Type Inference for Patterns

Type inference works for patterns too. Consider

fun leng [] = 0
    | leng (a::b) = 1 + leng b;

We first create type variables:

\[ T(\text{leng}) = \forall a \rightarrow b \]
\[ T(a) = \forall c \]
\[ T(b) = \forall d \]

From \( \text{leng} [] \) we conclude that

\[ \forall a = \forall \text{list} \]

From \( \text{leng} [] = 0 \) we conclude that

\[ \forall b = \text{int} \]

From \( \text{leng} (a::b) \) we conclude that

\[ \forall c = \forall \text{and} \quad \forall d = \forall \text{list} \]

Thus we have

\[ T(\text{leng}) = \forall \text{list} \rightarrow \text{int} \]
\[ T(a) = 'e \]
\[ T(b) = 'e \text{ list} \]

This solution is type correct throughout the definition.
Not Everything can be Automatically Typed in ML

Let’s try to type
fun f x = (x x);

We assume
T(f) = 'a -> 'b
t(x) = 'c

Now (as usual) 'a = 'c since x is the argument of f.

From the call (x x) we conclude that 'c must be of the form 'd -> 'e (since x is being used as a function).

Moreover, 'c = 'd since x is an argument in (x x).

Thus 'c = 'd -> 'e = 'c -> 'e.

But 'c = 'c -> 'e has no solution, so in ML this definition is invalid. We
can’t pass a function to itself as an argument—the type system doesn’t allow it.

In Scheme this is allowed:

\[(\text{define } (f \ x) \ (x \ x))\]

but a call like

\[(f \ f)\]

certainly doesn’t do anything good!
**Type Unions**

Let’s try to type

```haskell
fun f g = ((g 3), (g true));
```

Now the type of `g` is `'a -> 'b` since `g` is used as a function.

The call `(g 3)` says `'a = int` and the call `(g true)` says `'a = boolean`.

Does this mean `g` is polymorphic?

That is, is the type of `f`

```haskell
f : ('a->'b)->'b*'b
```

NO!

All functions have the type `'a -> 'b` but not all functions can be passed to `f`.

Consider `not : bool->bool`.

The call `(not 3)` is certainly illegal.
What we’d like in this case is a union type. That is, we’d like to be able to type \( g \) as \((\text{int}|\text{bool})\rightarrow'b\) which ML doesn’t allow.

Fortunately, ML does allow type constructors, which are just what we need.

Given

```ml
datatype T =
  I of int | B of bool;

we can redefine \( f \) as

```fun f g =
  (g (I(3)), g (B(true)));
val f = fn : (T -> 'a) -> 'a * 'a
```
Finally, note that in a definition like

```
let
  val f =
    fn x => x (* id function*)
in (f 3, f true)
end;
```

```
val it = (3, true) : int * bool
```

Type inference works fine:

Here we define $f$ in advance, so its type is known when calls to it are seen.
Reading Assignment

- Roosta: Chapters 10 and 11
- Webber: Chapters 19, 20 and 22
Prolog

Prolog presents a view of programming that is very different from most other programming languages.

A famous text book is entitled “Algorithms + Data Structures = Programs”

This formula represents well the conventional approach to programming that most programming languages support.

In Prolog there is an alternative rule of programming:

“Algorithms = Logic + Control”

This rule encompasses a non-procedural view of programming.
Logic (what the program is to compute) comes first.

Then control (how to implement the logic) is considered.

In Prolog we program the logic of a program, but the Prolog system automatically implements the control.

Logic is essential—control is just efficiency.
Logic Programming

Prolog implements logic programming. In fact Prolog means Programming in Logic.

In Prolog programs are statements of rules and facts. Program execution is deduction—can an answer be inferred from known rules and facts.

Prolog was developed in 1972 by Kowalski and Colmerauer at the University of Marseilles.
**Elementary Data Objects**

- In Prolog integers and atoms are the elementary data objects.
- Integers are ordinary integer literals and values.
- Atoms are identifiers that begin with a lower-case letter (much like symbolic values in Scheme).
- In Prolog data objects are called terms.
- In Prolog we define relations among terms (integers, atoms or other terms).
- A predicate names a relation. Predicates begin with lower-case letters.
- To define a predicate, we write clauses that define the relation.
• There are two kinds of program clauses, facts and rules.

• A fact is a predicate that prefixes a sequence of terms, and which ends with a period (".").

As an example, consider the following facts which define “fatherOf” and “motherOf” relations.

fatherOf(tom, dick).
fatherOf(dick, harry).
fatherOf(jane, harry).
motherOf(tom, judy).
motherOf(dick, mary).
motherOf(jane, mary).

The symbols fatherOf and motherOf are predicates. The symbols tom, dick, harry, judy, mary and jane are atoms.
Once we have entered rules and facts that define relations, we can make queries (ask the Prolog system questions).

Prolog has two interactive modes that you can switch between.

To enter **definition mode** (to define rules and facts) you enter

[user].

You then enter facts and rules, terminating this phase with \(^D\) (end of file).

Alternatively, you can enter

['filename'].

to read in rules and facts stored in the file named filename.
When you start Prolog, or after you leave definitions mode, you are in query mode.

In query mode you see a prompt of the form

| ?- or ?– (depending on the system you are running).

In query mode, Prolog allows you to ask whether a relation among terms is true or false.

Thus given our definition of motherOf and fatherOf relations, we can ask:

| ?– fatherOf(tom, dick). 

yes

A “yes” response means that Prolog is able to conclude from the facts and rules it has been given that the relation queried does hold.
?- fatherOf(georgeW, george).
no

A “no” response to a query means that Prolog is unable to conclude that the relation holds from what it has been told. The relation may actually be true, but Prolog may lack necessary facts or rules to deduce this.
Variables in Queries

One of the attractive features of Prolog is the fact that variables may be included in queries. A variable always begins with a capital letter.

When a variable is seen, Prolog tries to find a value (binding) for the variable that will make the queried relation true.

For example,

\[ \text{fatherOf}(X, \text{harry}) \]

asks Prolog to find a value for \( X \) such that \( X \)'s father is \( \text{harry} \).

When we enter the query, Prolog gives us a solution (if one can be found):

\[ ?- \text{fatherOf}(X, \text{harry}). \]

\[ X = \text{dick} \]
If no solution can be found, it tells us so:

| ?- fatherOf(Y, jane).
no

Since solutions to queries need not be unique, Prolog will give us alternate solutions if we ask for them. We do so by entering a “;” after a solution is printed. We get a “no” when no more solutions can be found:

| ?- fatherOf(X, harry).
X = dick ;
X = jane ;
no
Variables may be placed anywhere in a query. Thus we may ask
|  ?- fatherOf(jane,X).
  X = harry ;
  no
We may use more than one variable if we wish:
|  ?- fatherOf(X,Y).
  X = tom,  
  Y = dick ;
  X = dick, 
  Y = harry ;
  X = jane,
  Y = harry ;
  no
(This query displays all the fatherOf relations).
**Conjunction of Goals**

More than one relation can be included as the “goal” of a query. A comma (”,”) is used as an AND operator to indicate a conjunction of goals—all must be satisfied by a solution to the query.

