Science of Computer Programming, 14(2-3), 1990, pp 305-321.

Algorithm Theories and Design Tactics

Douglas R. Smith and Michael R. Lowry
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304-1216 USA

Abstract

Algorithm theories represent the structure common to a class of algorithms, such as divide-and-conquer or backtrack. An algorithm theory for a class $\mathcal{A}$ provides the basis for design tactics – specialized methods for designing $\mathcal{A}$-algorithms from formal problem specifications. We illustrate this approach with recent work on the theory of global search algorithms and briefly mention several others. Several design tactics have been implemented in the KIDS/CYPRESS system and have been used to semiautomatically derive many algorithms.

1. Introduction

We describe an approach to the formal derivation of algorithms that is based on the notion of an *algorithm theory* which represents the structure common to a class of algorithms, such as divide-and-conquer or backtrack. Algorithm theories are abstract in several senses, the most important being problem-independence. They also abstract away implementation concerns about control strategy, target programming language, and, to some extent, the target architecture. By factoring out what is common to a class we hope to make it easier to apply the abstraction to particular problems. We have developed specialized construction methods, called *design tactics*, for various algorithm theories. We illustrate this approach with recent work on the theory of global search algorithms [18] and briefly mention theories of divide-and-conquer [16], local search [8, 9], and other classes of algorithms. We have implemented several of these design methods in the KIDS/CYPRESS system [19] and used them to semiautomatically derive many algorithms.

There are several advantages to representing the structure of a class of algorithms as a theory. Firstly, it abstracts away concerns about programming language and style (e.g. functional vs. logical vs. imperative, recursive vs. iterative), control strategy (e.g. top-down vs. bottom-up, depth-first vs. breadth-first vs. best-first), and, to some extent, target architecture (e.g. sequential vs. parallel). These concerns can be factored in as later decisions in the design

process. Secondly, once and for all we can derive abstract programs (schemes) as theorems in the abstract theory and then apply them to concrete problems. This allows us to reduce the problem of constructing a correct concrete algorithm to constructing an algorithm theory for a given problem. Thirdly, we can develop generic and thus highly-reusable design tactics on the basis of the abstract theory. Those design steps that are common to all instances of the class can be done just once in the abstract theory. The tactics that we have developed to date are sound, well-motivated, and mostly automatic in their implemented form. Fourth, our approach has much in common with current approaches to abstract data types and algebraic specifications which provides opportunities for fruitful interactions in the future. For example, the concept of global search underlies a number of well-known data structures such as binary search trees, quad-trees, and B-trees. In the expanded setting of data structure design, the global search concept provides a way to structure and access a dictionary (a set plus access operation). Finally, algorithmic theories can be combined to allow the inference of hybrid algorithms.

Several well-known program derivation methodologies, e.g. [3, 10], are based on inference rules for various programming language constructs - rules for inferring statement sequences, conditionals, loops, etc. Our complementary approach can be viewed as providing inference rules for various problem-solving methods or algorithmic paradigms. In a related approach, Bird [2] advocates a calculus of functional programs which exploits theorems relating problem structure to program structure.

Problems can be specified by means of a problem theory (Section 2). Designing an algorithm for a problem is mainly a matter of constructing an algorithm theory (Section 3). There are two ways to view this construction. From the point of view of the problem theory, the algorithm theory is an extension that provides just enough structure to support the construction of a concrete algorithm (Section 4). From the point of view of an abstract algorithm theory $\mathcal{A}$, the construction is a theory morphism or interpretation [8, 22]. These two points of views are tied together in the categorical concept of a pushout (Section 5). Constructing an algorithm theory for a given problem is accomplished by specialized design tactics (Section 5). Several design tactics have been implemented in the KIDS/CYPRESS system (Section 6) and used semiautomatically to derive dozens of algorithms. Key concepts of this paper are illustrated by application to the problem of enumerating cyclic difference sets [1].

2. Problem Theories and Extensions

We briefly review some concepts based on the abstract data type literature. A *theory* is a structure $\langle S, \Sigma, A \rangle$ consisting of sorts S, operations over those sorts Σ, and axioms A to constrain the meaning of the operations. A *theory morphism* (*theory interpretation*) maps from the sorts and operations of one theory to the sorts and expressions over the operations of another theory such that the image of each source theory axiom is valid in the target theory. A *parameterized theory* has formal parameters that are themselves theories [5]. The binding of actual values to formal parameters is accomplished by a theory morphism.

