Tactical Synthesis Of Efficient Global Search Algorithms

Srinivas Nedunuri
Dept. of Computer Sciences
University of Texas at Austin
nedunuri@cs.utexas.edu

Douglas R. Smith
Kestrel Institute
smith@kestrel.edu

William R. Cook
Dept. of Computer Sciences
University of Texas at Austin
cook@cs.utexas.edu

Abstract

Algorithm synthesis transforms a formal specification into an efficient algorithm to solve a problem. Algorithm synthesis in Specware combines the formal speicifcation of a problem with a high-level algorithm strategy. To derive an efficient algorithm, a developer must define operators that refine the algorithm by combining the generic operators in the algorithm with the details of the problem specification. This derivation requires skill and a deep understanding of the problem and the algorithmic strategy. In this paper we introduce two tactics to ease this process. The tactics serve a similar purpose to tactics used for determining indefinite integrals in calculus, that is suggesting possible ways to attack the problem.

1 Background

There have been a variety of approaches to program synthesis (e.g. see [Kre98] for a survey). The focus of this paper is an algorithm class called Global Search (GS) [Smi88]. Using this algorithm class, Smith and his colleagues have successfully synthesized a number of practical algorithms, including, in one case, a scheduler that ran several orders of magnitude faster than comparable hand-written ones [SPW95]. The starting point is a specification $\langle D, R, O \rangle$, where D is an input type, R an output type, and $O : D \times R \rightarrow Boolean$ is an output or correctness condition, along with a global search theory extension (described below). Then the following program, given an input x, returns a solution $z : R$ satisfying the output condition, if one exists (there are some additional conditions on R which will be explained shortly):

```
f(x:D) : R =
    if propagate(x,r0(x))=None then None else gs(x,r)
gs (x:D, r:R) : R =
    let z=Extract(r) in if z/=None && O(x,z) then z else gsAux(x,Subspaces(r))
gsAux (x:D, subs:{R}) :  R =
        if subs={} then None
        else let (s, rest) = arbsplit(subs) in
             if propagate(s) = None then gsAux(x, rest)
             else let  z= gs(x,s) in
                  if z = None  then gsAux(x,rest) else z
propagate(x, r) =  if Φ(x,r) then iterateToFixedPoint(ψ, x, r)  else None
iterateToFixedPoint (f, x, r) =
    let fr = f(x,r) in if FP?(fr,r) then fr else iterateToFixedPoint(f, x, fr)
```

The program is a classic search algorithm. It works by taking an initial space of possible solutions (corresponding to the root node of a search tree), and unless it can immediately extract a feasible solution, partitioning it into subspaces (corresponding to child nodes), each of which is searched in turn until a feasible solution is found. In this paper we provide tactics for synthesizing the operators Φ and ψ.

The remaining functions are defined in the global search theory extension, *GS-ext,* supplied by the developer, which is an algebra over R with the following operators: $r_0 : D \rightarrow R$ returns a descriptor of the initial search space, $Extract{:}R \rightarrow R$ determines whether the given space is terminal and if so, returns a solution (otherwise the distinguished element None, denoting an empty space). $Subspaces : R \rightarrow \{R\}$

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 171-175

returns a set of subspaces of the current space,$\Phi : D \times R \rightarrow Boolean$ is a necessary filter - those spaces that do not pass Φ need not be examined. It can be any predicate over R satisfying $r \sqsubseteq z \wedge O(x,z) \Rightarrow \Phi(x,r)$. $\sqsubseteq$ is a refinement relation over R. The intent of $\sqsubseteq$ is that if $r \sqsubseteq s$ then s is is a subspace of r (any solution contained in s is contained in r) and is "more defined" than r . $\psi : D \times R \rightarrow R$ is called a necessary propagator. It "tightens" a given space to eliminate infeasible solutions and can be any predicate satisfying $\forall z.\, r \sqsubseteq z \wedge O(x,z) \Rightarrow \psi(x,r) \sqsubseteq z$. When $\langle R, \sqsubseteq \rangle$ forms a lattice, Smith et al. [SPW95] show how a monotone inflationary ψ can be iterated from any starting space to a fixpoint which is the tightest possible space that still preserves all the original feasible solutions. That is what the *propagate* function in the abstract program above does. An axiomatic definition of GS theory and proof of correctness of the abstract program can be found in [Smi88].

