15-819K: Logic Programming

Lecture 10

Polymorphism

Frank Pfenning

September 28, 2006

In this lecture we extend the system of simple types from the previous lecture to encompass polymorphism. There are some technical pitfalls, and some plausible systems do not satisfy type preservation. We discuss three ways to restore type preservation.

10.1 Heterogeneous and Homogeneous Lists

Types such as natural numbers, be it in binary or in unary notation, are easy to specify and use in the system from the previous lecture. Generic data structures such as lists, on the other hand, present difficulties. Recall the type predicate for lists:

```
list([]).
list([X|Xs]) :- list(Xs).
```

The difficulty is that for lists in general there is no restriction on `X`: it can have arbitrary type. When we try to give the declarations

```
list : type.
[] : list.
'.' : ?, list -> list.
```

we realize that there is nothing sensible we can put as the type of the first argument of cons.[1]

Two solutions suggest themselves. One is to introduce a universal type "any" and ensure that $t : any$ for all terms t. This destroys the property of

[1] Recall that `[X|Xs]` is just alternate syntax for `'.'(X, Xs)`

simple types that every well-typed term has a unique type and significantly complicates the type system. Following this direction it seems almost inevitable that some types need to be carried at runtime. A second possibility is to introduce type variables and think of the type of constructors as schematic in their free type variables.

```
list : type.
[] : list.
'.' : A, list -> list.
```

We will pursue this idea in this lecture. Although it is also not without problems, it is quite general and leads to a rich and expressive types system.

Since the typing rules above are schematic, we get to choose a fresh instance for `A` every time we use the cons constructor. This means the elements of a list can have arbitrarily different types (they are *heterogeneous*).

For certain programs it is important to know that the elements of a list all have the same type. For example, we can sort a list of integers, but not a list mixing integers, booleans, and other lists. This requires that `list` is actually a type constructor: it takes a type as an argument and returns a type. Specifically:

```
list : type -> type.
[] : list(A).
'.' : A, list(A) -> list(A).
```

With these declarations only homogeneous lists will type-check: a term of type `list(A)` will be a list all of whose elements are of type `A`.

10.2 Polymorphic Signatures

We now move to a formal specification of typing. We start with signatures, which now have a more general form. We write α for type variables and $\boldsymbol{\alpha}$ for a sequences of type variables. As in the case of clauses and ordinary variables, the official syntax quantifies over type variables in declarations.

Signature	Σ	$::=$	$\cdot$	empty signature
		$\mid$	$\Sigma, a : \mathbf{type}_n \to \mathit{type}$	type constructor declaration
		$\mid$	$\Sigma, f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau$	function symbol declaration
		$\mid$	$\Sigma, p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o$	predicate symbol declaration

Here, boldface "$\mathbf{type}_n$" stands for a sequence $\mathit{type}, \ldots, \mathit{type}$ of length n. As usual, if a sequence to the left of the arrow is empty, we may omit

the arrow altogether. Similarly, we may omit the quantifier if there are no type variables, and the argument to a type zero-ary types constructor $a()$. Moreover, function and predicate declarations should not contain any free type variables.

The language of types is also more elaborate, but still does not contain function types as first-class constructors.

$$\text{Types} \quad \tau \quad ::= \quad \alpha \mid a(\tau_1, \ldots, \tau_n)$$

10.3 Polymorphic Typing for Terms

The only two rules in the system for simple types that are affected by polymorphism are those for function and predicate symbols. We account for the schematic nature of function and predicate declarations by allowing a substitution $\hat{\theta}$ for the type variables $\boldsymbol{\alpha}$ that occur in the declaration. We suppose a fixed signature Σ.

$$\frac{\begin{array}{c}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau \in \Sigma \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash f(\mathbf{t}) : \tau\hat{\theta}}$$

We use the notation $\hat{\theta}$ to indicate a substitution of types for type variables rather terms for term variables.

Looking ahead (or back) at the required property of type preservation, one critical lemma is that unification produces a well-typed substitution. Unfortunately, in the presence of polymorphic typing, this property fails! You may want to spend a couple of minutes thinking about a possible counterexample before reading on. One way to try to find one (and also a good start on fixing the problem) is to attempt a proof and learn from its failure.