Theory $\mathcal{T}_2 = \langle S_2, \Sigma_2, A_2 \rangle$ *extends* (or is an *extension* of) theory $\mathcal{T}_1 = \langle S_1, \Sigma_1, A_1 \rangle$ if $S_1 \subseteq S_2$, $\Sigma_1 \subseteq \Sigma_2$, and $A_1 \subseteq A_2$. An extension can be represented by a special theory morphism called an *inclusion* that takes each sort and operation symbol to itself in the target theory. A category of theories can be formed by taking theories as objects, theory morphisms as arrows, and map composition as arrow composition.

Problem theories define a problem by specifying a domain of problem instances or inputs and the notion of what constitutes a solution to a given problem instance. Formally, a *problem theory* $\mathcal{B}$ has the following structure.

Sorts	D, R
Operations	$I : D \rightarrow Boolean$
	$O : D \times R \rightarrow Boolean$

The *input condition* $I(x)$ constrains the input domain D. The *output condition* $O(x, z)$ describes the conditions under which output domain value $z \in R$ is a *feasible solution* with respect to input $x \in D$. Theories of booleans and sets are implicitly imported. Problems of finding optimal feasible solutions can be treated as extensions of problem theory by adding a cost domain, cost function, and ordering on the cost domain (see [8, 18] for examples).

As a running example we use the problem of enumerating Cyclic Difference Sets (CDSs) [1, 14]. They are relatively rare sets that are somewhat analogous to primes in the natural numbers. The problem can be defined as follows. Given a modulus v, a set size k, and a constant ℓ, a $\langle v, k, \ell \rangle$*–cyclic difference set* C is a subset of $\{0..v-1\}$ that has size k. Furthermore, if we consider "rotating" C by adding an arbitrary constant i, where $i \neq 0 \ mod \ v$, to each element yielding a new set D, then C and D have exactly ℓ elements in common. For example, the simplest CDS is the $\langle 7, 3, 1 \rangle$-cds {0,1,3}. It has the property that for any i, $i \neq 0 \ mod \ 7$,

$$size(\{0, 1, 3\} \cap \{i + j \ mod \ 6 \mid j \in \{0, 1, 3\}\}) = 1,$$

for example, for $i = 4$ we have $size(\{0, 1, 3\} \cap \{4, 5, 1\}) = 1$.

Cyclic difference sets have been used for coding satellite communications, creating masks for X-ray telescopes, and other applications. Baumert [1] list all known CDSs for $k \leq 100$. These known sets were found by mathematical construction. Below we describe the derivation of a program to enumerate CDSs. We have used this program to discover a previously unknown CDS: the $\langle 13, 4, 1 \rangle$-CDS: {0,1,4,6}.

The problem of enumerating cyclic difference sets can be specified via a theory morphism $\mathcal{B} \mapsto \mathcal{B}_{CDS}$.

$$\begin{array}{lcl} D & \mapsto & Nat \times Nat \times Nat \\ I & \mapsto & \lambda\langle v, k, \ell \rangle.\ 1 \leq \ell \leq k < v \\ R & \mapsto & set(Nat) \\ O & \mapsto & \lambda\langle v, k, \ell \rangle, sub.\ sub \subseteq \{0..v-1\} \ \wedge \ size(sub) = k \\ & & \quad \wedge \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, sub) = \ell). \end{array}$$

where we define

$$self\_overlap\_under\_rotation(i, v, S) = size(S \cap \{\ (a + i)\ mod\ v \mid a \in S\ \}).$$

Derived laws in $\mathcal{B}_{CDS}$ include

$$self\_overlap\_under\_rotation(i, v, \{\}) = 0$$

$$self\_overlap\_under\_rotation(i, v, \{0..v-1\}) = v$$

$$S \subseteq T \implies self\_overlap\_under\_rotation(i, v, S) \leq self\_overlap\_under\_rotation(i, v, T).$$

3. Algorithm Theories

An *algorithm theory* represents the essential structure of a certain class of algorithms. Algorithm theory $\mathcal{A}$ extends problem theory $\mathcal{B}$ with any additional sorts, operators, and axioms needed to support the correct construction of an $\mathcal{A}$ algorithm for $\mathcal{B}$. The algorithm theories that we have studied can be arranged in a refinement hierarchy as in Figure 1. Below each algorithm theory in this hierarchy are listed various well-known classes of algorithms or computational paradigms that are based on it. More discussion of this hierarchy may be found in Section 7. Below we present a theory for the class of global search algorithms.

