Compiling and Executing Declarative Modeling Languages to Gecode

Raffaele Cipriano, Agostino Dovier, and Jacopo Mauro

Dipartimento di Matematica e Informatica
Università di Udine, via delle Scienze 208, 33100, Udine, Italy
(cipriano,dovier)@dimi.uniud.it

Abstract. We developed a compiler from SICStus Prolog CLP(FD) to Gecode and a compiler from MiniZinc to Gecode. We compared the running times of the executions of (standard) codes directly in the three languages and of the compiled codes for some classical problems. Performances of the compiled codes in Gecode improve those in the original languages and are comparable with running time of native Gecode code. This is a first step towards the definition of a unified declarative modeling tool for combinatorial problems.

1 Introduction

In the past decades a lot of techniques and solvers have been developed to cope with combinatorial problems (e.g., branch and bound/cut/price, constraint programming, local search) and several modeling languages have been proposed to easily model problems and interact with the solvers. In fact, it would be desirable to have both user-friendly modeling languages, that allows to define problems in an easy and flexible way, and efficient solvers, that cleverly explore the search space.

CLP(FD), Gecode, MiniZinc are recent modeling platforms, with different characteristics: Gecode has excellent performances, but is not as user-friendly as declarative approaches, like CLP(FD) or MiniZinc. We present a compiler from SICStus Prolog CLP(FD) to Gecode and a compiler from MiniZinc to Gecode. We have chosen a set of benchmarks, and compare their running times in the original paradigms (SICStus, MiniZinc, and Gecode) and in their Gecode translation. We also compare our results with the translation of MiniZinc into Gecode offered by MiniZinc developers.

The results are rather encouraging. Native code executions are typically faster in Gecode than in SICStus and MiniZinc. However, in all cases, compilation (and then execution) in Gecode improves the performance of the native execution, and, moreover, these times are comparable with running time of native Gecode code. This way, the user can model problems at high level keeping all the advantages of this programming style, without loosing efficiency w.r.t. C++ encoding. Moreover, our encoding of MiniZinc in Gecode outperforms the one presented in [1].

2 The Languages Used

We briefly introduce the languages CLP(FD) and MiniZinc, and the Gecode platform.

M. Garcia de la Banda and E. Pontelli (Eds.): ICLP 2008, LNCS 5366, pp. 744–748, 2008.

$CLP(\mathcal{D})$ is a declarative programming paradigm, first presented in 1986 (e.g., [2]). Combinatorial problems are usually encoded using constraints over *finite domains* ($\mathcal{D} = FD$), currently supported by all CLP systems based on Prolog. We focused on the library `clpfd` of SICStus Prolog [3], but what we have done can be repeated for the other CLP(FD) systems (e.g., ECLiPSe, GNU-Prolog, B-Prolog). We focus on the classical *constraint+generate* programming style. We report the fragment of the N-Queens problem where diagonal constraints are set.

```
1. safe( _,_,[]).
2. safe(X,D,[Q|Queens]) :-
3.  X + D #\= Q,  Q + D #\= X,  D1 is D + 1, safe(X,D1,Queens).
```

MiniZinc is a high-level modeling language developed by the NICTA research group [4]. It allows to express most CP problems easily, supporting sets, arrays, user defined predicates, some automatic coercions, and so on. But it is also low-level enough to be easily mapped onto existing solvers. FlatZinc [5] is a low-level solver-input language. The NICTA group provides a compiler from MiniZinc to FlatZinc that supports global constraints. The NICTA team also provide a solver that reads and executes a FlatZinc model. We report the diagonal constraints of the MiniZinc N-Queens model.

```
4. constraint
5.  forall (i in 1..n, j in i+1..n) (
6.   q[i] + i != q[j] + j /\ q[i] - i != q[j] - j; );
```

Gecode is an environment for developing constraint-based systems and applications [1]. It is implemented in C++ and offers competitive performance w.r.t. both runtime and memory usage. It implements a lot of data structures, constraints definitions and search strategies, allowing also the user to define his own ones. It is C++ like, and, thus, programmer should take care of several low-level details (there exists, however, a FlatZinc frontend). We report the extract of the N-Queens problem encoded in Gecode regarding diagonal constraints.

```
7. for (int i = 0; i<n; i++){
8.  for (int j = i+1; j<n; j++) {
9.   post(this, q[i]+i!=q[j]+j); post(this, q[i]-i!=q[j]-j);}}
```

3 Translation

The translation from SICStus and MiniZinc programs to Gecode is carried out in two stages: first we translate the high-level code into an intermediate language (CNT, defined ad-hoc) that lists explicitly all the constraints. Then we generate C++ code from CNT, using static analysis to improve the second part of the compilation.

