Representing Calendrical Algorithms and Data in Prolog and Prolog III Languages

Pavol Návrat and Mária Bieliková
Slovak Technical University, Dept. of Computer Science and Engineering,
Ilkovičova 3, 812 19 Bratislava, Slovakia
E-mail: {navrat,bielikova}@elf.stuba.sk

Abstract. The paper reports on a study to develop solutions for a chosen problem in two related, but different languages. Moreover, the languages reflect two related, but different programming paradigms: logic programing, and constraint logic programming, respectively. We use Prolog to describe calendars and their mutual conversions. Next, we use Prolog III to describe the same. We discuss suitability of both languages for this kind of task. Prolog III as a logic programming language with constraints allows writing a program which is both more general (i.e., covering a broader range of cases) and more abstract (i.e., expressed on a higher level of abstraction due to the use of constraints).

Key words: Logic Programming, Constraints, Calendar, Prolog, Prolog III

Introduction

Constraint logic programming plays an important role among the concepts related to declarative programming. This framework encapsulates both the paradigms of constraint solving and logic programming [4]. The constraint solving paradigm allows concise and natural representation of complex problems because of two main reasons:

- the constraints declare properties in the domain of discourse in a straightforward way as opposed to having these properties coded indirectly into say, Prolog terms or Lisp lists
- the constraints provide for representing properties implicitly by a relation-defining formula as opposed to having listed relevant bindings to variables. Moreover, in combination with a logic programming paradigm, there is available an overall rule-based framework to reason about constraints.

In this paper, we present an example of developing solutions to one problem in both Prolog and Prolog III languages. The example is taken from the area of calendrical calculations. Several calendars, both recent and historical were described using the functional programming paradigm in Common Lisp by [3, 7].

We develop a possibly more general description of calendrical calculations using the constraint logic programming paradigm (in Prolog III language) and compare it to description of the same using the logic programming paradigm (in Prolog language).

Our aim is to identify kinds of tasks that are with advantage approached with a language with constraints. Our method is to compare and analyze solutions of the same fairly simple problem in similar languages with and without constraints. Therefore, any conclusions we arrive at are to be viewed with this methodological limitation in mind.

Method for calendrical calculations

The method requires to describe each specific calendar in a way that allows converting from it to so called absolute date and converting from an absolute date into it. The method of absolute dates establishes an arbitrary starting point as day 1 and specifies a date by giving a day number relative to the starting point [5]. The variation of the method employed by Dershowitz and Reingold [3] assumes

that Monday 1st, 1 C.E. (Common era; or, A.D., Anno Domini) according to the Gregorian calendar is the absolute date 1. The choice is arbitrary and not of fundamental importance.

Our approach is to represent the relevant knowledge declaratively. We shall accomplish this in two ways: using the logic programming language Prolog and the constraint logic programming language Prolog III. We shall concentrate on those parts that describe calendars differently.

Prolog III Language

A program in Prolog III is a set of clauses. Each clause has the form (to achieve compatibility and readability of Prolog programs, we will make use of the standard Edinburgh syntax option):

$t_0 \ :- \ t_1, t_2, t_3, \ldots t_n \ \ S$

where n can be zero, t_i are terms and S is a possibly empty system of constraints (an empty system is simply not present). The constraint system is a finite sequence of constraints (i.e., syntactic objects, formed by the symbol that expresses a relation, and a term or pair of terms denoting values belonging to the relation) which are separated by commas and enclosed by braces. For example a clause

```
is_in_interval(Num, [Low, Up])  { Num >= Low, Num =< Up }.
```

asserts that relation *is_in_interval* between number Num and interval $[Low, \ Up]$ holds under the specified constraint. The clause from this example has an empty body (the right hand side of the rule). It is called a *fact*.

For comparison, in Prolog the same relation would have to be declared by a rule

```
is_in_interval(Number, [Low, Up]) :-
    Number >= Low, Number =< Up.
```

At the first glance, the difference between Prolog and Prolog III seems to be purely syntactical. However, the important concept of Prolog III making the difference is that in Prolog III as a constraint logic programming language, the unification algorithm that is used by Prolog is augmented by a solver for the particular domain. The solver must be able to decide at any moment whether the remaining constraints are solvable. A Prolog III program still needs to search a database of facts and rules, but it can use constraints to cut off many branches of the searched tree.

