In the Lecture Series Logic Programming and Constraints

Terms and Unification

Presented by Stéphane Bressan

Prolog Terms

Prolog terms are formed of variables and functors (function symbols).

f(g(X, Y), h(a))

- f is functor of arity 2 (noted f/2)
- h is a functor of arity 1 (noted h/1)
- a is a functor of arity 0 or atom (noted a/0)
- X is a variable

Prolog Terms

For convenience of the programmer and efficiency of programming there are other types of atomic terms than atom such as numbers and strings

Type Testing

Several built-in type testing procedures are available in ECLiPSe.

- atom(?Atom)
  - Succeeds if Atom is a Prolog atom.
- string(?String)
  - Succeeds if String is a string.
- number(?Number)
  - Succeeds if Number is a number
- atomic(?Atomicterm)
  - Succeeds if Atomicterm is an atom, a number, or a string.
- compound(?Term)
  - Succeeds if Term is of type compound, i.e. a structure or a list.

Type Testing

- var(?Var)
  - Succeeds if Var is a variable or an attributed variable.
- ground(?Term)
  - Succeeds if Term is ground, i.e. it does not contain variables.

Type Testing

- integer(?Integer)
  - Succeeds if Integer is an integer number.
- rational(?Rational)
  - Succeeds if Rational is a rational number.
- real(?Real)
  - Succeeds if Real is a real (float or breal) number.
- float(?Real)
  - Succeeds if Real is a floating point number.
- breal(?Breal)
  - Succeeds if Breal is a bounded real number.

Integers

The magnitude of integers is only limited by the available memory. However, integers that fit into the word size of your computer are represented more efficiently (this distinction is invisible to the user). Integers are written in decimal notation or in base notation.

0
3
-5
1024
16'f3ae
0'a
155112100433309859840000000

Rational Numbers

Rational numbers are ratios of two integers (numerator and denominator). ECLiPSe represents rational numbers in a canonical form where the greatest common divisor of numerator and denominator is 1 and the denominator is positive. Rational constants are written as numerator and denominator separated by an underscore

1_3
-30517578125_32768
0_1

Real or Floating Point Numbers

Floating point numbers conceptually correspond to the mathematical domain of real numbers, but are not precisely represented. Floats are written with decimal point and/or an exponent. ECLiPSe uses double precision floats

0.0
3.141592653589793
6.02e23
-35e-12
-1.0Inf

Bounded Real Numbers

A bounded real consists of a pair of floating point numbers which constitute a safe lower and upper bound for the real number that is being represented.

Bounded real numbers are written as two floating point numbers separated by two underscores.

-0.001__0.001
3.141592653__3.141592654
1e308__1.0Inf

Bounded real numbers are usually not typed in by the user, they are the result of a computation or type coercion. All computations with bounded real numbers give safe results, taking rounding errors into account. This is achieved by doing interval arithmetic on the bounds and rounding the results outwards. The resulting bounded real is then guaranteed to enclose the true real result.

A Note on the String Data Type

- The space consumption of a string is always less than that of the corresponding list of characters.
  - "abcd" versus ["a", "b", "c", "d"]
- For long strings, it is asymptotically 16 times more compact. Items of both types are allocated on the global stack, which means that the space is reclaimed on failure and on garbage collection.
- For the complexity of operations it must be kept in mind that the string type is essentially an array representation, ie. every character in the string can be immediately accessed via its index. The list representation allows only sequential access. The time complexity for extracting a substring when the position is given is therefore only dependent on the size of the substring for strings, while for lists it is also dependent on the position of the substring.
- Comparing two strings is of the same order as comparing two lists, but faster by a constant factor. If a string is to be processed character by character, this is easier to do using the list representation.
- The higher memory consumption of lists is sometimes compensated by the property that when two lists are concatenated, only the first one needs to be copied, while the list that makes up the tail of the concatenated list can be shared. When two string are concatenated, both strings must be copied to form the new one.

A Note on the String Data Type

- What is the difference between a string and an atom?
  abcd (or 'abcd') versus "abcd"
- a string is simply stored as a character sequence
- an atom is mapped into an internal constant (This mapping is done via a table called the *dictionary*.)

