A Type-System for DSLs in Logic Programming

Boaz Rosenan

January 2, 2009

Abstract

Logic programming has a variety of applications. It is most useful for expressing things not easily expressed with traditional, imperative languages. One reason for the power of logic programming lies in the fact that Domain Specific Languages (DSLs) can be defined easily and elegantly from within the language, so that programming can take any shape desired by its designer.

Most existing logic-programming languages (such as most Prolog variants) are dynamically typed. Other languages that are statically typed (such as Mercury) are typed in a way that does not support such DSLs. Static typing is desired since it can help the creator of a DSL lay out the rules for that DSL.

We introduce a type-system for a logic-programming language based on Prolog that will support and facilitate the definitions of DSLs.

1 Introduction

Logic programming has been known to be a powerful paradigm for solving difficult problems, ones not easily solved with main-stream imperative languages. Throughout the years it has been used for different aspects of artificial intelligence, natural language processing, expert systems and more. All these tasks require the language to be highly expressive in terms of the problem domain. The combination of simplicity and power of the computational model behind logic programming facilitated that expressiveness. Logic programming starts with first-order logic, but it can move anywhere.

Let's consider the following Prolog code example:

```
:- op(700, xfx, '::=').
% parse(Pattern, Text, Residue)
%   Succeeds if a prefix of Text matches Pattern, leaving Residue
parse(empty, Text, Text).
parse(First, [First | Rest], Rest).
parse((A, B), Text, Residue) :-
     parse(A, Text, AfterA),
     parse(B, AfterA, Residue).
parse(NonTerminal, Text, Residue) :-
```

```
    NonTerminal ::= Pattern,
    parse(Pattern, Text, Residue).
```

This is a definition of a predicate - *parse*/3, which parses a string using a given pattern. The pattern may contain terminals (elements that are expected in the string e.g. characters), and non-terminal symbols. The pattern may contain uses of the comma operator for concatenation. Non-terminals are interpreted by using the ::= /2 predicate, not defined here. This predicate converts a non-terminal symbol to a pattern. After converting it to a pattern, the *parse/3* predicate calls itself recursively to parse the underlying pattern.

While the ::= /2 predicate was not defined by this code, its meaning was. It was defined by usage. The definition of the ::= /2 predicate becomes the grammar accepted by the *parse*/3 predicate. The behavior of *parse*/3 is determined by the definition of ::= /2, which in this case is left to the user. Users can now define their own grammar using a BNF-like language, all by defining the ::= /2 predicate. The following Prolog code completes the example:

```
'S' ::= a, 'S', b.
'S' ::= empty.
:- parse('S', [a, a, b, b], []).
yes.
:- parse('S', [a, a, b], []).
no.
```

This is a usage example. Here we define a non-terminal symbol S, with 2 production rules stating that a matching string must contain a series of a symbols followed by a series of b symbols, where the number of as and bs must be the same. Then we test the *parse*/3 predicate with 2 strings - one conforming and one not.

On the Prolog level, this was a definition of ::= /2; but actually this was a definition of S. This definition did not wear the traditional logic-programming shape of a Horn Clause. In logic-programming terms this definition of ::= /2 consists of a set of "facts". An alternative way to look at this definition is to say that we've stepped out of logic programming and into another domain - context free grammars, in this particular case.

In a similar way we can show that Prolog code can be used in other domains as well - functional programming, imperative programming and more. All these extensions use the same technique of defining a predicate by its usage as a part of the "DSL" definition, and then having the user of the DSL define that predicate in the traditional sense.

The above example may raise a few questions: We used capital letters for non-terminal symbols and lowercase letters for characters, but this was no more than a convention. Is there a way we can differentiate between them? How can we make sure that grammars defined by the user are correct (i.e. have a single non-terminal symbol on the left-hand-side, and a legal term on the right hand side)?

One way of solving this is by using static typing. With static typing we can define the type signature of ::= /2 as a part of the DSL definition, to allow only correct production rules to be defined. The field of type-systems for logic-programming languages has been widely explored ([6, 8, 7]), but most existing type-systems assume "traditional" use, meaning they assume that predicates are defined using a closed set of consecutive Horn Clauses, and not in the way ::= /2 was defined in the above example. In this paper we define a type-system that allows and empowers DSL definition. The type information provided by this type system will complete the DSL definition with rules of correctness.

