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

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:

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

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 $(\mathcal{X}, \mathcal{Y}, O_r)$.

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 $(\text{list(term)} \times \text{integer}, \text{list}(V \times 1..W), O_{assign})$, where $O_{assign}$ is of the form:

$$\forall (X_1, Y_1) \in M. \forall (X_2, Y_2) \in M. \land_{i=1}^n P_i(X_1, Y_1, X_2, Y_2) \rightarrow Q_i(X_1, Y_1, X_2, Y_2) \tag{O_{assign}}$$

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:

$$\forall (R, C, A) : \text{list(term)} \times \text{integer} \times \text{list}(R \times R). \forall M : \text{list}(R \times 1..C).$$

$$\text{coloring}(R, C, A, M) \rightarrow \forall (R_1, C_1) \in M. \forall (R_2, C_2) \in M. \text{member}((R_1, R_2), A) \Rightarrow C_1 \neq C_2 \tag{S_{colour}}$$

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 $(T, A)$, where $\text{template} T$ is an open program showing the (problem-independent) data-flow and control-flow of programs constructed according to $M$, and $\text{axioms} A$ 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:

$$r(X, Y) \leftarrow initialise(X, D),$$
$$\hspace{1cm} \text{rgs}(X, D, Y),$$
$$\hspace{1cm} \text{generate}(Y, X)$$

$$\text{rgs}(X, D, Y) \leftarrow extract(X, D, Y),$$
$$\text{split}(D, X, D', \delta),$$
$$\text{constrain}(\delta, D, X),$$
$$\text{rgs}(X, D', Y) \tag{GS}$$

\(^1\) 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** \((D, X, D', \delta)\) means that descriptor \(D'\) describes a subspace of \(D\) w.r.t. problem \(X\), such that \(D'\) is obtained by adding \(\delta\) to descriptor \(D\);
- **constrain** \((\delta, D, X)\) means that adding \(\delta\) 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 \(D\) be the type of the search space descriptors, and let \(\Delta\) 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:

\[
\begin{align*}
\forall X : X. \forall D : D. \text{ initialise}(X, D) & \iff O_{\text{init}}(X, D) \quad (S_{\text{init}}) \\
\forall X : X. \forall D : D. \forall Y : Y. \text{ extract}(X, D, Y) & \iff \text{ extract}(X, D, Y) \\
\forall X, D' : \Delta. \forall Y : Y. \text{ split}(D, X, D', \delta) & \iff \text{ split}(D, X, D', \delta) \\
\forall Y : Y. \forall X : X. \text{ constrain}(\delta, D, X) & \iff \text{ constrain}(\delta, D, X) \\
\forall Y : Y. \forall X : X. \text{ generate}(Y, X) & \iff \text{ generate}(Y, X)
\end{align*}
\]

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 \(X\):

\[
\forall X : X. \forall Y : Y. \text{ O}_{r}(X, Y) \iff \exists D : D. \text{ O}_{\text{init}}(X, D) \land \text{satisfies}(Y, D) \quad (A_1)
\]

where \text{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 \text{split} to \(D\). Formally:

\[
\forall X : X. \forall Y : Y. \forall D : D. \quad \text{satisfies}(Y, D) \iff \exists k : \text{integer}. \exists D' : D. \exists \delta : \Delta. \text{split}^k(D, X, D', \delta) \land \text{O}_{\text{extract}}(X, D, Y)
\]

where:

\[
\text{split}^0(D, X, D', \delta) \iff D = D'
\]

and, for all \(k : \text{integer}:

\[
\text{split}^{k+1}(D, X, D', \delta) \iff \exists D'' : D. \exists \delta' : \Delta. \text{split}^k(D'', X, D', \delta') \land \text{split}^{k}(D'', X, D', \delta)
\]

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 \text{constrain} must succeed. Formally:

\[
\forall X : X. \forall Y : Y. \forall D, D', \Delta. \forall \delta : \Delta. \quad \text{O}_{r}(X, Y) \land \text{O}_{\text{split}}(D, X, D', \delta) \land \text{satisfies}(Y, D') \land \text{O}_{\text{constrain}}(\delta, D, X)
\]

Conversely, the contrapositive of this axiom shows that if \text{constrain} fails, then the new space described by \(D'\) (which is \(D\) plus \(\delta\)) does not contain any solution to \(X\). CLP languages contain a decision procedure, called \text{SAT}, which checks whether a constraint store is satisfiable \([7]\).

This last axiom sets up a necessary condition that \text{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 \text{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 \text{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 $(X, Y, O_r)$ if for all $X : X$ and $Y : Y$ 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:

$$\forall X : X. \forall D : D. \forall Y : Y. rgs(X, D, Y) \leftrightarrow satisfies(Y, D) \land O_r(X, Y)$$

(Srgs)

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 $D, \Delta, satisfies, O_{init}, O_{extr}, O_{split}$, 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 $(V, W, \ldots)$. Descriptors take the form $(T, M)$, and the idea is to gradually