1.1 A Constraint Satisfaction Theory

We are developing a specialization of Global Search to solve problems that involve multi-variable Constraint Satisfaction (CS) [Dec03]. Unlike generic constraint solvers [San94], which accept constraints as input and find a solution, in our approach the constraint is the output condition of the problem to be solved. This constraint is the starting point of algorithm synthesis, not dynamic constraint solving. In this way, many of the problems we will look at can be solved by constraint satisfaction,[Dec03]. For this reason, it is useful to have a specialization of the GS class for Constraint Satisfaction (CS) problems, which we can later extend to each specific problem as needed.

In a nutshell, constraint satisfaction does the following:: given a set of variables, $\{1..maxVar\}$, assign a value, drawn from some domain D_v, to each variable, in a manner that satisfies some set of constraints. The theory which does this, we call CST, is defined below. All other domain specific theories we will use will monotonically extend this theory.

$R \mapsto m : Map(Nat \rightarrow D_v) \times tbd : \{Nat\} \times ch : Map(Nat \mapsto \{D_v\})$
$D \mapsto maxVar : Nat \times vals : \{D_v\}$
$O \mapsto \lambda x, z.\ dom(z.m) = \{1..x.maxVar\}$
$r_0 \mapsto \lambda x.\ \{m = \emptyset, tbd = \{1..x.maxVar\}, ch = \{(v \mapsto x.vals) | v \in \{1..x.maxVar\}\})$
$Subspaces \mapsto \lambda x, \widehat{z}.\ \{\widehat{z}' : v = pick(\widehat{z}.tbd) \wedge a \in \widehat{z}.ch(v) \wedge \widehat{z}'.m = \widehat{z}.m \oplus \{v \mapsto a\} \wedge \widehat{z}'.tbd = \widehat{z}.tbd - \{v\}\}$
$Extract \mapsto \lambda z.$ if $z.tbd = \emptyset$ then z else $None$
$\sqsubseteq \mapsto \{(\widehat{z}, \widehat{z}') | \widehat{z}.m \subseteq \widehat{z}'.m\}$
$\Phi \mapsto \lambda x, \widehat{z}.\ True$
$\psi \mapsto \lambda x, \widehat{z}.\ \widehat{z}$

In this theory, branching occurs via the *subspaces* function. The *subspaces* function, after picking a variable from the set of variables not yet assigned a value (*tbd*), returns the subspaces formed by assigning to v each of the possible values (drawn from ch*(v)*, adding each pair to the map m, and removing v from tbd. The initial space r_o makes all the values in $x.vals$ available to every variable. The choice of which variable to pick does not matter functionally, but can have a significant impact on the efficiency of the actual program. We will often abbreviate $\widehat{z}.m(i)$ as $\widehat{z}_i$. Now with a definition for D_v, and whatever conditions are appropriate added to O, the abstract program given earlier becomes a working constraint satisfaction solver. The key to making it efficient are appropriate definitions for Φ and ψ[1] . This is what the next section examines.

[1]Often, further optimizations such as context-dependent simplification, finite differencing, and data structure selection have to be carried out before arriving at a final efficient algorithm . However, these latter operations are not the focus of this paper.

