15-819K: Logic Programming

Lecture 8

# Completion

Frank Pfenning

September 21, 2006

In this lecture we extend the ground backtracking semantics to permit free variables. This requires a stronger normal form for programs. After introducing this normal form related to the so-call *iff-completion* of a program, we give the semantics at which point we have a complete specification of the pure Prolog search behavior including unification, subgoal selection and backtracking. At this point we return to the logical meaning of Prolog programs and derive the iff-completion of a program via a process called *residuation*.

## 8.1 Existential Quantification

First we return to the backtracking semantics with the intent of adding free variables and unification to make it a fully specified semantics for pure Prolog.

The first problem is presented by the rules for atomic goals. In a ground calculus, the rule

$$\frac{\forall \mathbf{x}.\, P' \leftarrow B' \in \Gamma \qquad \begin{array}{l}\mathrm{dom}(\tau) = \mathbf{x} \\ \mathrm{cod}(\tau) = \emptyset \\ P'\tau = P\end{array} \qquad B'\tau \;/\; S \;/\; F}{P \;/\; S \;/\; F}$$

is correct only if we stipulate that for every ground atomic goal $P$ there is *exactly one* program clause for which the rule above can be applied. Otherwise, not all failure or choice points would be explicit in the semantics.

Now we need the property that for every atomic goal $P$ (potentially containing free variables) there is *exactly one* program clause that applies. Because goals can have the form $p(X_1, \ldots, X_n)$ for variables $X_1, \ldots, X_n$, this means that for every predicate $p$ there should be only one clause in the program. Moreover, the head of this clause must unify with *every* permissible goal. At first this may seem far-fetched, but since our extended language includes equality, we can actually achieve this by transforming the program so that all clause heads have the form $p(X_1, \ldots, X_n)$ for distinct variables $X_1, \ldots, X_n$.

As an example, consider the `member` predicate in the form appropriate for the ground semantics.

```
member(X, []) :- fail.
member(X, [Y|Ys]) :- X = Y ; member(X, Ys).
```

We can transform this further by factoring the two cases as

```
member(X, Ys) :-
        (Ys = [], fail) ;
        (Ys = [Y|Ys1], (X = Y ; member(X, Ys1))).
```

This can be simplified, because the first disjunct will always fail.

```
member(X, Ys) :- Ys = [Y|Ys1], (X = Y ; member(X, Ys1)).
```

Writing such a program would be considered poor style, since the very first one is much easier to read. However, as an internal representation it turns out to be convenient.

We take one more step which, unfortunately, does not have a simple rendering in all implementations of Prolog. We can simplify the treatment of atomic goals further if the only free variables in a clause are the ones appearing in the head. In the `member` example above, this is not the case, because `Y` and `Ys1` occur in the body, but not the head. If we had existential quantification $\exists x.\, A$, we could overcome this. In logical form, the rule would be the following.

$$\frac{\exists y.\, \exists ys_1.\, Ys \doteq [y|ys_1] \wedge (X \doteq y \vee \mathsf{member}(X, ys_1))\ \mathit{true}}{\mathsf{member}(X, Ys)\ \mathit{true}}$$

In some implementations of Prolog, existential quantification is available with the syntax `X^A` for $\exists x.\, A$.[1] Then the program above would read

[1] You should beware, however, that some implementations of this are unsound in that the variable `X` is visible outside its scope.

```
member(X, Ys) :-
        Y^Ys1^(Ys = [Y|Ys1], (X = Y ; member(X, Ys1))).
```

On the logical side, This requires a new form of proposition, $\exists x.\, A$, where $x$ is a *bound variable* with scope $A$. We assume that we can always rename bound variables. For example, we consider $\exists x.\, \exists y.\, p(x, x, y)$ and $\exists y.\, \exists z.\, p(y, y, z)$ to be identical. We also assume that bound variables (written as lower-case identifiers) and free variables (written as upper-case identifiers) are distinct syntactic classes, so no clash between them can arise.

