15-819K: Logic Programming

Lecture 9

Types

Frank Pfenning

September 26, 2006

In this lecture we introduce types into logic programming, primarily to distinguish meaningful from meaningless terms and propositions, thereby capturing errors early during program development.

9.1 Views on Types

Types are a multi-faceted concept, and also subject of much debate. We will discuss some of the views that have been advanced regarding the role of types in logic programming, and then focus on the one we find most useful and interesting.

Descriptive Types. In logic programming terminology, a type system is called *descriptive* if it captures some aspect of program behavior, where program behavior is defined entirely without reference to types. A common example is to think of a type as an approximation to the success set of the program, that is, the set of terms on which it holds. Reconsider addition on unary numbers:

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

We can see that if `plus(`t_1`,` t_2`,` t_3`)` is true according to this definition, then t_1 must be a natural number, that is, `nat(`t_1`)` according to the definition

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

On the other hand, nothing interesting can be said about the other two arguments, because the first clause as written permits non-sensical propositions such as `plus(z, [], [])` to be true.

The nature of descriptive types means that usually they are used to optimize program behavior rather than as an aid to the programmer for writing correct code.

Prescriptive Types. A type system is called *prescriptive* if it is an integral part of the meaning of programs. In the example above, if we prescribe that `plus` is a relation between three terms of type `nat`, then we suddenly differentiate well-typed expressions (such as `plus(z, s(z), P)` when `P` is a variable of type `nat`) from expressions that are not well-typed and therefore meaningless (such as `plus(z, [], [])`. Among the well-typed expressions we then further distinguish those propositions which are true and those which are false.

Prescriptive types, in a well-designed type system, are immediately useful to the programmer since many accidental mistakes in the program will be captured rather then leading to either failure or success, which may be very difficult to debug. In some ways, the situation in pure Prolog is worse than in other dynamically typed languages such as Lisp, because in the latter language you will often get a run-time error for incorrect programs, while in pure Prolog everything is either true or false. This is somewhat overstates the case, since many built-in predicates have dynamically enforced type restrictions. Nevertheless programming with no static and few dynamic errors becomes more and more difficult as programs grow larger. I recognize that it may be somewhat difficult to fully appreciate the point if you write only small programs as you do in this class (at least so far).

Within the prescriptive type approach, we can make further distinction as to the way types are checked or expressed.

Monadic Propositions as Types. In this approach types are represented as monadic predicates, such as `nat` above. This approach is prevalent in the early logic programming literature, and it is also the prevalent view classical logic. The standard answer to the question why the traditional predicate calculus (also known as classical first-order logic) is untyped is that types can be eliminated in favor of monadic predicates. For example, we can translate $\forall x{:}\mathsf{nat}.\, A$ into $\forall x.\, \mathsf{nat}(x) \supset A$, where x now ranges over arbitrary terms. Similarly, $\exists x{:}\mathsf{nat}.\, A$ becomes $\exists x.\, \mathsf{nat}(x) \wedge A$.

In a later lecture we will see that this view is somewhat difficult to sustain in the presence of higher-order predicates, that is, predicates that take other predicates as arguments. The corresponding *higher-order logic*, also known as Church classical theory of types, therefore has a different concept of type called *simple*. Fortunately, the two ideas are compatible, and it is possible to refine the simple type system using the ideas behind monadic propositions, but we have to postpone this idea to a future lecture.

Simple Types. In this approach types are explicitly declared as new entities, separately from monadic propositions. Constructors for types in the form of constants and function symbols are also separately declared. Often, types are disjoint so that a given term has a unique type. In the example above, we might declare

```
nat : type.
z : nat.
s : nat -> nat.
plus : nat, nat, nat -> o.
```

where `o` is a distinguished type of propositions. The first line declares `nat` to be a new type, the second and third declare `z` and `s` as constructors for terms of type `nat`, and the last line declares `plus` as a predicate relating three natural numbers.

With these declarations, an expression such as `plus(z, [], [])` can be readily seen as ill-typed, since the second and third argument are presumably of type `list` and not `nat`. Moreover, in a clause `plus(z, N, N)` it is clear that `N` is a variable ranging only over terms of type `nat`.

In this lecture we develop a system of simple types. One of the difficulty we encounter is that generic data structures, including even simple lists, are difficult to deal with unless we have a type of all terms, or permit variables types. For example, while lists of natural numbers are easy

```
natlist : type.
[] : natlist.
[_|_] : nat, natlist -> natlist.
```

there is no easy way to declare lists with elements of unknown or arbitrary type. We will address this shortcoming in the next lecture.