2 Tactics

In order to get an efficient final algorithm, the developer must typically find good instantiations of the operators Φ and ψ. The question of where to begin often arises. For this reason we propose to formulate a library of *tactics* that can be used by a developer attempting to instantiate one of the operators. The analogy is with tactics used for integration in calculus. Unlike differentiation, integration has no straightforward algorithm. Rather, there are a number of (some 7 or 8) tactics such as "integration by parts", "integration by partial fractions", "integration by change of variable", etc., that can be tried in order to try and determine the integral of a given formula. There are of course differences. Unlike integration, there is often no one "correct" answer. Also our tactics are often inspired by techniques used in algorithms in computer science and operations research, rather than calculus. But the basic principle is the same: to package up a number of tedious calculations into a pattern-matching rule. Furthermore, by expressing the technique in more abstract form as a tactic, it can be applied to other problem areas, *without requiring the developer be familiar with the implicit assumptions and notations when the technique is buried inside a specific algorithm*, The ultimate goal is that a competent developer will be able to use the approach we propose here to investigate a variety of solutions to their problem.

2.1 A Tactic for Calculating Φ

This tactic helps in constructing Φ (necessary) filters when the feasibility constraint takes a certain form.

TACTIC 1: *If a conjunct from O matches the form $\bigotimes_{i\in I} F_i(z_i) \preceq K$ where $\bigotimes$ is a monotone associative operator, and $\preceq$ forms a meet semi-lattice over range(F), then a possible Φ is one in which the combination of value assignments in the partial solution combined ($\otimes$) with the least possible value assignments for the remaining variables is $\preceq K$.*

The tactic is backed by the following theorem. Note, $\oplus$ denotes extending a partial solution, that is $\widehat{z}\oplus e$ means $\widehat{z}.m \cup e.m$ (unless otherwise stated, we will always be assuming $dom(\widehat{z}.m)\cap dom(e.m)=\emptyset$)

Theorem 1. If $O(x,z) \Rightarrow \bigotimes_{i\in I} F_i(z_i) \preceq K$ for some K, some family of functions $\{F_i\}$, $\bigotimes$ a monotone associative operator, and $\preceq$ forms a meet semi-lattice over $rng(F)$, *then*

$$O(x,\widehat{z}\oplus e) \Rightarrow (\bigotimes_{1\le i\le \#\widehat{z}} F_i(\widehat{z}_i) \otimes \bigotimes_{1\le i\le \#e} f_i) \preceq K \text{ where } f_i = \sqcap_{a\in x.vals} F_i(a)$$

Proof.

$O(x,z)$
$\Rightarrow$ {assumption}
$\bigotimes_{1\le i\le \#z} F_i(z_i) \preceq K$
$=$ {$z = \widehat{z}\oplus e$ and use associativity of $\otimes$}
$\bigotimes_{1\le i\le \#\widehat{z}} F_i(\widehat{z}_i) \otimes \bigotimes_{1\le i\le \#e} F_i(e_i) \preceq K$
$\Rightarrow$ {replace every $F_i(e_i)$ with $f_i = \sqcap_{a\in x.vals} F_i(a)$ and use polarity}
$\bigotimes_{1\le i\le \#\widehat{z}} F(\widehat{z}_i) \otimes \bigotimes_{1\le i\le \#e} f_i \preceq K$

□

Additionally, if F_i is monotone, and $x.vals$ has a least element, then, using the following Quantifier Elimination law: $\sqcap_{\check{a}\preceq a} F_i(a) = F_i(\check{a})$, we can rewrite the last line above as:$\bigotimes_{1\le i\le \#\widehat{z}} F_i(\widehat{z}_i) \otimes \bigotimes_{1\le i\le \#e} F_i(\check{a}) \preceq K$ where $\check{a}$ is the least value of $a \in x.vals$.

A symmetrical result is obtained by replacing $\preceq$ with $\succeq$, "meet semi-lattice" with "join semi-lattice", $\check{a} \preceq a$ with $\hat{a} \succeq a$, and $\sqcap$ with $\sqcup$ everywhere in the above theorem.