In the following section we shall explore the requirements we have for this type-system. In the third section we will describe the language constructs added to facilitate this type-system. In the forth section we shall describe the algorithm used to type-check the code, and in the fifth section we shall provide a discussion and comparison to other type systems.

2 Requirements

2.1 Implicit

We require that the type-system will be as seamless as possible. This is especially important for defined DSLs. For example, in the above DSL (context-free grammars) we allow the language definition (the first listing) to be altered to contain type definitions, but we would like to keep the user code (the second listing) virtually unchanged. In that example, the definition of S as a non-terminal symbol should be implicit, and any use of that symbol thereafter should be associated to that definition. This implies that we need HINDLEY-MILNER type inference, to allow our language to be *implicitly typed*.

2.2 Polymorphism

Another requirement is for type signatures to be polymorphic. For example, in the above example we may want to be able to parse lists of any kind of element (e.g. characters, tokens of some sort etc). This requires that the type signature of $parse/3$ should be polymorphic (i.e. depend on a type variable instead of all-concrete types). This kind of polymorphism is totally supported by HINDLEY-MILNER type inference. However, sometimes we may wish to restrain this polymorphism. One example would be if we want to keep the $parse/3$ predicate "open", meaning we would like to allow users to extend it to allow different kinds of patterns to be used. One example for this could be to allow character ranges (as often used in regular expressions) when the terminal type is a character. Allowing this requires that the $Pattern$ parameter provided to $parse/3$ would not be of a restricted type, but rather a member of a *type-class*. This type-class would be associated with the $parse/3$ predicate and any new *instance* of that class (a type conforming to it) would require an implementation of $parse/3$ that takes that instance type as its $Pattern$ parameter.

Type-classes provide great strength to the language. They allow terms to change their meaning based on the context, while maintaining the ability to infer types. In DSL terms, they can be used to reason about the semantic correctness of a piece of DSL code.

2.3 Extensibility

Here we require that the type-system will allow DSL definition. This requires that type inference and constraint checking associated with type-classes will not be based on the assumption that all clauses contributing to a certain predicate are grouped together in the source file. We can't even assume they are in the same source file or module. In the example above, the ::= /2 predicate can be "defined" in numerous different modules, in different parts of the software, each defining a different grammar. We would still like type-checking to be able to work correctly.

3 Language Extensions to support the Type-System

In this section we will introduce the language constructs needed for our type-system. We'll start with a definition of the HINDLEY-MILNER system, and then move on to define type-classes.

3.1 The HINDLEY-MILNER Type-Inference

3.1.1 Units

Because of the extensibility requirement we do not want to rely on the fact that clauses of a certain predicate are kept together to infer their types. It is for that same reason that we don't want to limit type inference to predicates. For this reason we define a new concept - a *unit*. A unit is a user-defined section of the code - a set of clauses, forming the scope in which one or more concepts[1] assume their type signature by type-inference. In other words, type inference "happens" inside units. Each concept that is defined in a unit has an "open" type signature inside this unit (meaning the type signature can be determined inside that unit), a "closed" signature after that unit (meaning that other clauses using that concept do not change its signature), and no signature before that unit (meaning that using that concept above the unit where it's defined will emit an error).

A unit definition looks like the following:

$$\begin{array}{l}
\text{unit } [name_1/arity_1, \ldots, name_n/arity_n]. \\
clause_1 \\
\vdots \\
clause_k \\
\text{end.}
\end{array}$$

[1]We will use the term *concept* here to refer to a *name/arity* pair. This can apply to both predicates and functions.

For example:

```
1 unit [p/2].
2 p(1, _).
3 p(_, X) :- p(X, _).
4 end.
```

This example defines a simple predicate which succeeds if given the value 1 as one of its arguments. Line 2 asserts that the first argument must be a number (or an integer, depending on how we decide to type numbers). Line 3 asserts that the first and second argument share type (and hence the second argument is also a number). The type signature provided by this definition is the combination of what we infer from both clauses. In this example we define a predicate, but units can be used to define any kind of concept.

