A Constraint Solver for Sequences

Nikolai Kosmatov

Université de Franche-Comté, LIFC, 25030 Besançon France
kosmatov@lifc.univ-fcomte.fr

1 Introduction

Research work of the last ten years has shown the effectiveness and fruitfulness of constraint logic programming in different areas of software engineering. The development of constraint solvers for some data types has already permitted the symbolic evaluation of formal models containing data elements of these types. For example, the tool BZTT [3] uses the solver CLPS-B [2] and allows the animation and test generation for formal models containing integers, sets, functions and relations. BZTT was used to validate and verify software in different projects for industry, transport, commerce and finance [2] (see also references in [2]), e.g. for PSA Peugeot Citroën, Schlumberger, Thales. The model evaluated by BZTT is written in a logic notation with sets like B or Z. Sequences are one of the data types used in these notations and representing finite lists of elements such as stacks, queues, communication channels, sequences of transitions or any other data with consecutive access to elements.

Nowadays there exists no validation and verification tool with an integrated constraint solver for sequences. The main motivation of this work is to develop such a solver, which is indispensable to take into account sequences during the validation and verification of software. This solver will be integrated into the existing constraint solvers for other data types such as CLPS-B [2] and used in validation and verification tools such as BZTT [3].

The problem of constraint solving for sequences is very close to that of words or lists. The fundamental result of Makanin [9] shows that the satisfiability of word equations (where the concatenation is the unique operation) is decidable. Kościelski and Pacholski [7] showed that for a given constant $c > 2$ the problem of the existence of a solution of length $\leq cd$ for an equation of length d is NP-complete. Therefore there does not exist any fast algorithm for word equations in the general case. The decidability of the existential theories of words is close to the borderline of decidability. Durnev [4] showed that the positive $\forall\exists^3$-theory of concatenation is unsolvable. The decidability of word equations with an additional equal-length predicate is still an open problem. We refer the reader to [1, 6] for more detail on the word equations.

The general constraint solving problem for sequences is even more complicated than that for words, because sequences generalize words and are usually considered with more operations. Therefore it is impossible to provide a general and efficient constraint solver for sequences terminating for all constraint problems. Nevertheless, even a partial constraint solving technique of reasonable complexity would be extremely useful for applications. Our work aims to study the problem of constraint solving for sequences from the practical point of view

and to provide at least a partial constraint solving technique for the problems which appear in practice.

Definition. Let E be a set. *A sequence* over E is a finite list of elements of E. *The size (length)* of a sequence S is the number of elements of S. The empty sequence (of size 0) is denoted by $[\,]$.

For example, let $E = \{1, 2, 3\}$, then $[\,]$, $[3]$, $[1, 2, 3]$, Let us recall the usual operations on sequences which are used, for example, in the formal notations B and Z. We give in brackets an example with the notation commonly used in B. For the convenience of the reader, we prefer to use this logic notation (e.g. $\mathrm{size}\,(S) = N$) rather than that of Prolog (e.g. `S size N`).

1. *the first element* $(\mathrm{first}\,[1, 2, 2, 2, 3] = 1)$;
2. *the last element* $(\mathrm{last}\,[1, 2, 2, 2, 3] = 3)$;
3. *the front, or deleting of the last element* $(\mathrm{front}\,[1, 2, 2, 2, 3] = [1, 2, 2, 2])$;
4. *the tail, or deleting of the first element* $(\mathrm{tail}\,[1, 2, 2, 2, 3] = [2, 2, 2, 3])$;
5. *prefix* $(1 \rightarrow [2, 2, 2, 3] = [1, 2, 2, 2, 3])$;
6. *append* $([1, 2, 2, 2] \leftarrow 3 = [1, 2, 2, 2, 3])$;
7. *the size* $(\mathrm{size}\,[1, 2, 2, 2, 3] = 5)$;
8. *take the first n elements* $([1, 2, 2, 2, 3] \uparrow 2 = [1, 2])$;
9. *remove the first n elements* $([1, 2, 2, 2, 3] \downarrow 2 = [2, 2, 3])$;
10. *the concatenation* $([1, 2, 2] \cap [2, 3] = [1, 2, 2, 2, 3])$.
11. *the reverse* $(\mathrm{rev}\,([1, 2, 3]) = [3, 2, 1])$.

