Schema-Guided Synthesis of Constraint Logic Programs

Pierre Flener
Dept of Information Science
Uppsala University
S-75105 Uppsala, Sweden
pf@csd.uu.se

Hamza Zidoum
UAE University
P.O. Box 15551
Al-Ain
United Arab Emirates

Brahim Hnich
Dept of Information Technology
Tampere University of Technology
SF-33101 Tampere
Finland

Abstract

By focusing on the families of assignment and permutation problems (such as graph colouring and n-Queens), we show how to adapt D.R. Smith's KIDS *approach for the synthesis of* constraint *programs (with implicit constraint satisfaction code thus), rather than applicative* Refine *programs with explicit constraint propagation and pruning code. Synthesis is guided by a global search schema and can be fully automated with little effort, due to some innovative ideas. CLP(Sets) programs are equivalent in expressiveness to our input specifications. The synthesised CLP(FD) programs would be, after optimising transformations, competitive with carefully hand-crafted ones.*

1. Introduction

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*) [12, 13, 14]. The basic idea of GS is to represent and manipulate sets of candidate solutions. Starting from an initial set that contains all solutions to the given problem, a GS program incrementally *extracts* solutions from a set, *splits* sets into subsets, eliminates sets via *filters*, and *cuts* sets, until no set remains to be split.

Instead of synthesising *Refine* programs, our work concentrates on synthesising constraint (logic) programs. Constraint Logic Programming (CLP) [8] is a paradigm especially suited for solving combinatorial problems, thanks to its double reasoning: symbolic reasoning expresses the logic properties of the problem, while constraint satisfaction reasoning (over several computational domains, such as reals, booleans, finite domains, sets, ...) uses constraint propagation to keep the search space manageable. 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.

Search problems can be classified into *decision problems*, which consist in finding some correct solution, and *optimisation problems*, which consist in finding an optimal correct solution given a cost function, and are thus an extension of decision problems.

Very few works deal with the synthesis and transformation of CLP programs. In [9], the possibility of synthesising steadfast CLP programs is shown, without exhibiting a synthesis method, though. A manual and informal method for constructing CLP programs from specifications is given in [3]. We here outline a completely automatic and formal method for synthesis, and leave optimising transformations for future work.

Schema-guided synthesis of CLP programs is also based on a GS schema. We use particular cases of that general schema to simultaneously instantiate all its place-holders. Although we are still working on it, we think that the number of these particular cases will be small (but probably more than the seven of KIDS [12, 13, 14]). We here only tackle the families of assignment and permutation problems.

This paper is organised as follows. Section 2 defines specifications as inputs to synthesis and discusses their forms for assignment and permutation problems. Section 3 introduces our GS schema for CLP programs. Section 4 defines particularisations as particular cases of a schema and exhibits particularisations of our GS schema for assignment and permutation problems. Section 5 defines when a specification reduces to another one, so that a program for the latter can be re-used towards implementing the former. Section 6 introduces a synthesis method guided by our GS schema. Section 7 contains benchmarks establishing the viability of our approach. Section 8 argues why our work is more than a transposition of Smith's results from *Refine* to CLP.

2. Specifications

Specifications are the input to program synthesis. In order to enable (or facilitate) *automated* synthesis, such inputs ought to be formal (though it would then be more adequate to say that the inputs are programs and that synthesis is compilation [10]). Without loss of generality, we only consider minimisation problems.

Definition 2.1 A *specification* of a program for a relation r is a first-order logic formula of the form:

$$\begin{array}{c}\forall X : \mathcal{X} \,.\, \forall Y : \mathcal{Y} \,.\, \forall W : \mathcal{W} \,. \\ I_r(X) \rightarrow (r(X,Y,W) \leftrightarrow O_r(X,Y,W))\end{array} \qquad (S_r)$$

where $X : \mathcal{X}$, $Y : \mathcal{Y}$, and $W : \mathcal{W}$ are (possibly empty) lists of sorted variables. Formula I_r is called the *input condition*, constraining the input domain $\mathcal{X}$, whereas O_r is called the *output condition*, describing when some output Y (of cost W) is a correct (optimal) solution for input (or problem) X. Usually O_r has the form

$$Objective(X,Y,W) \wedge minimal(Solution(X,Y),W)$$

where $Solution(X,Y)$ expresses that Y is a correct solution to problem X, and $Objective(X,Y,W)$ expresses that solution Y to problem X has cost W. The specification primitive $minimal(Solution(X,Y),W)$ expresses that W is the minimal cost of all correct solutions Y (according to $Solution$) to problem X.

To simplify some formulas, we consider I_r to be part of the definition of $\mathcal{X}$. Often, we then simply designate specifications by $\langle \mathcal{X}, \mathcal{Y}, \mathcal{W}, O_r \rangle$ tuples.

We distinguish two families of problems, namely assignment problems and permutation problems.

2.1. Specifications of Assignment Problems

We first consider the family of *decision assignment problems*, where a mapping M from a list V into the integer interval $1..W$ has to be found, satisfying certain constraints. Their specifications S^{dec}_{ass} take the form $\langle list(term) \times int, list(V \times 1..W), O^{dec}_{ass} \rangle$, with:

$$\begin{array}{c}\forall \langle I,J \rangle, \langle K,L \rangle \in M \,. \\ \wedge^{m}_{i=1} P_i(I,J,K,L) \rightarrow Q_i(I,J,K,L)\end{array} \qquad (O^{dec}_{ass})$$

where the P_i and Q_i are formulas. This can be considered a *specification template*; others are given below. This covers many problems, such as graph colouring (see below), Hamiltonian path, n-Queens, etc.