When Prolog III deals with the domain of real numbers, there is another feature in which it differs from Prolog. Prolog III can perform opeartions with uninstantiated variables, e.g. in the absence of complete information the answer might be a symbolic expression or even a constraint.

Another kind of clauses in Prolog III is a *rule*, such as the two clauses

```
leap_year(gregorian, Year) :-
    mod(Year, 4, 0),              % every 4th year is a leap year, except
    mod(Year, 100, Year_mod_100)  % every 100th year is not a leap year, except
    { Year_mod_100 # 0 }.
leap_year(gregorian, Year) :-
    mod(Year, 400, 0).            % every 400th year is a leap year
```

These assert the known fact that for any year in Gregorian calendar, it is a leap year if and only if it is divisible by 4 and it is not a century year (expressed by the first clause) or if it is divisible by 400 (expressed by the second clause).

For comparison, in Prolog the same relation would be declared by the rule

```
leap_year(gregorian, Year) :-
   0 is Year mod 4,           % every 4th year is a leap year, except
   not( 0 is Year mod 100).   % every 100th year is not a leap year, except
leap_year(gregorian, Year) :-
   0 is Year mod 400.         % every 400th year is a leap year
```

We note that due to the fact that Prolog III unfortunately does not allow operations such as *mod* to be used in constraints, the Prolog III rule is to be formulated with an additional auxiliary variable to allow referring to the value of *mod* within the constraint.

Conversion to/from absolute date

We have been able to develop a fairly uniform representation of calculations for different calendars. We shall concentrate mainly on Gregorian calendar. Our results can be easily applied for other calendars.

We shall represent dates essentially as triples of integer numbers. First number denotes month, the second number represents day in a particular month, and the third number represents a year. When it is more suitable to treat the triple as one structured data item, we write date as a three element list.

If we wish to convert dates from one calendar, say X to another, say Y, we need

1. to convert the date for calendar X to absolute date
2. to convert the absolute date into calendar Y date

To achieve a complete symmetry and flexibility, we need for each calendar to know how it is related to the absolute date. Declarative representation in Prolog requires to define two predicates *absolute_from_calendar_date* and *calendar_date_from_absolute*. Here we present the predicates for conversion to/from Gregorian calendar. The first parameter identifies the calendar. The predicates assume the second parameter is instantiated. The third parameter will be computed.

```
% Prolog description
absolute_from_calendar_date(gregorian, [Month,Day,Year], Absolute_date):-
    days_in_prior_months_in_year(gregorian, Month, Year, MDays),
    Absolute_date is
        Day +                %   Days so far this month
        MDays +              % + Days so far in this month
        (Year - 1)*365 +     % + Days in prior years
        (Year - 1)//4  -     % + Julian leap days in prior years
        (Year - 1)//100 +    % - prior century years
        (Year - 1)//400.     % + prior years divisible by 400

calendar_date_from_absolute(gregorian, Absolute_date, [Month,Day,Year]):-
    Approx is Absolute_date//366,       % approximation of year
    year_from_absolute(gregorian, Absolute_date, Approx, Year),
    month_from_absolute(gregorian, Absolute_date, Year, 1, Month),
    absolute_from_calendar_date(gregorian, [Month,1,Year], Absolute_date1),
    Day is Absolute_date - Absolute_date1 + 1.
```

The calculation of the absolute date from Gregorian date is done by counting the number of days in prior years (expression $(Year-1)*365 + (Year-1)//4 - (Year-1)//100 + (Year-1)//400$), the number of days in prior months of the current year (predicate *days_in_prior_months_in_year*), and the number of days in the current month (given by date in the second argument). Operator '//' denotes the (truncated) integer quotient of two integers.

Gregorian date is computed from the absolute date by approximating the year first. Using the approximate value for the year, search for precise values of year and month is performed (predicates *year_from_absolute* and *month_form_absolute*). The day of the month is then determined by subtraction.

Actually, the declarative description of algorithm for conversion between the Gregorian date and the absolute date is defined by the predicate *absolute_from_calendar_date*. Because constraint logic programming languages (e.g. Prolog III) are capable of treating also uninstantiated variables involved in numerical relations, in Prolog III it is possible to describe the relation between the absolute date and the Gregorian date in both directions by just one clause.