9.2 Signatures

Simple types rely on explicit declarations of new types, and of new constructors together with their type. The collection of such declarations is called a *signature*. We assume a special type constant o (omicron) that stands for the type of propositions. We use the letters τ and σ for newly declared types. These types will always be atomic and not include o.[1]

Signature	Σ	$::=$	$\cdot$	empty signature
		$\mid$	$\Sigma, \tau : \mathrm{type}$	type declaration
		$\mid$	$\Sigma, f : \tau_1, \ldots, \tau_n \to \tau$	function symbol declaration
		$\mid$	$\Sigma, p : \tau_1, \ldots, \tau_n \to o$	predicate symbol declaration

As usual, we will abbreviate sequences of types by writing them in boldface, $\boldsymbol{\tau}$ or $\boldsymbol{\sigma}$. Also, when a sequence of types is empty we may abbreviate $c : \cdot \to \tau$ by simply writing $c : \tau$ and similarly for predicates. All types, functions, and predicates declared in a signature must be distinct so that lookup of a symbol is always unique.

Despite the suggestive notation, you should keep in mind that "$\to$" is not a first class constructor of types, so that, for example, $\tau \to \tau$ is *not* a type for now. The only true types we have are atomic type symbols. This is similar to the way we developed logic programming: the only propositions we had were atomic, and the logical connectives only came in later to describe the search behavior of logic programs.

9.3 Typing Propositions and Terms

There are three basic judgments for typing: one for propositions, one for terms, and one for sequences of terms. All three require a context Δ in which the types of the free variables in a proposition or term are recorded.

$$\text{Typing Context} \quad \Delta \quad ::= \quad \cdot \mid \Delta, x{:}\tau$$

We assume all variables in a context are distinct so that the type assigned to a variable is unique. We write $\mathrm{dom}(\Delta)$ for the set of variables declared in a context.

A context Δ represents *assumptions* on the types of variables and is therefore written on the left side of a turnstile symbol '$\vdash$', as we did with

[1] Allowing τ to be o would make the logic higher order, which we would like to avoid for now.

logic programs before.

$$\begin{array}{ll} \Sigma; \Delta \vdash A : o & A \text{ is a valid proposition} \\ \Sigma; \Delta \vdash t : \tau & \text{term } t \text{ has type } \tau \\ \Sigma; \Delta \vdash \mathbf{t} : \boldsymbol{\tau} & \text{sequence } \mathbf{t} \text{ has type sequence } \boldsymbol{\tau} \end{array}$$

Because the signature Σ never changes while type-checking, we omit it from the judgments below and just assume that there is a fixed signature Σ in the background theory.

The rules for propositions are straightforward. As is often the case, these rules, read bottom-up, have an interpretation as an algorithm for type-checking.

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \wedge B : o} \qquad \frac{}{\Delta \vdash \top : o}$$

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \vee B : o} \qquad \frac{}{\Delta \vdash \bot : o}$$

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \supset B : o} \qquad \frac{p : \boldsymbol{\tau} \to o \in \Sigma \quad \Delta \vdash \mathbf{t} : \boldsymbol{\tau}}{\Delta \vdash p(\mathbf{t}) : o}$$

For equality, we demand that the terms we compare have the same type τ, whatever that may be. So rather than saying that, for example, zero is not equal to the empty list, we consider such a question meaningless.

$$\frac{\Delta \vdash t : \tau \quad \Delta \vdash s : \tau}{\Delta \vdash t \doteq s : o}$$

In the rules for quantifiers, we have to recall the convention that bound variables can be renamed silently. This is necessary to ensure that the variable declarations we add to the context do not conflict with existing declarations.

$$\frac{\Delta, x{:}\tau \vdash A : o \quad x \notin \mathrm{dom}(\Delta)}{\Delta \vdash \forall x{:}\tau.\, A : o} \qquad \frac{\Delta, x{:}\tau \vdash A : o \quad x \notin \mathrm{dom}(\Delta)}{\Delta \vdash \exists x{:}\tau.\, A : o}$$

While we have given the condition $x \notin \mathrm{dom}(\Delta)$ in these two rules, in practice they are often omitted by convention.