Example 2. 0-1 Integer Linear Programming (01-ILP) [2]

A GSO theory for 01-ILP is obtained by extending CST as follows (only the components that differ from CST are shown): $D_v \mapsto \{0,1\}$, $D \mapsto CST.D \times l : Nat \times \mathbf{A} : Map(\{1..l\} \times \{1..n\} \mapsto Real) \times \mathbf{b} : Map(\{1..n\} \mapsto Real)$, $O \mapsto \lambda(x,z).\ CST.O(x,z) \wedge (x.\mathbf{A}) \cdot (z.m) \leq x.\mathbf{b}$

To apply the tactic above, the operator $\bigotimes$ is interpreted as $\sum$ and F_i as $(\mathbf{A}_{hi}\cdot)$ for appropriate h , over the lattice $\langle Real, \leq, \min, \max\rangle$. Applying the tactic (but not the final simplification since $(\mathbf{A}_{hi}\cdot)$ is not monotone) gives the following filter Φ:$\forall h.\ 1 \leq h \leq 1.\ \sum_{i\in dom(\hat{z}.m)} \mathbf{A}_{hi} \cdot z_i + \sum_{i\in dom(e.m)} (\min_{a\in\{0,1\}}\{\mathbf{A}_{hi} \cdot a\}) \leq \mathbf{b}_h$ which is by case analysis: $\forall h.\ 1 \leq h \leq l.\ \sum_{i\in dom(\hat{z}.m)} \mathbf{A}_{hi} \cdot z_i + \sum_{j\in dom(e.m)} (\min\{\mathbf{A}_{hi} \cdot 0, \mathbf{A}_{hi} \cdot 1\}) \leq \mathbf{b}_h$, or after simplifying:

$$\forall h.\ 1 \leq h \leq l.\ \sum_{i\in dom(\hat{z}.m)} \mathbf{A}_{hi} \cdot z_i + \sum_{i\in dom(e.m)} (\min\{0, \mathbf{A}_{hi}\}) \leq \mathbf{b}_h$$

Using the same tactic we have obtained a filter for the Vehicle Routing Problem (VRP) equivalent to one used in algorithm textbooks. The next example shows that the generalization offered by the tactic is indeed useful enough to carry over to other qualitatively differet problems.

Example 3. The Set Covering Problem (SCP)

Suppose we are given a collection of subsets of a set S, each of which has a certain cost. The SCP is the problem of determining the minimum cost collection of subsets that "covers" the original set, ie. every element in S is in at least one subset in the resulting collection. The problem has many practical applications including airline crew scheduling, facility location, and logic circuit minimization. A GSO theory for SCP is obtained by extending CST as follows (only the components that differ from the base theory are shown): $D_v \mapsto \{False, True\}$, $D \mapsto CST.D \times ss : Map(Nat \mapsto \{Id\})$, $O \mapsto \lambda(x,z).\ CST.O \wedge \bigcup_{i|z_i} S_i = S$

Id is some user defined set element type. *x.ss* returns the actual subset given a variable from $\{1..x.maxVar\}$. S_i stands for the subset $x.ss(i)$, and S stands for $\bigcup_{i\in\{1..x.maxVar\}} S_i$. To apply the tactic, we instantiate $\bigotimes$ as $\cup$, F_i as $\lambda z_i.\ z_i \rightarrow S_i \,|\, \{\}$, over the join semi-lattice $\langle\{S\}, \subseteq, \{\}, \{S\}\rangle$. Certainly, $\bigcup_{i|z_i} S_i = S$ implies $\bigcup_{i|z_i} S_i \supseteq S$ that is, $\bigcup_i F_i(z_i) \supseteq S$. Applying the tactic gives us a filter $\bigcup_i F_i(\widehat{z}_i) \cup \bigcup_i F_i(S_i) \sqcup \{\}) \supseteq S$ = $\bigcup_i F_i(\widehat{z}_i) \cup \bigcup_i F_i(S_i) \supseteq S$, that is if at any point, the union of the selected sets in $\widehat{z}$ along with all the remaining sets is not at least S, then the space $\widehat{z}$ can be eliminated.

2.2 A Tactic for Calculating ψ

Observe that in the initial space r_o all value choices (from D_v) are available to every variable. The intent, though, is that propagation will narrow this set to only those that would lead to feasible solutions (analogous to hyper-arc consistency in CSP). If at any point a choice set becomes empty, then that space can be abandoned. This is the idea behind the following tactic for ψ . The tactic applies when the variables (*vars*) of the input can be viewed as, or represent, nodes in some kind of graph structure, so we can talk about the "neighborhood" around a variable.