In Prolog III, the numeric domain is understood to be the set of real numbers in the mathematical sense, including both rational numbers and irrational numbers. In computations, however, only rational numbers take part. It is a property of the language that if a variable is sufficiently constrained to represent a unique real number then this number is necessarily a rational number [2]. One of the

restrictions in Prolog III is that it is not possible to constrain a term to represent an integer value. Hence we adopt the following way of processing integers. We define the set of constraints that apply to rational numbers. At the end of the process, we perform a complete enumeration of the possible values of these variables by means of the predefined predicate *enum*.
The general structure of the program for conversion to/from absolute date for any calendar could be:

```
date_absolute(Calendar, [Month, Day, Year], Absolute_date) :-
    constraints over predicates,
    generator for the Month, Day, Year
    {primitive constraints}.
```

where *generator* is used to instantiate the date in a given calendar (when the date is uninstantiated). The generator forces the solution to be in the required (finite and integer) domain.

```
% Prolog III description
date_absolute(gregorian, [Month, Day, Year], Absolute_date) :-
                % Month starts on a day From (integer) and ends ...
                %  ... on a day To in a year Year
    delayed_month_days_in_year(gregorian, Month, Year, [From, To]),
    delayed_div(Year - 1, 4, Year_div_4),
    delayed_div(Year - 1, 100, Year_div_100),
    delayed_div(Year - 1, 400, Year_div_400),

    enum(Year),      % enumerate Year when it is not specified
    enum(Month),     % enumerate Month when it is not specified
    enum(Day),       % enumerate Day when it is not specified
    !                     % we are interested only in a first ...
                          %  ... solution which is the only one
    { Absolute_date =
           Day  +               %   Days so far this month
           From +               % + Days so far in this month
           (Year - 1) * 365 +   % + Days in prior years
           Year4   -            % + Julian leap days in prior years
           Year100 +            % - prior century years
           Year400,             % + prior years divisible by 400

      Year >= Absolute_date/366,    % approximation of a year
      Month =< 12,  Month >= 1,     % bounds for a month
      Day =< To - From, Day >= 1    % bounds for a day
    }.
```

It is to be noted that the order of predicates in the body of the rule is important. Also note that we have adopted a "test and generate" paradigm [6]. It can radically improve the search. Instrumental here were the properties of Prolog III. Let us assume for a moment we would have followed the usual "generate and test" paradigm. The date would be generated first. To arrive at the final, accepted instantiation of $Year$, $Month$ and Day for a given absolute date, i.e. to generate the date, enumeration of years (starting possibly from the approximated year to reduce the search, similarly to the Prolog solution above), then of months in these years and finally of days in these months is necessary. On the other hand, in the solution given above a day is calculated simply by resolving constraints.

Because the arguments of constraints over predicates are not generated at the time of their execution we use delay mechanism by means of the predefined (in Prolog III) predicate *freeze*. When a constraint (satisfiability of which cannot be determined) is delayed, the computation simply proceeds. The delayed constraint is awakened when required arguments become instantiated. An example of the use of the delay mechanism will be given in the next section.

Predicate *date_absolute* specified in Prolog III language solves both tasks of conversion (calendar date to and from absolute). In Prolog, there are two more clauses necessary:

```
% Prolog description
date_absolute(Calendar, Date, Absolute_Date) :-
    calendar(Calendar),         %Calendar is known calendar to the system
    nonvar(Date), !,            %absolute_from_Calendar
    absolute_from_calendar_date(Calendar, Date, Absolute_Date).
date_absolute(Calendar, Date, Absolute_Date) :-
    calendar(Calendar),         %Calendar is known calendar to the system
    nonvar(Absolute_Date), !,   %Calendar_from_absolute
    calendar_date_from_absolute(Calendar, Absolute_Date, Date).
```

The example of the predicate *date_absolute* manifests clearly the capability of Prolog III to allow writing more general solutions than it is possible in Prolog.

Conversion between calendars

With conversion between different calendars, the situation is similar to converting to/from absolute dates. Let us assume that we have specified not only the Gregorian calendar, but also other calendars in a way similar to the above. More specifically, the definitions of the predicate *date_absolute* in Prolog III, or of the pair of predicates *absolute_from_calendar_date* and *calendar_date_from_absolute* in Prolog were augmented by the corresponding sets of clauses such that the first parameter in their heads is a constant denoting the indicated calendar.