False Claim 10.1 *If $\Delta \vdash t : \tau$ and $\Delta \vdash s : \tau$ and $\Delta \vdash t \doteq s \mid \theta$ then $\Delta \vdash \theta$ subst and similarly for sequences of terms.*

Proof attempt: We proceed by induction on the derivation $\mathcal{D}$ of the unification judgment, applying inversion to the given typing judgments in each case. We focus on the problematic one.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \Delta \vdash \mathbf{t} \doteq \mathbf{s} \mid \theta\end{array}}{\Delta \vdash f(\mathbf{t}) \doteq f(\mathbf{s}) \mid \theta}$.

We note that we could complete this case if we could appeal to the induction hypothesis on $\mathcal{D}'$, since this would yield the well-typedness of θ. We can appeal to the induction hypothesis if we can show that $\mathbf{t}$ and $\mathbf{s}$ have the same sequence of types. Let's see what we can glean from applying inversion to the giving typing derivations. First, we note that there must be a *unique* type declaration for f in the signature, say

$$f : \boldsymbol{\sigma} \to \tau' \in \Sigma \text{ for some } \boldsymbol{\sigma} \text{ and } \tau'.$$

Now we write out the inversions on the given typing derivations, using the uniqueness of the declaration for f.

$\Delta \vdash f(\mathbf{t}) : \tau$	Assumption
$\tau = \tau'\hat{\theta}_1$ and $\Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}_1$ for some $\hat{\theta}_1$	By inversion
$\Delta \vdash f(\mathbf{s}) : \tau$	Assumption
$\tau = \tau'\hat{\theta}_2$ and $\Delta \vdash \mathbf{s} : \boldsymbol{\sigma}\hat{\theta}_2$ for some $\hat{\theta}_2$	By inversion

At this point we would like to conclude

$$\boldsymbol{\sigma}\hat{\theta}_1 = \boldsymbol{\sigma}\hat{\theta}_2$$

because then $\mathbf{t}$ and $\mathbf{s}$ would have the same sequence of types and we could finish this case by the induction hypothesis.

Unfortunately, this is not necessarily the case because all we know is

$$\tau = \tau'\hat{\theta}_1 = \tau'\hat{\theta}_2.$$

From this we can only onclude that θ_1 and θ_2 agree on the type variables free in τ', but they could differ on variables that occur only in $\boldsymbol{\sigma}$ but not in τ'.

$\diamond$

From this we can construct a counterexample. Consider heterogeneous lists

$$\begin{array}{lcl} \mathsf{nil} & : & \mathsf{list} \\ \mathsf{cons} & : & \forall\alpha.\, \alpha, \mathsf{list} \to \mathsf{list} \end{array}$$

Then

$$\begin{array}{lcl} x{:}\mathsf{nat} \vdash \mathsf{cons}(x, \mathsf{nil}) & : & \mathsf{list} \\ x{:}\mathsf{nat} \vdash \mathsf{cons}(\mathsf{nil}, \mathsf{nil}) & : & \mathsf{list} \end{array}$$

and

$$x{:}\mathsf{nat} \vdash \mathsf{cons}(x, \mathsf{nil}) \doteq \mathsf{cons}(\mathsf{nil}, \mathsf{nil}) \mid (\mathsf{nil}/x)$$

but the returned substitution (nil/x) is not well typed because x:nat and nil:list.

Because unification does not return well-typed substitutions, the operational semantics in whichever form we presented does also not preserve types. The design of the type system is flawed.

We can pursue two avenues to fix this problem: restricting the type system or rewriting the operational semantics.

Type Restriction. In analyzing the failed proof above we can see that at least this case would go through if we require for a declaration $f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau$ that every type variable that occurs in $\boldsymbol{\sigma}$ also occurs in τ. Function symbols of this form are called *type preserving*.[2] When all function symbols are type preserving, the falsely claimed property above actually does hold—the critical case is the one we gave.

Requiring all function symbols to be type preserving rules out heterogeneous lists, and we need to apply techniques familiar from functional programming to inject elements into a common type. I find this tolerable, but many Prolog programmers would disagree.