Global search generalizes the computational paradigms of binary search, backtracking, branch-and-bound, constraint satisfaction, heuristic search, and others. The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are (*i*) to create an initial space that contains all feasible solutions, (*ii*) to *extract* candidate solutions from a set, and (*iii*) to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts candidates, splits sets, and eliminates sets via filters until no sets remain to be split. The process is often described as a tree (or DAG) search in which a node represents a set of candidates and an arc represents the split relationship between set and subset. The filters serve to prune off branches of the tree that cannot lead to solutions.

The sets of candidate solutions are often infinite and even when finite they are rarely represented extensionally. Thus the intuitive notion of global search can be formalized as the extension of problem theory with an abstract data type of intensional representations called *descriptors*. In addition to the extraction and splitting operations mentioned above, the type also includes a *satisfaction* predicate that determines when a candidate solution is in the set denoted by a descriptor. For the sake of simplifying the presentation we will use the term *space* (or *subspace*) to denote both the descriptor and the set that it denotes. It should be clear from context which meaning is intended.

Formally, *gs–theory* $\mathcal{G}$ consists of the following structure:

Figure 1: Refinement Hierarchy of Algorithm Theories

Sorts $D, R, \hat{R}$

Operations

$I : D \rightarrow \mathit{boolean}$

$O : D \times R \rightarrow \mathit{boolean}$

$\hat{I} : D \times \hat{R} \rightarrow \mathit{boolean}$

$\hat{r_0} : D \rightarrow \hat{R}$

$\mathit{Satisfies} : R \times \hat{R} \rightarrow \mathit{boolean}$

$\mathit{Split} : D \times \hat{R} \times \hat{R} \rightarrow \mathit{boolean}$

$\mathit{Extract} : R \times \hat{R} \rightarrow \mathit{boolean}$

Axioms

GS0. $I(x) \implies \hat{I}(x, \hat{r_0}(x))$

GS1. $I(x) \wedge \hat{I}(x, \hat{r}) \wedge \mathit{Split}(x, \hat{r}, \hat{s}) \implies \hat{I}(x, \hat{s})$

GS2. $I(x) \wedge O(x, z) \implies \mathit{Satisfies}(z, \hat{r_0}(x))$

GS3. $I(x) \wedge \hat{I}(x, \hat{r}) \implies (\mathit{Satisfies}(z, \hat{r}) = \exists(\hat{s})\ (\ \mathit{Split}^*(x, \hat{r}, \hat{s}) \wedge \mathit{Extract}(z, \hat{s})))$

where $\hat{R}$ is the type of space descriptors, $\hat{I}$ defines legal space descriptors, $\hat{r}$ and $\hat{s}$ vary over descriptors, $\hat{r_0}(x)$ is the descriptor of the initial set of candidate solutions, $\mathit{Satisfies}(z, \hat{r})$ means that z is in the set denoted by descriptor $\hat{r}$ or that z satisfies the constraints that $\hat{r}$ represents, $\mathit{Split}(x, \hat{r}, \hat{s})$ means that $\hat{s}$ is a subspace of $\hat{r}$ with respect to input x, and $\mathit{Extract}(z, \hat{r})$ means that z is directly extractable from $\hat{r}$. Axiom GS0 asserts that the initial descriptor $\hat{r_0}(x)$ is a legal descriptor. Axiom GS1 asserts that legal descriptors split into legal descriptors. Axiom GS2 gives the denotation of the initial descriptor — all feasible solutions are contained in the initial space. Axiom GS3 gives the denotation of an arbitrary descriptor $\hat{r}$ — an output object z is in the set denoted by $\hat{r}$ if and only if z can be extracted after finitely many applications of *Split* to $\hat{r}$ where

$$\mathit{Split}^*(x, \hat{r}, \hat{s}) = \exists(k : Nat)\ (\ \mathit{Split}^k(x, \hat{r}, \hat{s})\ )$$

and

$$\mathit{Split}^0(x, \hat{r}, \hat{t}) = \hat{r} = \hat{t}$$

and for all natural numbers k

$$\mathit{Split}^{k+1}(x, \hat{r}, \hat{t}) = \exists(\hat{s} : \hat{R})\ (\ \mathit{Split}(x, \hat{r}, \hat{s}) \wedge \mathit{Split}^k(x, \hat{s}, \hat{t})).$$

Note that all variables are assumed to be universally quantified unless explicitly specified otherwise.