This is quite natural for both languages and reflects their declarative style. For comparison, in a functional style it is more natural to define special functions for each calendar e.g., the pair *absolute_from_gregorian* and *gregorian_from_absolute* would define the Gregorian calendar. Dershowitz et al. [3, 7] have written such pairs of functions in Common Lisp for several recent and historical calendars. To write just two complex functions in Common Lisp and distinguishing several cases according to respective calendars inside them is technically feasible, but it would inevitably lead to functions several pages long, which contradicts any principle of a good programming style. More importantly perhaps, it would force grouping pieces of code according to a rather secondary criterion rather than grouping all the description of a particular calendar together. Here, it would be another interesting exercise to investigate how suitable for this problem would be a combination of an object-oriented paradigm with the presented declarative approach.

To calculate for a given date in a given calendar the corresponding date in another given calendar, we define predicate *convert*. Because of the required symmetry of the conversion relation, i.e. for given both calendars, say $Calendar1$ and $Calendar2$, either date for $Calendar2$ is calculated from given date for $Calendar1$, or vice versa, in Prolog we need two clauses to declare the predicate *convert*:

```
% Prolog description
convert(Calendar1, Calendar2, Date1, Date2) :-
    nonvar(Date1), !,                          %Date1 specified
    absolute_from_calendar_date(Calendar1, Date1, Absolute_Date),
    calendar_date_from_absolute(Calendar2, Absolute_Date, Date2).
convert(Calendar1, Calendar2, Date1, Date2) :-
    !, nonvar(Date2),                          %Date2 specified
    absolute_from_calendar_date(Calendar2, Date2, Absolute_Date),
    calendar_date_from_absolute(Calendar1, Absolute_Date, Date1).
```

In the Prolog III definition of the predicate *convert*, we use the delay mechanism:

```
% Prolog III description
convert(Calendar1, Calendar2, Date1, Date2) :-
    delayed_date_absolute(Calendar1, Date1, Absolute_Date),
    delayed_date_absolute(Calendar2, Date2, Absolute_Date).

delayed_date_absolute(Calendar, Date, Absolute_date) :-
    freeze(Date, date_absolute1(Calendar, Date, Absolute_date)),
    freeze(Absolute, date_absolute1(Calendar, Date, Absolute_date)).
```

Predicate *delayed_date_absolute* delays calculation of the date for a given calendar until one of the two terms *Date* and *Absolute_date* is known.

It should be noted that the predicate *delayed_date_absolute* can be written in a more efficient way, so that *date_absolute* is not executed twice:

```
% Prolog III description
delayed_date_absolute(Calendar, Date, Absolute_date) :-
    freeze(Date, date_absolute1(Calendar, Date, Absolute_date, X)),
    freeze(Absolute, date_absolute1(Calendar, Date, Absolute_date, X)).

delayed_date_absolute1(_, _, _, X) :-  known(X), !.
delayed_date_absolute1(Calendar, Date, Absolute_date, 1') :-
    date_absolute(Calendar, Date, Absolute_date).
```

The predicate *delayed_date_absolute1* uses an auxiliary variable X to avoid executing *date_absolute* twice if *Date* is known. The first clause of *delayed_date_absolute1* will fail since X is not known. Once the absolute date is calculated it will assign 1' to X (i.e., Boolean value *true*). Subsequent calls to execute *delayed_date_absolute1* will succeed with the first clause which does not instantiate any constraint.

Incorporating other relevant data

In this section, we give an example which further enhances the capabilities of the presented solution to the problem of calendrical calculations. We wish to modify the solutions presented so far in such a way that they would reflect more faithfully the actual state of affairs. For example, let us consider the Gregorian calendar. It is a well known fact that the calendar has been adopted only in the sixteenth century, and originally only in a very few states. Thus an answer to the question "Which of the two dates: January 4th, 1643 in Italy and December 25th, 1642 in England describes an earlier date ?" is slightly more complicated than it might appear. In this case the first date (it is in the Gregorian calendar) is earlier than the second one (it is in the Julian calendar which was still in use in England: in Gregorian, the date was January 5th, 1643).