A Note on the String Data Type

- copying and comparing atoms is a unit time operation, while for strings both is proportional to the string length.
- each time an atom is read into the system, it has to be looked up and possibly entered into the dictionary
- The dictionary is a much less dynamic memory area than the global stack. That means that once an atom has been entered there, this space will only be reclaimed by a relatively expensive dictionary garbage collection. It is therefore in general not a good idea to have a program creating new atoms dynamically at runtime.

Strings vs. Atoms

```
/* father_SvA.pl

afather(mary, george).
afather(john, george).
afather(sue, harry).
afather(george, edward).
sfather("mary", "george").
sfather("john", "george").
sfather("sue", "harry").
sfather("george", "edward").
```


Strings vs. Atoms

Compare:

- [eclipse 2]: afather(sue,X).
- [eclipse 3]: sfather("sue",X).

The predicate with atoms is ***indexed***, that means that the matching clause is directly selected and the *determinacy* of the call is recognized (the system does not prompt for more solutions).

Question: is there indexing of clauses with other terms (e.g compound terms)?

How to Know the Details

Examples of the types lexical and value spaces the documentation of the type testing procedures from ECLiPSe Reference Manual

- Success
  - atom(atom).
  - atom('Anything').
  - atom(*).
  - atom([]). % notice the empty list is an atom
  - atom(#).
  - atom($).
  - atom(/).
  - atom(\).
- Fail:
  - atom(1).
  - atom(this(is,a,structure)).
  - atom(X).

How to Know the Details

Definitions of the type lexical space are from the syntax section of the User Manual

Discussions on the value space are in the User Manual

Type Conversion

Several built-in type conversion procedures are available in ECLiPSe.

- fix(+Number, ?Result)
- rational(+Number, ?Result)
- float(+Number, ?Result)

Notice that they can be used with is/2:

```
:- X is rational(25 – 22).
```

Type Conversion

Several built-in type conversion procedures are available in ECLiPSe.

- term_string(?Term, ?String)
- number_string(?Number, ?String)
- atom_string(?Atom, ?String)
- Etc.

Documentation

- ECLiPSe User Manual
- ECLiPSe Reference Manual

Unification

```
/* uni.pl */
*Unification Example
p(f(g(Y), a, Z)).

:- p(f(X, Y, g(T))).
```


Unification

Unifies f(X, Y1, g(T))and f(g(Y2), a, Z)

Variables are local to a goal, a rule or a fact

Unification

:- f(X, Y, g(T)) = f(g(Y), a, Z)

Unification

- How to make the two terms identical?
  - X and Y
    - $\{X \rightarrow a, Y \rightarrow a\}$, or
    - $\{X \rightarrow b, Y = b\}$, or
    - $\{X \rightarrow f(Z, b), Y \rightarrow f(Z, b)\}$, or
    - $\{X \rightarrow Y\}$
  - f(X, Y, g(T)) and f(g(Y), a, Z)
    - $\{X \rightarrow g(Y), Y \rightarrow a, Z \rightarrow g(a)\}$, or
    - $\{X \rightarrow g(Y), Y \rightarrow a, Z \rightarrow g(b)\}$, or
    - $\{X \rightarrow g(Y), Y \rightarrow a, Z \rightarrow g(T)\}$
  - f(X, Y, g(b)) and f(g(Y), a, X)
    - impossible

Substitution and Unifier

- A ***substitution*** is an object of the form $X \rightarrow t$ where X is a variable and t a term
- Applied to a term, a substitution rewrites the occurrences of the variable X into the term t

- A ***unifier*** of two terms is a set of substitutions that makes the two terms identical (only variables of the two terms appear in a substitution and a variable appear sat most once on the left hand side of a substitution)
- Two terms are ***unifiable*** if there exists a unifier
- The term resulting from the application of the substitutions in the unifier is the ***unified term***

Most General Unifier (mgu)

- A ***most general unifier*** mgu of two terms t1 and t2 is the unifier such that the unified term t it defines is unifiable with any unified term t' obtained with a unifier of t1 and t2
- If two terms are unifiable there exist an mgu
- There can be several mgu, but they are syntactic variants (the unified term is the same, except possibly for the name of variables)
  - f(X) = f(Y)
    - mgu1: $\{X \rightarrow Y\}$
    - mgu2: $\{Y \rightarrow X\}$