Example 2.1 Given a map, the *graph colouring* problem consists of finding a mapping M from the list R of its regions to a set of colours (numbered $1..C$) so that any two adjacent regions (as indicated in an adjacency list A) have different colours. Formally:

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

where $\in$ is a primitive (with the usual meaning).

In *optimisation assignment problems*, a mapping M from a list V into the integer interval $1..W$ has to be found, satisfying certain constraints and minimising W. Their specifications S^{opt}_{ass} take the form $\langle list(term), list(V \times 1..W), int, O^{opt}_{ass} \rangle$, with:

$$\begin{array}{c}\forall \langle I,J \rangle, \langle K,L \rangle \in M \,. \\ \wedge^{m}_{i=1} P_i(I,J,K,L) \rightarrow Q_i(I,J,K,L) \\ \wedge\, W = max\{U | \langle \_,U \rangle \in M\} \wedge minimal(W,V)\end{array} \qquad (O^{opt}_{ass})$$

where the specification primitive $max\ \mathcal{S}$ returns the maximal element in number set $\mathcal{S}$. This also covers many problems, such as optimal k-graph colouring, optimal Hamiltonian path, etc.

2.2. Specifications of Permutation Problems

We also consider the family of *decision permutation problems*, where a permutation S of the interval $1..N$ has to be found, satisfying certain constraints. Their specifications S^{dec}_{perm} take the form $\langle int, list(1..N), O^{dec}_{perm} \rangle$, with:

$$perm(N,S) \wedge \forall V_1,V_2 \,.\, P(V_1,V_2,S) \rightarrow Q(V_1,V_2) \qquad (O^{dec}_{perm})$$

where specification primitive $perm(U,V)$ holds iff list V is a permutation of the interval $1..U$; atomic formula P involves either $anyTwo(V_1,V_2,S,P_1,P_2)$ (V_1 and V_2 occur respectively at positions P_1 and P_2 in list S), or $consTwo(V_1,V_2,S)$ (V_1 immediately precedes V_2 in list S), or $precTwo(V_1,V_2,S)$ (V_1 precedes V_2 in list S), and Q is a formula. This covers many problems, such as Hamiltonian path, job scheduling, n-Queens, etc. (The big overlap with assignment problems happens because permutation problems essentially are assignment problems with a bijectiveness constraint on the assignment. The corresponding algorithms will be quite different, though.)

Finally, we consider *optimisation permutation problems*, where an optimal permutation S of the interval $1..N$ has to be found, satisfying certain constraints and optimising a certain cost W of S. Their specifications S^{opt}_{perm} take the form $\langle int, list(1..N), int, O^{opt}_{perm} \rangle$, with:

$$\begin{array}{c}perm(N,S) \wedge \forall V_1,V_2 \,.\, P(V_1,V_2,S) \rightarrow Q(V_1,V_2) \\ \wedge\, W = F\{E | consTwo(V_1,V_2,S) \wedge Q(V_1,V_2)\} \\ \wedge\, minimal(W,N)\end{array} \qquad (O^{opt}_{perm})$$

where function F is either sum or $product$, with specification primitive $sum\ \mathcal{S}$ ($product\ \mathcal{S}$) returning the sum (product) of all elements in number set $\mathcal{S}$. Formula Q must have E as a free variable. This also covers many problems, such as optimal Hamiltonian path (see below), optimal job scheduling, etc.

Example 2.2 Given C cities, the *optimal Hamiltonian path* problem consists of finding a permutation H of the interval $1..C$ that minimises the total distance D of visiting the cities as ordered in H, given an adjacency list A of triples $\langle C_1, C_2, E\rangle$ indicating that the distance between adjacent cities C_1 and C_2 is E. Formally:

$$\begin{gathered}\forall\langle C, A\rangle : int \times list(1..C \times 1..C \times int)\,.\,\forall H : list(1..C)\,.\\ \forall D : int\,.\,hamPath(\langle C,A\rangle, H, D) \leftrightarrow perm(C,H)\\ \wedge\,\forall C_1, C_2\,.\,consTwo(C_1,C_2,H) \rightarrow \langle C_1, C_2, \_\rangle \in A\\ \wedge\, D = sum\{E \mid consTwo(C_1,C_2,H) \wedge \langle C_1,C_2,E\rangle \in A\}\\ \wedge\, minimal(D, \langle C, A\rangle)\end{gathered} \tag{S_{ham}^{opt}}$$

3. A Global Search Program Schema for CLP

A *program schema* [4] 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. (An *open program* is a program in which at least one non-primitive (relation or function) symbol, called an *open symbol*, is undefined; a *closed program* is a program without any open symbols.)

We now formalise our global search (GS) schema for CLP programs. The basic idea is to start from an *initialised* descriptor of the search space, to incrementally *split* that space into sub-spaces, 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 (optimal) solution is *generated*, by instantiation of the variables in the variablised solution. Compared to Smith's GS schema, ours only computes one correct (optimal) solution rather than all, because this is standard practice in CLP. In any case, all solutions can easily be obtained in CLP, due to its built-in backtracking.

3.1. The Global Search Template

Our global search template is the open program:

$$\begin{array}{rcl} r(X,Y,W) & \leftarrow & initialise(X,D),\\ && rgs(X,D,Y),\\ && objective(X,Y,W),\\ && minof(generate(Y,X),W)\\ rgs(X,D,Y) & \leftarrow & extract(X,D,Y)\\ rgs(X,D,Y) & \leftarrow & split(D,X,D',\delta),\\ && constrain(\delta,D,X),\\ && rgs(X,D',Y)\end{array} \tag{GS_{opt}}$$

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 sub-space of D wrt problem X, such that D' is obtained by adding δ to descriptor D;
- $constrain(\delta,D,X)$ iff adding δ to descriptor D leads to a descriptor defining a sub-space of D that may contain correct (optimal) solutions to problem X;
- $objective(X,Y,W)$ iff arithmetic expression W is the cost of correct solution Y to problem X;
- $generate(Y,X)$ iff correct (optimal) solution Y to problem X is enumerated (by instantiations in the initially variablised solution Y) from the constraint store, which is an implicit parameter representing X.