Type Passing. We can also modify the operational semantics so that types are passed and unified at run-time. Then we can use the types to prevent the kind of failure of preservation that arose in the counterexample above. Passing types violates the phase separation and therefore has some overhead. On the other hand, it allows data structures such as heterogeneous lists without additional coding. The language λProlog uses a type passing approach, together with some optimizations to avoid unnecessary passing of types. We return to this option below.

Before we can make a choice between the two, or resolve the apparent conflict, we must consider the type preservation theorem to make sure we understand all the issues.

[2] Function symbols are constructors, so this is not the same as type preservation in a functional language. Because of this slightly unfortunate terminology this property has also been called *transparent*.

10.4 Polymorphic Predicates

As it turns out, requiring all function symbols to be type preserving is insufficient to guarantee type preservation. The problem is presented by predicate symbols, declared now as $p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow o$. If we just unify $\Delta \vdash p(\mathbf{t}) \doteq p(\mathbf{s})$, we run into the same problem as above because predicate symbols are *never* type preserving unless they do not have any type variables at all.

Disallowing polymorphic predicates altogether would be too restrictive, because programs that manipulate generic data structures must be polymorphic. For example, for homogeneous lists we have

```
append : list(A), list(A), list(A) -> o.
```

which is polymorphic in the type variable `A`.

Moreover, the program clauses themselves also need to be polymorphic. For example, in the first clause for `append`

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

the variable `Ys` should be of type `list(A)` for a type variable `A`.

Before we can solve our problem with type preservation, we must account for the presence of type variables in program clauses, which now quantify not only over term variables, but also over type variables. The general form is $\forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{t}) \leftarrow G$ where the type variables $\boldsymbol{\alpha}$ contain all free type variables in the clause. We will come back to the normal form used in the free variables operational semantics below.

The meaning of universal quantification over types is specified via substitution in the focusing judgment.

$$\frac{\Gamma; D(\tau/\alpha)\ \mathit{true} \vdash P\ \mathit{true}}{\Gamma; \forall \alpha.\, D\ \mathit{true} \vdash P\ \mathit{true}}$$

This just states that an assumption that is schematic in a type variable α can be instantiated to any type τ. Since logical deduction is ground, we implicitly assume in the rule above that τ does not contain any free type variables.

This forces a new typing rule for clauses, which in turn means we have to slightly generalize contexts to permit declarations $\alpha\ \mathit{type}$ for type variables.

$$\frac{\Delta, \alpha\ \mathit{type} \vdash A : o}{\Delta \vdash \forall \alpha.\, A : o}$$

To be complete and precise, we also need a judgment that types themselves are well-formed ($\Delta \vdash \tau \ \mathit{type}$) and similarly for type substitutions ($\Delta \vdash \hat{\theta} \ \mathit{tsubst}$); they can be found in various figures at the end of these notes.

$$\frac{\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \qquad p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o \in \Sigma \quad \Delta \vdash \hat{\theta} \ \mathit{tsubst} \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}}{\Delta \vdash p(\mathbf{t}) : o}$$

Now we reconsider how to solve the problem of polymorphic predicates. Looking at the predicate `append`

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

for homogenous lists, we see that no run-time type error can actually arise. This is because the heads of the clauses cover the most general case for a goal. For example, `Ys` has type `list(A)` for an arbitrary type `A`. It can therefore take on the type of the list in the goal. For example, with a goal

```
append([], [1,2,3], Zs).
```

with `Zs : list(int)`, no problem arises because we instantiate the clause to `A = int` and then use that instance.

On the other hand, if we added a clause

```
append([1], [], [1]).
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

which is certainly not logically incorrect albeit redundant, then a run-time type error would occur if we called it with a goal such as

