S}{\tau \vdash G_1 \wedge G_2 \mathbin{/} S} \qquad \frac{\tau \vdash G_2 \mathbin{/} S}{\tau \vdash \top \mathbin{/} G_2 \wedge S} \qquad \frac{}{\tau \vdash \top \mathbin{/} \top}$$

$$\frac{(\forall \mathbf{x}.\, P' \leftarrow G) \in \Gamma \quad P'\rho \doteq P\tau \mid \theta \quad \tau\theta, \rho\theta \vdash G \mathbin{/} S}{\tau \vdash P \mathbin{/} S}$$

In the last rule, ρ is a substitution renaming $\mathbf{x}$ to a new set of logic variables $\mathbf{X}$, that is, $\mathrm{dom}(\rho) = \mathbf{x}$, $\mathrm{cod}(\rho) \cap \mathrm{cod}(\tau) = \emptyset$. We also assume that the variables $\mathbf{x}$ have been renamed so that $\mathbf{x} \cap \mathrm{dom}(\tau) = \emptyset$.

From this semantics it is easily seen that progress is a question regarding atomic goals, because the cases for conjunction and truth always apply.

19.3 Input and Output Coverage

Focusing in, we rewrite the rules for predicate calls assuming the predicate $p(\mathbf{t}, \mathbf{s})$ has a mode declaration which divides the arguments into input arguments $\mathbf{t}$, which come first, and output arguments $\mathbf{s}$, which come second.

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G) \in \Gamma \quad (\mathbf{t}'\rho, \mathbf{s}'\rho) \doteq (\mathbf{t}\tau, \mathbf{s}\tau) \mid \theta \quad \tau\theta, \rho\theta \vdash G \;/\; S}{\tau \vdash p(\mathbf{t}, \mathbf{s}) \;/\; S}$$

We have ensured progress if such a clause and unifier θ always exist.

Breaking it down a bit further, we see we must have

> *There exists a θ such that (1) $\mathbf{t}'\rho\theta = \mathbf{t}\tau\theta$, and (2) $\mathbf{s}'\rho\theta = \mathbf{s}\tau\theta$ where $\mathbf{t}'$ are the input terms in the clause head, $\mathbf{s}'$ are the output terms in the clause head, $\mathbf{t}$ are the input arguments to the predicate call, and $\mathbf{s}$ are the output arguments in the predicate call.*

We refer to part (1) as *input coverage* and part (2) as *output coverage*. In the problem analysis above a single substitution θ is required, but we will approximate this by two separate checks. In the next two sections we will describe the analysis for the two parts of coverage checking. We preview them here briefly.

For input coverage we need to recall the assumption that predicates are well-moded. This means that the input arguments in the call, $\mathbf{t}\tau\theta$ will be ground. Hence input coverage is satisfied if for any sequence $\mathbf{t}$ of ground terms of the right types, there exists a clause head such that its input arguments $\mathbf{t}'$ can be instantiated to $\mathbf{t}$.

Output coverage is trickier. The problem is that mode analysis does not tell us anything about the output argument $\mathbf{s}\tau$ of the call $p(\mathbf{t}\tau, \mathbf{s}\tau)$. What we know if that if p succeeds with substitution $\tau\theta'$, then $\mathbf{s}\tau\theta'$ will be ground, but this does not help. From examples, like plus above, we can observe that output coverage is satisfied because the output argument of the call (in the second clause for plus it is P) is a variable, and will remain a variable until the call is made. This means we have to sharpen mode checking to verify that some variables remain free, which we tackle below.

19.4 Input Coverage

Given a program Γ and a predicate $p : (\boldsymbol{\tau}, \boldsymbol{\sigma}) \rightarrow o$ with input arguments of type $\boldsymbol{\tau}$. We say that p satisfies input coverage in Γ if for any sequence of ground terms $\mathbf{t} : \boldsymbol{\tau}$ there exists a clause $\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G$ and a substitution $\theta : (\mathbf{x}{:}\boldsymbol{\tau}')$ such that $\mathbf{t}'\theta = \mathbf{t}$.

For the description of the algorithm, we will need a slightly more general form. We write $\Delta \vdash \Gamma_p \gg \mathbf{t}$ (read: Γ_p immediately covers $\mathbf{t}$) if there exists a clause $\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G$ in Γ_p and a substitution $\Delta \vdash \theta : (\mathbf{x} : \boldsymbol{\tau}')$ such that $\mathbf{t}'\theta = \mathbf{t}$. We write $\Delta \vdash \Gamma_p > \mathbf{t}$ (read: Γ_p covers $\mathbf{t}$) if for every ground instance $\mathbf{t}\sigma$ with $\sigma : \Delta$ there exists a clause $\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}) \leftarrow G$ in Γ_p and a subsitution $\Delta \vdash \theta : (\mathbf{x}{:}\boldsymbol{\tau}')$ such that $\mathbf{t}'\theta = \mathbf{t}\sigma$. Clearly, immediate coverage implies coverage, but not vice versa.

