Schema-Guided Synthesis of Constraint Logic Programs *

Hamza Zidoum, Pierre Flener, and Brahim Hnich
Email to: pf@info.fundp.ac.be

Abstract

By focusing on the family of assignment problems (such as graph colouring, n-Queens, etc), we show how to adapt D.R. Smith's KIDS approach for the synthesis of *constraint* programs, rather than applicative *Refine* programs with explicit constraint propagation and pruning code. Synthesis is guided by a global search program 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. After optimisations, the synthesised programs would be competitive with hand-crafted ones.

1 Introduction

This work is inspired by D.R. Smith's research on synthesising global search programs (in the *Refine* language) from first-order logic specifications (also in *Refine*) [10, 11, 12]. The basic idea of global search is to represent and manipulate sets of candidate solutions. Starting from an initial set that contains all solutions to the given problem, a global search 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) [7] is a programming paradigm especially suited for solving combinatorial problems, thanks to its double reasoning: the symbolic reasoning expresses the logic properties of the problem, while the 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. In this paper, we only deal with decision problems, since optimisation problems can be treated as simple extensions of decision problems, namely by adding a cost domain and a cost function.

Very few works deal with the synthesis and transformation of CLP programs. The authors of [8] show the possibility of synthesising steadfast CLP programs, however they do not exhibit such a synthesis method. 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 global search schema. We use particular cases of that general schema to instantiate 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 like for KIDS [10, 11, 12]). In this paper, we only tackle the family of assignment problems.

As argued in the conclusion, our work is not simply a transposition of Smith's results from the *Refine* world to the CLP world, because there are several important innovations.

2 Specifications

Specifications are the input to program synthesis. In order to enable (or at least facilitate) automated synthesis, such inputs ought to be formal (though a more adequate terminology would then be to say

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15–19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

that the inputs are programs and that synthesis is compilation).

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

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

where $X : \mathcal{X}$ and $Y : \mathcal{Y}$ are (possibly empty) lists of sorted variables. Formula I_r is called the *input condition*, constraining the input domain $\mathcal{X}$, whereas formula O_r is called the *output condition*, describing when some output value Y is a correct solution for input value (or 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 $\langle \mathcal{X}, \mathcal{Y}, O_r \rangle$ triples.

In this paper, we only consider the family of *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_{assign} take the form $\langle list(term) \times integer, list(V \times 1..W), O_{assign} \rangle$, where O_{assign} is of the form:

$$\forall \langle X_1, Y_1 \rangle \in M \,.\, \forall \langle X_2, Y_2 \rangle \in M \,.\, \wedge_{i=1}^{m} P_i(X_1, Y_1, X_2, Y_2) \rightarrow Q_i(X_1, Y_1, X_2, Y_2) \qquad (O_{assign})$$

where the P_i and Q_i are formulas. This can be considered a *specification template*. 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 two adjacent regions (as indicated in an adjacency list A) have different colours. Formally:

$$\begin{array}{c} \forall \langle R, C, A \rangle : list(term) \times integer \times list(R \times R) \,.\, \forall M : list(R \times 1..C) \,. \\ colouring(\langle R, C, A \rangle, M) \leftrightarrow \forall \langle R_1, C_1 \rangle \in M \,.\, \forall \langle R_2, C_2 \rangle \in M \,.\, member(\langle R_1, R_2 \rangle, A) \rightarrow C_1 \neq C_2 \end{array} \qquad (S_{colour})$$

where *member* is a primitive (with the usual meaning).

3 A Global Search Program Schema for CLP

Let us first recall a definition of (program) schemas [4]:

Definition 3.1 (Program Schemas)
A *program schema* for a programming methodology M 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 according to M, and *axioms* A constrain the (problem-dependent) programs for the open relations in T such that the overall program really reflects M.