```
?- append([X], [], Zs), plus(X, s(z), s(s(z))).
```

where `X : nat`.

What is required is that each clause head for a predicate p is *maximally general* or *parametric* with respect to the declaration of p. More precisely, we say a clause

$$\forall \boldsymbol{\beta}.\, \forall \mathbf{x}{:}\boldsymbol{\tau}.\, p(\mathbf{t}) \leftarrow G \qquad \text{with} \qquad p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o \in \Sigma$$

is *parametric* if there is a type substitution $\hat{\theta}$ with $\mathrm{dom}(\hat{\theta}) = \boldsymbol{\beta}$ with $\boldsymbol{\alpha}\ \mathbf{type} \vdash \hat{\theta} \ \mathit{tsubst}$ such that

$$\boldsymbol{\alpha}\ \mathbf{type}, \mathbf{x}{:}\boldsymbol{\tau}\hat{\theta} \vdash \mathbf{t} : \boldsymbol{\sigma}.$$

In other words, the types of the arguments $\mathbf{t}$ to p in the clause head must match $\boldsymbol{\sigma}$ in their full generality, keeping all types $\boldsymbol{\alpha}$ fixed.

The proof of type preservation in the presence of this restriction is technically somewhat involved[3] so we will omit it here. In the next section we will see an alternative approach for which type preservation is easy.

10.5 Polymorphic Residuation

When previously describing residuation, we actually were somewhat cavalier in the treatment of atomic programs (that is, clause heads).

$$\frac{}{p'(\mathbf{s}) \vdash p(\mathbf{x}) > p'(\mathbf{s}) \doteq p(\mathbf{x})}$$

Strictly speaking, the resulting equation is not well-formed because it relates two atomic propositions rather than two terms. We can eliminate this inaccuracy by introducing equality between term sequences as a new goal proposition, writing is as $\mathbf{t} \doteq \mathbf{s}$ in overloaded notation. Then we can split the residuation above into two rules:

$$\frac{}{p(\mathbf{s}) \vdash p(\mathbf{x}) > \mathbf{s} \doteq \mathbf{x}} \qquad \frac{p \neq p'}{p(\mathbf{s}) \vdash p'(\mathbf{x}) > \bot}$$

The new pair of rules removes equality between propositions. However, the first rule now has the problem that if p is not maximally general, the residuated equation $\mathbf{s} \doteq \mathbf{x}$ may not be well-typed!

The idea now is to do some checking during residuation, so that it fails when a clause head is not maximally general. Since residuation would normally be done statically, as part of compilation of a logic program, we discover programs that violate the condition at compile time, before the program is executed. Following this idea requires passing in a context Δ. to residuation so we can perform type-checking. The two rules above then become

$$\frac{\Delta \vdash \mathbf{x} : \boldsymbol{\sigma} \quad \Delta \vdash \mathbf{s} : \boldsymbol{\sigma}}{\Delta; p(\mathbf{s}) \vdash p(\mathbf{x}) > \mathbf{s} \doteq \mathbf{x}} \qquad \frac{p \neq p'}{\Delta; p(\mathbf{s}) \vdash p'(\mathbf{x}) > \bot}$$

For the program fragment D_p defining p with

$$p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o$$

[3] As I am writing these notes, I do not have a complete and detailed proof in the present context.

we initially invoke residuation with the most general atomic goal $p(\mathbf{x})$ as in

$$\boldsymbol{\alpha}\ \mathbf{type}, \mathbf{x}{:}\boldsymbol{\sigma}; D_p \vdash p(\mathbf{x}) > G_p$$

leading to the residuated program clause

$$\forall\boldsymbol{\alpha}.\, \forall\mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p.$$

Residuation is easily extended to handle universally quantified type variables in programs: we just have to guess how to instantiate them so that when reaching the head the arguments have the types $\boldsymbol{\sigma}$.

$$\frac{\Delta \vdash \tau\ \mathit{type} \quad \Delta; D(\tau/\beta) \vdash p(\mathbf{x}) > G}{\Delta; \forall\beta.\, D \vdash p(\mathbf{x}) > G}$$

Note that this residuation never generates any residual existential quantification over types. This means that the operational semantics should not allow any free type variables at run-time. This makes sense from a practical perspective: even though programs are generic in the types of data they can manipulate, when we execute programs they operate on concrete data. Moreoever, since we do not actually carry the types around, their role would be quite unclear. Nevertheless, an extension to residuate types that cannot be guessed at compile-time is possible (see Exercise 10.1).

The operational semantics refers to the residuated program. Since it contains no equations involving predicates, and we assume all function symbols are type preserving, type preservation is now a relatively straightforward property. The rule for predicate invocation looks as follows:

$$\frac{\begin{array}{l}(\ p : \forall\boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow o\ ) \\ (\ \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}(\boldsymbol{\tau}/\boldsymbol{\alpha})\ ) \\ \forall\boldsymbol{\alpha}.\, \forall\mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p \qquad \Delta \vdash G_p(\boldsymbol{\tau}/\boldsymbol{\alpha})(\mathbf{t}/\mathbf{x})\ /\ S\ /\ F\end{array}}{\Delta \vdash p(\mathbf{t})\ /\ S\ /\ F}$$