CNT. The language CNT is used for listing the constraints and specifying some searching parameters and the output variables. It is very similar to FlatZinc, and in the next future we intend to use only FlatZinc instead of CNT. The complete CNT grammar is defined in the file `parser.y` [6]. An example of CNT code is the following:

```
10. domain [_1, _2, _3, _4], 1, 4;
11. all_different [_1, _2, _3, _4];
```

```
12. ((_1+1)!=_2); ((_2+1)!=_1); ((_1+2)!=_3); ((_3+2)!=_1);
13. ((_1+3)!=_4); ((_4+3)!=_1); ((_2+1)!=_3); ((_3+1)!=_2);
14. ((_2+2)!=_4); ((_4+2)!=_2); ((_3+1)!=_4); ((_4+1)!=_3);
```

CLP(FD) to CNT. For translating SICStus to CNT, we automatically create a new SICStus program where constraints definition is replaced by a printing stage. The execution of the modified SICStus code prints all the constraints in the CNT form, and thus generates CNT code. For instance, the code 15–18 is converted into code 19–22:

```
15. test(X,N):-
16.  length(X,N),
17.  domain(X,1,N),
18.  all_different(X).
```

```
19. test(X,N) :-
20.  length(X,N),
21.  format("domain ~q,~q,~q;\n",[X,1,N]),
22.  format("all_different ~q;\n",[X]).
```

Some problems arise when there is a unification. In fact, in some programs the logic variables are known to be FD-variables only at runtime and therefore every time in the program there is a unification we have to add some equality constraints. We developed some particular cases to cope with this and other minor technical problems.

For instance, the execution of the modified version of the SICStus N-Queens code with $N = 4$ generates the CNT code 10–14.

MiniZinc (FlatZinc) into CNT. We took advantage of the existing compiler from MiniZinc to FlatZinc [4] and thus focused on the translation from FlatZinc to CNT. Being these two languages rather similar in spirit, the translation is straightforward. For example, the FlatZinc constraints 23–25 can be defined into CNT code 26–28:

```
23. array[0 .. 2] of var 0 .. 2: v;
24. constraint int_eq(v[0], 0);
25. constraint all_different([v[0],v[1],v[2]]);
```

```
26. domain [_0,_1,_2], 0, 2;
27. (_0 == 0);
28. all_different [_0,_1,_2];
```

In principle, starting from MiniZinc one can exploit the existing "for" to obtain a more compact CNT code. Since we pass through FlatZinc, however, this information is lost. Prolog does not have "for" loops, and one could infer some of them through program analysis, but this is, in general, an undecidable problem. One solution (reasonable only from MiniZinc) could be to enrich CNT (or, better, FLatZinc) with a construct that keeps the "for" cycles information.

CNT into Gecode. We have developed a compiler from CNT to C++/Gecode. Before the compilation we perform some simplifying transformation on the CNT code (e.g., precomputation of numerical expressions). Moreover, using static analysis, the compiler groups the constraints that can be defined within a `for` cycle to reduce the size of the final C++ program. This can lead to a dramatic reduction of time needed by Gecode for the compilation of the .cc file with its libraries. For instance, the Gecode file obtained by the instance 100-Queens with this optimization is compiled with the Gecode libraries in 5.8s, while the code obtained by "flat" CNT requires 13 hours and 40 minutes.

The translation is performed using the following tools: for SICStus to CNT: SICStus; FlatZinc to CNT: gcc, Bison and Flex; for CNT to Gecode we used Haskell tools.

4 Experimental Results

We considered instances of four well-known problems, i.e. N-Queens, Sudoku, Golomb Rulers, and Knapsack. Sudoku 16x16 instances are taken from `www.live-sudoku.com/play-online/geant`, and 25x25 ones are taken from `www.eleves.ens.fr/home/frisch/sudoku.html`; N-Queens instances range from $N = 100$ to $N = 115$; Golomb Rulers instances have order from 6 to 13, with two different lengths (the biggest satisfiable and the shorter unsatisfiable) for each order; knapsack instances are the same used in [7].

We modeled each problem in SICStus Prolog, MiniZinc, and Gecode. When available, we used the modeling offered by languages libraries. We also considered their translation with the tools described in the paper. All codes and instances, together with all the running times, are available at [6].