We reconsider the plus predicate, with the first two arguments considered as inputs.

$$\begin{array}{l}\mathsf{plus}(\mathsf{z}, N, N).\\ \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P).\end{array}$$

By the preceding remark, in order to show that input coverage holds, it is sufficient to show that

$$x_1{:}\mathsf{nat}, x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (x_1, x_2).$$

Clearly, immediate coverage does not hold, because x_1 is not an instance of either z or $\mathsf{s}(M)$. On the other hand, x_2 is an instance of N.

At this point we need to exploit the assumption $x_1{:}\mathsf{nat}$ by applying an appropriate left rule. This is acceptable because we move from the usual open world assumption (any predicate and type is inherently open-ended) to the closed world assumption (all predicates and types are given completely by their definition). The closed world assumption is necessary because progress (and coverage) can only be establish with respect to a fixed set of types and clauses and could immediately be violated by new declarations (e.g., the additional declaration $\omega : \mathsf{nat}$ causes input coverage for plus to fail).

To see what the left rules would look like, we can take a short detour through type predicates. The declarations

$$\begin{array}{lcl}\mathsf{z} & : & \mathsf{nat}.\\ \mathsf{s} & : & \mathsf{nat} \rightarrow \mathsf{nat}.\end{array}$$

correspond to

$$\begin{array}{l}\mathsf{nat}(\mathsf{z}).\\ \mathsf{nat}(\mathsf{s}(N)) \leftarrow \mathsf{nat}(N).\end{array}$$

The iff-completion yields

$$\mathsf{nat}(N) \quad \leftrightarrow \quad N \doteq \mathsf{z} \vee \exists N'.\, N \doteq \mathsf{s}(N') \wedge \mathsf{nat}(N').$$

The left rule for $\mathsf{nat}(x)$ (which is not the most general case, but sufficient for our purposes) for an arbitrary judgment J on the right-hand side can then be derived as

$$\dfrac{\dfrac{\vdash J(\mathsf{z}/x)}{x \doteq \mathsf{z} \vdash J} \quad \dfrac{\dfrac{\mathsf{nat}(x') \vdash J(\mathsf{s}(x')/x)}{x \doteq s(x') \wedge \mathsf{nat}(x') \vdash J}}{\exists N'.\, x \doteq \mathsf{s}(N') \wedge \mathsf{nat}(N') \vdash J}}{x \doteq \mathsf{z} \vee \exists N'.\, x \doteq \mathsf{s}(N') \wedge \mathsf{nat}(N') \vdash J}}{\mathsf{nat}(x) \vdash J}$$

or, in summary:

$$\dfrac{\vdash J(\mathsf{z}/x) \quad \mathsf{nat}(x') \vdash J(\mathsf{s}(x')/x)}{\mathsf{nat}(x) \vdash J}$$

Translated back to types:

$$\dfrac{\Delta \vdash J(\mathsf{z}/x) \quad \Delta, x'{:}\mathsf{nat} \vdash J(\mathsf{s}(x')/x)}{\Delta, x{:}\mathsf{nat} \vdash J}$$

Using this rule, we can now prove our goal:

$$\dfrac{x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (\mathsf{z}, x_2) \quad x_1'{:}\mathsf{nat}, x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (\mathsf{s}(x_1'), x_2)}{x_1{:}\mathsf{nat}, x_2{:}\mathsf{nat} \vdash \Gamma_{\mathsf{plus}} > (x_1, x_2)}$$

Both of the premisses now follow by immediate coverage, using the first clause for the first premiss and the second clause for the second premiss, using the critical rule

$$\dfrac{\Delta \vdash \Gamma_p \gg \mathbf{t}}{\Delta \vdash \Gamma_p > \mathbf{t}}$$

For immediate coverage, there is but one rule.

$$\dfrac{(\forall \mathbf{x}{:}\boldsymbol{\tau}'.\, p(\mathbf{t}', \mathbf{s}') \leftarrow G) \in \Gamma_p \quad \mathbf{t}'\theta = \mathbf{t} \quad \text{for } \Delta \vdash \theta : (\mathbf{x}{:}\boldsymbol{\tau}')}{\Delta \vdash \Gamma_p \gg \mathbf{t}}$$

We do not write out the left rules, but it should be clear how to derive them from the type declarations, at least for simple types. We call this process *splitting* of a variable $x{:}\tau$.