The CLP primitive $minof(generate(Y,X),W)$ holds iff W is the minimal cost of all correct solutions Y enumerated by $generate$ for problem X. Formalising these informal specifications is the role of the axioms, shown below.

For decision problems, GS_{opt} specialises to:

$$\begin{array}{rcl} r(X,Y) & \leftarrow & initialise(X,D),\\ && rgs(X,D,Y),\\ && generate(Y,X)\\ rgs(X,D,Y) & \leftarrow & extract(X,D,Y)\\ rgs(X,D,Y) & \leftarrow & split(D,X,D',\delta),\\ && constrain(\delta,D,X),\\ && rgs(X,D',Y)\end{array} \tag{GS_{dec}}$$

but we (mostly) continue with the general version.

3.2. The Global Search Axioms

Let $\mathcal{D}$ be the type of search space descriptors, and Δ be the type of the elements of the partial solutions stored in descriptors. The first axioms are the specifications of the open relations of the GS_{opt} template:

$$\begin{gathered}\forall X : \mathcal{X}\,.\,\forall D : \mathcal{D}\,.\\ initialise(X,D) \leftrightarrow O_{init}(X,D)\end{gathered} \tag{S_{init}}$$

$$\begin{gathered}\forall X : \mathcal{X}\,.\,\forall D : \mathcal{D}\,.\,\forall Y : \mathcal{Y}\,.\\ extract(X,D,Y) \leftrightarrow O_{extr}(X,D,Y)\end{gathered} \tag{S_{extr}}$$

$$\begin{gathered}\forall D, D' : \mathcal{D}\,.\,\forall X : \mathcal{X}\,.\,\forall \delta : \Delta\,.\\ split(D,X,D',\delta) \leftrightarrow O_{split}(D,X,D',\delta)\end{gathered} \tag{S_{split}}$$

$$\begin{gathered}\forall \delta : \Delta\,.\,\forall D : \mathcal{D}\,.\,\forall X : \mathcal{X}\,.\\ constrain(\delta,D,X) \leftrightarrow O_{constr}(\delta,D,X)\end{gathered} \tag{S_{constr}}$$

$$\forall X : \mathcal{X} \,.\, \forall Y : \mathcal{Y} \,.\, \forall W : \mathcal{W} \,. \quad objective(X, Y, W) \leftrightarrow Objective(X, Y, W) \tag{S_{obj}}$$

$$\forall Y : \mathcal{Y} \,.\, \forall X : \mathcal{X} \,. \quad generate(Y, X) \leftrightarrow Solution(X, Y) \tag{S_{gen}}$$

The output conditions of some of these specifications are constrained by the next axioms. The output conditions of the other specifications are directly made of parts of the output condition O_r.

Second, the following axiom expresses that all correct solutions Y to problem X are contained in the computed initial space for X:

$$\forall X : \mathcal{X} \,.\, \forall Y : \mathcal{Y} \,.\, Solution(X, Y) \rightarrow \exists D : \mathcal{D} \,.\, O_{init}(X, D) \wedge satisfies(Y, D) \tag{A_1}$$

where $satisfies(Y, D)$ means that (possibly variablised) solution Y is in the space described by descriptor D, which is the case if Y can be extracted after a finite number of applications of $split$ to D. Formally:

$$\begin{array}{l}
\forall X : \mathcal{X} \,.\, \forall Y : \mathcal{Y} \,.\, \forall D : \mathcal{D} \,. \\
\quad satisfies(Y, D) \leftrightarrow \exists k : int \,.\, \exists D' : \mathcal{D} \,.\, \exists \delta : \Delta \,. \\
\qquad split^k(D, X, D', \delta) \wedge O_{extr}(X, D, Y) \\
\text{where :} \\
\quad split^0(D, X, D', \delta) \leftrightarrow D = D' \\
\text{and, for all } k : int : \\
\quad split^{k+1}(D, X, D', \delta) \leftrightarrow \exists D'' : \mathcal{D} \,.\, \exists \delta' : \Delta \,. \\
\qquad O_{split}(D, X, D'', \delta') \wedge split^k(D'', X, D', \delta)
\end{array} \tag{A_2}$$

Finally, we want to fully exploit CLP features to eliminate spaces from further consideration. Constraint satisfaction can be used to prune off branches of the search tree that cannot yield solutions. Given a space described by D and a (possibly still variablised) solution Y to problem X, if splitting D into D' makes D' contain the solution Y, then $constrain$ must succeed. Formally:

$$\begin{array}{l}
\forall X : \mathcal{X} \,.\, \forall Y : \mathcal{Y} \,.\, \forall D, D' : \mathcal{D} \,.\, \forall \delta : \Delta \,. \\
\quad Solution(X, Y) \wedge O_{split}(D, X, D', \delta) \\
\quad \wedge\, satisfies(Y, D') \rightarrow O_{constr}(\delta, D, X)
\end{array} \tag{A_3}$$

Conversely, the contrapositive of A_3 shows that if $constrain$ fails, then the new space described by D' (which is D plus δ) does not contain any solution to X. CLP languages contain the SAT decision procedure, checking whether a constraint store is satisfiable [8].