Strictly speaking we should differentiate between substitutions for variables $x$ that may be bound, and for logic variables $X$ (also called meta-variables). At this point, the necessary machinery for this distinction would yield little benefit, so we use the same notations and postulate the same properties for both.

Existential quantification is now defined by

$$\frac{A(t/x)\ true}{\exists x.\, A\ true}\ \exists I$$

where the the substitution term $t$ is understood to have no free variables since logical deduction is ground deduction.

When carrying out a substitution $\theta$, we must take care when encountering a quantifier $\exists x.\, A$. If $x$ is in the domain of $\theta$, we should first rename it to avoid possible confusion between the bound $x$ and the $x$ that $\theta$ substitutes for. The second condition is that $x$ does not occur in the co-domain of $\theta$. This condition is actually vacuous here ($t$ is closed), but required in more general settings.

$$(\exists x.\, A)\theta \quad = \quad \exists x.\, (A\theta) \quad \text{provided } x \notin \mathrm{dom}(\theta) \cup \mathrm{cod}(\theta)$$

Recall the convention that bound variables can always be silently renamed, so we can always satisfy the side condition no matter what $\theta$ is.

The search semantics for existential quantification comes in two flavors: in the ground version we guess the correct term $t$, in the free variable version we substitute a fresh logic variable.

$$\frac{A(t/x)\ /\ S}{\exists x.\, A\ /\ S} \qquad \frac{A(X/x)\ /\ S \mid \theta \quad X \notin \mathrm{FV}(\exists x.\, A\ /\ S)}{\exists x.\, A\ /\ S \mid \theta}$$

Since $X$ does not occur in $\exists x.\, A\ /\ S$, we could safely restrict $\theta$ in the conclusion to remove any substitution term for $X$.

## 8.2 Backtracking Semantics with Free Variables

Now we assume the program is in a normal form where for each atomic predicate $p$ of arity $n$ there is exactly one clause

$$\forall x_1 \ldots \forall x_n.\, p(x_1, \ldots x_n) \leftarrow B'$$

where the $\mathrm{FV}(B') \subseteq \{x_1, \ldots, x_n\}$.

We will not endeavor to return the answer substitution from the free variable judgment, but just describe the computation to either success or failure. The extension to compute an answer substitution requires some thought, but does not add any essentially new elements (see Exercise 8.1). Therefore, the we write $A \;/\; S \;/\; F$ where $A$, $S$, and $F$ may have free variables. The intended interpretation is that if $A \;/\; S \;/\; F$ in the free variable semantics then there exists a grounding substitution $\sigma$ such that $A\sigma \;/\; S\sigma \;/\; F\sigma$ in the ground semantics. This is not very precise, but sufficient for our purposes.

First, the rules for conjunction and truth. They are the same in the free and ground semantics.

$$\frac{A \;/\; B \wedge S \;/\; F}{A \wedge B \;/\; S \;/\; F} \qquad \frac{B \;/\; S \;/\; F}{\top \;/\; B \wedge S \;/\; F} \qquad \frac{}{\top \;/\; \top \;/\; F}$$

Second, the rules for disjunction and falsehood. Again, it plays no role if the goals are interpreted as closed or with free variables.

$$\frac{A \;/\; S \;/\; (B \wedge S) \vee F}{A \vee B \;/\; S \;/\; F} \qquad \frac{B \;/\; S' \;/\; F}{\bot \;/\; S \;/\; (B \wedge S') \vee F} \qquad \begin{array}{c}\textit{fails (no rule)}\\ \bot \;/\; S \;/\; \bot\end{array}$$

Third, the equality rules. Clearly, these involve unification, so substitutions come into play.

$$\frac{t \doteq s \mid \theta \quad \top \;/\; S\theta \;/\; F}{t \doteq s \;/\; S \;/\; F} \qquad \frac{\text{there is no } \theta \text{ with } t \doteq s \mid \theta \quad \bot \;/\; S \;/\; F}{t \doteq s \;/\; S \;/\; F}$$

When unification succeeds, the most general unifier $\theta$ must be applied to the success continuation $S$ which shares variables with $t$ and $s$. But we do not apply the substitution to $F$. Consider a goal of the form $(X \doteq a \wedge p(X)) \vee (X \doteq b \wedge q(X))$ to see that while we try the first disjunction, $X \doteq a$ the instantiations of $X$ should not affect the failure continuation.

