15-819K: Logic Programming

Lecture 24

Metavariables

Frank Pfenning

November 28, 2006

In this lecture we return to the treatment of logic variables. In Prolog and some extensions we have considered, logic variables are global, and equations involving logic variables are solved by unification. However, when universal goals $\forall x.\, A$ are allowed in backward chaining, or existential assumptions $\exists x.\, A$ in forward chaining, new parameters may be introduced into the proof search process. Ordinary unification on logic variables is now unsound, even with the occurs-check. We generalize logic variables to *metavariables*, a terminology borrowed from proof assistants and logical frameworks, and describe unification in this extended setting.

24.1 Parameters

When proving a universally quantified proposition we demand that proof to be *parametric*. In the rule this is enforced by requiring that x be new.

$$\frac{\Gamma; \Delta \Vdash A \quad x \notin \mathrm{FV}(\Gamma, \Delta)}{\Gamma; \Delta \Vdash \forall x.\, A}\ \forall R$$

The condition on x can always be satisfied by renaming the bound variable. Operationally, this means that we introduce a new parameter into the derivation when solving a goal $\forall x.\, A$.

We already exploited the parametricity of the derivation for the admissibility of cut by substituting a term t for x in the subderivation. The admissibility of cut, rewritten here for lax linear logic, has the following form, with J standing for either $C\ true$ or $C\ lax$:

$$\textit{If } \Gamma; \Delta_D \overset{\mathcal{D}}{\Vdash} A \textit{ and } \Gamma; \Delta_E, A \overset{\mathcal{E}}{\Vdash} J \textit{ then } \Gamma; \Delta_E, \Delta_D \overset{\mathcal{F}}{\Vdash} J.$$

Putting aside some issues related to the validity and lax judgments, this is proved by a nested induction, first on the structure of the formula A, then the two derivations $\mathcal{D}$ and $\mathcal{E}$. One of the critical cases for quantification:

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma; \Delta_D \Vdash A_1 \quad x \notin \mathrm{FV}(\Gamma, \Delta_D)\end{array}}{\Gamma; \Delta_D \Vdash \forall x.\, A_1}\, \forall R$ where $A = \forall x.\, A_1$ and

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 \\ \Gamma; \Delta_E, A_1(t/x) \Vdash J\end{array}}{\Gamma; \Delta_E, \forall x.\, A_1 \Vdash J}\, \forall L.$$

$\Gamma; \Delta_D \Vdash A_1(t/x)$	By $\mathcal{D}_1(t/x)$, noting $x \notin \mathrm{FV}(\Gamma, \Delta_D)$
$\Gamma; \Delta_E, \Delta_D \Vdash J$	By ind.hyp. on $A_1(t/x)$, $\mathcal{D}_1(t/x)$, $\mathcal{E}_1$

The induction hypothesis applies in (first-order) lax linear logic because $A_1(t/x)$ may be considered a subformula of $\forall x.\, A_1$ since it contains fewer quantifiers and connectives. The condition $x \notin \mathrm{FV}(\Gamma, \Delta_D)$ is critical to guarantee that $\mathcal{D}_1(t/x)$ is a valid proof of $\Gamma; \Delta_D \Vdash A_1(t/x)$.

Parameters behave quite differently from logic variables in that during unification they may not be instantiated. Indeed, carrying out such a substitution would violate parametricity. But parameters interact with logic variables, which means that simply treating parameters as constants during unifications is unsound.

24.2 Parameter Dependency

We consider two examples, $\forall x.\, \exists y.\, x \doteq y$, which should obviously be true (pick x for y), and $\exists y.\, \forall x.\, x \doteq y$, which should obviously be false (in general, there is not a single y equal to all x).

We consider the proof search behavior in these two examples. First, the successful proof.

$$\dfrac{\dfrac{\dfrac{}{\Vdash x \doteq x}\, \doteq R}{\Vdash \exists y.\, x \doteq y}\, \exists R}{\Vdash \forall x.\, \exists y.\, x \doteq y}\, \forall R$$

With logic variables and unification, this becomes the following, assuming

we do not actually care to return the substitution.

$$\dfrac{\dfrac{\dfrac{x \doteq Y \mid (x/Y)}{\Vdash x \doteq Y}\doteq R}{\Vdash \exists y.\, x \doteq y}\exists R}{\Vdash \forall x.\, \exists y.\, x \doteq y}\forall R$$

Applying the substitution (x/Y) in the derivation leads to the first proof, which is indeed valid.