This last axiom sets up a necessary condition that $constrain$ must establish. Given the left-hand side of the implication, such a condition can be derived using automated theorem proving (ATP), as shown in [11, 12]. Of course, we are not interested in too weak such a condition, such as the trivial solution $true$, but rather in a stronger one. However, deriving the absolutely strongest one (which establishes equivalence rather than implication) is impractical, because finding it may take too much time or may even turn out to be beyond current ATP possibilities, and because such a perfect $constrain$ would be too expensive to evaluate (since it would eliminate all backtracking in the solution generation). So we should (automatically, if possible) derive the strongest "possible and reasonable" condition, the criteria for these qualities being rather subjective. Fortunately, for the families of assignment and permutation problems, it turns out that this condition can be easily manually precomputed (see below) at schema-design time, for *any* such problems, in an optimal way, so that no ATP technology is then necessary at synthesis time!

The derivation of the output condition of $constrain$ depends on its calling context, namely that it is invoked after $split$: this gives rise to rather effective (namely incremental) constraint-posing code [and stands in contrast to Smith's calling-context-independent derivation of filters [12, 13] and cuts [14], which thus may be non-incremental]. (Sentences between [...] are for understanding the differences with Smith's work.) Notice that $constrain$ just *poses* constraints on the search space, the actual solutions being *enumerated* by $generate$ once *all* constraints have been posed, because we use a constraint language.

3.3. Correctness of the Global Search Schema

Now we define a notion of correctness, and establish that our GS schema is correct.

Definition 3.1 A closed program P_r for a relation r is *totally correct* wrt its specification $\langle \mathcal{X}, \mathcal{Y}, \mathcal{W}, O_r \rangle$ if for all $X : \mathcal{X}$, $Y : \mathcal{Y}$, and $W : \mathcal{W}$ we have that $O_r(X, Y, W)$ iff $P_r \vdash r(X, Y, W)$.

This can be generalised to open programs, the correctness criterion being then called *steadfastness* [4].

Theorem 3.1 Given a specification S_r for a relation r, any closed program $GS_{opt} \cup P_{init} \cup P_{extr} \cup P_{split} \cup P_{constr} \cup P_{obj} \cup P_{gen}$ such that P_{init}, P_{extr}, P_{split}, P_{constr}, P_{obj}, P_{gen} are totally correct wrt S_{init}, S_{extr}, S_{split}, S_{constr}, S_{obj}, S_{gen}, respectively, and such that the axioms A_1 to A_3 hold, is totally correct wrt S_r.

Proof. Outline (analogous to [12]):
Let P_r be the first clause of GS_{opt}, and let P_{rgs} be the remaining two clauses of GS_{opt}. First, prove that P_{rgs} is steadfast wrt the specification

$$\forall X : \mathcal{X} \,.\, \forall D : \mathcal{D} \,.\, \forall Y : \mathcal{Y} \,.\, rgs(X, D, Y) \leftrightarrow satisfies(Y, D) \wedge Solution(X, Y) \tag{S_{rgs}}$$

and the other axioms of the GS schema. Second, prove that P_r is steadfast wrt to S_r and S_{rgs}. □

4. Schema Particularisations

In theory, one could use the global search (GS) schema in a way analogous to the way the divide-and-conquer schema was used in [11, 4] to guide synthesis, namely by following a *strategy* of (a) arbitrarily choosing programs for *some* of the open relations (satisfying the axioms of course) from a pool of frequently used such programs, (b) propagating their concrete specifications across the axioms to set up concrete specifications for the remaining open relations, (c) calling a (schema-guided) synthesiser to generate programs from these specifications, and (d) assembling the overall synthesised program from the template, the chosen programs, and the generated programs. However, in general this puts heavy demands on ATP technology, and in particular this turns out much more difficult for the GS schema than for the divide-and-conquer one [12]. Fortunately, a very large percentage of GS programs falls into one of seven families identified by Smith, each representing a particular case of the global search schema (in the sense that programs for *all* its open relations are adequately chosen in advance), here called a *particularisation*. We here investigate the families of assignment and permutation problems, other families enumerating sublists of (given or bounded) length k over a given list, enumerating sequences over a given list, etc [12].

Definition 4.1 A *particularisation* of the GS schema is a set of formulas defining $\mathcal{D}$, Δ, $satisfies$, O_{init}, O_{extr}, O_{split}, O_{constr}, such that axioms A_1 to A_3 are satisfied.

We now discuss a few sample particularisations.

4.1. Particularisations for Assignment Problems

The formulas below, denoted by P^{dec}_{ass}, constitute a particularisation of the GS schema for *decision assignment* problems. It enumerates mappings from a list V into an interval $1..W$, where the problem tuple X has the form $\langle V, W, \ldots \rangle$. Descriptors take the form $\langle T, M \rangle$, and the idea is to gradually build up the (initially empty) mapping M (represented as a list of pairs), with a sublist of V as domain and $1..W$ as range, such that list T has the elements of V that have not been mapped to elements in $1..W$ yet. Formally:

$$\mathcal{D} = \{\langle T, M \rangle | T \subseteq V \wedge M \in list((V \setminus T) \times 1..W)\}$$

$$\Delta = \{\langle I, J \rangle | I \in V \wedge J \in 1..W\} = V \times 1..W$$

$$\begin{array}{l} \forall Y : \mathcal{Y} \,.\, \forall D : \mathcal{D} \,.\, satisfies(Y, D) \leftrightarrow \exists M : \mathcal{Y} \,. \\ \quad D = \langle \_, M \rangle \wedge \forall \langle I, J \rangle \in M \,.\, \langle I, J \rangle \in Y \end{array}$$

$$\forall X : \mathcal{X} \,.\, \forall D : \mathcal{D} \,.\, O_{init}(X, D) \leftrightarrow D = \langle V, [\,] \rangle$$

$$\begin{array}{c} \forall X : \mathcal{X} \,.\, \forall D : \mathcal{D} \,.\, \forall Y : \mathcal{Y} \,. \\ O_{extr}(X, D, Y) \leftrightarrow D = \langle [\,], Y \rangle \end{array}$$