In this rule, formally, we determine a type substitution $\boldsymbol{\tau}/\boldsymbol{\alpha}$, but this is only a technical device in order to make it easier to state and prove the type preservation theorem. We have indicated this by parenthesizing the extraneous premisses. During the actual operation of the abstract machine, quantifiers are not annotated with types, and type substitutions are neither computed nor applied.

Now type preservation follows in a pretty straightforward way. A critical lemma is *parametricity over types*[4]: if $\boldsymbol{\alpha}\ \mathbf{type}, \Delta \vdash t : \sigma$ then $\Delta(\boldsymbol{\tau}/\boldsymbol{\alpha}) \vdash$

[4]This notion is related to, but not the same as the semantic notion of parametricity in functional programming.

$t : \sigma(\boldsymbol{\tau}/\boldsymbol{\alpha})$ and similarly for proposition. Essentially, if we keep a type variable fixed in a typing derivation, we can substitute an arbitrary type for the variable and still get a proper derivation. This can be proven easily by induction over the structure of the given typing derivation.

We also have that if $\Delta \vdash D : o$ and $\Delta \vdash p(\mathbf{x}) : o$ for a program proposition D and $\Delta; D \vdash p(\mathbf{x}) > G$ then $\Delta \vdash G : o$. Of course, this theorem is possible precisely because we check types in the case where D is atomic.

For reference, we recap some of the judgments and languages. We have simplified equalities by using only equality of term sequences, taking the case of single terms as a special case. Recall that $G \supset D$ and $D \leftarrow G$ are synonyms.

Signatures	Σ	::=	$\cdot \mid \Sigma, a : \mathbf{type}_n \rightarrow \mathit{type} \mid \Sigma, f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow \tau$
			$\mid \Sigma, p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \rightarrow o$
Contexts	Δ	::=	$\cdot \mid \Delta, \alpha\ \mathit{type} \mid \Delta, x{:}\tau$
Types	τ	::=	$a(\boldsymbol{\tau}) \mid \alpha$
Programs	Γ	::=	$\cdot \mid \Gamma, \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p$
Clauses	D	::=	$p(\mathbf{t}) \mid D_1 \wedge D_1 \mid \top \mid G \supset D \mid \forall x{:}\tau.\, D \mid \forall \alpha.\, D$
Goals	G	::=	$p(\mathbf{t}) \mid G_1 \wedge G_2 \mid \top \mid G_1 \vee G_2 \mid \bot \mid \mathbf{t} \doteq \mathbf{s} \mid \exists x{:}\tau.\, G$
Goal Stacks	S	::=	$\top \mid G \wedge S$
Failure Conts	F	::=	$\bot \mid (G \wedge S) \vee F$

Rules defining typing and well-formedness judgments on these expressions are given at the end of these notes. Programs are in normal form so that they can be used easily in a backtracking free-variable semantics.

Now we can state and prove the type preservation theorem in its polymorphic form.

Theorem 10.2 *Assume a well-formed signature Σ, program Γ, and context Δ. Further assume that all function symbols are type preserving. If $\Delta \vdash G \ / \ S \ / \ F$ state and $(\Delta \vdash G \ / \ S \ / \ F) \Rightarrow (\Delta' \vdash G' \ / \ S' \ / \ F')$ then $\Delta' \vdash G' \ / \ S' \ / \ F'$ state.*

Proof: By distinguishing cases on the transition relation, applying inversion on the given derivations. In some cases, previously stated lemmas such as the soundness of unification and preservation of types under well-formed substitutions are required. □

10.6 Parametric and Ad Hoc Polymorphism

The restrictions on function symbols (type preserving) and predicate definitions (parametricity) imply that no types are necessary during the execu-

tion of well-typed programs.

The condition that clause heads must be maximally general implies that the programs behave parametrically in their type. The `append` predicate, for example, behaves identically for lists of all types. This is also a characteristic of parametric polymorphism in functional programming, so we find the two conditions neatly relate the two paradigms.

On the other hand, the parametricity restriction can be somewhat unpleasant on occasion. For example, we may want a generic predicate `print` that dispatches to different, more specialized predicates, based on the type of the argument. This kind of predicate is *not* parametric and in its full generality would require a type passing interpretation. This is a form of *ad hoc polymorphism* which is central in object-oriented languages.

We only very briefly sketch how *existential types* might be introduced to permit a combination of parametric polymorphism with ad hoc polymorphism, the latter implemented with type passing.

For each function symbol, we shift the universal quantifiers that do not appear in the result type into an existential quantifier over the arguments. That is,

$$f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau$$

is transformed into

$$f : \forall \boldsymbol{\alpha}_1.\, (\exists \boldsymbol{\alpha}_2.\, \boldsymbol{\sigma}) \to \tau$$

where $\boldsymbol{\alpha} = \boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2$ and $\boldsymbol{\alpha}_1 = FV(\tau)$. By the latter condition, the declaration can now be considered type preserving.

To make this work with the operational semantics we apply f not just to terms, but also to the types corresponding to $\boldsymbol{\alpha}_2$. For example, heterogeneous lists