3.1.2 Explicit Type Signature

An alternative way to provide a type signature is by specifying it explicitly. This is done using the *::/2* operator, in the following way:

$$name(\tau_1, \ldots, \tau_n) :: \tau$$

Where $\tau, \tau_1, \ldots, \tau_n$ are types. For example, here is a definition of a list:

```
[] :: list(_).
[T | list(T)] :: list(T).
```

This is a definition of the polymorphic type *list(T)*, which has 2 concepts that conform to it: *[]/0* and *(.)/2*. Since this is an explicit definition of the type signature of both concepts, this code should not be contained in a unit defining these concepts. However, this is also the definition of the concept *list/1* as a type, so this code can be written like this to implicitly provide a signature for *list/1*:

```
unit [list/1].
[] :: list(_).
[T | list(T)] :: list(T).
end.
```

As can be seen in the above example, types are first-order elements in this type-system. They all have the type *type*. In this example *list/1* receives the signature:

```
list(type) :: type
```

Predicates are concepts with type *pred*. This allows for meta-predicates to take predicates as parameters. For example, the predicate *p/2* from the above example will have the following signature:

```
p(number, number) :: pred
```

3.1.3 Implementing the Parsing example with Type Inference

The example from section 1 should look like this:

```
1  % The DSL Definition:
2  :- op(700, xfx, '::=').
3  unit [pattern/1].
4  parse(pattern(C), list(C), list(C)) :: pred
5  end.
6
7  unit [empty/0, cat/2, terminal/1, '::='/2].
8  parse(empty, Text, Text).
9  parse(terminal(First), [First | Rest], Rest).
10 parse(cat(A, B), Text, Residue) :-
11     parse(A, Text, AfterA),
12     parse(B, AfterA, Residue).
13 parse(NonTerminal, Text, Residue) :-
14     NonTerminal ::= Pattern,
15     parse(Pattern, Text, Residue).
16 end.
17
18 % DSL usage
19 unit ['S'/0].
20 'S' ::= cat(cat(terminal(1), 'S'), terminal(2)).
21 'S' ::= empty.
22 end.
23 :- parse('S', [1, 1, 2, 2], []).
24 yes.
25 :- parse('S', [1, 1, 2], []).
26 no.
```

This example consists of 3 units - 2 in the DSL definition and 1 in the usage. The first unit defines a polymorphic type - *pattern*/1, which is the type of all allowed patterns. Its type parameter determines the type of tokens it is supposed to match in the input string. The signature of *parse*/3 is given explicitly on line 4. This signature provides the connection between the types of the arguments.

The second unit defines the clauses of *pred*/3, thus inferring the types of the constructs forming a valid pattern. This unit also infers the signature of ::= /2, which is to be defined by the user. The content of the unit is the same as the definition of *parse*/3 in section 1, except for use of *cat*/2 instead of (,)/2 and the use of *terminal*/1. These are needed here because in our type-system, like many other HINDLEY-MILNER type systems, a concept can only have one signature (no overloading). This means that if (,)/2 already has a signature, we cannot "overload" it. Similarly, on line 9, if we omit the *terminal* wrapper, we will get a type-mismatch, since *First* has type *pattern*(C) on the first argument, but type C on the second argument. Adding these constructs makes the DSL less expressive, but we will fix that later in this section.

The third unit is in the "user" code. Here the new concept we define is $S/0$. It gets its type from its usage as the first argument of $::= /2$. $S/0$ is given the type *pattern(number)*, by inferring the type of the right-hand-side of the definition on line 20. We used numbers instead of atoms in this example to avoid the "atom vs. concept" problem. This is a caveat in the Prolog language, and our way of dealing with it is out of the scope of this paper.

3.2 Type Classes and Type Constraints

Type-classes originated from functional programming, and are best known as a feature of the *Haskell* type system [2]. The term *Type Class* implies *A set of types*. This is the case for some implementations, such as the one in *Haskell98*, where a type-class refers to a single type. However, some extensions to *Haskell* implemented by *GHC* and *Hugs* allow a class to attribute more than a single type, making them predicates over types more than just sets. It has been shown that this extension, while crutual to some applications such as container classes, may be unsafe in some cases [4].