Second, the unsuccessful proof. In the calculus without logic variables we fail either because in the $\exists R$ step the substitution is capture-avoiding (so we cannot use x/y), or in the $\forall R$ step where we cannot rename x to y.

$$\dfrac{\dfrac{\begin{array}{c}\text{fails}\\ \Vdash x \doteq y\end{array}}{\Vdash \forall x.\, x \doteq y}\forall R}{\Vdash \exists y.\, \forall x.\, x \doteq y}\exists R$$

In the presence of free variables we can apparently succeed:

$$\dfrac{\dfrac{\dfrac{x \doteq Y \mid (x/Y)}{\Vdash x \doteq Y}\doteq R}{\Vdash \forall x.\, x \doteq Y}\forall R}{\Vdash \exists y.\, \forall x.\, x \doteq y}\exists R$$

However, applying the substitution (x/Y) into the derivation does not work, because Y occurs in the scope of a quantifier on x.

In order to prevent the second, erroneous solution, we need to prevent (x/Y) as a valid result of unification.

24.3 Skolemization

At a high level, there are essentially three methods for avoiding the above-mentioned unsoundness. The first, traditionally used in classical logics in a pre-processing phase, is *Skolemization*. In classical logic we usually negate the theorem and then try to derive a contradiction, in which case Skolemization has a natural interpretation. Given an assumption $\forall y.\, \exists x.\, A(x, y)$, for every y there exists an x such that $A(x, y)$ is true. This means that there must be a function f such that $\forall y.\, A(f(y), y)$ because f can simply select the appropriate x for every given y.

I don't know how to explain Skolemization in the direct, positive form except as a syntactic trick. If we replace universal quantifiers by a Skolem function of the existentials in whose scope it lies, then $\exists y.\, \forall x.\, x \doteq y$ is transformed to $\exists y.\, f(y) \doteq y$. Now if we pick an existential variable Y for y, then $f(Y)$ and Y are not unifiable due to the occurs-check.

Unfortunately, Skolemization is suspect for several reasons. In Prolog, there is no occurs-check, so it will not work directly. In logics with higher-order term languages, Skolemization creates a new function symbol f for every universal quantifier, which could be used incorrectly in other places. Finally, in intuitionistic logics, Skolemization can no longer be done in a preprocessing phase, although it can still be employed.

24.4 Raising

Raising is the idea that existential variables should *never* be allowed to depend on parameters. When confronted with an unsolvable problem such as $\exists y.\, \forall x.\, x \doteq y$ this is perfect.

However, when a solution does exist, as in $\forall x.\, \exists y.\, x \doteq y$ we need to somehow permit y to depend on x. We accomplish this by rotating the quantifier outward and turning it into an explicit function variable, as in $\exists y.\, \forall x.\, x \doteq y(x)$. Now y can be instantiated with the identity function $\lambda z.\, z$ to solve the equation. Note that y is does not contain any parameters as specified.

Raising works better than Skolemization, but it does require a term language allowing function variables. While this seems to raise difficult issues regarding unification, we can make the functional feature so weak that unification remains decidable and most general unifiers continue to exist. This restriction to so-called *higher-order patterns* stipulates that function variables be applied only to a list of distinct bound variables. In the example above this is the case: y is applies only to x. We briefly discuss this further in the section on *higher-order abstract syntax* below.

