1 Introduction

Optimisation problems — where appropriate values for the variables of the problem have to be found, subject to some constraints, such that some cost function on these variables takes an optimal value — are ubiquitous in industry. Examples are production planning subject to demand and resource availability so that profit is maximised, air traffic control subject to safety protocols so that flight times are minimised, transportation scheduling subject to initial and final location of the goods and the transportation resources so that delivery time and fuel expenses are minimised, etc. A particular case are decision problems, where there is no cost function that has to take an optimal value. Many of these problems can be expressed as constraint programs and then be solved using constraint solvers.

However, effective constraint programming (CP) [9] is very difficult, even for application domain experts, and hence time-consuming. Moreover, many of these problems are ill-behaved, in the sense that it can be shown that solving them requires an amount of time that is worse than polynomial in the size of the input data, hence making solving times prohibitively long.

To address the programming time problem, ever more expressive and declarative programming languages are being developed. To address the solving time problem, a search procedure and redundant constraints can be added so as to accelerate the default solution enumeration. If expressive declarative languages cannot be compiled into acceptably fast code, then the advantage of decreased programming time is neutralised by the disadvantage of increased solving time. We call modeling the usage of a CP language for expressing an optimisation/decision problem. This results in a model, which thus has a constraint part and an optional search part.

We here report on our initial results regarding the compilation of very expressive, purely declarative, typed, first-order set constraint logic programs into clp(FD) programs [3]. For simplicity, we only consider decision problems.

In Section 2, we discuss our set constraint programming language as well as (syntactic forms of programs of) the target language clp(FD). Then, in Section 3, our notion of schema-guided compilation is presented. Finally, in Section 4, we conclude, compare with related work, and discuss our directions of future work. (For reasons of space, undefined concepts are set between single quotes.)

2 Programs

All programs, whether of the input or target language, are here theories in some typed, first-order logic. The input language, let us call it $S$ until we have decided on a suitable name for it, is partially introduced below. The target language here is clp(FD), but our compilation technique is independent of that choice. We consider open (or: parametric) programs, together with the corresponding notion of ‘open equivalence’ (or: parametric equivalence). Hence the following definitions.

Definition 2.1 A relation symbol $r$ occurring in a theory $T$ in a language $L$ is open in $T$ if it is neither ‘defined’ in $T$, nor a primitive symbol in $L$. A non-open symbol in $T$ is a closed symbol in $T$. A theory with at least one open symbol is called an open theory; otherwise it is a closed theory.

Definition 2.2 A program for a relation $r$ is a possibly open theory that ‘defines’ $r$.

Program $P$, whether open or closed, is a refinement of open program $T$ under extension $\theta$ if $\theta$ is a set of programs ‘defining’ some of the open symbols of $T$ such that $P$ and $T \cup \theta$ are ‘open-equivalent’.

Note that closed programs thus do not have refinements. We ask the reader to overlook the fact that in our examples we use a much more general definition of refinements.

Example 2.1 Let us first look at our input language $S$. The chosen representation of sets in $S$, called the external representation, is the classical one, with curly braces. Programs in $S$ are iff-programs, expressing that, under input condition $i$, on input $X$, a program for relation $r$ must succeed if and only if output condition $o_r$ on $X$ and output $Y$ holds. Formally, this gives rise to the following open program:

$$\forall X : \text{term}. \ \forall Y : \text{term}. \ \ i_r(X) \Rightarrow (r(X,Y) \Leftrightarrow o_r(X,Y))$$

The only open symbols are relations $i_r$ and $o_r$. We here only consider programs for $k$-subsets decision problems, where a set $W$ of $K$ subsets of a given finite set $T$ (an integer set here) has to be found, such that $W$ satisfies an (open) constraint $g$ and an arbitrary two different elements of $W$ satisfy an (open) constraint $g'$; in addition, each element $S$ of $W$ satisfies an (open) constraint $q$ and an arbitrary two different elements of $S$ satisfy an (open) constraint $q'$. Hence