The rule for existentials just introduces a globally fresh logic variable.

$$\frac{A(X/x) \;/\; S \;/\; F \quad X \notin \mathrm{FV}(\exists x.\, A \;/\; S \;/\; F)}{\exists x.\, A \;/\; S \;/\; F}$$

Finally the rule for atomic propositions. Because of the normal form for each clause in the program, this rule no longer involves unification or generation of fresh variables. Such considerations have now been relegated to the cases for equality and existential quantification.

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow B') \in \mathcal{P} \quad B'(\mathbf{t}/\mathbf{x}) \;/\; S \;/\; F}{p(\mathbf{t}) \;/\; S \;/\; F}$$

Here we wrote $\mathbf{t}/\mathbf{x}$ as an abbreviation for the substitution $t_1/x_1, \ldots, t_n/x_n$ where $\mathbf{t} = t_1, \ldots, t_n$ and $\mathbf{x} = x_1, \ldots, x_n$.

## 8.3 Connectives as Search Instructions

The new operational semantics, based on the normal form for programs, beautifully isolates various aspects of the operational reading for logic programs. It is therefore very useful as an intermediate form for compilation.

**Procedure Call ($p(\mathbf{t})$).** An atomic goal $p(\mathbf{t})$ now just becomes a procedure call, interpreting a predicate $p$ as a procedure in logic programming. We use a substitution $\mathbf{t}/\mathbf{x}$ for parameter passing if the clause is $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow B'$ and $\mathrm{FV}(B') \subseteq \mathbf{x}$.

**Success ($\top$).** A goal $\top$ simply succeeds, signaling the current subgoal has been solved. Stacked up subgoals are the considered next.

**Conjunctive choice ($A \wedge B$).** A conjunction $A \wedge B$ represents two subgoals that have to be solved. The choice is which one to address first. The rule for conjunction says $A$.

**Failure ($\bot$).** A goal $\bot$ simply fails, signaling the current subgoal fails. We backtrack to previous choice points, exploring alternatives.

**Disjunctive choice ($A \vee B$).** A disjunction $A \vee B$ represents a choice between two possibly path towards a solution. The rules for disjunction say we try $A$ first and later $B$ (if $A$ fails).

**Existential choice ($\exists x.\, A$).** An existential quantification $\exists x.\, A$ represents the choice which term to use for $x$. The rule for existential quantification says to postpone this choice and simply instantiate $x$ with a fresh logic variable to be determined later during search by unification.

**Unification ($t \doteq s$).** An equality $t \doteq s$ represents a call to unification, to determine some existential choices postponed earlier in a least committed way (if a unifier exists) or fail (if no unifier exists).

Let us read the earlier `member` program as a sequence of these instructions.

```
member(X, Ys) :-
        Y^Ys1^(Ys = [Y|Ys1], (X = Y ; member(X, Ys1))).
```

Given a *procedure call*

```
?- member(t, s).
```

we substitute actual arguments for formal parameters, reaching

```
?- Y^Ys1^(s = [Y|Ys1], (t = Y ; member(t, Ys1))).
```

We now *create fresh variables* `Y` and `Ys1` (keeping their names for simplicity), yielding

```
?- (s = [Y|Ys1], (t = Y ; member(t, Ys1))).
```

Now we have *two subgoals* to solve (a conjunction), which means we solve the left side first by *unifying* $s$, the second argument in the call to `member`, with `[Y|Ys1]`. If this fails, we fail and backtrack. If this succeeds, we apply the substitution to the second subgoal.