4
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:

$$D = \{ (T, M) \mid T \subseteq V \land M \in \text{list}(V \setminus T) \times 1..W \}$$

$$\Delta = \{ (X_1, Y_1) \mid X_1 \in V \land Y_1 \in 1..W \}$$

$$\forall Y : \forall D : \Delta . \text{ satisfies}(Y, D) \implies \exists M : Y : D = (\_, M) \land \forall (X_1, Y_1) \in M . (X_1, Y_1) \in Y$$

$$\forall X : X . \forall D : \Delta . \text{ Oinit}(X, D) \implies D = (V, [])$$

$$\forall X : X . \forall D : \Delta . \forall Y : \forall \text{ Oextr}(X, D, Y) \implies D = ([\_], Y)$$

$$\forall D, D' : \Delta . \forall X : X . \forall \delta : \Delta . \text{ Osplit}(D, X, D', \delta) \implies D = ([X_1[T], M] \land Y_1 \in 1..W) \land \delta = (X_1, Y_1) \land D' = (T, [\_][M])$$

$$\forall (X_1, Y_1) : \Delta . \forall M : Y . \forall X : X . \text{ Oconstr}((X_1, Y_1), (\_, M), X) \implies \forall (X_2, Y_2) \in M . N_{i=1}^m P_t(X_1, Y_1, X_2, Y_2) \implies Q_t(X_1, Y_1, X_2, Y_2)$$

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

Especially notice the definition of $\text{Oconstr}$: once $\text{satisfies}$ and $\text{Osplit}$ had been chosen, and considering that $O_r$ has the form of $\text{Oassign}$ (see Section 2), it became possible for us to hand-derive the indicated $\text{Oconstr}$ 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: $\delta = (X_1, Y_1)$ being the most recently added couple (by $\text{split}$) to the descriptor $D$, which contains the partial mapping $M$ constructed so far, it suffices to backward-check whether $(X_1, Y_1)$ is consistent with every $(X_2, Y_2)$ of $M$. Note that this constraint is thus nothing but $\text{Oassign}$ 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 $\text{generate}$) 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 $\text{Passign}$)**

The programs $P_{\text{init}}$, $P_{\text{extr}}$, $P_{\text{split}}$, $P_{\text{constr}}$, $P_{\text{gen}}$ below, denoted by $\text{Cassign}$ (where the $C$ stands for closure, because it “closes” the open program $GS$), are totally correct wrt the axioms $S_{\text{init}}$, $S_{\text{extr}}$, $S_{\text{split}}$, $S_{\text{constr}}$, $S_{\text{gen}}$, respectively, after they have been unfolded wrt $\text{satisfies}$, $\text{Oinit}$, $\text{Oextr}$, $\text{Osplit}$, $\text{Oconstr}$, using the particularisation $\text{Passign}$ above.

$$P_{\text{init}} : \text{ initialise}(X, D) \implies$$

$$D = ([\_], [])$$

$$P_{\text{extr}} : \text{ extract}(\_ D, Y) \implies$$

$$D = ([\_], Y)$$

$$P_{\text{split}} : \text{ split}(D, X, D', \delta) \implies$$

$$D = ([X_1[T], M], Y_1 \in 1..W, \delta = (X_1, Y_1), D' = (T, [\_][M])$$

$$P_{\text{constr}} : \text{ constrain}(\_ D, \_) \implies$$

$$D = ([\_], [])$$

$$\text{ constrain}(\delta, D, X) \implies$$

$$\delta = (X_1, Y_1), D = ([X_2, Y_2][M']), N_{i=1}^m P_t(X_1, Y_1, X_2, Y_2) \implies Q_t(X_1, Y_1, X_2, Y_2), \text{ constrain}(\delta, ([\_], M'), X)$$

$$P_{\text{gen}} : \text{ generate}(M, \_) \implies$$

$$M = []$$

$$\text{ generate}(M, \_) \implies$$

$$M = ([\_], Y_1)[M'], \text{ indomain}(Y_1), \text{ generate}(M', \_)$$

5
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, note 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 \( S_r = (X_r, Y_r, O_r) \) for a relation \( r \) reduces to a specification \( S_g = (X_g, Y_g, O_g) \) for \( r \) with substitution \( \theta \) if \( \forall X_r : X_r \cdot \exists X_g : X_g \cdot \forall Y_r : Y_r \cdot X_r = X_g \theta \land Y_r = Y_g \theta \land O_r(X_r, Y_r) = O_g(X_g, Y_g) \theta \).

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 \( \theta \) 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:

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

Note that \( A \) is free in the \( \lambda \)-term substituted for \( P_1 \); this does not pose a problem because \( (R, C, A) \) 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 \( \theta \) under which it reduces to the generic specification \( S_g \) attached to some particularisation \( P_g \) of the global search schema, and then apply \( \theta \) 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 \( O \) 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:

\[
\text{constrain}(\delta, D, \{\_, \_, A\}) \leftarrow \\
\delta \equiv (R_1, C_1), \\
D \equiv (\{\langle R_2, C_2 \rangle | M'\}), \\
\text{member}(\langle R_1, R_2 \rangle, A) \leftarrow C_1 \neq C_2, \\
\text{constrain}(\delta, \{\_, M'\}, \{\_, \_, A\})
\]

by virtue of the substitution \( \theta \) (see Example 5.1). Note that we here use \( P \leftarrow Q \) to denote not \( P \); \( Q \), where \( \lor \) 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.
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).

![Graph](image)

**Figure 1**: Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Map Colouring (France)</th>
<th>Hamiltonian Path</th>
<th>8-Queens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesised CLP(FD) programs</td>
<td>27,150 ms</td>
<td>50 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td>Synthesised LP programs</td>
<td>overflow</td>
<td>527 ms</td>
<td>3200 ms</td>
</tr>
<tr>
<td>Hand-crafted CLP(FD) programs [2]</td>
<td>5,230 ms</td>
<td>20 ms</td>
<td>30 ms</td>
</tr>
</tbody>
</table>

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], Conjunto [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 *constraint-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 Passign particularisation, the substitution under which a given specification
reduces to a specification template like Sassign 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 Passign 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


[5] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical


Kestrel Institute, 1988.