---

*This paper is a summary of [5], but tackles a different class of programs in its examples.*
The only open symbols are relations \( g, g', q, \overline{q} \) (assuming that \( \text{size} \) and \( \neq \) are primitives of \( S \), with the usual meaning). Note how the type of \( W \) ‘depends’ on \( T \). The open program \( \text{ksubsets} \) is a refinement of open program \( \text{iff} \) above. It has itself as refinements programs for many problems, such as finding \( k \) cliques of a graph (see below), crew scheduling, Steiner, etc. For instance, the closed program:

\[
\forall (T, K) : \text{set(int)} \times \text{int}. \forall W : \text{set(set(T))}. \\
\text{ksubsets}((T, K), W) \leftrightarrow \text{size}(W, K) \land g(W) \land \forall W_1, W_2 : W. W_1 \neq W_2 \rightarrow g'(W_1, W_2) \land \forall S : W. g(S) \land \forall I, J : S. I \neq J \rightarrow \overline{q}(I, J), \\
(\text{ksubsets}_{dec})
\]

The only open symbols are relations \( g, g', q, \overline{q} \) (assuming that \( \text{size} \) and \( \neq \) are primitives of \( S \), with the usual meaning). Note how the type of \( W \) ‘depends’ on \( T \). The open program \( \text{ksubsets} \) is a refinement of open program \( \text{iff} \) above. It has itself as refinements programs for many problems, such as finding \( k \) cliques of a graph (see below), crew scheduling, Steiner, etc. For instance, the closed program:

\[
\forall (V, K, A) : \text{set(int)} \times \text{int} \times \text{set(set(V))}. \\
\forall C : \text{set(set(V))}. \text{cliques}((V, K, A), C) \leftrightarrow \text{size}(C, K) \land \text{unionall}(C, V) \land \forall C_1, C_2 : C. C_1 \neq C_2 \rightarrow (\exists C_3 : \text{set(V)}). \exists F : \text{int}. \text{intersect}(C_1, C_2, C_3) \land \text{size}(C_3, F) \land F \leq 1 \land \forall E : C. \exists Z : \text{int}. \text{sum}(E, Z) \land Z > 10 \land \forall M, N : E. M \neq N \rightarrow \langle M, N \rangle \in A, \\
(\text{cliques}_{dec})
\]

is a refinement of \( \text{ksubsets}_{dec} \), under the extension

\[
\forall V : \text{set(int)}. \forall C : \text{set(set(V))}. \\
g(C) \leftrightarrow \text{unionall}(C, V) \\
\forall V : \text{set(int)}. \forall C : \text{set(set(V))}. \forall C_1, C_2 : C. \\
g'(C_1, C_2) \leftrightarrow \exists C_3 : \text{set(V)}. \exists F : \text{int}. \text{intersect}(C_1, C_2, C_3) \land \text{size}(C_3, F) \land F \leq 1 \land \forall V : \text{set(int)}. \forall C : \text{set(set(V))}. \forall E : C. \\
g(E) \leftrightarrow \exists Z : \text{int}. \text{sum}(E, Z) \land Z > 10 \\
\forall (V, A) : \text{set(int)} \times \text{set(set(V))}. \forall C : \text{set(set(V))}. \\
\forall A : C. \forall M, N : E. g(M, N) \leftrightarrow \langle M, N \rangle \in A, \\
(\sigma)
\]

assuming that \( \text{sum}, \text{intersect}, \text{unionall}, >, \leq, \) and \( \in \) are primitives of \( S \), with the obvious meanings. It is a program for a particular case of the \( k \)-cliques problem, namely finding \( k \) cliques (or maximally connected components) that divide an undirected graph (which is given through its integer-labeled vertex set \( V \) and its arc set \( A \)), such that for every clique the sum of its integer labels exceeds 10, and for every two different cliques, the cardinality of their intersection is at most one.