24.5 Contextual Metavariables

A third possibility is to record with every logic variable (that is, metavariable) the parameters it may depend on. We write $\Sigma \vdash X$ if the substitution term for X may depend on all the parameters in Σ. As we introduce parameters into a deduction we collect them in Σ. As we create metavariables, we collect them into another different context Θ, together with their contexts. We write $\Theta; \Sigma; \Gamma; \Delta \Vdash A$. No variable may be declared more than once. The

right rules for existential and universal quantifiers are:

$$\frac{\Theta; \Sigma, x; \Gamma; \Delta \Vdash A}{\Theta; \Sigma; \Gamma; \Delta \Vdash \forall x.\, A}\ \forall R \qquad \frac{\Theta, (\Sigma \vdash X); \Sigma; \Gamma; \Delta \Vdash A(X/x)}{\Theta; \Sigma; \Gamma; \Delta \Vdash \exists x.\, A}\ \exists R$$

By the convention that variables can be declared only once, we now omit the condition and use renaming to achieve freshness of X and x. Unification now also depends on Θ and Σ so we write $\Theta; \Sigma \vdash s \doteq t \mid \theta$.

Let us revisit the two examples above. First, the successful proof.

$$\dfrac{\dfrac{\dfrac{(x \vdash Y); x \vdash x \doteq Y \mid (x/Y)}{(x \vdash Y); x \Vdash x \doteq Y}\ \doteq R}{\cdot; x; \cdot; \cdot \Vdash \exists y.\, x \doteq y}\ \exists R}{\cdot; \cdot; \cdot; \cdot \Vdash \forall x. \exists y.\, x \doteq y}\ \forall R$$

The substitution in the last step is valid because Y is allowed to depend on x due to its declaration $x \vdash Y$.

In the failing example we have

$$\dfrac{\dfrac{\dfrac{\text{fails}}{(\cdot \vdash Y); x \vdash x \doteq Y \mid \_}}{(\cdot \vdash Y); x; \cdot; \cdot \Vdash x \doteq Y}\ \doteq R}{(\cdot \vdash Y); \cdot; \cdot; \cdot \Vdash \forall x.\, x \doteq Y}\ \forall R}{\cdot; \cdot; \cdot; \cdot \Vdash \exists y. \forall x.\, x \doteq y}\ \exists R$$

Now unification in the last step fails because the parameter x on the left-hand side is not allowed to occur in the substitution term for Y.

We call metavariables $\Sigma \vdash X$ *contextual metavariables* because they carry the context in which they were created.

24.6 Unification with Contextual Metavariables

The unification algorithm changes somewhat to account for the presence of parameters. The first idea is relatively straightforward: if $(\Sigma_X \vdash X)$ and we unify $X \doteq t$, then we fail if there is a parameter in t not in Σ_X.

But unification is a bit more subtle. Consider, for example, $(x \vdash X)$ and $(x, y \vdash Y)$ and the problem $X \doteq f(Y)$. In this case the substitution term for X may only depend on x, but not on y. But Y is allowed to depend on y, so just substituting $f(Y)/X$ would be unsound if later Y were instantiated

with y. So we need to restrict Y to depend only on x. In general, for a problem $X \doteq t$, we need to restrict any metavariable in t by intersecting its context with Σ_X.

Unfortunately, this means that unification must return a new Θ' as well as a substitution θ such that every free variable in the codomain of θ is declared in Θ. We write

$$\Theta; \Sigma \vdash t \doteq s \mid (\Theta' \vdash \theta)$$

for this unification judgment, and similarly for term sequences.

$$\frac{\mathbf{t} \doteq \mathbf{s} \mid (\Theta' \vdash \theta)}{\Theta; \Sigma \vdash f(\mathbf{t}) \doteq f(\mathbf{s}) \mid (\Theta' \vdash \theta)} \qquad \frac{x \in \Sigma}{\Theta; \Sigma \vdash x \doteq x \mid (\Theta' \vdash \cdot)}$$

$$\frac{\Theta; \Sigma \vdash t \doteq s \mid (\Theta_1 \vdash \theta_1) \quad \Theta_1; \Sigma \vdash \mathbf{t}\theta_1 \doteq \mathbf{s}\theta_1 \mid (\Theta_2 \vdash \theta_2)}{\Theta; \Sigma \vdash (t, \mathbf{t}) \doteq (s, \mathbf{s}) \mid (\Theta_2 \vdash \theta_1\theta_2)}$$

$$\frac{}{\Theta; \Sigma \vdash (\cdot) \doteq (\cdot) \mid (\Theta \vdash \cdot)}$$

Second, the cases for metavariables. We fold the occurs-check into the restriction operation.