Example: Enumerating Subsets

Consider the problem of enumerating subsets of a given finite set S. A space can be described by a pair $\langle U, V \rangle$ of disjoint sets that denotes the set of all subsets of $U \uplus V$ that extend U. The descriptor for the initial space is just $\langle \{\ \}, S \rangle$. Formally, the descriptor $\langle U, V \rangle$ denotes the set

$$\{T \mid U \subseteq T \ \wedge \ T \subseteq V \uplus U\}.$$

Splitting is accomplished by either adding or not adding an arbitrary element $a \in V$ to U. If V is empty then the subset U can be extracted as a solution. This global search theory for enumerating subsets $\mathcal{G}_{subsets}$ can be presented via a theory morphism from abstract gs-theory $\mathcal{G}$.

$$
\begin{array}{rcl}
D & \mapsto & set(\alpha) \\
R & \mapsto & set(\alpha) \\
I & \mapsto & \lambda S.\, true \\
O & \mapsto & \lambda S, T.\, T \subseteq S \\
\hat{R} & \mapsto & set(\alpha) \times set(\alpha) \\
\hat{I} & \mapsto & \lambda S, \langle U, V \rangle.\, U \uplus V \subseteq S \ \wedge\ U \cap V = \{\} \\
Satisfies & \mapsto & \lambda T, \langle U, V \rangle.\, U \subseteq T \ \ \wedge\ \ T \subseteq V \uplus U \\
\hat{r_0} & \mapsto & \lambda S.\, \langle emptyset, S \rangle \\
Split & \mapsto & \lambda S, \langle U, V \rangle, \langle U', V' \rangle.\, V \neq \{\,\} \ \wedge\ a = arb(V) \\
 & & \qquad \wedge\ (\langle U', V' \rangle = \langle U, V - a \rangle \ \vee\ \langle U', V' \rangle = \langle U + a, V - a \rangle) \\
Extract & \mapsto & \lambda T, \langle U, V \rangle.\, empty(V) \ \ \wedge\ \ T = U
\end{array}
$$

End of Example.

In addition to the above components of global search theory, there are various derived operations which may play a role in producing an efficient algorithm. Filters, described next, are crucial to the efficiency of a global search algorithm. Filters correspond to the notion of pruning branches in backtrack algorithms and to pruning via lower bounds and dominance relations in branch-and-bound. A *feasibility filter* $\psi : D \times \hat{R} \rightarrow Boolean$ is used to eliminate spaces from further processing. The *ideal feasibility filter* decides the question "Does there exist a feasible solution in space $\hat{r}$?", or, to be more precise,

$$\exists(z : R)(\ Satisfies(z, \hat{r}) \ \wedge\ O(x, z)\ ). \tag{1}$$

However, to use (1) directly as a filter would usually be too expensive, so instead we use various approximations to it. These approximations can be classified as either

1. *necessary feasibility filters* where (1) $\implies \psi(x, \hat{r})$;
2. *sufficient feasibility filters* where $\psi(x, \hat{r}) \implies$ (1); or
3. *heuristic feasibility filters* which bear other relationships to (1).

Necessary filters only eliminate spaces that do not contain solutions, so they are generally useful. Sufficient filters are mainly used when only one solution is desired. Heuristic filters offer no guarantees, but a fast heuristic approximation to (1) may have the best performance in practice.