There are two kinds of compile times: time of the compilation from high level code to Gecode C++ file and time needed by Gecode for internal compilation and libraries linking. With the proposed automatic detection of "`for`" loops both of them are rather low (the order of some seconds). Of course, for some small instances this time cannot be ignored w.r.t. execution time, but it becomes negligible for difficult instances.

There is a wide variety of results, that it still has to be investigated. However, the following general considerations can be done.

Native Gecode code is always the fastest, save for some 25x25 Sudoku instances.

Gecode models obtained compiling SICStus prolog models often speeds-up SICStus native, save for some instances of N-Queens. Moreover it has, in average, comparable times with Gecode native code. Let us observe, however, that it solves all the Sudoku instances, while native Gecode does not.

Gecode models obtained compiling SICStus and MiniZinc have substantially the same behavior, while in average they outperforms the MiniZinc to FlatZinc models obtained by the tools provided by NICTA and Gecode team, which are the standard ways to run MiniZinc (FlatZinc) models. Precisely, for Sudoku, N-Queens and Golomb rulers the Gecode encoding obtained from Minizinc with our tool is faster than the ones obtained with NICTA tools of various order of magnitude, while it is slightly slower in the case of Knapsack.

We think that the performances of the Gecode models obtained by our translations are better than the NICTA and Gecode ones because of our precomputations and static analysis (see section 3), that simplify domains and constraints, w.r.t the execution of the flatzinc models. However, the utility provided by NICTA research group and the one of the Gecode Team directly execute the flatzinc files, without returning any intermediate encoding, so we can't perform an exhaustive comparison of the encodings.

5 Conclusion and Future Work

Our compiler from SICStus and MiniZinc to Gecode, although in its preliminary version, shows that Constraint Programming can be done at high level using well-known languages and then executed in new paradigms since running times are comparable w.r.t. those of this latter paradigm. This allows to benefit from both the flexibility of high-level modeling languages and the efficiency of the new low-level solvers.

We would like to improve the static analysis of the generated code to further speed-up the overall process (compilation+execution) and to extend its compiling mechanism (up to now limited to CSP). We also would like to discard the use of CNT and write a front-end from CLP(FD) to FlatZinc to take advantage of FlatZinc solvers, and to port our tools to the Gecode 2.1.1.

The present work is part of a more general project of developing a programming tool for combinatorial problems, made of three main parts: the *modeling part*, where the user will define in a high-level style the problem to solve and the algorithm to use (e.g., constraint programming search, eventually interleaved with local search, heuristics or meta-heuristics phases); the *translating part*, where the model and the meta-algorithm defined will be automatically compiled into the solver languages, e.g. Gecode; and the *solving part*, where the overall compiled program will be run and the various solvers will interact in the way specified, to find the solution of the problem modeled.

We are planning to test the tool on different families of problems including those we have already cope with: a hospital rostering (timetable) problem [8], the protein structure prediction problem [9], and planning problems [10].

Acknowledgements. The work is partially supported by MUR FIRB RBNE03B8KK.

References

1. Team, G.: Gecode: Generic Constraint Development Environment, `http://www.gecode.org`
2. Jaffar, J., Maher, M.J.: Constraint Logic Programming: A survey. Journal of Logic Programming (19/20), 503–581 (1994)
3. Carlsson, M., Ottosson, G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver, 191–206
4. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a Standard CP Modelling Language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
5. Nethercote, N.: Specification of FlatZinc, `http://www.g12.cs.mu.oz.au/minizinc/flatzinc-spec.pdf`
6. Cipriano, R., Dovier, A., Jacopo, M.: Tools for Compiling SICStus and Minizinc in Gecode, `http://www.dimi.uniud.it/dovier/MISIGE/`
7. Dovier, A., Formisano, A., Pontelli, E.: A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 67–82. Springer, Heidelberg (2005)
8. Cipriano, R., Di Gaspero, L., Dovier, A.: Hybrid Approaches for Rostering: a Case Study in The Integration of Constraint Programming and Local Search. In: Almeida, F., Blesa Aguilera, M.J., Blum, C., Moreno Vega, J.M., Pérez Pérez, M., Roli, A., Sampels, M. (eds.) HM 2006. LNCS, vol. 4030, pp. 110–123. Springer, Heidelberg (2006)
9. Cipriano, R., Dal Palù, A., Dovier, A.: A Hybrid Approach Mixing Local Search and Constraint Programming Applied to the Protein Structure Prediction Problem. In: WCB 2008, Paris (2008)
10. Dovier, A., Formisano, A., Pontelli, E.: Multivalued Action Languages with Constraints in CLP(FD). In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 255–270. Springer, Heidelberg (2007)