Unification Algorithm

```
Input t1, t2
Output θ or Fail
Algorithm:
    θ := ∅; % θ a list of substitutions
    σ := ∅; % σ is a stack
    push t1 = t2 on σ;
    while σ <> ∅
    do
        pop 's1 = s2' from σ;
        switch
        case s1 is a variable
            do
            substitute s2 for s1 in σ and θ
            θ := θ ∪ {s1 → s2};
            enddo
        case s2 is a variable
            do
            substitute S1 for S2 in σ and θ
            θ := θ ∪ {s2 → s1};
            enddo
        case s1= f(u1, … un) and s2=f(v1, …, vn) % same functor and arity n
            do
            for i= n to 1 push ui = vi on σ;
            enddo
        otherwise goto FAIL;
    enddo;
    output θ; exit;
FAIL: output "Fail"; exit;
```


Unification Algorithm

f(X, g(a, Z)) = f(Z, g(Z, Y))

1. $\theta = \emptyset$; $\sigma = \emptyset$;
2. $\theta = \emptyset$; $\sigma = \{f(X, g(a, Z)) = f(Z, g(Z, Y))\}$;
3. $\theta = \emptyset$; $\sigma = \{X = Z, g(a, Z) = g(Z, Y)\}$;
4. $\theta = \{X \rightarrow Z\}$; $\sigma = \{g(a, Z) = g(Z, Y)\}$;
5. $\theta = \{X \rightarrow Z\}$; $\sigma = \{a = Z, Z = Y\}$;
6. $\theta = \{X \rightarrow a, Z \rightarrow a\}$; $\sigma = \{a = Y\}$;
7. $\theta = \{X \rightarrow a, Z \rightarrow a, Y \rightarrow a\}$; $\sigma = \emptyset$;

Solution: $\theta = \{X \rightarrow a, Z \rightarrow a, Y \rightarrow a\}$
is the most general unifier

Most General Unifier

f(X, g(a, Z)) = f(Z, g(Z, Y))
with $\theta = \{X \rightarrow a, Z \rightarrow a, Y \rightarrow a\}$

- f(X, g(a, Z))
  - $X \rightarrow a$: f(a, g(a, Z))
  - $Z \rightarrow a$: f(a, g(a, a))
  - $Y \rightarrow a$: f(a, g(a, a))
- f(Z, g(Z, Y))
  - $X \rightarrow a$: f(Z, g(Z, Y))
  - $Z \rightarrow a$: f(a, g(a, Y))
  - $Y \rightarrow a$: f(a, g(a, a))

Unification Algorithm

f(X, g(a, Z)) = f(Z, g(X, X))

1. $\theta = \emptyset$; $\sigma = \emptyset$;
2. $\theta = \emptyset$; $\sigma = \{f(X, g(a, Z)) = f(Z, g(X, X))\}$;
3. $\theta = \emptyset$; $\sigma = \{X = Z, g(a, Z) = g(X, X)\}$;
4. $\theta = \{X \rightarrow Z\}$; $\sigma = \{g(a, Z) = g(Z, Z)\}$;
5. $\theta = \{X \rightarrow Z\}$; $\sigma = \{a = Z, Z = Z\}$;
6. $\theta = \{X \rightarrow a, Z \rightarrow a\}$; $\sigma = \{a = a\}$;
7. $\theta = \{X \rightarrow a, Z \rightarrow a\}$; $\sigma = \emptyset$;

Solution: $\theta = \{X \rightarrow a, Z \rightarrow a\}$
is the most general unifier

Most General Unifier

f(X, g(a, Z)) = f(Z, g(X, X))
with $\theta = \{X \rightarrow a, Z \rightarrow a\}$

- f(X, g(a, Z))
  - $X \rightarrow Z$: f(a, g(a, Z))
  - $Z \rightarrow a$: f(a, g(a, a))
- f(Z, g(X, X))
  - $X \rightarrow a$: f(Z, g(a, a))
  - $Z \rightarrow a$: f(a, g(a, a))

Unification Algorithm

f(X, g(a, Z)) = f(Z, g(X, b))

1. $\theta = \emptyset$; $\sigma = \emptyset$;
2. $\theta = \emptyset$; $\sigma = \{f(X, g(a, Z)) = f(Z, g(X, b))\}$;
3. $\theta = \emptyset$; $\sigma = \{X = Z, g(a, Z) = g(X, b)\}$;
4. $\theta = \{X \rightarrow Z\}$; $\sigma = \{g(a, Z) = g(Z, b)\}$;
5. $\theta = \{X \rightarrow Z\}$; $\sigma = \{a = Z, Z = b\}$;
6. $\theta = \{X \rightarrow a, Z \rightarrow a\}$; $\sigma = \{a = b\}$;
7. Fail