$$\begin{array}{c} \forall D, D' : \mathcal{D} \,.\, \forall X : \mathcal{X} \,.\, \forall \delta : \Delta \,. \\ O_{split}(D, X, D', \delta) \leftrightarrow D = \langle [I|T], M \rangle \\ \wedge\ J\ in\ 1..W \wedge \delta = \langle I, J \rangle \wedge D' = \langle T, [\delta|M] \rangle \end{array}$$

$$\begin{array}{c} \forall \langle I, J \rangle : \Delta \,.\, \forall M : \mathcal{Y} \,.\, \forall X : \mathcal{X} \,. \\ O_{constr}(\langle I, J \rangle, \langle \_, M \rangle, X) \leftrightarrow \forall \langle K, L \rangle \in M \,. \\ \wedge_{i=1}^{m} P_i(I, J, K, L) \rightarrow Q_i(I, J, K, L) \end{array}$$

where in is a primitive (with the usual meaning).

Especially notice the definition of O_{constr}: once $satisfies$ and O_{split} had been chosen, and considering that $Solution$ has the form of O^{dec}_{ass} (see Section 2.1), it became possible for us to hand-derive the indicated O_{constr} in a way satisfying axiom A_3. It is indeed as strong a necessary condition as "possible and reasonable," as it just poses an incremental consistency constraint: $\delta = \langle I, J \rangle$ being the most recently added couple (by $split$) to the descriptor D, which contains the partial mapping M constructed so far, it suffices to backward-check whether $\langle I, J \rangle$ is consistent with every $\langle K, L \rangle$ of M. Note that this constraint is thus nothing but O^{dec}_{ass} where the outermost universal quantification has been stripped away! It is also important to understand that [as opposed to Smith's filters and cuts] no forward constraint needs to be posed (establishing whether the new partial mapping can possibly be part of a correct solution), not even for efficiency reasons, due to the way in which 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.)

Theorem 4.1 The programs P_{init}, P_{extr}, P_{split}, P_{constr}, P_{gen} below, denoted by C^{dec}_{ass} (where the C is for *closure*, because it "closes" the open program GS_{dec}), are totally correct wrt the axioms S_{init}, S_{extr}, S_{split}, S_{constr}, S_{gen}, respectively, after they have been unfolded wrt $satisfies$, O_{init}, O_{extr}, O_{split}, O_{constr} using the particularisation P^{dec}_{ass} above.

$$\begin{array}{ll} P_{init}: & initialise(X, D) \leftarrow \\ & \quad D = \langle V, [\,] \rangle \\ P_{extr}: & extract(\_, D, Y) \leftarrow \\ & \quad D = \langle [\,], Y \rangle \\ P_{split}: & split(D, X, D', \delta) \leftarrow \\ & \quad D = \langle [I|T], M \rangle, \\ & \quad J\ in\ 1..W, \\ & \quad \delta = \langle I, J \rangle, \\ & \quad D' = \langle T, [\delta|M] \rangle \end{array}$$

```
$P_{constr}$ :  $constrain(\_, D, \_) \leftarrow$
                    $D = \langle \_, [\,] \rangle$
                $constrain(\delta, D, X) \leftarrow$
                    $\delta = \langle I, J \rangle,$
                    $D = \langle \_, [\langle K, L \rangle | M'] \rangle,$
                    $\wedge_{i=1}^{m} P_i(I, J, K, L) \rightarrow Q_i(I, J, K, L),$
                    $constrain(\delta, \langle \_, M' \rangle, X)$
$P_{gen}$ :     $generate(M, \_) \leftarrow$
                    $M = [\,]$
                $generate(M, \_) \leftarrow$
                    $M = [\langle \_, J \rangle | M'],$
                    $indomain(J),$
                    $generate(M', \_)$
```

Note that all but the recursive clause for $constrain$ of these programs are problem-independent. We have thus hand-synthesised in advance programs for the relations defined by the particularisation: some of these syntheses were trivial, for the others we used a divide-and-conquer schema for guidance [11, 4]. Finally, notice that S_{ass}^{dec} (see Section 2.1), P_{ass}^{dec}, and C_{ass}^{dec} share the free variables V, W, m, P_i, Q_i (which represent the problem to be solved): therefore, if a problem-dependent substitution for these variables is applied to S_{ass}^{dec}, then it must also be applied to P_{ass}^{dec} and C_{ass}^{dec}. Finding such a substitution is the objective of the notion of specification reduction, which we examine in Section 5.

For *optimisation assignment* problems, space reasons preclude the inclusion of the extended versions of the particularisation and closure above.

4.2. Particularisations for Permutation Problems

For *decision permutation* problems, three particularisations and closures have been designed (one pair for each specification type, depending on the primitive used in the P formula), but, for space reasons, we can here only discuss their optimisation versions.

The formulas below, denoted by P_{perm}^{opt}, constitute a particularisation of the GS schema for *optimisation permutation* problems whose specifications use $consTwo$ in formula P. (Space reasons preclude presenting the other two particularisations, for permutation problems whose specifications use $precTwo$ or $anyTwo$.) It enumerates permutations S of the $1..N$ interval, where the problem tuple X has the form $\langle N, \ldots \rangle$. Descriptors take the form $\langle S, U \rangle$, and the idea is to gradually build up (from the rear, because in CLP there is no constant-time way of adding an element to the end of a list) the (initially empty) permutation S, which is a sublist of $[1..N]$, such that U is the length of list S. Formally:

$$\mathcal{D} = \{\langle S, U \rangle | S \subseteq [1..N] \wedge length(S, U)\}$$