Next we come to typing terms and sequences of terms.

$$\frac{f : \boldsymbol{\tau} \to \sigma \in \Sigma \quad \Delta \vdash \mathbf{t} : \boldsymbol{\tau}}{\Delta \vdash f(\mathbf{t}) : \sigma} \qquad \frac{x{:}\tau \in \Delta}{\Delta \vdash x : \tau}$$

$$\frac{\Delta \vdash t : \tau \quad \Delta \vdash \mathbf{t} : \boldsymbol{\tau}}{\Delta \vdash (t, \mathbf{t}) : (\tau, \boldsymbol{\tau})} \qquad \frac{}{\Delta \vdash (\cdot) : (\cdot)}$$

This system does not model the overloading for predicate and function symbols at different arities that is permitted in Prolog. In this simple first-order language this is relatively easy to support, but left as Exercise 9.2.

9.4 Typing Substitutions

In order to integrate types fully into our logic programming language, we need to type all the artifacts of the operational semantics. Fortunately, the success and failure continuations are propositions for which typing is already defined, and the same is true for programs. This leaves substitutions, as calculated by unification.

For a substitution, we just demand that if we substitute t for x and x has type τ, the t also must have type τ. We write the judgment as $\Delta \vdash \theta \ \mathit{subst}$, expressing that θ is a well-typed substitution.

$$\frac{x{:}\tau \in \Delta \quad \Delta \vdash t : \tau}{\Delta \vdash (\theta, t/x) \ \mathit{subst}} \qquad \frac{}{\Delta \vdash (\cdot) \ \mathit{subst}}$$

Our prior conventions that all the variables defined by a substitution are distinct, and that the domain and codomain of a substitution are disjoint also still apply. Because we would like to separate typing from other considerations, they are not folded into the rules above which could easily be done.

9.5 Types and Truth

Now that we have extended our language to include types, we need to consider how this affects the various judgment we have. The most basic one is truth. Since this is defined for ground expression (that is, expression without free variables), we do not need to generalize this to carry a context Δ, although it will have to carry a signature Σ. We leave this implicit as in the presentation of the typing judgments.

We presuppose that any proposition we write is a well-typed proposition, that is, has type o. In other words, if we write $A\ true$ we implicitly assume that $\cdot \vdash A : o$. We have to be careful that our rules maintain this property, and in which direction the rule is read. For example, when the rule

$$\frac{\Gamma \vdash A\ true}{\Gamma \vdash A \vee B\ true}\ \vee I$$

is read from the premiss to the conclusion, then we would need a second premiss to check that $B : o$.

However, we prefer to read it as "*Assuming $A \vee B$ is a valid proposition, $A \vee B$ is true if A is true.*" In this reading, the condition on B is implicit.

Since the bottom-up reading of rules is pervasive in logic programming, we will adopt the same here. Then, only the rules for quantifiers require an additional typing premiss. Recall that Γ represents a fixed program.

$$\frac{\cdot \vdash t : \tau \quad \Gamma \vdash A(t/x)\ true}{\Gamma \vdash \exists x{:}\tau.\ A\ true}\ \exists I \qquad \frac{\cdot \vdash t : \tau \quad \Gamma; A(t/x)\ true \vdash P\ true}{\Gamma; \forall x{:}\tau.\ A \vdash P\ true}\ \forall L$$

This assumes that if we substitute a term of type τ for a variable of type τ, the result will remain well-typed. Fortunately, this property holds and is easy to prove.

Theorem 9.1 *Assume $\Delta \vdash \theta\ subst$.*

1. *If $\Delta \vdash t : \tau$ then $\Delta \vdash t\theta : \tau$.*
2. *If $\Delta \vdash \mathbf{t} : \boldsymbol{\tau}$ then $\Delta \vdash \mathbf{t}\theta : \boldsymbol{\tau}$.*
3. *If $\Delta \vdash A : o$ then $\Delta \vdash A\theta : o$.*

Proof: By mututal induction on the structure of the given typing derivations for t, $\mathbf{t}$, and A. □

9.6 Type Preservation

A critical property tying together a type system with the operational semantics for a programming language is *type preservation*. It expresses that if we start in a well-typed state, during the execution of a program all intermediate states will be well-typed. This is an absolutely fundamental property without which a type system does not make much sense. Either the type system or the operational semantics needs to be revised in such a case so that they match at least to this extent.