**Example 2.2** Let us now look at the target language, namely clp(FD) [3]. Among the many possible forms of programs, there are the global search [11, 12] programs. Formally, their dataflow and control-flow can be captured in the following open clp(FD) program for \( r \) [4]:

\[
\begin{align*}
\text{r}(X, Y) & \leftarrow \text{initialise}(X, D), \\
& \text{rgs}(X, D, Y'), \\
& \text{generate}(Y', Y, X) \\
\text{rgs}(X, D, Y') & \leftarrow \text{extract}(X, D, Y'), \\
& \text{split}(D, X, D', \delta), \\
& \text{constrain}(\delta, D, X), \\
& \text{rgs}(X, D', Y')
\end{align*}
\]

The only open symbols are relations \( \text{initialise}, \text{generate}, \text{extract}, \text{split}, \) and \( \text{constrain} \). Note that \( \text{constrain} \) just poses constraints on the search space, the actual solutions being enumerated by \( \text{generate} \) once all constraints have been posed, because we use a constraint language.

The \( gs_{dec} \) program can be refined for \( k \)-subsets decision problems, yielding a still (open) program \( gs_{dec}^{\text{ksubsets}} \) by extending \( gs_{dec} \) with \( k \)-subsets (see below), where \( in, \# \land, =, >, \) and \( \text{labeling} \) are primitives of clp(FD) (with the obvious meanings). The relation \( \text{flatten} \) has the usual meaning.

\[
\begin{align*}
\text{initialise}(X, D) & \leftarrow X = \langle V, Co, \rangle, \\
& \text{kinit}(Co, \langle L, H \rangle), \\
& D = \langle V, L, H \rangle, \\
\text{extract}(\text{...,} D, Y') & \leftarrow D = \{\}, V, W), \\
& Y' = \langle V, W \rangle, \\
& V = \langle [T], W = \langle [T], \rangle, \\
& \text{pose}_g(Y'), \text{pose}_g(Y', \langle T, T' \rangle), \\
& g(Y') \\
\text{pose}_g(Y') & \leftarrow Y' = \{\}, [1] \\
\text{pose}_g(Y') & \leftarrow Y' = \langle \{A[T], [B[T] \rangle, \\
& Y_1 = \langle A, B \rangle, q(Y_1), \\
& \text{pose}_g(T, T') \\
\text{pose}_g(\text{...,} Q) & \leftarrow Q = \{\}, [1] \\
\text{pose}_g(Y', Q) & \leftarrow Y' = \langle \{A[T], [B[T] \rangle, \\
& Q = \langle \{C[T_2], [D[T_3] \rangle, \\
& Y_1 = \langle A, B \rangle, \text{pose}_g(\langle T, T_1 \rangle, \langle T_2, T_3 \rangle) \\
\text{split}(D, X, D', \delta) & \leftarrow D = \langle \{A[T], L, K \rangle, \\
& X = \langle \text{...,} Co, \rangle, \\
& \text{ss}(A, L, K, F, M, \delta, Co), \\
& D' = \langle T, F, M \rangle, \\
\text{constrain}(\delta, D, \text{...,} \rangle & \leftarrow \delta = \{\}, D = \langle \{\}, [1] \rangle \\
\text{constrain}(\delta, D, X) & \leftarrow \delta = \{\delta_1, \delta_2 \}, \\
& D = \langle A[T], [B[T] \rangle, \\
& \text{cons} \text{aux}(\delta_1, \langle A, B \rangle, X), \\
& \text{constrain}(\delta_2, \langle A, T_1, T_2 \rangle, X) \\
\text{cons} \text{aux}(\delta, D, \text{...,} \rangle & \leftarrow D = \langle A, B \rangle, \\
& D = \langle \{E[V], [F[W] \rangle, \\
& B \# \land F \rightarrow p(A, E), \\
& \text{cons} \text{aux}(\delta, \langle V, W \rangle, X) \\
\text{generate}(Y', Y, \text{...,} \rangle & \leftarrow Y' = \langle V, W \rangle, \\
& \text{flatten}(V, W'), \\
& \text{labeling}(\{], W'), \\
& \text{int2ext}(\langle V, W' \rangle, Y') \\
\text{kinit}(Co, Q) & \leftarrow \ldots \\
\text{ss}(\text{...,} \rangle & \leftarrow \ldots \\
E \in S & \leftarrow \ldots \\
\text{sum}(S, N) & \leftarrow \ldots \\
\ldots \ldots & \leftarrow \ldots \\
(\sigma_{ksubsets})
\end{align*}
\]
3 Schema-Guided Compilation