Let us assume $s = [s_1|s_2]$, so that the substitution will be $s_1/Y, s_2/Y s_1$. Then we have to solve

```
?- t = s1 ; member(t, s2).
```

Now we have a *disjunctive choice*, so we first try to *unify* $t$ with $s_1$, pushing the alternative onto the failure continuation. If unification succeeds, we succeed with the unifying substitution. Note that besides the unifying substitution, we have also changed the failure continuation by pushing a call to `member` onto it. If the unification fails, we try instead the second alternative, calling `member` recursively.

```
?- member(t, s2).
```

I hope this provides some idea how the body of the `member` predicate could be compiled to a sequence of instructions for an abstract machine.

## 8.4 Logical Semantics Revisited

So far, we have carefully defined truth for all the connectives and quantifers except for universal quantification and implication which appeared only in the program. These introduction rules show how to establish that a given proposition is true. For universal quantification and implication we follow a slight different path, because from the logical point of view the program is a set of *assumptions*, not something we are trying to prove. In the language of judgments, we are dealing with a so-called *hypothetical judgment*

$$\Gamma \vdash A \ \mathit{true}$$

where $\Gamma$ represents the program. It consists of a collection of propositions $D_1 \ \mathit{true}, \ldots, D_n \ \mathit{true}$.

In logic programming, occurrences of logical connectives are quite restricted. In fact, as we noted at the beginning, pure Prolog has essentially no connectives, just atomic predicates and inference rules. We have only extended the language in order to accurately describe search behavior in a logical notation. The restrictions are different for what is allowed as a program clause and what is allowed as a goal. So for the remainder of this lecture we will use $G$ to stand for legal goals and $D$ to stand for legal program propositions.

We summarize the previous rules in this slightly generalized form.

$$\frac{\Gamma \vdash G_1 \ \mathit{true} \quad \Gamma \vdash G_2 \ \mathit{true}}{\Gamma \vdash G_1 \wedge G_2 \ \mathit{true}} \wedge I \qquad \frac{}{\Gamma \vdash \top \ \mathit{true}} \top I$$

$$\frac{\Gamma \vdash G_1 \ \mathit{true}}{\Gamma \vdash G_1 \vee G_2 \ \mathit{true}} \vee I_1 \qquad \frac{\Gamma \vdash G_2 \ \mathit{true}}{\Gamma \vdash G_1 \vee G_2 \ \mathit{true}} \vee I_2 \qquad \frac{\textit{no rule}}{\Gamma \vdash \bot \ \mathit{true}}$$

$$\frac{}{\Gamma \vdash t \doteq t \ \mathit{true}} \doteq I \qquad \frac{\Gamma \vdash G(t/x) \ \mathit{true}}{\Gamma \vdash \exists x.\, G \ \mathit{true}} \exists I$$

When the goal is atomic, we have to use an assumption, corresponding to a clause in the program. Choosing a particular assumption and then breaking down its structure as an assumption is called *focusing*. This is a new judgment $\Gamma; D \ \mathit{true} \vdash P \ \mathit{true}$. The use of the semi-colon here ";" is unrelated to its use in Prolog where it denotes disjunction. Here is just isolates a particular assumption $D$. We call this the *focus* rule.

$$\frac{D \in \Gamma \quad \Gamma; D \ \mathit{true} \vdash P \ \mathit{true}}{\Gamma \vdash P \ \mathit{true}} \text{ focus}$$

When considering which rules should define the connectives in $D$ it is important to keep in mind that the rules now define the *use* of an assumption, rather than how to prove its truth. Such rules are called *left rules* because the apply to a proposition to the left of the turnstile symbol '$\vdash$'.

First, if the assumption is an atomic fact, it must match the conclusion. In that case the proof is finished. We call this the *init* rule for *initial sequent*.

$$\frac{}{\Gamma; P\ true \vdash P\ true}\ \text{init}$$

Second, if the assumption is an implication we would have written $P \leftarrow B$ so far. We observe that $B$ will be a subgoal, so we write it as $G$. Further, $P$ does not need to be restricted to be an atom—it can be an arbitrary legal program formula $D$. Finally, we turn around the implication into the more customary form $G \supset D$.

$$\frac{\Gamma; D\ true \vdash P\ true \quad \Gamma \vdash G\ true}{\Gamma; G \supset D\ true \vdash P\ true}\ \supset L$$

Here, $G$ actually appears as a *subgoal* in one premise and $D$ as an assumption in the other, which is a correct given the intuitive meaning of implication to represent clauses.