```prolog
?- fatherOf(jane,X), motherOf(jane,Y).
X = harry,
Y = mary ;
no
```

A given variable may appear more than once in a query. The same value of the variable must be used in all places in which the variable appears (this is called **unification**).
For example,

```
| ?- fatherOf(tom,X),fatherOf(X,harry).
X = dick ;
no
```
Rules in Prolog

Rules allow us to state that a relation will hold depending on the truth (correctness) of other relations. In effect a rules says, “If I know that certain relations hold, then I also know that this relation holds.”

A rule in Prolog is of the form

\[ \text{rel}_1 \ :- \ \text{rel}_2, \ \text{rel}_3, \ldots \ \text{rel}_n. \]

This says \( \text{rel}_1 \) can be assumed true if we can establish that \( \text{rel}_2 \) and \( \text{rel}_3 \) and all relations to \( \text{rel}_n \) are true. \( \text{rel}_1 \) is called the head of the rule. \( \text{rel}_2 \) to \( \text{rel}_n \) form the body of the rule.
Example

The following two rules define a `grandMotherOf` relation using the `motherOf` and `fatherOf` relations:

```prolog
grandMotherOf(X, GM) :-
    motherOf(X, M),
    motherOf(M, GM).
```

```prolog
grandMotherOf(X, GM) :-
    fatherOf(X, F),
    motherOf(F, GM).
```

| ?- grandMotherOf(tom, GM). 
  GM = mary ; no |
| ?- grandMotherOf(dick, GM). no |
| ?- grandMotherOf(X, mary). X = tom ; no |
As is the case for all programming, in all languages, you must be careful when you define a rule that it correctly captures the idea you have in mind.

Consider the following rule that defines a sibling relation between two people:

\[
sibling(X,Y) :-
\text{motherOf}(X,M), \text{motherOf}(Y,M), \text{fatherOf}(X,F), \text{fatherOf}(Y,F).
\]

This rule says that \(X\) and \(Y\) are siblings if each has the same mother and the same father.

But the rule is wrong!

Why?
Let’s give it a try:

?- sibling(X,Y).

X = Y = tom

Darn! That’s right, you can’t be your own sibling. So we refine the rule to force X and Y to be distinct:

sibling(X,Y) :-
    motherOf(X,M), motherOf(Y,M),
    fatherOf(X,F), fatherOf(Y,F),
    \+(X=Y).

In Quintus prolog “\+” represents not; most other Prologs include a not relation.

?- sibling(X,Y).
X = dick,
Y = jane ;
X = jane,
Y = dick ;
no
Note that distinct but equivalent solutions
(like \(x = \text{dick}, y = \text{jane}\) vs. 
\(x = \text{jane}, y = \text{dick}\)) often appear in
Prolog solutions. You may sometimes
need to “filter out” solutions that are
effectively redundant (perhaps by
formulating stricter or more precise
rules).