$$\frac{}{\Theta; \Sigma \vdash X \doteq X \mid (\Theta; \cdot)}$$

$$\frac{\Theta \vdash t|_X > \Theta'}{\Theta; \Sigma \vdash X \doteq t \mid (\Theta' \vdash t/X)} \qquad \frac{t = f(\mathbf{t}) \quad \Theta \vdash t|_X > \Theta'}{\Theta; \Sigma \vdash t \doteq X \mid (\Theta' \vdash t/X)}$$

Finally, the restriction operator:

$$\frac{(\Sigma_X \vdash X) \in \Theta \quad x \in \Sigma_X}{\Theta \vdash x|_X > \Theta} \qquad \begin{array}{c}\text{no rule for } \Sigma_X \vdash X, x \notin \Sigma_X \\ \Theta \vdash x|_X > \theta\end{array}$$

$$\frac{\Theta \vdash \mathbf{t}|_X > \Theta'}{\Theta \vdash f(\mathbf{t})|_X > \Theta'} \qquad \frac{}{\Theta \vdash (\cdot)|_X > \Theta} \qquad \frac{\Theta \vdash t|_X > \Theta_1 \quad \Theta_1 \vdash \mathbf{t}|_X > \Theta_2}{\Theta \vdash (t, \mathbf{t})|_X > \Theta_2}$$

$$\frac{X \neq Y; (\Sigma_X \vdash X) \in \Theta}{\Theta, (\Sigma_Y \vdash Y) \vdash Y|_X > \Theta, (\Sigma_Y \cap \Sigma_X \vdash Y)} \qquad \begin{array}{c}\text{no rule for} \\ \Theta \vdash X|_X > \_\end{array}$$

As indicated before, restriction can fail in two ways: in $x|_X$ if $x \notin \Sigma_X$ and when trying $X|_X$. The first we call a parameter dependency failure, the second an occurs-check failure. Overall, we also call restriction an *extended occurs-check*.

24.7 Types

One of the reasons to be so pedantic in the judgments above is the now straightforward generalization to the typed setting. The parameter context Σ contains the type declarations for all the variables, and declaration $\Sigma_X \vdash X : \tau$ contains all the types for the parameters that may occur in the substitution term for X. We can take this quite far to a dependent and polymorphic type theory and metavariables will continue to make sense. We only mention this here; details can be found in the literature cited below.

24.8 Higher-Order Abstract Syntax

It has been my goal in this class to present logic programming as a general paradigm of computation. It is my view that logic programming arises from the study of the structure of proofs (since computation is proof search) and that model-theoretic considerations are secondary. The liberation from the usual concerns about Herbrand models and classical reasoning has opened up a rich variety of new possibilities, including, for example, linear logic programming for stateful and concurrent systems.

At the same time I have been careful to keep my own interests in applications of logic programming in the background, and have drawn examples from a variety of domains such as simple list manipulation, algorithms on graphs, solitaire puzzles, decision procedures, dataflow analysis, etc. In the remainder of this lecture and the next one I will go into some examples of the use of logic programming in a logical framework, where the application domain itself also consists of logics and programming languages.

One of the pervasive notions in this setting is *variable binding*. The names of bound variables should not matter, and we should be able to substitute for them in a way that avoids capture. For example, in a proposition $\exists y.\, \forall x.\, x \doteq y$ we cannot substitute x for y because the binder on x would incorrectly *capture* the substitution term for y. Substitution into a quantified proposition is then subject to some conditions:

$$(\forall x.\, A)(t/y) = \forall x.\, A(t/y) \quad \text{provided } x \neq y \text{ and } x \notin \mathrm{FV}(t).$$

These conditions can always be satisfied by (usually silently) renaming bound variables, here x.