If we have a universally quantified proposition, we can instantiate it with an arbitrary (closed) term.

$$\frac{\Gamma; D(t/x)\ true \vdash P\ true}{\Gamma; \forall x.\, D\ true \vdash P\ true}\ \forall L$$

It is convenient to also allow conjunction to combine multiple clauses for a given predicate. Then the use of a conjunction reduces to a choice about which conjunct to use, yielding two rules.

$$\frac{\Gamma; D_1\ true \vdash P\ true}{\Gamma; D_1 \wedge D_2\ true \vdash P\ true}\ \wedge L_1 \qquad \frac{\Gamma; D_2\ true \vdash P\ true}{\Gamma; D_1 \wedge D_2\ true \vdash P\ true}\ \wedge L_2$$

We can also allow truth, but there is no rule for it as an assumption

$$\frac{\textit{no rule}}{\Gamma; \top\ true \vdash P\ true}$$

since the assumption that $\top$ is true gives us no information for proving $P$.

We have explicitly *not* defined how to *use* disjunction, falsehood, equality, or existential quantification, or how to *prove* implication or universal quantification. This is because attempting to add such rules would significantly change the nature of logic programming. From the rules, we can read off the restriction to goals and program as conforming to the following grammar.

$$\begin{array}{lrcl} \text{Goals} & G & ::= & P \mid G_1 \wedge G_2 \mid \top \mid G_1 \vee G_2 \mid \bot \mid t \doteq s \mid \exists x.\, G \\ \text{Clauses} & D & ::= & P \mid D_1 \wedge D_2 \mid \top \mid G \supset D \mid \forall x.\, D \end{array}$$

Clauses in this form are equivalent to so-called *Horn clauses*, which is why it is said that Prolog is based on the Horn fragment of first-order logic.