```
list : type.
nil : list.
cons : A, list -> list.
```

are interpreted as

$$\begin{array}{lcl} \mathsf{list} & & \textit{type} \\ \mathsf{nil} & : & \mathsf{list} \\ \mathsf{cons} & : & (\exists \alpha.\, \alpha, \mathsf{list}) \to \mathsf{list} \end{array}$$

A source level term

```
cons(1, cons(z, nil))
```

would be represented as

$$\mathsf{cons}(\mathsf{int}; 1, \mathsf{cons}(\mathsf{nat}; \mathsf{z}, \mathsf{nil}))$$

where we use a semi-colon to separate type arguments from term arguments. During unification, the type argument as well as the term arguments must be unified to guarantee soundness.

For predicates, type parameters that are not treated parametrically must be represented as existential quantifiers over the arguments, the parametric ones remain universal. For example,

```
append([1], [], [1]).
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

for homogeneous lists is ad hoc polymorphic because of the first clause and should be given type

$$\mathsf{append} \quad : \quad (\exists\alpha.\, \mathsf{list}(\alpha), \mathsf{list}(\alpha), \mathsf{list}(\alpha)) \to o$$

At least internally, but perhaps even externally, `append` now has one type argument and three term arguments.

```
append(int; [1], [], [1]).
append(A; [], Ys, Ys).
append(A; [X|Xs], Ys, [X|Zs]) :- append(A; Xs, Ys, Zs).
```

Unification of the type arguments (either implicitly or explicitly) prevents violation of type preservation, as can be seen from the earlier counterexample

```
?- append(nat; [X], [], Zs), plus(X, s(z), s(s(z))).
```

which no fails to match the first clause of `append`.

The extension of polymorphic typing by using type-passing existential quantifiers is designed to have the nice property that if the program is parametric and function symbols are type-preserving, then no types are passed at runtime. However, if function symbols or predicates are needed which violate these restrictions, they can be added with some feasible and local overhead.[5]

[5] At the point of this writing this is speculation—I have not seen, formally investigated, or implemented such a system.

10.7 Historical Notes

The first proposal for parametrically polymorphic typing in Prolog was made by Mycroft and O'Keefe [3] and was strongly influenced by the functional language ML. Hanus later refined and extended this proposal [1]. Among other things, Hanus considers a type passing interpretation for ad hoc polymorphic programs and typed unification. The modern dialect λProlog [2] incorporates polymorphic types and its Teyjus implementation contains several sophisticated optimizations to handle run-time types efficiently [4].