We focus our attention on the resolution of constraints for sequences with the operations 1–11, using an external numerical constraint solver such as CLP(FD) to solve the numerical constraints on the sequence size. The existing constraint solvers can be used for constraints of other data types.

2 Constraint Solver for Sequences

Our technique of constraint solving for sequences was implemented in the CHR language [5] in SICStus Prolog, and can be consulted and executed from [8]. The implementation in CHR has the advantage to be very clear and easy to experiment with. The algorithm is inspired by the generalized concatenation for lists implemented by Thom Frühwirth. *The generalized concatenation* $S = \mathrm{conc}\,(S_1, \ldots, S_k)$ is equivalent to $(k-1)$ simple concatenations of the sequences $S_1, \ldots, S_k$, that is, to $S = S_1 \cap \ldots \cap S_k$. The first step of the algorithm is rewriting of the constraints 1–6, 8–10 in terms of conc and size. Some of the rewriting rules are given below, where T and Y denote new variables standing for a sequence and an element respectively.

$$
\begin{array}{rcc}
\mathrm{first}\,(S) = X & \Leftrightarrow & S = \mathrm{conc}\,([X], T). \\
\mathrm{front}\,(S) = S_1 & \Leftrightarrow & S = \mathrm{conc}\,(S_1, [Y]). \\
X \rightarrow S_1 = S & \Leftrightarrow & S = \mathrm{conc}\,([X], S_1). \\
S \uparrow N = S_1 & \Leftrightarrow & S = \mathrm{conc}\,(S_1, T),\ \mathrm{size}\,(S_1) = N. \\
S \downarrow N = S_2 & \Leftrightarrow & S = \mathrm{conc}\,(T, S_2),\ \mathrm{size}\,(T) = N. \\
S_1 \cap S_2 = S & \Leftrightarrow & S = \mathrm{conc}\,(S_1, S_2).
\end{array}
$$

```
:- use_module(library(clpfd)).
:- use_module(library(chr)).

handler sequences.
constraints conc/2, size/2, rev/2, labeling/0.
operator(700,xfx,conc). % 'List conc Seq' means conc(List)=Seq
operator(700,xfx,size). % 'Seq size N' means size(Seq)=N

r01@ rev(R,S)                      <=>  reverse(R,S).
r02@ [] conc L                     <=>  L=[].
r03@ [R] conc L                    <=>  R=L.
r04@ [R|Rs] conc []                <=>  R=[], Rs conc [].
r05@ [[X|R]|Rs] conc L             <=>  L=[X|L1], [R|Rs] conc L1.
r06@ Rs conc L                     <=>  delete([],Rs,Rs1) | Rs1 conc L.
r07@ Rs conc L                     <=>  delete(L,Rs,Rs1) | Rs1 conc [].
r08@ R conc L                      ==>  lenPropagate(R,L).
r09@ [] size N                     <=>  N#=0.
r10@ [_|L] size N                  <=>  N#=M+1, L size M.
r11@ L size N                      <=>  ground(N) | N1 is N, length(L,N1).
r12@ (X size N1)#Id \ X size N2    <=>  N1=N2 pragma passive(Id).
r13@ labeling, ([R|Rs] conc L)#Id <=>  true |
     ( var(L) -> length(L,_) ; true),
     ( R=[], Rs conc L ; L=[X|L1], R=[X|R1], [R1|Rs] conc L1 ),
     labeling pragma passive(Id).

reverse([],[]).
reverse(R,L):- R size N, L size N, X size 1,
     [X,R1] conc R, [L1,X] conc L, reverse(R1,L1).

delete( X, [X|L],  L).
delete( Y, [X|Xs], [X|Xt]) :- delete( Y, Xs, Xt).

lenPropagate([], []).
lenPropagate([R|Rs],L) :- R size NR, L size NL, L1 size NL1,
     NL #= NR + NL1, lenPropagate(Rs,L1).
```

Fig. 1. The essential part of the solver in CHR

This rewriting is executed at most once for each new constraint and leaves in the constraint store the constraints conc , size and rev only.