## 8.5 Residuation

A program in the form described above is rather general, but we can transform it into the procedure call form described earlier with a straightforward and elegant algorithm. To begin, for any predicate $p$ we collect all clauses contributing to the definition of $p$ into a single proposition $D_p$, which is the conjunction of the universal closure of the clauses whose head has the form $p(\mathbf{t})$. For example, given the program

```
nat(z).
nat(s(N)) :- nat(N).

plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

we generate a logical rendering in the form of two propositions:

$$\begin{array}{rcl} D_{\mathsf{nat}} & = & \mathsf{nat}(\mathsf{z}) \wedge \forall n.\, \mathsf{nat}(n) \supset \mathsf{nat}(\mathsf{s}(n)), \\ D_{\mathsf{plus}} & = & (\forall n.\, \mathsf{plus}(\mathsf{z}, n, n)) \\ & & \quad \wedge \forall m.\, \forall n.\, \forall p.\, \mathsf{plus}(m, n, p) \supset \mathsf{plus}(\mathsf{s}(m), n, \mathsf{s}(p)) \end{array}$$

The idea now is that instead of playing through the choices for breaking down $D_p$ when searching for a proof of

$$\Gamma; D_p \vdash p(\mathbf{t})$$

we residuate those choices into a goal whose search behavior is equivalent. If we write the residuating judgment as

$$D_p \vdash p(\mathbf{x}) > G$$

then the focus rule would be

$$\frac{D_p \in \Gamma \quad D_p \vdash p(\mathbf{x}) > G_p \quad \Gamma \vdash G_p(\mathbf{t}/\mathbf{x})}{\Gamma \vdash p(\mathbf{t})}$$

Residuation must be done *deterministically* and is not allowed to fail, so that we can view $G_p$ as the compilation of $p(\mathbf{x})$, the parametric form of a call to $p$.

To guide the design of the rules, we will want that if $D_p \vdash p(\mathbf{x}) > G$ then $\Gamma; D_p \vdash p(\mathbf{t})$ iff $\Gamma \vdash G(\mathbf{t}/\mathbf{x})$. Further more, if $D_p$ and $p(\mathbf{x})$ are given, then there exists a unique $G$ such that $D_p \vdash p(\mathbf{x}) > G$.

$$\frac{}{p'(\mathbf{s}) \vdash p(\mathbf{x}) > p'(\mathbf{s}) \doteq p(\mathbf{x})} \qquad \frac{D_1 \vdash p(\mathbf{x}) > G_1 \quad D_2 \vdash p(\mathbf{x}) > G_2}{D_1 \wedge D_2 \vdash p(\mathbf{x}) > G_1 \vee G_2}$$

$$\frac{}{\top \vdash p(\mathbf{x}) > \bot} \qquad \frac{D \vdash p(\mathbf{x}) > G_1}{G \supset D \vdash p(\mathbf{x}) > G_1 \wedge G}$$

$$\frac{D \vdash p(\mathbf{x}) > G \quad y \notin \mathbf{x}}{\forall y.\, D \vdash p(\mathbf{x}) > \exists y.\, G}$$

The side condition in the last rule can always be satisfied by renaming of the bound variable $x$.

First, the soundness of residuation. During the proof we will discover a necessary property of deductions, called a *substitution property*. You may skip this and come back to it once you understands its use in the proof below.

**Lemma 8.1** *If $D \vdash p(\mathbf{x}) > G$ and $y \notin \mathbf{x}$, then $D(s/y) \vdash p(\mathbf{x}) > G(s/y)$ for any closed term $s$. Moreover, if the original derivation is $\mathcal{D}$, the resulting derivation $\mathcal{D}(s/y)$ has exactly the same structure as $\mathcal{D}$.*

**Proof:** By induction on the structure of the derivation for $D \vdash p(\mathbf{x}) > G$. In each case we just apply the induction hypothesis to all premisses and rebuild the same deduction from the results. □

Now we can prove the soundness.

**Theorem 8.2** *If $D \vdash p(\mathbf{x}) > G$ for $\mathbf{x} \cap \mathrm{FV}(D) = \emptyset$ and $\Gamma \vdash G(\mathbf{t}/\mathbf{x})$ for ground $\mathbf{t}$ then $\Gamma; D \vdash p(\mathbf{t})$.*

**Proof:** By induction on the structure of the deduction of the given residuation judgment, applying inversion to the second given deduction in each case. All cases are straightforward, except for the case of quantification which we show.

**Case:** $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ D_1 \vdash p(\mathbf{x}) > G_1 \quad y \notin \mathbf{x}\end{array}}{\forall y.\, D_1 \vdash p(\mathbf{x}) > \exists y.\, G_1}$ where $D = \forall y.\, D_1$ and $G = \exists y.\, G_1$.

| | |
|---|---|
| $\Gamma \vdash (\exists y.\, G_1)(\mathbf{t}/\mathbf{x})$ | Assumption |