$$\frac{\Delta \vdash G_1 \;/\; G_2 \wedge S \;/\; F}{\Delta \vdash G_1 \wedge G_2 \;/\; S \;/\; F} \qquad \frac{\Delta \vdash G_2 \wedge S \;/\; F}{\Delta \vdash \top \;/\; G_2 \wedge S \;/\; F} \qquad \frac{}{\Delta \vdash \top \;/\; \top \;/\; F}$$

$$\frac{\Delta \vdash G_1 \;/\; S \;/\; (G_2 \wedge S) \vee F}{\Delta \vdash G_1 \vee G_2 \;/\; S \;/\; F} \qquad \frac{\Delta \vdash G_2 \;/\; S' \;/\; F}{\Delta \vdash \bot \;/\; S \;/\; (G_2 \wedge S') \vee F} \qquad \begin{array}{c}\textit{fails (no rule)}\\ \Delta \vdash \bot \;/\; S \;/\; \bot\end{array}$$

$$\frac{\Delta \vdash t \doteq s \mid \theta \quad \Delta \vdash \top \;/\; S\theta \;/\; F}{\Delta \vdash t \doteq s \;/\; S \;/\; F} \qquad \frac{\begin{array}{c}\text{there is no } \theta \text{ with}\\ \Delta \vdash t \doteq s \mid \theta\end{array} \qquad \Delta \vdash \bot \;/\; S \;/\; F}{\Delta \vdash t \doteq s \;/\; S \;/\; F}$$

$$\frac{\Delta, x{:}\tau \vdash G \;/\; S \;/\; F \quad x \notin \mathrm{dom}(\Delta)}{\Delta \vdash \exists x{:}\tau.\, G \;/\; S \;/\; F}$$

$$\frac{(\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow G) \in \Gamma \quad \Delta \vdash G(\mathbf{t}/\mathbf{x}) \;/\; S \;/\; F}{\Delta \vdash p(\mathbf{t}) \;/\; S \;/\; F}$$

Figure 1: Operational Semantics Judgment

It is worth stepping back to make explicit in which way the inference rules for our last judgment (with goal stack, failure continuation, and free variables) constitute a transition system for an abstract machine. We will add a context Δ to the judgment we had so far in order to account for the types of the free variables. For, in the form of inference rules from last lecture. We assume the clauses for each predicate p are in a normal form $\forall \mathbf{x}.\, p(\mathbf{x}) \leftarrow B'$, all collected in a fixed program Γ, and that a signature Σ is also fixed.

Each rule with one premise can be seen as a state transition rule. For example, the very first rule becomes

$$(\Delta \vdash A \wedge B \;/\; S \;/\; F) \Rightarrow (\Delta \vdash A \;/\; B \wedge S \;/\; F).$$

We do not write down the others, which can be obtained by simple two-dimensional rearrangement.

The state $\Delta \vdash \top \;/\; \top \;/\; F$ is a final state (success) since the corresponding rule has no premiss, as is the state $\Delta \vdash \bot \;/\; S \;/\; \bot$ (failure) since there is no corresponding rule.

The rules for equality have two premisses and make up two conditional

transition rules.

$$\begin{array}{lcll} (\Delta \vdash t \doteq s \;/\; S \;/\; F) & \Rightarrow & (\Delta \vdash \top \;/\; S\theta \;/\; F) & \text{provided } \Delta \vdash t \doteq s \mid \theta \\ (\Delta \vdash t \doteq s \;/\; S \;/\; F) & \Rightarrow & (\Delta \vdash \bot \;/\; S\theta \;/\; F) & \text{provided there is no } \theta \text{ with} \\ & & & \Delta \vdash t \doteq s \mid \theta \end{array}$$

In order to state type preservation, we need a judgment of typing for a state of the abstract machine, $\Delta \vdash G \;/\; S \;/\; F \; \mathit{state}$. It is defined by a single rule.

$$\frac{\Delta \vdash G : o \quad \Delta \vdash S : o \quad \Delta \vdash F : o}{\Delta \vdash G \;/\; S \;/\; F \; \mathit{state}}$$

In addition, we assume that G, S, and F have the shape of a goal, goal stack, and failure continuation, respectively, following this grammar:

$$\begin{array}{lcll} \text{Goals} & G & ::= & G_1 \wedge G_2 \mid \top \mid G_1 \vee G_2 \mid \bot \mid t \doteq s \mid \exists x{:}\tau.\, G \mid p(\mathbf{t}) \\ \text{Goal Stacks} & S & ::= & \top \mid G \wedge S \\ \text{Failure Conts} & F & ::= & \bot \mid (G \wedge S) \vee F \end{array}$$

The preservation theorem now shows that if state s is valid and $s \Rightarrow s'$, then s' is valid. To prove this we first need a lemma about unification.