10.8 Exercises

Exercise 10.1 *Define an extension of residuation in the presence of polymorphism that allows free type variable in the body of a clause.*

Exercise 10.2 *Write out the rules for typing, unification, operational semantics, and sketch type preservation for a type-passing interpretation of existential types as outlined in this lecture.*

Exercise 10.3 *Write a type-checker for polymorphic Prolog programs, following the starter code and instructions available on the course website.*

10.9 References

[1] Michael Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[2] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[3] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, July 1984.

[4] Gopalan Nadathur and Xiaochu Qi. Optimizing the runtime processing of types in a higher-order logic programming language. In G. Suffcliff and A. Voronkov, editors, *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, pages 110–125, Montego Bay, Jamaica, December 2005. Springer LNAI 3835.

10.10 Appendix: Judgment Definitions

We collect the rules for various judgments here for reference. General assumptions apply without being explicitly restated, such as the uniqueness of declarations in signatures, contexts, and substitutions, or tacit renaming of bound variables (both at the type and the term level). Also, many judgment implicitly carry a signature or program which never changes, so we elide them from the rules. If necessary, they are shown explicitly on the left of the judgment, separated by a semi-colon from other hypotheses.

$$\frac{\Delta \vdash \mathbf{x} : \boldsymbol{\sigma} \quad \Delta \vdash \mathbf{s} : \boldsymbol{\sigma}}{\Delta; p(\mathbf{s}) \vdash p(\mathbf{x}) > \mathbf{s} \doteq \mathbf{x}} \qquad \frac{p \neq p'}{\Delta; p(\mathbf{s}) \vdash p'(\mathbf{x}) > \bot}$$

$$\frac{\Delta; D_1 \vdash p(\mathbf{x}) > G_1 \quad \Delta; D_2 \vdash p(\mathbf{x}) > G_2}{\Delta; D_1 \wedge D_2 \vdash p(\mathbf{x}) > G_1 \vee G_2}$$

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

$$\frac{\Delta, y{:}\tau; D \vdash p(\mathbf{x}) > G}{\Delta; \forall y{:}\tau.\, D \vdash p(\mathbf{x}) > \exists y{:}\tau.\, G}$$

$$\frac{\Delta \vdash \tau \ \mathit{type} \quad \Delta; D(\tau/\beta) \vdash p(\mathbf{x}) > G}{\Delta; \forall \beta.\, D \vdash p(\mathbf{x}) > G}$$

Figure 1: Residuation Judgment $\Sigma; \Delta; D \vdash p(\mathbf{x}) > G$

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

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

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

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

$$\frac{\begin{array}{l}(\; p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o)\\ (\; \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}(\boldsymbol{\tau}/\boldsymbol{\alpha})\;)\\ \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p \in \Gamma \qquad \Delta \vdash G_p(\boldsymbol{\tau}/\boldsymbol{\alpha})(\mathbf{t}/\mathbf{x}) \;/\; S \;/\; F\end{array}}{\Delta \vdash p(\mathbf{t}) \;/\; S \;/\; F}$$

Figure 2: Operational Semantics Judgment $\Sigma; \Gamma; \Delta \vdash G \;/\; S \;/\; F$

Propositions $\Sigma; \Delta \vdash A : o$

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

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

$$\frac{\Delta \vdash A : o \quad \Delta \vdash B : o}{\Delta \vdash A \supset B : o} \qquad \frac{\Delta \vdash \mathbf{t} : \boldsymbol{\tau} \quad \Delta \vdash \mathbf{s} : \boldsymbol{\tau}}{\Delta \vdash \mathbf{t} \doteq \mathbf{s} : o}$$

Terms $\Sigma; \Delta \vdash t : \tau,\ \Sigma; \Delta \vdash \mathbf{t} : \boldsymbol{\tau}$

$$\frac{\begin{array}{c}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o \in \Sigma \quad \Delta \vdash \hat{\theta}\ \mathit{tsubst} \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash p(\mathbf{t}) : o}$$

$$\frac{\Delta, x{:}\tau \vdash A : o}{\Delta \vdash \forall x{:}\tau.\, A : o} \qquad \frac{\Delta, x{:}\tau \vdash A : o}{\Delta \vdash \exists x{:}\tau.\, A : o} \qquad \frac{\Delta, \alpha\ \mathit{type} \vdash A : o}{\Delta \vdash \forall \alpha.\, A : o}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau \in \Sigma \quad \Delta \vdash \hat{\theta}\ \mathit{tsubst} \quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash f(\mathbf{t}) : \tau\hat{\theta}}$$