$$\Delta = \{\delta | \delta \in 1..N\} = 1..N$$

$$\forall Y : \mathcal{Y} \,.\, \forall D : \mathcal{D} \,.\, satisfies(Y, D) \leftrightarrow \exists S, L : \mathcal{Y} \,.\, D = \langle S, \_ \rangle \wedge append(L, S, Y)$$

$$\forall X : \mathcal{X} \,.\, \forall D : \mathcal{D} \,.\, O_{init}(X, D) \leftrightarrow D = \langle [\,], 0 \rangle$$

$$\forall X : \mathcal{X} \,.\, \forall D : \mathcal{D} \,.\, \forall Y : \mathcal{Y} \,.\, O_{extr}(X, D, Y) \leftrightarrow D = \langle Y, N \rangle \wedge alldifferent(Y)$$

$$\forall D, D' : \mathcal{D} \,.\, \forall X : \mathcal{X} \,.\, \forall \delta : \Delta \,.\, O_{split}(D, X, D', \delta) \leftrightarrow \exists S : \mathcal{Y} \,.\, \exists U : int \,.\, D = \langle S, U \rangle \wedge \delta \; in \; 1..N \wedge D' = \langle [\delta | S], U + 1 \rangle$$

$$\forall \delta : \Delta \,.\, \forall D : \mathcal{D} \,.\, \forall X : \mathcal{X} \,.\, O_{constr}(\delta, D, X) \leftrightarrow D = \langle [\,], 0 \rangle \vee \exists V : 1..N \,.\, D = \langle [V | \_], \_ \rangle \wedge Q(\delta, V)$$

where $length$, $append$, $alldifferent$ are primitives (with the usual meanings). Again notice how a strongest "possible and reasonable" O_{constr} could be hand-derived in advance.

Theorem 4.2 The programs P_{init}, P_{extr}, P_{split}, P_{constr}, P_{obj}, P_{gen} below, denoted by C_{perm}^{opt}, are totally correct wrt the axioms S_{init}, S_{extr}, S_{split}, S_{constr}, S_{obj}, S_{gen}, respectively, after they have been unfolded wrt $satisfies$, O_{init}, O_{extr}, O_{split}, O_{constr} using the particularisation P_{perm}^{opt}.

```
$P_{init}$ :    $initialise(\_, D) \leftarrow$
                    $D = \langle [\,], 0 \rangle,$
$P_{extr}$ :    $extract(X, D, Y) \leftarrow$
                    $D = \langle Y, N \rangle,$
                    $alldifferent(Y)$
$P_{split}$ :   $split(D, X, D', \delta) \leftarrow$
                    $D = \langle S, U \rangle,$
                    $\delta \; in \; 1..N,$
                    $D' = \langle [\delta | S], U + 1 \rangle$
$P_{constr}$ :  $constrain(\delta, D, X) \leftarrow$
                    $D = \langle [V | \_], \_ \rangle,$
                    $Q(\delta, V)$
                $constrain(\_, D, \_) \leftarrow$
                    $D = \langle [\,], 0 \rangle$
$P_{obj}$ :     $objective(X, Y, W) \leftarrow$
                    $objective'(Y, X, 0, W)$
                $objective'([\,], \_, W, W) \leftarrow$
                $objective'([\_], \_, W, W) \leftarrow$
                $objective'([J, L | Y], X, W, Z) \leftarrow$
                    $Q(J, L),$ % $E$ must be free in $Q$
                    $NewW = W + E,$
                    $objective'([L | Y], X, NewW, Z)$
$P_{gen}$ :     $generate(Y, \_) \leftarrow$
                    $Y = [\,]$
                $generate(Y, \_) \leftarrow$
                    $Y = [J | Y'],$
                    $indomain(J),$
                    $generate(Y', \_)$
```

All but some clauses for $constrain$ and $objective'$ are problem-independent; we have again hand-synthesised in advance programs for the relations defined by the particularisation. Note that P_{obj} is specific to problems where a sum has to be minimised (that is, where $F = sum$). Finally, notice that S^{opt}_{perm} (see Section 2.2), P^{opt}_{perm}, and C^{opt}_{perm} share the free variables N and Q (which represent the problem to be solved): therefore, if a problem-dependent substitution for these variables is applied to S^{opt}_{perm}, then it must also be applied to P^{opt}_{perm} and C^{opt}_{perm}.

5. Specification Reduction

Given a specification S_r for which no program has been written yet, and a specification S_g for which a program P_g has already been written, we examine the conditions under which it suffices to invoke P_g in order to (partially) implement S_r. We then say that S_r *reduces to* S_g. Basically, this requires that the set of correct solutions to S_g contains those to S_r, provided there later is an elimination of the solutions to S_g that are not solutions to S_r. Formally:

Definition 5.1 A specification $S_r = \langle \mathcal{X}_r, \mathcal{Y}_r, \mathcal{W}_r, O_r \rangle$ for a relation r *reduces to* a specification $S_g = \langle \mathcal{X}_g, \mathcal{Y}_g, \mathcal{W}_g, O_g \rangle$ for r *with* substitution θ if

$$\begin{array}{l} \forall X_r : \mathcal{X}_r \,.\, \exists X_g : \mathcal{X}_g \,.\, \forall Y_r : \mathcal{Y}_r \,.\, \forall W_r : \mathcal{W}_r \,. \\ \quad X_r = X_g\theta \wedge \mathcal{Y}_r = \mathcal{Y}_g\theta \wedge \mathcal{W}_r = \mathcal{W}_g\theta \\ \quad \wedge\, O_r(X_r, Y_r, W_r) = O_g(X_g, Y_r, W_r)\theta \end{array}$$

Computing such a substitution involves second-order semi-unification, which is decidable but NP-complete in general, though linear in the case of higher-order patterns [7], where all predicate variables (such as the P_i and Q_i) apply to distinct variables only, which is the case here.