TACTIC 2: *If one of the conjuncts of O matches the form $\forall j \in N_i.\ z_i \neq z_j$ where N_i is some neighborhood of points around i then a possible ψ is* one in which the choice of values available to variable *j* does not contain the value assigned to variable *i*.

The tactic is backed by the following theorem

[2] To simplify the presentation we have omitted the optimization aspect of many of the examples we discuss since none of our tactics pertain to optimization. In our actual implementation we use a generalization of GS that incorporates optimization.

Theorem 4. *If* $O(x,z) \Rightarrow \forall j \in N_i.\, z_i \neq z_j$ *for some set* $N_i \subseteq x.vars$ *then*

$$\widehat{z} \sqsubseteq z \wedge O(x,z) \Rightarrow \psi(x,\widehat{z}) \sqsubseteq z \text{ where } \psi(x,\widehat{z}) = \widehat{z}\{ch(j) = \widehat{z}.ch(j) - \widehat{z}_i \mid j \in N_i\}$$

where the notation $o\{f(i) = v \mid P(i)\}$ denotes the object obtained by replacing the value of the ith index of field f of object o with v when $P(i)$ holds. The value is unchanged otherwise.

Example 5. Maximum Independent Segment Sum Problem (MISS), [SHT00]

This is a variant of the well-known maximum segment sum problem (MSS) in which the goal is to maximize the sum of a selection of elements from a given array, with the restriction that no two adjacent elements can be selected. The specification of the problem is as follows: $D_v \mapsto \{False, True\}$, $D \mapsto CST.D \times data : [Int]$, $O \mapsto \lambda(x,z).\, CST.O \wedge \forall i : 1 \leq i < \#z.m.\ : z_i \Rightarrow \neg z_{i+1}$

Now let N_i be the left and right neighbors of i, i.e. $i-1$ and $i+1$, if z_i and $\{\}$ otherwise. Then in the case where z_i holds, $\psi(\widehat{z}) = \widehat{z}\{ch(i+1) = ch(i+1) - \{True\}\}$ which is just $\widehat{z}\{ch(i+1) = \{True\}\}$.

Using this tactic we have also derived a ψ function for the Graph Coloring Problem and a variety of puzzles including n-Queens and Sudoku.

2.3 Summary and Future Work

We have shown how for certain problem types, calculation of the operators Φ and ψ can be replaced by pattern matching and substitution. The lesson here for program synthesis is that narrowing down the range of problem types can lead to much faster program design. We have developed a number of other such tactics, which space does not permit us to describe here. We can also handle optimization problems by incorporating dominance relations [Smi88] and bounds tests into our approach, and have developed a number of tactics for their calculation. Using one such tactic, we have synthesized a previously unpublished greedy solution to the Unbounded Knapsack Problem, and another tactic for dominance relations led us to fast solutions to variants of the Maximum Segment Sum problem that improve on the work of Sasano et al.,[SHT00]. Our eventual goal is to have a library of tactics sufficient to tackle significant Global Search problems such as synthesizing fast planners and efficiently mapping platform independent models to platform specific models.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant CCF-0724979

References

[Dec03] R Dechter. *Constraint Processing*. Morgan Kauffman, 2003.

[Kre98] Christoph Kreitz. Program synthesis. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III, chapter III.2.5, pages 105–134. Kluwer, 1998.

[San94] Michael Sannella. The skyblue constraint solver and its applications. In *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*, pages 385–406. MIT Press, 1994.

[SHT00] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. In *Proc. Intl. Conf. on Functional Prog.(ICFP)*, 2000.

[Smi88] D R Smith. Structure and design of global search algorithms. Technical Report Kes.U.87.12, Kestrel Institute, 1988.

[SPW95] Douglas R. Smith, Eduardo A. Parra, and Stephen J. Westfold. Synthesis of high-performance transportation schedulers. Technical report, Kestrel Institute, 1995.