Theorem 9.2 *If $\Delta \vdash t : \tau$ and $\Delta \vdash s : \tau$ and $\Delta \vdash s \doteq t \mid \theta$ then $\Delta \vdash \theta \; \mathit{subst}$. Similarly, if $\Delta \vdash \mathbf{t} : \boldsymbol{\tau}$ and $\Delta \vdash \mathbf{s} : \boldsymbol{\tau}$ and $\Delta \vdash \mathbf{s} \doteq \mathbf{t} \mid \theta$ then $\Delta \vdash \theta \; \mathit{subst}$.*

Proof: By mutual induction on the structures of $\mathcal{D}$ of $\Delta \vdash s \doteq t \mid \theta$ and $\mathcal{D}'$ of $\Delta \vdash \mathbf{s} \doteq \mathbf{t} \mid \theta$, applying inversion to the derivations of $\Delta \vdash t : \tau$ and $\Delta \vdash s : \tau$ as needed. □

Now we can state (and prove) preservation.

Theorem 9.3 *If $\Delta \vdash G \;/\; S \;/\; F \; \mathit{state}$ and $(\Delta \vdash G \;/\; S \;/\; F) \Rightarrow (\Delta' \vdash G' \;/\; S' \;/\; F')$ then $\Delta' \vdash G' \;/\; S' \;/\; F' \; \mathit{state}$.*

Proof: First we apply inversion to conclude that G, S, and F are all well-typed propositions. Then we distinguish cases on the transition rules, applying inversion to the typing derivations for G, S, and F as needed to reassemble the derivations that G', S', and F' are also well-typed.

In the case for unification we appeal to the preceding lemma, and the lemma that applying well-typed substitutions preserves typing.

In the case for existential quantification, we need an easy lemma that we can always add a new typing assumption to a given typing derivation. □

9.7 The Phase Distinction

While the operational semantics (including unification) preserves types, it does not refer to them during execution. In that sense, the types appearing with quantifiers or the context Δ are not necessary to execute programs. This means that our type system obeys a so-called *phase distinction*: we can type-check our programs, but then execute programs without further reference to types. This is a desirable property since it tells us that there is no computational overhead to types at all. On the contrary: types could help a compiler to optimize the program because it does not have to account for the possibility of ill-typed expressions. Types also help us to sort out ill-formed expressions, but they do not change the meaning or operational behavior of the well-formed ones. Generally, as type systems become more expressive, this property is harder to maintain as we will see in the next lecture.

9.8 Historical Notes

Type systems have a long and rich history, having been developed originally to rule out problems such as Russell's paradox [3] in the formulation of expressive logics for the formalization of mathematics. Church's theory of types [1] provided a great simplification and is also known as classical higher-order logic. It uses a type o (omicron) of propositions, a single type ι (iota) for individuals, and closes types under function spaces.

In logic programming, the use of types was slow to arrive since the predicate calculus (its logical origin) does not usually employ them. Various articles on notion of types in logic programming are available in an edited collection [2].

9.9 Exercises

Exercise 9.1 *The notion of type throws a small wrinkle on the soundness and non-deterministic completeness of an operational semantics with free variables. The new issue is the presence of possibly empty types, either because there are no constructors, or the constructors are such that no ground terms can be formed. Discuss the issues.*

Exercise 9.2 *Write out an extension of the system of simple types given here which permits Prolog-like overloading of function and predicate symbols at different arity. Your extension should continue to satisfy the preservation theorem.*

Exercise 9.3 *Write a Prolog program for type checking propositions and terms. Extend your program to type inference where the types on quantifiers are not explicit, generating an explicitly typed proposition (if it is indeed well-typed).*

Exercise 9.4 *Rewrite the operational rules so that unification is explicitly part of the transitions for the abstract machine, rather than a condition.*

Exercise 9.5 *Show the cases for unification, existential quantification, and atomic goals in the proof of the type preservation theorem in detail.*

Exercise 9.6 *Besides type preservation, another important property for a language is* progress*: any well-formed state of an abstract machine is either an explicitly characterized final state, or can make a further transition.*

In our language, this holds with or without types, if we declare states $\top$ / $\top$ / F (for any F) and $\bot$ / S / $\bot$ (for any S) final, and assume that G, S, and F satisfy the grammar for goals, goal stacks and failure continuations shown in this lecture.

Prove the progress theorem.

9.10 References

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

[2] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[3] Bertrand Russell. Letter to Frege. In J. van Heijenoort, editor, *From Frege to Gödel*, pages 124–125. Harvard University Press, 1967. Letter written in 1902.