Example 5.1 The specification S^{dec}_{col} (see Example 2.1) reduces to S^{dec}_{ass} (see Section 2.1) with:

$$\begin{array}{c} \theta_1 = \{X/\langle R, C, A\rangle,\ V/R,\ W/C,\ m/1, \\ P_1/\lambda J, K, L, M \,.\, \langle J, L\rangle \in A,\ Q_1/\lambda J, K, L, M \,.\, K \neq M\} \end{array}$$

Note that A is free in the λ-term substituted for P_1: this is no problem because $\langle R, C, A\rangle$ is substituted for X, which is universally quantified wherever P_1 occurs.

Example 5.2 The specification S^{opt}_{ham} (see Example 2.2) reduces to S^{opt}_{perm} (see Section 2.2) with:

$$\theta_2 = \{X/\langle C, A\rangle,\ N/C,\ Q/\lambda J, K \,.\, \langle J, K, E\rangle \in A\ \}$$

Note that A is free in the λ-term substituted for Q: this is no problem because $\langle C, A\rangle$ is substituted for X, which is universally quantified wherever Q occurs. Also note that, as required earlier, the summation variable E is free in the λ-term substituted for Q.

6. The Synthesis Method

The synthesis method is apparent now: given a specification S_r, find a substitution θ under which it reduces to the generic specification S_g attached to some particularisation P_g of the global search schema, and then apply θ to P_g and to the closure C_g, so as to obtain a (closed) program that correctly implements S_r by taking the GS_{opt} template and $C_g\theta$.

For assignment and permutation problems, note how the elimination of the solutions to S^{dec}_{ass} or S^{opt}_{perm} that are not solutions to S_r is performed [without explicitly inserting O_r at the end of the synthesised program, like Smith does]: for instance, O^{dec}_{ass} has predicate variables P_i and Q_i, which also appear in P^{dec}_{ass} (and thus in C^{dec}_{ass}) and which become instantiated to the particular conditions in O_r, which thus wind up, as we have seen, in the recursive clause for $constrain$. [In Smith's approach, O^{dec}_{ass} is $true$, and the post-condition O_r of the particular problem can thus not appear in the search part of the synthesised code, except maybe in a filter, whose derivation is however often not fully automatic and which filter is not necessarily "reasonable."]

Example 6.1 Given the specification S^{dec}_{col} (see Example 2.1), the fully automatically synthesised program thus consists of the GS_{dec} template (see Section 3.1) and the closure C^{dec}_{ass} of Theorem 4.1, where the problem-dependent recursive clause for $constrain$ is:

$$\begin{array}{l} constrain(\delta, D, \langle \_, \_, A\rangle) \leftarrow \\ \quad \delta = \langle R_1, C_1\rangle, \\ \quad D = \langle \_, [\langle R_2, C_2\rangle | M']\rangle, \\ \quad \langle R_1, R_2\rangle \in A \rightarrow C_1 \neq C_2, \\ \quad constrain(\delta, \langle \_, M'\rangle, \langle \_, \_, A\rangle) \end{array}$$

by virtue of the substitution θ_1 (see Example 5.1). We here use $P \rightarrow Q$ to denote $not(P); Q$, where $;/2$ denotes disjunction and can easily be implemented by the two clauses $P; Q \leftarrow P$ and $P; Q \leftarrow Q$, using the meta-variable facility of CLP. The usage of negation-as-failure (denoted by not) is not dangerous here, because the synthesised program guarantees that the thus negated atom is ground at that moment.

Example 6.2 Given the specification S^{opt}_{ham} (see Example 2.2), the fully automatically synthesised program consists of the GS_{opt} template (see Section 3.1) and the closure C^{opt}_{perm} of Theorem 4.2, where the problem-dependent clauses for $constrain$ and $objective'$ are:

$$\begin{array}{l} constrain(\delta, D, \langle \_, A\rangle) \leftarrow \\ \quad D = \langle [V|\_], \_\rangle, \\ \quad \langle \delta, V, \_\rangle \in A \\ objective'([V_1, V_2|Y], \langle \_, A\rangle, W, Z) \leftarrow \\ \quad \langle V_1, V_2, E\rangle \in A, \\ \quad NewW = W + E, \\ \quad objective'([V_2|Y], \langle \_, A\rangle, NewW, Z) \end{array}$$

by virtue of the substitution θ_2 (see Example 5.2).

7. Benchmarks

In Table 1, we first compare our synthesised CLP programs (run under $clp(FD)$ [2]) with the (non-CLP) logic program counterparts (also run under $clp(FD)$) of KIDS-synthesised *Refine* programs (with hand-derived filters). These tests (for colouring the departmental map of France (DeptMap), for finding a Hamiltonian path through the countries of Europe (HamPath), and for solving the 8-Queens problem) show that at least one order of magnitude is gained in efficiency by switching from an ordinary symbolic language to a constraint one (a comparison with the more recent $SpecWare$ and $PlanWare$ [14] of Kestrel Institute is underway). We chose *Finite Domains* (FD) as constraint domain because of the well-known high performance of CLP(FD).

	DeptMap	HamPath	8-Queens
Synth'd CLP(FD)	27,150 ms	50 ms	100 ms
Synth'd LP/Refine	overflow	527 ms	3260 ms
Publ'd CLP(FD) [2]	5,230 ms	20 ms	30 ms

Table 1. Benchmarks

Further tests show that our automatically synthesised CLP(FD) programs are only 3 to 5 times slower than carefully hand-crafted, published CLP(FD) programs, which is encouraging since none of the obvious problem-specific optimising transformations have been performed yet on our programs. Since our synthesis is fully automatic, starting from short and elegant specifications, our approach thus seems viable.

