Schema-Guided Synthesis of CLP Programs *

Hamza Zidoum[1], Pierre Flener[2], and Brahim Hnich[3]

[1] UAE University, PO Box 15551, Al-Aïn, United Arab Emirates
[2] Dept of Info Science, Uppsala Univ., S-751 05 Uppsala, Sweden, pierref@csd.uu.se
[3] Dept of Info Technology, Tampere Univ. of Technology, SF-33101 Tampere, Finland

1 Introduction and Specifications

This work is inspired by D.R. Smith's research on synthesising global search (GS) programs (in the *Refine* language) from first-order logic specifications (also in *Refine*) [8–10]. We concentrate on synthesising constraint logic programs (CLP) [6] instead. We thus only have to synthesise code that (incrementally) *poses* the constraints, because the actual constraint propagation and pruning are performed by the CLP system. We here only tackle the family of decision assignment problems; the families of optimisation assignment problems, decision permutation problems, and optimisation permutation problems are covered in [4].

Specifications are the input to program synthesis. In *decision assignment problems*, a mapping M from a list V into the integer interval $1..W$ has to be found, satisfying certain constraints. Their formal specifications take the form

$$\begin{array}{c} \forall \langle V, W\rangle : list(term) \times int \,.\, \forall M : list(V \times 1..W) \,. \\ r(\langle V, W\rangle, M) \leftrightarrow \forall \langle I, J\rangle, \langle K, L\rangle \in M \,.\; \wedge_{i=1}^{m} P_i(I, J, K, L) \rightarrow Q_i(I, J, K, L) \end{array} \quad (1)$$

where the P_i and Q_i are formulas. This can be considered a *specification template*. This covers many problems, such as Hamiltonian path, n-Queens, and *graph colouring*, which problem consists of finding a mapping M from the list R of the regions of a map to a set of colours (numbered $1..C$) so that any two adjacent regions (as indicated in an adjacency list A) have different colours:

$$\begin{array}{c} \forall \langle R, C, A\rangle : list(term) \times int \times list(R \times R) \,.\, \forall M : list(R \times 1..C) \,. \\ col(\langle R, C, A\rangle, M) \leftrightarrow \forall \langle R_1, C_1\rangle, \langle R_2, C_2\rangle \in M \,.\, \langle R_1, R_2\rangle \in A \rightarrow C_1 \neq C_2 \end{array} \quad (2)$$

2 A Global Search Program Schema for CLP

A *program schema* [3] for a programming methodology M (such as divide-and-conquer, generate-and-test, ...) is a couple $\langle T, A\rangle$, where *template* T is an open program showing the (problem-independent) data-flow and control-flow of programs constructed following M, and *axioms* A constrain the (problem-dependent) programs for the open relations in T such that the overall (closed) program will really be a program constructed following M.

* A full version of this extended abstract is published as [4].

The basic idea of our GS schema for CLP is to start from an *initialised* descriptor of the search space, to incrementally *split* that space into subspaces, while declaring the domains of the involved variables and *constraining* them to achieve partial consistency, until no splits are possible and a variablised solution can be *extracted*. Then a correct solution is *generated*, by instantiation of the variables in the variablised solution. Our GS template is thus the open program:

$$\begin{array}{rl} r(X,Y) \leftarrow & initialise(X,D), \\ & rgs(X,D,Y), \\ & generate(Y,X) \\ rgs(X,D,Y) \leftarrow & extract(X,D,Y) \\ rgs(X,D,Y) \leftarrow & split(D,X,D',\delta), \\ & constrain(\delta,D,X), \\ & rgs(X,D',Y) \end{array} \qquad (GS)$$

where the open relations are informally specified as follows: $initialise(X,D)$ iff D is the descriptor of the initial space of candidate solutions to problem X; $extract(X,D,Y)$ iff the variablised solution Y to problem X is directly extracted from descriptor D; $split(D,X,D',\delta)$ iff descriptor D' describes a subspace of D wrt problem X, such that D' is obtained by adding δ to descriptor D; $constrain(\delta,D,X)$ iff adding δ to D leads to a descriptor defining a subspace of D that may contain correct solutions to problem X; $generate(Y,X)$ iff correct solution Y to problem X is enumerated from the constraint store, which is an implicit parameter representing X. Formalising this is the role of the axioms, as shown in [4]. There we also establish in what sense our GS schema is correct.

3 Schema Particularisations and the Synthesis Method

Using the GS schema like the divide-and-conquer schema was used in [7, 3] to guide synthesis puts heavy demands on automated theorem proving and turns out much more difficult [8]. Fortunately, a large percentage of GS programs falls into one of 7 families identified by Smith, each representing a particular case of the GS schema (in the sense that programs for *all* its open relations are adequately chosen in advance), here called a *particularisation*. In [4], we exhibit our particularisations for assignment and permutation problems, and show how to implement them as programs, called *closures*, because they "close" the open program GS. We also define the notion of *specification reduction*, expressing when it suffices to invoke a program P_g of specification S_g to implement a new specification S_r. The *synthesis method* is then as follows: Given a specification S_r, find (through a linear subcase of the decidable second-order semi-unification) a substitution θ under which S_r reduces to the specification template S_g attached to some particularisation P_g of the GS schema, and then obtain a (closed) program that correctly implements S_r by taking the GS template and $C_g\theta$.