We can now define the semantic notion of program schema, which is intended to represent entire families of similar programs.

Definition 3.1 A program schema is a couple \( \langle T, A \rangle \), where template \( T \) is an open program, and axioms \( A \) are open formulas constraining the refinements of \( T \).

Program \( P \) is a refinement of program schema \( \langle T, A \rangle \) if \( P \) is a refinement of \( T \) under some extension \( \theta \), provided the ‘definitions’ in \( P \) ‘satisfy’ the axioms \( A \).

The template of a program schema captures the problem-independent dataflow and control-flow of an entire family of programs, whereas some refinement thereof (such that the axioms are ‘satisfied’) captures the problem-dependent constraints of a member of that family.

Example 3.1 Let \( K_{\text{dec}} \) denote the \( S \) program schema obtained from template \( k_{\text{dec}} \) (of Example 2.1) and the empty set of axioms, as we do not wish to impose any conditions on the (open) relations \( g, g', q, q' \).

Example 3.2 Let \( G_{k_{\text{dec}}} = \langle g_{k_{\text{dec}}}, \emptyset \rangle \) denote the \( S \) program schema obtained from template \( g_{k_{\text{dec}}} \) (of Example 2.2) and the empty set of axioms, as we again do not wish to impose any conditions on the (open) relations \( g, g', q, q' \).

We next introduce the semantic notion of programming schemas, which are useful for guiding our compilation.

Definition 3.2 A programming schema is a triple \( \langle D_1, E, D_2 \rangle \), where \( D_1 \) and \( D_2 \) are program schemas, and open formula \( E \) is the condition under which any refinement of \( D_2 \) under some extension \( \theta \) is ‘open-equivalent’ to the corresponding refinement of \( D_1 \) under \( \theta \).

Example 3.3 The triple \( \langle K_{\text{_dec}}, \text{true}, G_{k_{\text{dec}}} \rangle \) is a programming schema, capturing the compilation into global search clp(FD) programs of \( k \)-subsets decision programs in \( S \). Note that the open symbols in its input and target program schemas are the same, namely \( g, g', q, q' \).

A very large percentage of global search programs falls into one of seven families identified by Smith [11], each being a refinement of the \( g_{\text{dec}} \) template where programs for all its open relations are chosen in advance so as to ‘satisfy’ the axioms in [4]. These refinements are then still open, though no longer in the problem-independent relations of the target \( g_{\text{dec}} \) template, but now in the problem-dependent relations of the input \( S \) template (such as in Example 3.3).

A particular case of schema-guided compilation is thus apparent now: given a program \( P \) in \( S \) and a programming schema \( \langle D_1, E, D_2 \rangle \), where the open symbols in \( D_1 \) and \( D_2 \) are the same, find an extension \( \theta \) under which \( P \) refines \( D_1 \), verify the equivalence condition (i.e., whether \( \theta \models E \)), and return the refinement \( D_2 \cup \theta \), which is in clp(FD). In other words, schema-guided compilation then simply amounts to replacing the template part of a refinement by another template, if the equivalence condition holds.

4 Conclusion

In this progress report on our research, we have introduced a novel approach to the compilation of very expressive, purely declarative, typed, first-order set constraint logic programs. While awaiting (more) efficient set constraint solvers, we have decided to exploit the well-known efficiency of finite-domain solvers and to compile (or: reformulate) set constraints into finite-domain constraints. Our approach is independent of the target language.