We now formalise our global search (GS) schema for CLP programs. Intuitively, the basic idea is as follows: starting from an *initialised* descriptor of the full search space, incrementally *split* that space into sub-spaces, while declaring the domains of the involved variables[1] and *constraining* these variables so as 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. Compared to Smith's global search schema, ours only computes one correct solution rather than all of them, 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 following open program:

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

[1] Usually, in CLP, domain declaration is done in a separate loop preceding the constraint-posing loop, but our approach is slightly more efficient because we gain a loop.

where the open relations are informally specified as follows:

- *initialise*(X, D) means that D is the descriptor of the initial space of candidate solutions to X;
- *extract*(X, D, Y) means that the variablised solution Y to problem X is directly extracted from descriptor D;
- *split*(D, X, D', δ) means that descriptor D' describes a subspace of D wrt problem X, such that D' is obtained by adding δ to descriptor D;
- *constrain*(δ, D, X) means that adding δ to descriptor D leads to a descriptor defining a sub-space of D that may contain correct solutions to problem X;
- *generate*(Y, X) means that correct solution Y to problem X is enumerated (by instantiation of the variables in the initially variablised solution Y) from the constraint store, which represents X.

Formalising this is the role of the axioms, shown next.

3.2 The Global Search Axioms

Let $\mathcal{D}$ be the type of the search space descriptors, and let Δ be the type of the elements of the partial solutions stored in descriptors. First, the following axioms are the specifications of the open relations of the *GS* template:

$$\forall X : \mathcal{X} \,.\, \forall D : \mathcal{D} \,.\, initialise(X, D) \leftrightarrow O_{init}(X, D) \qquad (S_{init})$$
$$\forall X : \mathcal{X} \,.\, \forall D : \mathcal{D} \,.\, \forall Y : \mathcal{Y} \,.\, extract(X, D, Y) \leftrightarrow O_{extr}(X, D, Y) \qquad (S_{extr})$$
$$\forall D, D' : \mathcal{D} \,.\, \forall X : \mathcal{X} \,.\, \forall \delta : \Delta \,.\, split(D, X, D', \delta) \leftrightarrow O_{split}(D, X, D', \delta) \qquad (S_{split})$$
$$\forall \delta : \Delta \,.\, \forall D : \mathcal{D} \,.\, \forall X : \mathcal{X} \,.\, constrain(\delta, D, X) \leftrightarrow O_{constr}(\delta, D, X) \qquad (S_{constr})$$
$$\forall Y : \mathcal{Y} \,.\, \forall X : \mathcal{X} \,.\, generate(Y, X) \leftrightarrow O_r(X, Y) \qquad (S_{gen})$$

The output conditions of these specifications are constrained by the next axioms.

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

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

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:

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

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:

$$\begin{array}{c}
\forall X : \mathcal{X} \,.\, \forall Y : \mathcal{Y} \,.\, \forall D, D' : \mathcal{D} \,.\, \forall \delta : \Delta \,. \\
O_r(X, Y) \wedge O_{split}(D, X, D', \delta) \wedge satisfies(Y, D') \rightarrow O_{constr}(\delta, D, X)
\end{array} \qquad (A_3)$$

Conversely, the contrapositive of this axiom 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 a decision procedure, called *SAT*, which checks whether a constraint store is satisfiable [7].

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) technology, as shown in [9, 10] for instance. 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 qualifiers being rather subjective. Fortunately, for the family of assignment problems tackled in this paper, it turns out that this condition can be easily manually pre-computed (see Example 4.1) at schema-design time, for *any* such problems, in an optimal way, so that no ATP technology is then necessary at synthesis time!

Also note that the derivation of the output condition of *constrain* depends on the calling context of *constrain*, 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 [10, 11] and cuts [12], which thus tend to 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 Program Schema

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

Definition 3.2 (Total correctness)
A closed program P_r for a relation r is *totally correct* wrt its specification $\langle \mathcal{X}, \mathcal{Y}, O_r \rangle$ if for all $X : \mathcal{X}$ and $Y : \mathcal{Y}$ we have that $O_r(X,Y)$ iff $P_r \vdash r(X,Y)$.