If we want to represent objects with variable binders (such as such as quantified propositions) as terms in a metalanguage, the question arises on how to represent bound variables. By far the most elegant means of accomplishing this is to represent them by corresponding bound variables in the metalanguage. This means, for example, that substitution in the object language is modeled accurately by substitution in the metalanguage without any additional overhead. This is the basic idea behind *higher-order abstract syntax*. A simple grammar decomposes terms into either *abstractions* or *applications*.

$$\begin{array}{lcll} \text{Abstractions} & b & ::= & x.\, b \mid t \\ \text{Applications} & t & ::= & h(b_1, \ldots, b_n) \\ \text{Heads} & h & ::= & x \mid f \end{array}$$

An abstraction $x.\, b$ binds the variable x with scope b. An application is just the term structure from first-order logic we have considered so far, except that the head of the term may be a variable as well as a function symbol, and the arguments are again abstractions.

These kinds of terms with abstractions are often written as λ-terms, using the notation $\lambda x.\, M$. However, here we do not use abstraction to form functions in the sense of functional programming, but simply to indicate variable binding. We therefore prefer to think of $\lambda x.\, M$ as $\lambda(x.\, M)$ and $\forall x.\, A$ as $\forall(x.\, A)$, clearly representing variable binding in each case and thinking of λ and $\forall$ as simple constructors.

We we substitute an abstraction for a variable, we may have to hereditarily perform substitution in order to obtain a term satisfying the above grammar. For example, if we have a term $\mathsf{lam}(x.\, E(x))$ and we substitute $(y.y)/E$ then we could not return $(y.y)\,(x)$ (which is not legal, according to our syntax), but substitute x for y in the body of the abstraction to obtain $\mathsf{lam}(x.\, x)$.

$$\begin{array}{l} (\mathsf{lam}(x.\, E(x)))((y.y)/E) \\ = \mathsf{lam}(x.\, (E(x))((y.y)/E)) \\ = \mathsf{lam}(x.\, y(x/y)) \\ = \mathsf{lam}(x.\, x) \end{array}$$

Since one form of substitution may engender another substitution on embedded terms we call it *hereditary substitution*.

A more detailed analysis of higher-order abstract syntax, hereditary substitution, and the interaction of these notions with typing is beyond the

scope of this course. We will use it in the next lecture in order to specify the operational semantics of a programming language.

24.9 Historical Notes

Traditionally, first-order theorem provers have used Skolemization, either statically (in classical logic), or dynamically (in non-classical logics) [13]. A clear presentation and solution of many issues connected to quantifier dependencies and unification has been given by Miller [5].

The idea of higher-order abstract syntax goes back to Church's type theory [1] in which all variable binding was reduced to λ-abstraction. Martin-Löf's system of arities (mostly unpublished) was a system of simple types including variable binding. Its important role in logical frameworks was identified by several groups in 1986 and 1987, largely independently, and with different applications: theorem proving [8], logic programming [6], and logical frameworks [2].

In programming languages, the idea of mapping bound variables in an object language to bound variables in a metalanguage is due to Huet and Lang [3]. Its use in programming environments was advocated and further analyzed by Elliott and myself [11]. This paper also coined the term "*higher-order abstract syntax*" and is therefore sometimes falsely credited with the invention of concept.

There are many operational issues raised by variable dependencies and higher-order abstract syntax, most immediately unification which is important in all three types of applications (theorem proving, logic programming, and logical frameworks). The key step is Miller's discovery of *higher-order patterns* [4] for which most general unifiers still exist. I generalized this latter to various type theories [9, 10].

The most recent development in this area is *contextual modal type theory* [7] which gives metavariables first-class status within a type theory, rather than consider them a purely operational artifact. A presentation of unification in a slighly simpler version of this type theory can be found in Pientka's thesis [12].

24.10 Exercises

Exercise 24.1 *Show an example that leads to unsoundness if parameter dependency is not respected during unification using only hereditary Harrop formulas, that it, the asynchronous fragment of intuitionistic logic.*

24.11 References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[3] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[4] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[5] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[6] 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.

[7] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. Submitted, September 2005.

[8] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[9] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[10] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[11] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[12] Brigitte Pientka. *Tabled Higher-Order Logic Programming*. PhD thesis, Department of Computer Science, Carnegie Mellon University, December 2003. Available as Technical Report CMU-CS-03-185.

[13] N. Shankar. Proof search in the intuitionistic sequent calculus. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, pages 522–536, Saratoga Springs, New York, June 1992. Springer-Verlag LNCS 607.