| $\Gamma \vdash \exists y.\, G_1(\mathbf{t}/\mathbf{x})$ | Since $y \notin \mathbf{x}$ and $\mathbf{t}$ ground |
| $\Gamma \vdash G_1(\mathbf{t}/\mathbf{x})(s/y)$ for some ground $s$ | By inversion |
| $\Gamma \vdash G_1(s/y)(\mathbf{t}/\mathbf{x})$ | Since $y \notin \mathbf{x}$ and $\mathbf{t}$ and $s$ ground |
| $D_1(s/y) \vdash p(\mathbf{x}) > G_1(s/y)$ | By substitution property for residuation |
| $\Gamma; D_1(s/y) \vdash p(\mathbf{t})$ | By i.h. on $\mathcal{D}_1(s/y)$ |
| $\Gamma; \forall y.\, D_1 \vdash p(\mathbf{t})$ | By rule |

We may apply the induction hypothesis to $\mathcal{D}_1(s/y)$ because $\mathcal{D}_1$ is a subdeduction of $\mathcal{D}$, and $\mathcal{D}_1(s/y)$ has the same structure as $\mathcal{D}_1$.

□

Completeness follows a similar pattern.

**Theorem 8.3** *If $D \vdash p(\mathbf{x}) > G$ with $\mathbf{x} \cap \mathrm{FV}(D) = \emptyset$ and $\Gamma; D \vdash p(\mathbf{t})$ for ground $\mathbf{t}$ then $\Gamma \vdash G(\mathbf{t}/\mathbf{x})$*

**Proof:** By induction on the structure of the given residuation judgment, applying inversion to the second given deduction in each case. In the case for quantification we need to apply the substitution property for residuation, similarly to the case for soundness. □

Finally, termination and uniqueness.

**Theorem 8.4** *If $D$ and $p(\mathbf{x})$ are given with $\mathbf{x} \cap \mathrm{FV}(D) = \emptyset$, then there exists a unique $G$ such that $D \vdash p(\mathbf{x}) > G$.*

**Proof:** By induction on the structure of $D$. There is exactly one rule for each form of $D$, and the propositions are smaller in the premisses. □

## 8.6 Logical Optimization

Residuation provides an easy way to transform the program into the form needed for the backtracking semantics with free variables. For each predicate $p$ we calculate

$$D_p \vdash p(\mathbf{x}) > G_p$$

and then replace $D_p$ by

$$\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G_p.$$

With respect to the focusing semantics, $D_p$ is equivalent to the new formulation (see Exercise 8.4).

We reconsider the earlier example.

$$\begin{array}{lcl} D_{\mathsf{nat}} & = & \mathsf{nat}(\mathsf{z}) \wedge \forall n.\, \mathsf{nat}(n) \supset \mathsf{nat}(\mathsf{s}(n)), \\ D_{\mathsf{plus}} & = & (\forall n.\, \mathsf{plus}(\mathsf{z}, n, n)) \\ & & \quad \wedge \forall m. \forall n. \forall p.\, \mathsf{plus}(m, n, p) \supset \mathsf{plus}(\mathsf{s}(m), n, \mathsf{s}(p)) \end{array}$$

Running our transformation judgment, we find

$$D_{\mathsf{nat}} \vdash \mathsf{nat}(x) > \mathsf{nat}(\mathsf{z}) \doteq \mathsf{nat}(x) \vee \exists n.\, \mathsf{nat}(\mathsf{s}(n)) \doteq \mathsf{nat}(x) \wedge \mathsf{nat}(n)$$

$$\begin{array}{l} D_{\mathsf{plus}} \vdash \mathsf{plus}(x_1, x_2, x_3) > (\exists n.\, \mathsf{plus}(\mathsf{z}, n, n) \doteq \mathsf{plus}(x_1, x_2, x_3)) \vee \\ \quad (\exists m. \exists n. \exists p.\, \mathsf{plus}(\mathsf{s}(m), n, \mathsf{s}(p)) \doteq \mathsf{plus}(x_1, x_2, x_3) \wedge \mathsf{plus}(m, n, p)). \end{array}$$

These compiled forms can now be the basis for further simplification and optimizations. For example,

$$G_{\mathsf{plus}} = (\exists n.\, \mathsf{plus}(\mathsf{z}, n, n) \doteq \mathsf{plus}(x_1, x_2, x_3)) \vee \ldots$$

Given our knowledge of unification, we can simplify this equation to three equations.

$$G'_{\mathsf{plus}} = (\exists n.\, \mathsf{z} = x_1 \wedge n = x_2 \wedge n = x_3) \vee \ldots$$

Since $n$ does not appear in the first conjunct, we can push in the existential quantifier, postponing the creation of an existential variable.

$$G''_{\mathsf{plus}} = (\mathsf{z} = x_1 \wedge \exists n.\, n = x_2 \wedge n = x_3) \vee \ldots$$

The next transformation is a bit trickier, but we can see that there exists an $n$ which is equal to $x_2$ and $x_3$ iff $x_2$ and $x_3$ are equal. Since $n$ is a bound variable occurring nowhere else, we can exploit this observation to elimination $n$ altogether.

$$G'''_{\mathsf{plus}} = (\mathsf{z} = x_1 \wedge x_2 = x_3) \vee \ldots$$

The optimized code will unify the first argument with z and, if this succeeds, unify the second and third arguments.