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

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

Proof. Outline: Let P_r be the first clause of GS, and let P_{rgs} be the remaining two clauses of GS. First, prove that P_{rgs} is steadfast wrt the following specification:

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

and the 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 schema in a way analogous to the way the divide-and-conquer schema was used in [9, 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 global search schema than for the divide-and-conquer one [10]. Fortunately, a very large percentage of global search 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 family of assignment problems, which amounts to enumerating mappings from a finite list into a finite integer interval, other families enumerating permutations of a given list, sublists of (given or bounded) length k over a given list, sequences over a given list, etc [10].

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

Example 4.1 The formulas below, denoted by P_{assign}, constitute a particularisation of the global search schema for 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, whose domain is a sublist of V and whose range is $1..W$, such that list T has the elements of V that have not been mapped to elements in $1..W$ yet. Formally:

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

$$\Delta = \{\langle X_1, Y_1\rangle | X_1 \in V \wedge Y_1 \in 1..W\}$$

$$\forall Y : \mathcal{Y} \,.\, \forall D : \mathcal{D} \,.\, satisfies(Y, D) \leftrightarrow \exists M : \mathcal{Y} \,.\, D = \langle \_, M\rangle \wedge \forall \langle X_1, Y_1\rangle \in M \,.\, \langle X_1, Y_1\rangle \in Y$$

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

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

$$\begin{gathered}\forall D, D' : \mathcal{D} \,.\, \forall X : \mathcal{X} \,.\, \forall \delta : \Delta \,.\, O_{split}(D, X, D', \delta) \leftrightarrow \\ D = \langle [X_1|T], M\rangle \wedge Y_1 \ in\ 1..W \wedge \delta = \langle X_1, Y_1\rangle \wedge D' = \langle T, [\delta|M]\rangle\end{gathered}$$

$$\begin{gathered}\forall \langle X_1, Y_1\rangle : \Delta \,.\, \forall M : \mathcal{Y} \,.\, \forall X : \mathcal{X} \,.\, O_{constr}(\langle X_1, Y_1\rangle, \langle \_, M\rangle, X) \leftrightarrow \\ \forall \langle X_2, Y_2\rangle \in M \,.\, \wedge_{i=1}^{m} P_i(X_1, Y_1, X_2, Y_2) \rightarrow Q_i(X_1, Y_1, X_2, Y_2)\end{gathered}$$

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 O_r has the form of O_{assign} (see Section 2), 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: $\delta = \langle X_1, Y_1\rangle$ 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 $\langle X_1, Y_1\rangle$ is consistent with every $\langle X_2, Y_2\rangle$ of M. Note that this constraint is thus nothing but O_{assign} 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 (Implementation of P_{assign})
The programs P_{init}, P_{extr}, P_{split}, P_{constr}, P_{gen} below, denoted by C_{assign} (where the C stands for *closure*, because it "closes" the open program GS), 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_{assign} above.

$$\begin{array}{rl}
P_{init}: & initialise(X, D) \leftarrow \\
 & \quad D = \langle V, [\,]\rangle \\
P_{extr}: & extract(_, D, Y) \leftarrow \\
 & \quad D = \langle [\,], Y\rangle \\
P_{split}: & split(D, X, D', \delta) \leftarrow \\
 & \quad D = \langle [X_1|T], M\rangle, \\
 & \quad Y_1\ in\ 1..W, \\
 & \quad \delta = \langle X_1, Y_1\rangle, \\
 & \quad D' = \langle T, [\delta|M]\rangle \\
P_{constr}: & constrain(_, D, _) \leftarrow \\
 & \quad D = \langle _, [\,]\rangle \\
 & constrain(\delta, D, X) \leftarrow \\
 & \quad \delta = \langle X_1, Y_1\rangle, \\
 & \quad D = \langle _, [\langle X_2, Y_2\rangle | M']\rangle, \\
 & \quad \wedge_{i=1}^{m} P_i(X_1, Y_1, X_2, Y_2) \rightarrow Q_i(X_1, Y_1, X_2, Y_2), \\
 & \quad constrain(\delta, \langle _, M'\rangle, X) \\
P_{gen}: & generate(M, _) \leftarrow \\
 & \quad M = [\,] \\
 & generate(M, _) \leftarrow \\
 & \quad M = [\langle _, Y_1\rangle | M'], \\
 & \quad indomain(Y_1), \\
 & \quad generate(M', _)
\end{array}$$