Solution: Fail

Unification Algorithm

f(Z, X) = f(g(a, X), X)

1. $\theta = \emptyset$; $\sigma = \emptyset$;
2. $\theta = \emptyset$; $\sigma = \{f(Z, X) = f(g(a, X), X)\}$;
3. $\theta = \emptyset$; $\sigma = \{Z = g(a, X), X = X\}$;
4. $\theta = \{Z \rightarrow g(a, X)\}$; $\sigma = \{X = X\}$;
5. $\theta = \{Z \rightarrow g(a, X), X \rightarrow X\}$; $\sigma = \emptyset$;

Solution: $\theta = \{Z \rightarrow g(a, X), X \rightarrow X\}$;
f(g(a, X), X)

Unification Algorithm

f(Z, X, Y) = f(a, g(X), X)

1. $\theta = \emptyset$; $\sigma = \emptyset$;
2. $\theta = \emptyset$; $\sigma = \{f(Z, X, Y) = f(a, g(X), X)\}$;
3. $\theta = \emptyset$; $\sigma = \{Z = a, X = g(X), Y = X\}$;
4. $\theta = \{Z \rightarrow a\}$; $\sigma = \{X = g(X), Y = X\}$;
5. $\theta = \{Z \rightarrow a, X \rightarrow g(g(g(\ldots g(\ldots)\ldots))\}$; $\sigma = \{Y = g(g(g(\ldots g(\ldots)\ldots))\}$;
6. $\theta = \{Z \rightarrow a, X \rightarrow g(g(g(\ldots g(\ldots)\ldots)), Y \rightarrow g(g(g(\ldots g(X)\ldots))\}$;
   $\sigma = \emptyset$;

Solution: $\theta = \{Z \rightarrow a, X \rightarrow g(g(g(\ldots g(\ldots)\ldots)), Y = g(g(g(\ldots g(X)\ldots))\}$;

f(a, g(g(g(…g(…)…)), g(g(g(…g(…)…)))

Unification Algorithm with Occur Check

```
Input t1, t2
Output θ or Fail
Algorithm:
        θ := ∅; % θ a list of substitutions
        σ := ∅; % σ is a stack
        push t1 = t2 on σ;
        while σ <> ∅
        do
                pop 's1 = s2' from σ;
                switch
                case s1 is a variable
                                do
                                if s1 occurs in s2 then goto FAIL;
                                substitute s2 for s1 in σ and θ
                                θ := θ ∪ {s1 → s2};
                                enddo
                case s2 is a variable
                                if s2 occurs in s1 then goto FAIL;
                                do
                                substitute S1 for S2 in σ and θ
                                θ := θ ∪ {s2 → s1};
                                enddo
                case s1= f(u1, ... un) and s2=f(v1, ..., vn) % same functor and arity n
                                do
                                for i= n to 1 push ui = vi on σ;
                                enddo
                otherwise goto FAIL;
        enddo;
        output θ; exit;
FAIL: output "Fail"; exit;
```


Unification as Rewriting

t1 = t2
create the set $\{t1 \rightarrow t2\}$

1. f(s1, …, sn) $\rightarrow$ f(t1, …, tn):
   delete the equation and replace with equations
   $s1 \rightarrow t1, \ldots, sn \rightarrow tn$
2. f(s1, …, sn) $\rightarrow$ g(t1, …, tn):
   fail
3. $X \rightarrow X$:
   delete the equation
4. $t \rightarrow X$ where t is not a variable:
   Delete the equation and replace with $X \rightarrow t$
5. $X \rightarrow t$ and X does not occur in t:
   substitute t for X in all other equations.
6. $X \rightarrow t$ and X occurs in t:
   fail

Unification as Rewriting

f(Z, X) = f(g(a, X), X)

- $\varepsilon = \{f(Z, X) \rightarrow f(g(a, X), X)\}$
- $\varepsilon = \{Z \rightarrow g(a, X), X \rightarrow X\}$ by 1
- $\varepsilon = \{Z \rightarrow g(a, X)\}$ by 3

f(g(a, X), X)

Unification in Prolog

- Write a procedure unify/3 that succeeds if the first and second arguments are unifiable and the third argument is the unified term and fails otherwise

unify(X, X, X).