We take the approach of *Parametric Type Classes* [1, 4], where the type constraint $class(\tau_1, \tau_2, \ldots, \tau_n)$ actually means: $\tau_1 \in class(\tau_2, \ldots, \tau_n)$.

3.2.1 Defining a Type Class

A definition of a new type-class takes the following form:

$$\text{class } name(\tau_1, \ldots, \tau_n) \text{ where } [\sigma_1, \ldots, \sigma_k].$$

Where *name* is a the name of the type-class, $\tau_1, \ldots, \tau_n$ are type variables, and $\sigma_1, \ldots, \sigma_k$ are type signatures of predicates depending on $\tau_1, \ldots, \tau_n$. Since all these signatures relate to predicates, $\sigma_1, \ldots, \sigma_k$ consist of only the left-hand-side of the signature. The right-hand-side is assumed to be *pred*. This definition emits a type-class called *name* that defines a mapping between $n - 1$ types (the class parameters) to a set of types. It also requires, for each instance of that class, that the predicates defined by $\sigma_1, \ldots, \sigma_k$ shall be defined with the concrete types related to that instance. Here is an example of such class definition, related to our parsing example:

```
class pattern(P, C) where [parse(P, list(C), list(C))].
```

In this example *pattern* is defined as a type-class (and not as a type as in the example in the previous subsection). As such it maps between C - the character or token type, to a set of pattern types, for which P is a representative. Instances of this type-class provide the connection between P and C. Each such instance is required to provide one or more clauses to $parse/3$, where the argument types match the types associated with P and C for that instance.

3.2.2 Type Constraints

Type constraints are mentions of a type-class used to constraint type variables. Type constraints are used in type signatures and in instance definitions. Type constraints have the following syntax:

$$class(\tau_1, \tau_2, \ldots, \tau_n)$$

Where *class* is a name of a type-class, n is the arity of *class*, and $\tau_1, \ldots, \tau_n$ are types or type variables. Semantically, the above syntax means:

$$\tau_1 \in class(\tau_2, \ldots, \tau_n)$$

Type constraints are used in type-signatures using the following syntax:

$$C_1, \ldots, C_m \Rightarrow name(\tau_1, \ldots, \tau_n) :: \tau$$

Where $C_1, \ldots, C_m$ are type constraints, *name* is the name of the concept, n is the arity, $\tau_1, \ldots, \tau_n$ are the argument types and τ is the type of the concept $name/n$. For example, the type signature for $parse/3$ inferred by the example above should be this:

```
pattern(P, C) => parse(P, list(C), list(C)) :: pred
```

3.2.3 Instance Definitions

An instance definition associates concrete types with a type-class. Instance definitions have the following form:

$$\begin{array}{l} \text{instance } C_1, \ldots, C_m \Rightarrow class(\tau_1, \ldots, \tau_n). \\ clause_1 \\ \vdots \\ clause_k \\ \text{end.} \end{array}$$

Where $C_1, \ldots, C_m$ are type-constraints, *class* is a name of a type-class, n is the arity of *class*, $\tau_1, \ldots, \tau_n$ are types and type-variables, and $clause_1, \ldots, clause_k$ are clauses for the predicates associated with *class* - at least one clause per predicate. Instances definitions follow some strict rules to assure unambiguous interpretation of types: τ_1 must be in the form $name(\upsilon_1, \ldots, \upsilon_m)$ for some m, where $name/m$ is a type concept and $\upsilon_1, \ldots, \upsilon_m$ are type variables. $\tau_2, \ldots, \tau_n$ can be of any form, but thie variables must depend (either directly or indirectly through $C_1, \ldots, C_m$) on $\upsilon_1, \ldots, \upsilon_m$.

Here is an example of an instance definition:

```
% We assume that the following is defined somewhere:
(A, B) :: pair(A, B).
```

```
instance pattern(T1, C), pattern(T2, C) => pattern(pair(T1, T2), C).
parse((A, B), Text, Residue) :-
    parse(A, Text, AfterA),
    parse(B, AfterA, Residue).
end.
```

3.2.4 Meta-Predicates and Rewrite-Rules

Unlike type inferrence, type-constraint checking is a directional process. As shown in [3], type constraints are first checked for the head of clauses, collecting the assertions - the things that can be assumed for the types of the variables in the head of the clause, and then with these assertions they check that the body of the clause meets the constraints.