Note that all but the recursive clause for *constrain* of these programs are problem-independent. Also note that 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 [9, 4]. Finally, notice that S_{assign} (see Section 2), P_{assign}, and C_{assign} 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_{assign}, then it must also be applied to P_{assign} and C_{assign}. Finding such a substitution is the objective of the notion of specification reduction, which we examine now.

5 Specification Reduction

Given a specification S_r for which no program has been written yet, and given a specification S_g for which a program P_g has already been written, we now 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, or that S_g *generalises* S_r. 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 (Specification Reduction)
A specification $S_r = \langle \mathcal{X}_r, \mathcal{Y}_r, O_r \rangle$ for a relation r *reduces to* a specification $S_g = \langle \mathcal{X}_g, \mathcal{Y}_g, O_g \rangle$ for r *with* substitution θ if $\forall X_r : \mathcal{X}_r \,.\, \exists X_g : \mathcal{X}_g \,.\, \forall Y_r : \mathcal{Y}_r \,.\, X_r = X_g\theta \wedge \mathcal{Y}_r = \mathcal{Y}_g\theta \wedge O_r(X_r, Y_r) = O_g(X_g, Y_r)\theta$.

Computing such a substitution often involves second-order semi-unification, which is decidable but NP-complete in general, though linear in the case of higher-order patterns [6], where all predicate variables (such as the P_i and Q_i) apply to distinct variables only, which is the case here. It even turns out that θ can be manually pre-computed, for *any* assignment problem, as illustrated in the following example:

Example 5.1 The specification S_{colour} (see Example 2.1) reduces to S_{assign} (see Section 2) with:

$$\theta = \{X/\langle R, C, A\rangle,\ V/R,\ W/C,\ m/1,\ P_1/\lambda J, K, L, M \,.\, member(\langle J, L\rangle, A),\ Q_1/\lambda J, K, L, M \,.\, K \neq M\}$$

Note that A is free in the λ-term substituted for P_1: this does not pose a problem because $\langle R, C, A\rangle$ is substituted for X, which is universally quantified wherever P_1 occurs.

6 The Synthesis Method

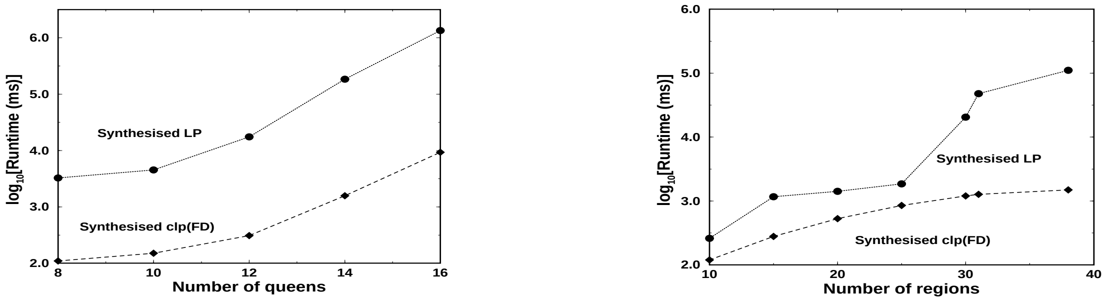
The synthesis method becomes apparent now: given a specification S_r, find a substitution θ under which it reduces to the generic specification S_g attached to some particularisation P_g of the global search schema, and then apply θ to P_g and to the closure C_g, so as to obtain a (closed) program correctly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

For assignment problems, note how the elimination of the solutions to S_{assign} that are not solutions to S_r is performed [without explicitly inserting O_r at the end of the synthesised program, like Smith does]: O_{assign} has predicate variables P_i and Q_i, which also appear in P_{assign} (and thus in the closure C_{assign}) and which become instantiated to the particular conditions in S_r, which thus wind up, as we have seen, in the recursive clause for *constrain*. [In Smith's approach, O_{assign} 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 or a cut, whose derivation is however often not fully automatic and which filter or cut is not necessarily "reasonable".]