The second step of the algorithm deals with these three constraints. The CHR rules for this part of the algorithm are given in Figure 1. Recall that a simplification rule `C1,...,Ci <=> D1,...,Dj` in CHR replaces the constraints `C1,...,Ci` in the constraint store by the list of constraints or Prolog goals `D1,...,Dj`. A guarded version `C1,...,Ci <=> Guard | D1,...,Dj` is applied if in addition the Prolog goal `Guard` is true. The rule `C1,...,Ci ==> D1,...,Dj` will add (or execute) the constraints (or Prolog goals) `D1,...,Dj` if the constraint store contains the constraints `C1,...,Ci` , which are not deleted. We do not detail the passive constraint declaration `#Id ... pragma passive(Id)`, which is just an optimiza-

tion and is not crucial. We refer the reader to [5] for more detail on the CHR language.

The rule r01 aims to propagate the constraint rev using an additional Prolog predicate `reverse`. The rules r02–r05 propagate the generalized concatenation `Rs conc L` until the first element of `Rs` is a non valuated variable. If it is the case, we can jump over this variable only by deleting empty sequences `[]` or the sequence `L` itself in `Rs` as shown in rules r06–r07. If it is still not sufficient for a constraint $\mathrm{conc}\,[R_1, R_2, ..., R_i] = L$, the rule r08 and `lenPropagate` establish the relation $\mathrm{size}\,(R_1) + \ldots + \mathrm{size}\,(R_i) = \mathrm{size}\,(L)$ between the sizes of the sequences, which can help in propagation. The propagation for the constraint `L size N` (where `N` can be an arithmetic expression) is provided by the rules r08–r11. To avoid repetitions, the rule r12 replaces the constraint `X size N2` in presence of `X size N1` by `N1 #= N2`. The rule r13 defines the labeling constraint, which is written at most once at the very end of the constraint list. It tries to find all the possible solutions of the constraint problem and does not necessarily terminate.

Our experiments show that our method is rather efficient on big constraint sets and gives satisfactory results even on sets of several hundreds of constraints. It is due to the uniformity of the approach: although we express some simple constraints in terms of a more complicated and more general constraint conc, it minimizes the number of different constraints and the number of constraint handling rules and finally provides a much faster solver for big constraint sets.

References

1. Algebraic Combinatorics on Words. Cambridge University Press, 2002. ISBN 0521812208.
2. F. Bouquet, B. Legeard, F. Peureux. CLPS-B – Constraint solver to animate a B specification. International Journal on Software Tools for Technology Transfer, 6 (2004), 143–157.
3. The BZ-Testing-Tools web site, http://lifc.univ-fcomte.fr/∼bztt, Université de Franche-Comté, Besançon.
4. V. G. Durnev. Studying algorithmic problems for free semi-groups and groups. In: S. Adian, A. Nerode (Eds). Logical Foundations of Computer Science (LFCS 97). Lect. Notes Comp. Sci., 1234(1997), 88–101. Springer-Verlag.
5. T. Frühwirth. Theory and Practice of Constraint Handling Rules. In: P. Stuckey, K. Marriot (Eds.). Special Issue on Constraint Logic Programming. Journal of Logic Programming, 37(1998), 95–138.
6. J. Karhumäki, J. Maňuch, W. Plandowski. The expressivity of languages and relations by word equations. Journal of the Association for Computing Machinery, 47(2000), 483–505.
7. A. Koscielski, L Pacholski. Complexity of Makanin's Algorithm. Journal of the ACM, 43(1996), no.4, 670-684.
8. N. Kosmatov. Constraint solving for sequences web site. http://lifc.univ-fcomte.fr/∼kosmatov/sequences, Université de Franche-Comté, Besançon.
9. G. S. Makanin. The problem of solvability of equations in a free semi-group. Mat. Sbornik (N.S.), 103 (1977), no.2, 147–236 (in Russian). English translation in: Math. URSS Sbornik, 32(1977) 129–198.