In our type-system we cannot stop with clauses. Consider the ::= /2 predicate: it helps to define new concepts (such as S in the above example). Semantically, the clause

$$H ::= B$$

is equivalent to the clause

$$parse(H, T, R) : -parse(B, T, R)$$

where H appears in the head, and B appears in the body. This means that the assumptions made by B can be weaker than the ones made by H, but not vice versa. If only considered deductions for this analysis, $H ::= B$ would have been counted as: $(H ::= B) : -\text{true}$, and H and B would have been treated symmetrically.

For this reason we introduce a new concept: a *meta-predicate*. Meta-predicates are predicates that are used in the extension of the language into a new domain. ::= /2 is an example for such a predicate. Just like regular predicates, meta-predicates are also first-order elements in the language and are therefore typed as *metapred*. Unlike predicates, meta-predicates give their first argument a special meaning: It is called the *head argument*, while the rest of its arguments are called the *body arguments*. This implies that the arity of a meta-predicate is at least 1.

Meta-predicates can appear in the left-hand-side of a deduction ($\vdash$), but they cannot appear in goals. Instead of goals meta-predicates are used with a special construct: rewrite rules. A rewrite-rule has the following syntax:

$$metapred(h, b_1, \ldots, b_n) \rightsquigarrow clause.$$

Where *metapred* is a name of a meta-predicate with arity $n + 1$, h is a variable, $b_1, \ldots, b_n$ are the body arguments for *metapred* and *clause* is a clause. The variable h can appear in *clause*, but only in its head. It may not appear in $b_1, \ldots, b_n$. Semantically, rewrite rules are interpreted as deductions, so that a rewrite rule of the form:

$$metapred \rightsquigarrow head \vdash body$$

is equivalent to:

$$head \vdash metapred, body$$

In the parsing example we can define the meaning of ::= /2 as a rewrite rule:

```
NonTerminal ::= Pattern ~> (parse(NonTerminal, Text, Residue) :-
    parse(Pattern, Text, Residue)).
```

This is equivalent to:

```
parse(NonTerminal, Text, Residue) :-
    NonTerminal ::= Pattern,
    parse(Pattern, Text, Residue).
```

But it allows the type-system to evaluate type constraints in the correct order.

3.2.5 Implementing the Parsing Example with Type Classes and Rewrite Rules

In the previous example (using type-inference only) we defined *pattern*(C) to be a type. This kind of definition paused some restrictions on our implementation of the parsing example - it required some awkward modifications to the syntax of our DSL, making it less expressive. In this example we will eliminate these modifications and use a syntax closer to the original - the one from the untyped implementation.

In this implementation we will replace the use of a single type for all patterns with a use of a type-class. This type-class will provide a relation between the type of the predicate and the type of the characters or tokens it accepts. The different operators are now instances.

```
unit [pair/2].
(A, B) :: pair(A, B).
end.

class pattern(P, C) where [parse(P, list(C), list(C)].
:- op(700, xfx, '::=').
(T ::= T) :: metapred.
(H ::= B) ~> (parse(H, T, R) :- parse(B, T, R)).

instance pattern(list(C), C).
parse([First | Pattern], [First | Text], Rest) :-
    parse(Pattern, Text, Rest).
parse([], Rest, Rest).
end.

instance pattern(TA, C), pattern(TB, C) =>
    pattern(pair(TA, TB), C).
```

```
18 parse((A, B), Text, Residue) :-
19       parse(A, Text, AfterA),
20       parse(B, AfterA, Residue).
21 end.
22
23 :- op(800, yfx, ';;').
24 unit [alt/0].
25 pattern(A, C), pattern(B, C) => (A ;; B) :: alt(C).
26 end.
27
28 instance pattern(TA, C), pattern(TB, C) =>
29       pattern(alt(C), C).
30 parse(A;;B, Text, Residue) :-
31       parse(A, Text, Residue).
32 parse(A;;B, Text, Residue) :-
33       parse(B, Text, Residue).
34 end.
```