Example 6.1 Given the specification S_{colour} (see Example 2.1), the fully automatically synthesised program thus consists of the GS template (see Section 3.1) and the closure C_{assign} of Theorem 4.1, where the problem-dependent recursive clause for *constrain* is:

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

by virtue of the substitution θ (see Example 5.1). Note that 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.



(a) n-Queens (b) Graph Colouring

Figure 1: Benchmarks

7 Benchmarks

In the following table, we first compare our synthesised CLP programs (run under *clp*(FD) [2]) with the (standard) logic program counterparts (also run under *clp*(FD)) of KIDS-synthesised *Refine* programs (with hand-derived filters). This shows 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* [12] systems of Kestrel Institute is underway). We chose *Finite Domains* (FD) as constraint domain because of the well-known high performance of CLP(FD).

	Map Colouring (France)	Hamiltonian Path	8-Queens
Synthesised CLP(FD) programs	27,150 ms	50 ms	100 ms
Synthesised LP programs	overflow	527 ms	3260 ms
Hand-crafted CLP(FD) programs [2]	5,230 ms	20 ms	30 ms

Table 1: Benchmarks

We also compare our synthesised CLP(FD) programs with hand-crafted CLP(FD) programs. This shows that our automatically synthesised CLP(FD) programs are only 3 to 5 times slower than carefully hand-crafted ones, 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 to the CLP(Sets) programming languages (such as *CLPS* [1], *Cojunto* [5]); we thus do not aim at synthesising CLP(Sets) programs, but rather at alternative ways of compiling them. Comparing execution times is however still meaningless because of the prototypical nature of CLP(Sets) compilers (which sort-of normalise the programs into Prolog programs and add constraint-solving code in Prolog).

In Figure 1 above, 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.

8 Conclusion

We have outlined how to fully automatically synthesise CLP programs for assignment problems, and we have shown that our results are competitive. We hope to replicate this effort for the other six families of global search problems identified by Smith [10].

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 not just a simple transcription of the KIDS approach from *Refine* to CLP, but they also reflect new ideas, as indicated all over this paper. In summary:

- We fully exploited CLP features [as opposed to *Refine*, which is "only" an ordinary symbolic language], by significantly modifying the original global search 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 transformation/optimisation time, is already part of the CLP(FD) *language* and is implemented there once and for all in a particularly efficient way.

- We introduced the notion of *specification template*, by illustrating it on the family of assignment problems. This has widespread effects on the KIDS approach, as shown below.
- As we showed for the P_{assign} particularisation, the substitution under which a given specification reduces to a specification template like S_{assign} can be pre-computed, so that there is no need to use a theorem prover, at synthesis time, to derive it.
- As we showed for the P_{assign} particularisation, the derivation of consistency-constraint-posing code can be calling-context-dependent [as opposed to Smith's filter and cut derivation]. Also, such code can even be pre-synthesised, for a given particularisation, so that there is no need to use a theorem prover, at synthesis time, to derive its specification.

All this means that synthesis can be fully automatic, without any usage of a theorem prover, for certain families of problems. We plan to add an automated reasoning layer for problems that do not fit our predetermined families. There are a lot of opportunities for automatically transforming/optimising the synthesised programs, hopefully bringing them on a par with hand-crafted programs.

Acknowledgments

We wish to thank Doug Smith for his pioneering work that inspired us. The first author is supported by a post-doctoral scholarship awarded by Bilkent University.

References

[1] F. Ambert, B. Legeard, and E. Legros. Programmation en logique avec contraintes sur ensembles et multi-ensembles 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] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.

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

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

[8] 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.

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

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

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

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