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

Substitutions $\Sigma; \Delta \vdash \theta\ \mathit{subst}$

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

Figure 3: Typing Judgments

Types $\Sigma; \Delta \vdash \tau\ \mathit{type},\ \Sigma; \Delta \vdash \boldsymbol{\tau}\ \mathbf{type}_n$

$$\frac{a : \mathbf{type}_n \to \mathit{type} \in \Sigma \quad \Delta \vdash \boldsymbol{\tau}\ \mathbf{type}_n}{\Delta \vdash a(\boldsymbol{\tau})\ \mathit{type}} \qquad \frac{\alpha\ \mathit{type} \in \Delta}{\Delta \vdash \alpha\ \mathit{type}}$$

$$\frac{}{\Delta \vdash (\cdot)\ \mathbf{type}_0} \qquad \frac{\Delta \vdash \tau\ \mathit{type} \quad \Delta \vdash \boldsymbol{\tau}\ \mathbf{type}_n}{\Delta \vdash (\tau, \boldsymbol{\tau})\ \mathbf{type}_{n+1}}$$

Type Substitutions $\Sigma; \Delta \vdash \hat{\theta}\ \mathit{tsubst}$

$$\frac{}{\Delta \vdash (\cdot)\ \mathit{tsubst}} \qquad \frac{\Delta \vdash \hat{\theta}\ \mathit{tsubst} \quad \Delta \vdash \tau\ \mathit{type}}{\Delta \vdash (\hat{\theta}, \tau/\alpha)\ \mathit{tsubst}}$$

Signatures $\Sigma\ \mathit{sig}$

$$\frac{}{(\cdot)\ \mathit{sig}} \qquad \frac{\Sigma\ \mathit{sig}}{(\Sigma, a : \mathbf{type}_n \to \mathit{type})\ \mathit{sig}}$$

$$\frac{\Sigma\ \mathit{sig} \quad \Sigma; \boldsymbol{\alpha}\ \mathbf{type} \vdash \boldsymbol{\sigma}\ \mathbf{type} \quad \Sigma; \boldsymbol{\alpha}\ \mathbf{type} \vdash \tau\ \mathit{type}}{(\Sigma, f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau)\ \mathit{sig}}$$

$$\frac{\Sigma\ \mathit{sig} \quad \Sigma; \boldsymbol{\alpha}\ \mathbf{type} \vdash \boldsymbol{\sigma}\ \mathbf{type}}{(\Sigma, p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o)\ \mathit{sig}}$$

Contexts $\Sigma; \Delta\ \mathit{ctx}$

$$\frac{}{(\cdot)\ \mathit{ctx}} \qquad \frac{\Delta\ \mathit{ctx}}{(\Delta, \alpha\ \mathit{type})\ \mathit{ctx}} \qquad \frac{\Delta\ \mathit{ctx} \quad \Delta \vdash \tau\ \mathit{type}}{(\Delta, x{:}\tau)\ \mathit{ctx}}$$

Programs $\Sigma; \Gamma\ \mathit{prog}$

$$\frac{}{(\cdot)\ \mathit{prog}} \qquad \frac{\Gamma\ \mathit{prog} \quad (p : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to o) \in \Sigma \quad \cdot \vdash \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p : o}{(\Gamma, \forall \boldsymbol{\alpha}.\, \forall \mathbf{x}{:}\boldsymbol{\sigma}.\, p(\mathbf{x}) \leftarrow G_p)\ \mathit{prog}}$$

States $\Sigma; \Gamma; \Delta \vdash G\ /\ S\ /\ F\ \mathit{state}$

$$\frac{\Sigma\ \mathit{sig} \quad \Gamma\ \mathit{prog} \quad \Delta\ \mathit{ctx} \quad \Delta \vdash G : o \quad \Delta \vdash S : o \quad \Delta \vdash F : o}{\Sigma; \Gamma; \Delta \vdash G\ /\ S\ /\ F\ \mathit{state}}$$

Figure 4: Well-Formedness Judgments