Lines 1-3 are supposed to be defined somewhre global, as this is the general definition of the comma (,) operator. Since we do not support overloading, there is just one chance to define each operator - so we assume this is it for comma. Line 5 provides the class definition for *pattern* that we already discussed, followed by the definition of ::= /2 in lines 6-8. These lines provide a syntactic definition (6), a type signature (7) and a semantic definition using a rewrite rule (8). Lines 10-13 provide the definition of the first instance - a list pattern. This kind of pattern (a list of tokens) did not appear in the original example (without types). It actually replaces the rules:

```
parse(First, [First | Rest], Rest).
parse(empty, Text, Text).
```

The reason for this is that the restrictions on instance definitions do not allow us to use the character type as a pattern type. Instead, we need to use a compound type that may take the character type as a parameter. $list(C)$ is such a type. This instance definition provides a solution for both a list of characters and an empty string (which is a private case).

Lines 16-21 define the comma operator applied on 2 patterns as a pattern, providing an implementation of $parse/3$ to support this kind of pattern. On lines 23-26 we define a new operator - ;;, which is to be used as alternation of patterns. In the previous implementations, alternation was implemented by providing more than one production rules using the same symbol on the left hand side. For example (using lists to wrap characters/tokens):

```
'S' ::= [].
'S' ::= [1], 'S', [2].
```

On both previous implementations that would work perfectly - on the first we do no type-checking, and on the second both sides of both equations are typed *pattern*(*number*) thus there is no conflict. When using type-classes, different patterns have different types. Since ::= /2 is defined to take both sides with the same type, the first line above will emit the signature: $S :: list(X)$, where X is an unbound type variable. The second line however will emit: $S :: pair\,(pair\,(list(number), \tau(S))\,, list(number))$, where $\tau(S)$ represents the type of S. This signature by itself is illegal because it is cyclic, but even if it were not - it conflicts with the first definition of S. This is why we introduce alternation as an operator, so that each new concept will be defined in a single clause and will have a single, non-conflicting type. For example:

```
'S' ::= [] ;; [1], 'S', [2].
```

The type of ;;/2 as defined on line 25 does not reflect the type of its arguments. The types of its arguments are refferred to as *existential types* [5], meannig that they are quantified by ∀ in the type signature of ;;/2. In our type-system this quantifier is implicit wherever a type variable appears in the arguments' types but does not appear in the concetp's type. Since predicates and meta-predicates are first order elements in our type-system, all polymorphic predicates and meta-predicates use existential types. We'll discuss existential types further when we will discuss the type-inference algorithm.

The reason we use existential types here is because of the second problem we demonstrated - the cyclic type emited by recursive definitions. By giving ;;/2 a type that does not depend on its arguments we eliminate that problem for any production rule for which any recursive production has an alternative. This is a good thing, because if we have a recursive production that dosn't have an alternative, the language this grammar describes does not accept any finite input.

To conclude this example, here is a usage example of the DSL defined above:

```
unit ['S'/0].
'S' ::= [] ;; [1], 'S', [2].
end.
:- parse('S', [1, 1, 2, 2], []).
yes.
:- parse('S', [1, 1, 2], []).
no.
```

References

[1] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes (extended abstract. In *In ACM conference on LISP and Functional Programming*, pages 170–181, 1992.

[2] C.V. Hall, K. Hammond, S.L.P. Jones, and P.L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[3] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in mercury draft.

[4] Mark P. Jones. Type classes with functional dependencies. pages 230–244. Springer-Verlag, 2000.

[5] K. Laeufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, 1996.

[6] A. Mycroft and R.A. O'Keefe. A polymorphic type system for PROLOG. *Artificial Intelligence*, 23(3):295–307, 1984.

[7] T. Schrijvers, V. Santos Costa, J. Wielemaker, and B. Demoen. Towards typed Prolog. *status: accepted.*

[8] Z. Somogyi, F.J. Henderson, and TC Conway. Mercury, an efficient purely declarative logic programming language. *AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS*, 17:499–512, 1995.