We here have presented the compilation of the family of \( k \)-subsets decision problems, which is not among the families of global search problems identified by Smith. Elsewhere, we have already discussed how to do this for the families of assignment problems and permutation problems [4], binary split problems [7], and subset problems [5]. The remaining family identified by Smith (two of his seven families actually are particular cases of other ones), namely finding sequences of (given or bounded) size over a given set, has been tackled but is not presented here due to the space restrictions.

We are ready to sacrifice any general-purpose nature of our input language, by just developing programming schemas for some problem families, if this is what it takes to facilitate (and speed up) constraint program development and to speed up the compiled programs.

Related Work

This work is inspired by D.R. Smith’s research on synthesising global search programs (in Refine) from first-order
logic specifications (also in Refine) with KIDS [11, 12] and its successor DESIGNWARE [13]. Our work concentrates on generating constraint programs instead. We thus only have to generate code that (incrementally) poses the constraints, because the actual constraint propagation and pruning are performed by the CP system. We have thus detached the problem-specific model (constraint part + search part) from the problem-independent solver. This allows us to generate only the model and to re-use the solver, whereas DESIGNWARE synthesizes both. This required a significant re-engineering [4] of the original global search schema, so that it reflects a constrain-and-generate programming methodology.

Tests [4] have shown that at least one order of magnitude is gained in efficiency (before optimisation) by switching from an “ordinary” symbolic language, such as Refine, to a constraint one. In addition, our generated clp(FD) programs behave much more gracefully when the problem size increases, rather than seeing their run-times degenerate with problem size. These tests also showed that our automatically generated clp(FD) programs are only 3 to 5 times slower than carefully hand-crafted, published clp(FD) programs, which is encouraging since no optimisations are performed yet on our programs. Since our compilation is fully automatic, starting from short and elegant programs, our approach seems viable.

The other novelty compared to the DESIGNWARE approach is that we advocate that program schemas can be of (much) lower granularity than those of Smith (i.e., our templates can be refinements of his templates), so that the selection of the most appropriate programming schema, via the notion of refinement, can be done through rather trivial theorem proving, such as by performing matching. Moreover, our differentiation between several program schemas where Smith only considers a single one allows the manual, offline pre-computation, at schema design time, of more details of the corresponding equivalent program schema (such as the filters of global search), which can otherwise only be found, at compilation time, through sophisticated theorem proving. Hence we facilitated (and speeded up) compilation and could even further speed up the compiled programs.

Other than inherently being a computationally incomplete language (by being limited to a finite set of problem families), our input language compares as follows to other languages. Compared to the CLPS [1], COJUNTO [6], and NP-SPEC [2] set constraint languages, our input language is more expressive (by not being limited to sets of initially known size), and by design much faster. The OZ [14] language also allows sets of a priori unknown size [10], and is faster than ours, but it is less expressive and less declarative. The OPL [15] constraint language sets new standards in expressiveness, but is currently limited to membership constraints with ground sets (of known size). The DESIGNWARE system [13] maybe allows the formulation of constraints on possibly infinite sets and on sets whose elements are drawn from an infinite domain, but its fully automated synthesis sub-system can only generate very slow programs (though its user-guided transformation sub-system can optimise them into extremely fast code.)

**Future Work**

Our plan now is to complete our investigation on adapting Smith’s work to the CP paradigm, and then to port the results to the very recent OPL [15], which is much more expressive, but not (significantly) faster or more declarative than clp(FD). We have already detected expressiveness and declarativeness gaps in OPL, so we can continue to deploy our approach.

We wish to achieve automated optimisation of the compiled programs by the generation of a problem specific search part and by adding redundant constraints that often accelerate the solver.

**Acknowledgements**

We sincerely thank our advisor, Pierre Flener, for his invaluable advice and encouragement. We also thank Hamza Zidoum (UAE University), for insightful ideas during the incubation time for this research.

**References**