Example Given specification (2), the fully automatically synthesisable program consists of the GS template and the following code:

$$
\begin{aligned}
P_{init} : & \mathit{initialise}(X, D) \leftarrow \\
& \quad D = \langle V, [\,] \rangle \\
P_{extr} : & \mathit{extract}(\_, D, Y) \leftarrow \\
& \quad D = \langle [\,], Y \rangle \\
P_{split} : & \mathit{split}(D, X, D', \delta) \leftarrow \\
& \quad D = \langle [I|T], M \rangle, \\
& \quad J \ \mathit{in}\ 1..W, \\
& \quad \delta = \langle I, J \rangle, \\
& \quad D' = \langle T, [\delta|M] \rangle \\
P_{constr} : & \mathit{constrain}(\_, D, \_) \leftarrow \\
& \quad D = \langle \_, [\,] \rangle \\
& \mathit{constrain}(\delta, D, \langle \_, \_, A \rangle) \leftarrow \\
& \quad \delta = \langle R_1, C_1 \rangle, \\
& \quad D = \langle \_, [\langle R_2, C_2 \rangle | M'] \rangle, \\
& \quad \langle R_1, R_2 \rangle \in A \rightarrow C_1 \neq C_2, \\
& \quad \mathit{constrain}(\delta, \langle \_, M' \rangle, \langle \_, \_, A \rangle) \\
P_{gen} : & \mathit{generate}(M, \_) \leftarrow \\
& \quad M = [\,] \\
& \mathit{generate}(M, \_) \leftarrow \\
& \quad M = [\langle \_, J \rangle | M'], \\
& \quad \mathit{indomain}(J), \\
& \quad \mathit{generate}(M', \_)
\end{aligned}
$$

4 Conclusion

At least one order of magnitude is gained in efficiency by switching from an ordinary symbolic language to a constraint one, and our automatically synthesisable CLP(FD) [2] programs are only 3 to 5 times slower than carefully hand-crafted ones [2], which is encouraging since none of the obvious problem-specific optimising transformations have been performed yet on our programs. Since our synthesis is fully automatable, starting from short and elegant formal specifications (which can even be generated from some form of controlled English [5]), our approach seems viable. Our formal specification language is equivalent in its expressiveness to CLP(Sets) programming languages, such as *CLPS* [1]. We thus aim at new ways of compiling CLP(Sets) programs. Comparing execution times is still meaningless because of the prototypical nature of CLP(Sets) compilers.

The synthesised programs are not small (minimum 33 atoms, in a very expressive programming language), and making them steadfast reusable components for a programming-in-the-large approach by embedding their whole development in a framework-based approach [3] is straightforward.

Our results are more than a simple transcription of the KIDS approach from *Refine* to CLP, as they also reflect some new ideas.

First, we fully exploited CLP [as opposed to *Refine*, which is "only" an ordinary symbolic language], by significantly modifying the original GS schema, so

that it reflects a *constrain-and-generate* programming methodology. Much of the constraint solving machinery that needs to be pushed into *Refine programs*, at synthesis time or at optimisation time, is already part of the CLP(FD) *language* and is implemented there once and for all in a particularly efficient way.

Second, we introduced the notion of specification template. This has nice effects on the KIDS approach, as shown in the next two items.

Third, the substitution under which a specification reduces to a specification template can be easily computed, so that there is no need of an automated theorem prover, at synthesis time, to compute it.

Fourth, the derivation of consistency-constraint-posing code can be calling-context-dependent, leading to rather effective (namely incremental) constraint-posing code [in contrast to Smith's calling-context-independent derivation of filters [8, 9] and cuts [10], which may be non-incremental]. Such code can even be pre-synthesised, for a given particularisation, so that there is no need of an automated theorem prover, at synthesis time, to derive its specification. [As opposed to filters and cuts] no forward constraints need to be posed, not even for efficiency reasons, due to the way CLP programs work [as opposed to *Refine* ones]: Solution construction (through *generate*) actually only starts in CLP once *all* constraints have been posed, and posing any forward constraints would thus be not only superfluous but also a way of slowing down the program, because the forward constraints of time t will become backward constraints at times larger than t and all constraints would thus have been posed twice. This does not prevent CLP from performing forward checks during solution generation.

All this means that synthesis can be fully automatic, without using any automated theorem prover, for some families of problems.

References

1. F. Ambert, B. Legeard, et E. Legros. Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis. *TSI* 15(3):297–328, 1996.
2. D. Diaz and Ph. Codognet. A minimal extension of the WAM for clp(FD). In: D.S. Warren (ed), *Proc. of ICLP'93*, pp. 774–790. The MIT Press, 1993.
3. P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. *Proc. of ASE'97*, pp. 153–160. IEEE Computer Society Press, 1997.
4. P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of constraint logic programs. *Proc. of ASE'98*. IEEE Computer Society Press, 1998.
5. N.E. Fuchs and U. Schwertel. Attempto Controlled English — Not just another logic specification language. This volume.
6. J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. of Logic Programming* 19–20:503–582, 1994.
7. D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.
8. D.R. Smith. The structure and design of global search algorithms. TR KES.U.87.12, Kestrel Institute, 1988.
9. D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Software Engineering* 16(9):1024–1043, 1990.
10. D.R. Smith. Towards the synthesis of constraint propagation algorithms. In: Y. Deville (ed), *Proc. of LOPSTR'93*, pp. 1–9, Springer-Verlag, 1994.