We show how we represent knowledge on the date when and where a particular calendar was adopted. This is important when there is a need to relate dates in different localities, such as countries or, more generally (administrative) districts. Let us describe the above in Prolog. We declare the calendar adopted by a particular district by means of a predicate *current_calendar*. It defines relation among a particular district, interval of absolute dates and calendar which was adopted during the specified interval in a given district.

```
current_calendar(italy, [0, 577735], julian).
current_calendar(italy, [577736], gregorian).
current_calendar(egypt, [227014], islamic(egypt)).
current_calendar( ......
```

Interval specified by one element list represents dates from the indicated absolute date up till now. Next we modify the predicate *date_absolute* which converts the date in a given calendar to and from absolute date to take into account also a given district. This can be accomplished in Prolog by two clauses because of the required symmetry with respect to date and absolute date:

```
% Prolog description
date_absolute(District, Date, Absolute_Date) :-
    nonvar(Absolute_Date), !,      %Absolute_Date to Date in a District
    current_calendar(District, Interval, Calendar),
    is_in_interval(Absolute_Date, Interval),
    calendar_date_from_absolute(Calendar, Absolute_Date, Date).
date_absolute(District, Date, Absolute_Date) :-
    nonvar(Date),                  %Date to Absolute_Date in a District
    current_calendar(District, Interval, Calendar),
    absolute_from_calendar_date(Calendar, Date, Absolute_Date),
    is_in_interval(Absolute_Date, Interval).
```

First clause solves the simpler situation: there is given the absolute date. Hence interval for a particular district and absolute date is found by means of the *is_in_interval* predicate and the date is calculated.

When absolute date is to be calculated for a particular district in Prolog, intervals for a given district are retrieved and absolute date is computed. Then (after calculation) it is tested by a predicate *is_in_interval*.

In Prolog III, we can declare the same by one clause as follows:

```
% Prolog III description
date_absolute(District, Date, Absolute_Date) :-
    current_calendar(District, Interval, Calendar),
    is_in_interval(Absolute_Date, Interval),
    date_absolute(Calendar, Date, Absolute_Date).
```

The rule constrains the variables first and only then the date (or the absolute date) is calculated with respect to these constraints.

Conclusions

We have shown how a class of calendrical algorithms and data can be represented in a declarative style, in two different but related languages. We are convinced that the declarative representation either in Prolog or in Prolog III is superior to e.g. a procedural one. We support this claim with the reference to locality dependent features of calendars. Besides the aforementioned intervals of applicability of particular calendars in given districts there are other aspects of some calendars that cannot be expressed algorithmically. As an example, we mention the Islamic calendar, which defines beginnings of (some) months and (some) holidays by proclamation, not by any algorithmic calculation. In order to be able to perform correct calculations, we have suggested to incorporate records of such events in the calendar description [1]. This can be done quite naturally in a declarative language.

Our presentation provides material for comparisons between Prolog and Prolog III, which has been our main goal. Although the problem domain is not very typical for constrained relations, and most of the calculations follow straightforward algorithms, nevertheless we were able to identify relations which can be described in Prolog III more simply and naturally than in Prolog. We were able to write predicates in Prolog III that are more general than the corresponding ones in Prolog. Moreover, due to the fact that part of the relationships among objects can be expressed implicitly by means of constraints in Prolog III, whereas in Prolog all of them must be stated explicitly as goals, predicates in Prolog III tend to be more abstract than those in Prolog.

We have suggested to describe the problem domain alternatively also in a language that incorporates object-oriented features. There have been several attempts to combine logic programming and object-oriented programming paradigms, e.g. [8].

References

1. M. Bieliková and P. Návrat. On declarative presentation of calculations for the Islamic calendar. Technical report, Slovak Technical University, Bratislava, 1995.
2. A. Colmerauer. An introduction to Prolog III. Technical report, Aix-Marseille II University, Marseille, 1990.
3. N. Dershowitz and E.M. Reingold. Calendrical calculations. *Software - Practice and Experience*, 20(9):899–928, 1990.
4. J. Jaffar and J.-L. Lassez. Constrained logic programming. In *14th ACM Symposium on the Principles of Programming Languages*, pages 111–119, 1987.
5. L. Lamport. On the proof of correctness of a calendar program. *CACM*, 22(10):554–556, 1979.
6. L. Matyska. Constraint logic programming: An overview. In *Conf. Proc. SOFSEM'93*, pages 133–164, 1993.
7. E.M. Reingold, N. Dershowitz, and S. Clamen. Calendrical calculations, II: Three historical calendars. *Software - Practice and Experience*, 23(4):383–404, 1993.
8. D. Xu and G. Zheng. Logical objects with constraints. *ACM SIGPLAN Notices*, 30(1):5–10, 1995.