An interesting aspect of the left rules is that they are asynchronous. However, always applying them leads to non-termination, so we have to follow some terminating strategy. This strategy can be summarized informally as follows, given a goal $\Delta \vdash \Gamma_p > \mathbf{t}$.

1. Check if $\Delta \vdash \Gamma_p \gg \mathbf{t}$. If so, succeed.
2. If not, pick a variable $x{:}\tau$ in Δ and apply inversion as sketched above. This yields a collection of subgoals $\Delta_i \vdash \Gamma_p \gg \mathbf{t}_i$. Solve each subgoal.

Picking the right variable to split is crucial for termination. Briefly, we pick a variable x that was involved in a clash $f(\mathbf{t}'') \doteq x$ when attempting immediate coverage, where $f(\mathbf{t}'')$ is a subterm of $\mathbf{t}'$. Now one of the possibilities for x will have f as it top-level function symbol, reducing the clash the next time around. Thus the splitting process is bounded by the total size of the input terms in Γ_p. See the reference below for further discussion and proof of this fact.

19.5 Output Coverage

Output coverage is to ensure that for every goal $p(\mathbf{t}, \mathbf{s})$ encountered while executing a program, the output positions $\mathbf{s}'$ of the relevant clause head $p(\mathbf{t}', \mathbf{s}')$ are an instance of $\mathbf{s}$ (if $\mathbf{t}$ and $\mathbf{t}'$ unify). The problem is that ordinary mode checking does not tell us anything about $\mathbf{s}$: we do not know whether it will be ground or partially ground or consist of all free variables. However, if we knew that $\mathbf{s}$ consisted of pairwise distinct free variables when the goal $p(\mathbf{t}, \mathbf{s})$ arose, then output coverage would be satisfied since the variables in $\mathbf{s}$ cannot occur in a clause head and therefore the unification process must succeed.

So we can guarantee output coverage with a sharpened mode-checking process where an output arguments must be distinct free variables when a predicate is invoked. Moreover, they must become ground by the time the predicate suceeds. This is actually very easy: just change the abstract domain of the mode analysis from $\mathsf{u} > \mathsf{g}$ (unknown and ground) to $\mathsf{f} > \mathsf{g}$ (free and ground). If we also have bidirectional arguments in addition to input and output we three abstract values with $\mathsf{f} > \mathsf{u} > \mathsf{g}$. The remainder of the development is just as in a previous lecture (see Exercise 19.1). The only slightly tricky aspect is that the output arguments must be *distinct* free variables, otherwise the individual substitutions may not compose to one for all output arguments simultaneously.

Returning to our example,

$$\begin{array}{l}\mathsf{plus}(\mathsf{z}, N, N).\\ \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \leftarrow \mathsf{plus}(M, N, P).\end{array}$$

the only output argument is P, which is indeed a free variable when that subgoal is executed.

We show a couple of cases for failure of output coverage, to illustrate some points. Assume we have already checked progress for plus. The program

$$\mathsf{test} \leftarrow \mathsf{plus}(\mathsf{z}, \mathsf{z}, \mathsf{s}(P)).$$

trivially satisfies input coverage, but yet fails (and hence cannot satisfy progress). This is because the output argument in the only call is $\mathsf{s}(P)$, which is not a free variable. This will be noted by the sharpened mode checker.

Similarly, the program

$$\begin{array}{l}\mathsf{test} \leftarrow \\ \quad \mathsf{plus}(\mathsf{z}, \mathsf{s}(\mathsf{z}), P), \\ \quad \mathsf{plus}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{z}), P).\end{array}$$

trivially satisfies input coverage but it does not pass the sharpened mode checker because the second occurrence of P will be ground (from the first call) rather than free when the second subgoal is executed. And, indeed, plus will fail and hence cannot satisfy progress.

Finally, a the predicate $\mathsf{nexttwo}(+, -, -)$

$$\mathsf{nexttwo}(N, \mathsf{s}(N), \mathsf{s}(\mathsf{s}(N))).$$

satisfies progress, but

$$\mathsf{test} \leftarrow \mathsf{nexttwo}(\mathsf{s}(\mathsf{z}), P, P).$$

does not, because the two occurrences of P would have to be $\mathsf{s}(\mathsf{s}(\mathsf{z}))$ and $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))$ simultaneously.

19.6 Historical Notes

Progress and coverage do not appear to have received much attention in the logic programming literature, possibly because they requires types to be interesting, and their main application lies in verifying proofs of meta-theorems which is a recent development. An algorithm for coverage in the richer setting with dependent types is given by Schürmann and myself [1], which also contains some pointers to earlier literature in functional programming.

19.7 Exercises

Exercise 19.1 *Write out the rules for a sharpened mode checker with only input and output arguments where output arguments must be distinct free variables when a predicate is invoked.*

19.8 References

[1] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.