Our specification language is equivalent in its high expressiveness with CLP(Sets) programming languages (such as $CLPS$ [1], $Cojunto$ [6], ...). We thus do not aim at synthesising CLP(Sets) programs, but rather at different ways of compiling them. Comparing execution times is however still meaningless because of the prototypical nature of CLP(Sets) compilers (which normalise the programs into Prolog programs and add constraint-solving code in Prolog).

Comparisons with handwritten non-constraint programs for performing the same computations have not been made yet, but we expect such programs to perform somewhere near handwritten constraint programs (whether CLP or not), unless they are very naive or super-optimised. Since our synthesised CLP(FD) programs perform within the same order of magnitude as handwritten CLP(FD) programs (and thus most non-constraint programs), the crucial difference may well lie in the time it takes to *construct* the programs, and then the benefits of constraint languages in general, as well as of automated constraint program synthesis (such as ours) in particular, will kick in.

In Figures 1 and 2, a further comparison is made between the synthesised CLP(FD) programs and the corresponding synthesised LP programs, for the n-Queens and graph colouring problems. The minimum one order of magnitude gain confirms that we fully exploit constraint propagation to reduce the search space by cutting off spaces that do not lead to correct solutions.

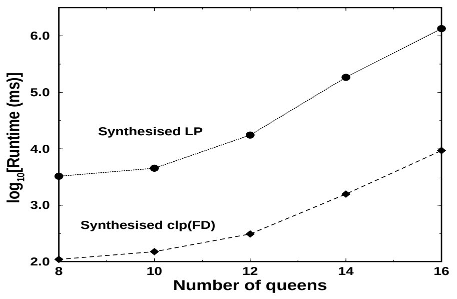

Figure 1. n-Queens

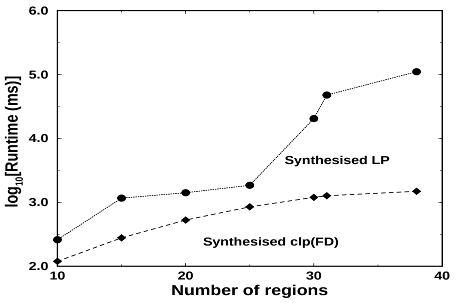

Figure 2. Graph Colouring

8. Conclusion

We outlined how to fully automatically synthesise CLP programs for assignment and permutation problems (whose specification templates in Section 2 thus determine the current scope of our approach), and we showed that our results are competitive. We see no reason why we would not be able to replicate this achievement for the other 5 families of global search problems identified by Smith [12]. We now need to investigate how this work scales up to more complexly specified (if not hybrid) search problems.

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 [4] should not be too difficult.

The results presented in this paper are however more than a simple transcription of the KIDS approach from *Refine* to CLP, as they also reflect new ideas. In summary:

- 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. We argue for our choice of CLP(FD) as target language by the fact that it is especially suited for solving combinatorial problems. Indeed, much of the constraint solving machinery that needs to be pushed into *Refine programs*, be it 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. We thus established that the features of the target computing environment can be successfully exploited by a synthesis mechanism.

- We introduced the notion of *specification template*, by illustrating it on the families of assignment and permutation problems. This has nice effects on the KIDS approach, as shown below.

- As shown for some particularisations, 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 derive it.

- As shown for some particularisations, the derivation of consistency-constraint-posing code can be calling-context-dependent [as opposed to filter derivation]. Such code can even be pre-synthesised, for a given particularisation, so that there is no need of a theorem prover, at synthesis time, to derive its specification.

This means that synthesis can be fully automatic, without using any theorem prover, for some families of problems. There are a lot of opportunities for automatically optimising the synthesised programs, hopefully bringing them on a par with hand-crafted programs.

Our work can also be seen as being of methodological nature: indeed, the identification of problem/specification families and of efficient corresponding programs is a contribution to constraint logic programming methodology, and not unlike what is advocated by the *patterns* community [5] (except that we aim at full formalisation and automation, whereas patterns are mostly informal). Moreover, unlike the top-down decomposition methodology, for instance, which is solution/program-centered, our methodology is problem/specification-centered and thus quite useful.

Acknowledgments

We thank Doug Smith for his inspiring pioneering work, and the anonymous referees for their constructive comments.

References

[1] F. Ambert, B. Legeard, et E. Legros. Programmation en logique avec contraintes sur ensembles et multiensembles héréditairement finis. *Techniques et Sciences Informatiques* 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] Y. Deville and P. Van Hentenryck. Construction of CLP programs. In D.R. Brough (ed), *Logic Programming: New Frontiers*, pp. 112–135, Kluwer Academic Publishers, 1992.

[4] P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In M. Lowry and Y. Ledru (eds), *Proc. of ASE’97*, pp. 153–160. IEEE Computer Society, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.

[7] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. D.S. Warren (ed), *Proc. of ICLP’93*, pp. 441–455. The MIT Press, 1993.

[8] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. of Logic Programming* 19–20:503–582, 1994.

[9] K.-K. Lau and M. Ornaghi. A formal approach to deductive synthesis of constraint logic programs. In J.W. Lloyd (ed), *Proc. of ILPS’95*, pp. 543–557. The MIT Press, 1995.

[10] B. Le Charlier and P. Flener. Specifications are necessarily informal, or: Some more myths of formal methods. *J. of Systems and Software* 40(3):275-296, 1998.

[11] D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.

[12] D.R. Smith. The structure and design of global search algorithms. TR *KES.U.87.12*, Kestrel Institute, 1988.

[13] D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Software Engineering* 16(9):1024–1043, 1990.

[14] D.R. Smith. Towards the synthesis of constraint propagation algorithms. In Y. Deville (ed), *Proc. of LOPSTR’93*, pp. 1–9, Springer-Verlag, 1994.