## 8.7 Iff Completion

We now revisit the normal form $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G_p$. Since this represents the only way of succeeding in a proof of $p(\mathbf{t})$, we can actually turn the implication around, replacing it by an *if and only if* ($\leftrightarrow$)

$$\forall \mathbf{x}.\, p(\mathbf{x}) \leftrightarrow G_p.$$

Of course, this proposition is outside the fragment that is amenable to logic programming search (considering the right-to-left implication), but it has found some application in the notion of definitional reflection, where it is usually written as

$$\forall \mathbf{x}.\, p(\mathbf{x}) \stackrel{\triangle}{=} G_p.$$

This allows us to draw conclusions that the program alone does not permit, specifically about the falsehood of propositions.

We do not have the formal reasoning rules available to us at this point, but given the (slightly optimized) iff-completion of $\mathsf{nat}$,

$$\forall x.\, \mathsf{nat}(x) \leftrightarrow \mathsf{z} \doteq x \vee \exists n.\, \mathsf{s}(n) \doteq x \wedge \mathsf{nat}(n)$$

we would be able to prove, explicitly with formal rules, that $\mathsf{nat}(\mathsf{a})$ is false for a new constant $\mathsf{a}$, because $\mathsf{a}$ is neither equal to $\mathsf{z}$ nor to $\mathsf{s}(n)$ for some $n$.

Unfortunately the expressive power of the completion is still quite limited in that it is much weaker than induction.

## 8.8 Historical Notes

The notion of iff-completion goes back to Clark [2] who investigated notions of negation and their justification in the early years of logic programming.

The use of goal-directed search and focusing to explain logic programming goes back to Miller et al. [3], who tested various extensions of the logic presented here for suitability as the foundation for a logic programming language.

The use of residuation for compilation and optimization has been proposed by Cervesato [1], who also shows that the ideas are quite robust by addressing a much richer logic.

The notion of definitional reflection goes back to Schroeder-Heister [5] who also examined its relationship to completion [4]. More recently, reflection has been employed in a theorem prover derived from logic programming [6] in the Bedwyr system.

## 8.9 Exercises

**Exercise 8.1** *Extend the free variable semantics with backtracking to explicitly return an answer substitution $\theta$. State the soundness and completness of this semantics with respect to the one that does not explicitly calculate the answer substitution. If you feel brave, prove these two theorems.*

**Exercise 8.2** *Prove that the free variable backtracking semantics given in lecture is sound and complete with respect to the ground semantics.*

**Exercise 8.3** *Prove the completeness of residuation.*

**Exercise 8.4** *Given soundness and completeness of residuation, show that if we replace programs $D_p$ by $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G_p$ where $D_p \vdash p(\mathbf{x}) > G_p$ then the focusing semantics is preserved.*

## 8.10 References

[1] Iliano Cervesato. Proof-theoretic foundation of compilation in logic programming languages. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 115–129, Manchester, England, June 1998. MIT Press.

[2] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[3] 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.

[4] Peter Schroeder-Heister. Definitional reflection and the completion. In R. Dyckhoff, editor, *Proceedings of the 4th International Workshop on Extensions of Logic Programming*, pages 333–347. Springer-Verlag LNCS 798, March 1993.

[5] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Proceedings of the 8th Annual Symposium on Logic in computer Science (LICS'93)*, pages 222–232, Montreal, Canada, July 1993. IEEE Computer Society Press.

[6] Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In C.Benzmüller, J.Harrison, and C.Schürmann, editors, *Proceedings of the Workshop on Empirically Successful Automated Reasnoing in Higher-Order Logics (ESHOL'05)*, pages 79–98, Montego Bay, Jamaica, December 2005.