- But how about (X, X, a)?

Unification in Prolog

- Write a procedure unify/2 that unifies the first and second argument

unify(X,X).

- This procedure is =/2

Occur Check in ECLiPSe

- Try

:-X = f(X).

- ECLiPSe can handle infinite terms
- But the occur check can be set
  :- get_flag(occur_check, X).
  :- set_flag(occur_check, on).

Term Manipulation

- ?Term =.. ?List
  - Univ --- Succeeds if List is the list which has Term's functor as its first element and Term's arguments, if any, as its successive elements.
- functor(?Term, ?Functor, ?Arity)
  - Succeeds if the compound term Term has functor Functor and arity Arity or if Term and Functor are atomic and equal, and Arity is 0.
- arg(+N, +Term, ?Arg)
  - Succeeds if Arg is the Nth argument of the compound term Term.
- term_variables(?Term, ?VarList)
  - Succeeds if VarList is the list of all variables in Term.
- copy_term(+OldTerm, ?NewTerm)
  - A copy of OldTerm with new variables is created and unified with NewTerm.
- ?Term1 == ?Term2
  - Succeeds if Term1 and Term2 are identical terms.

Syntax Settings

- current_op(?Precedence, ?Associativity, ?Name)
  - Succeeds if Name is a visible operator with precedence Precedence and associativity Associativity.

  :- current_op(P, A, '+').
  P = 500
  A = xfy

  - Infix:
    - xfx non-associative
    - xfy right to left
    - yfx left to right
  - Prefix
    - fx non-associative
    - fy left to right
  - Postfix:
    - xf non-associative
    - yf right to left

Syntax Settings

- op(+Precedence, +Associativity, +Name)
  - Declare operator syntax.

  :-op(500, xfy, father).

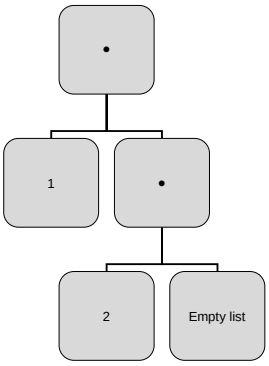
  :- X = "Dauphin Louis“ father "Louis XV" father "Louis, Duke of Burgundy".
  :- X = "Dauphin Louis“ father "Louis XV" father "Louis, Duke of Burgundy“, display(X).

Lists

We can represent lists as binary trees where list elements are left children

‘.’(1, ‘.’(2, ()))

Is the list [1,2]

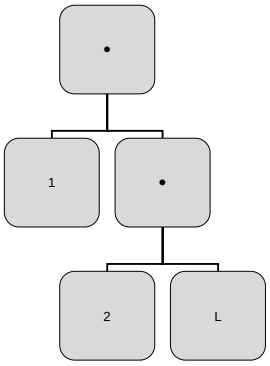


Lists

We can represent lists as binary trees where sub-lists are right children

‘.’(1, ‘.’(2, L))

Is the list [1,2| L]



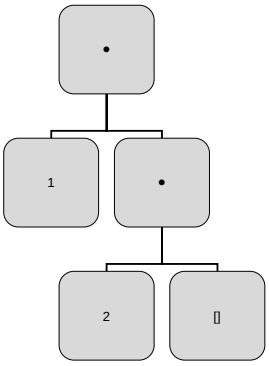
Lists

The empty list is the atom []

‘.’(1, ‘.’(2, []))

Is the list [1,2| []] which is the list [1,2]

:- [1,2|[]] = [1,2].



Write the following procedures

- car(?List, ?Head)
  - Succeeds if Head is the first element of List (the head)
- cdr(?List, ?Tail)
  - Succeeds if Tail is the sub-list of List without its first element (the tail)
- cons(?List, ?Head, ?Tail)
  - Succeeds if Head is the first element of List (the head) and Tail is the sub-list of List without its first element (the tail)

Implement a Stack

- push(?Element, +OldSctack, -NewStack)
- pop(+OldStack, -NewStack, -Element)
- top(+Stack, -Element)
- empty(+Stack)

Credits

Clipart and media are licensed from Microsoft Office Online Clipart and Media