4. Program Theories

A program theory represents an executable program and its properties such as invariants, termination, and correctness with respect to a problem theory. Formally, a *program theory* $\mathcal{P}$ is parameterized with an algorithm theory or, more generally, an extended problem theory. The sort and operator symbols of the theory parameter can be used in defining programs in $\mathcal{P}$. Parameter instantiation, which is expressed as a theory morphism from the parameter theory, results in the replacement of each sort and operator symbol in $\mathcal{P}$ by its image under the theory morphism. The program theory introduces operator symbols for various functions and defines them and their correctness conditions via axioms. The main function would be defined as follows in the case where all feasible solutions are desired.

$$\begin{array}{ll} \textbf{Operations} & F : D \rightarrow set(R) \\ & \ldots \\ \textbf{Axioms} & \forall (x : D)(\ I(x) \implies F(x) = \{z \mid O(x,z)\}\ ) \\ & \forall (x : D)(\ I(x) \implies F(x) = Body(x)\ ) \\ & \ldots \end{array}$$

where $Body$ is code that can be executed to compute F. In order to express $Body$ it is generally necessary to import a programming language and extend it with specification language features. In this paper we assume a straightforward mathematical language that uses set-theoretic data types and operators and serves both as specification and program language. Consistency of the program theory entails that the function computed by the code ($Body$) must return all feasible solutions. The axioms for other functions would be similar.

Program theories can be expressed in a somewhat more conventional format and called a *program specification*:

function $F\ (x : D) : set(R)$
  where $I(x)$
  returns $\{z \mid O(x,z)\}$
  $= Body(x)$

Depending on choices of control strategy and programming language, a range of abstract programs can be inferred in abstract global search theory [18]. We are interested in those program theories whose consistency can be established for all possible input theories; that is, those program theories whose consistency can be established solely on the basis of the parameter theory. One such theory is presented below. Given a global search theory, the following theorem shows how to infer a correct program for enumerating all feasible solutions. In this theorem the auxiliary function $F\_gs(x, \hat{r})$ computes the set of all feasible solutions z in space $\hat{r}$.

Theorem 4..1 *Let $\mathcal{G}$ be a global search theory. If Φ is a necessary feasibility filter then the following program specification is consistent*

function $F(x : D) : set(R)$
  where $I(x)$
  returns $\{z \mid O(x, z)\}$
  $= \{z \mid \Phi(x, \hat{r}_0(x)) \ \wedge \ z \in F\_gs(x, \hat{r}_0(x))\}$

function $F\_gs(x : D, \hat{r} : \hat{R}) : set(R)$
  where $I(x) \ \wedge \ \hat{I}(x, \hat{r}) \ \wedge \ \Phi(x, \hat{r})$
  returns $\{z \mid Satisfies(z, \hat{r}) \ \wedge \ O(x, z)\}$
  $= \{z \mid Extract(z, \hat{r}) \ \wedge \ O(x, z)\}$
  $\cup \, reduce(\cup, \{ F\_gs(x, \hat{s}) \mid Split(x, \hat{r}, \hat{s}) \ \wedge \ \Phi(x, \hat{s})\}).$

The proof may be found in [18]. In words, the abstract global search program works as follows. On input x the program F calls F_gs with the initial space $\hat{r}_0(x)$ if the filter holds (otherwise there are no feasible solutions and the set-former evaluates to the emptyset). The program $F\_gs(x, \hat{r})$ unions together two sets; (1) all solutions that can be directly extracted from the space $\hat{r}$, and (2) the union of all solutions found recursively in spaces $\hat{s}$ that are obtained by splitting $\hat{r}$ and that survive the filter. Note that Φ becomes an input invariant in F_gs.

5. Design Tactics

Theorem 4..1 and its analogues reduce the problem of constructing a program to the problem of constructing an algorithm theory for a given problem F. The task of constructing an $\mathcal{A}$-algorithm theory for F is described by the following commutative diagram (a pushout in the category of theories)

$$\begin{array}{ccc} \mathcal{B} & \stackrel{m}{\rightarrow} & \mathcal{B}_F \\ \downarrow e & & \downarrow e' \\ \mathcal{A} & \stackrel{m'}{\rightarrow} & \mathcal{A}_F \end{array}$$

where e and e' are inclusions (theory extensions) and m and m' are theory morphisms. That is, the construction of an $\mathcal{A}$-theory for F can be viewed both as an extension of $\mathcal{B}_F$ and as a theory morphism $\mathcal{A} \ \rightarrow \ \mathcal{A}_F$.

For each of several algorithm theories that we have explored (see Figure 1), we have developed specialized design tactics. An $\mathcal{A}$-design tactic constructs an $\mathcal{A}$-algorithm theory for a given problem theory. Our tactic for designing global search algorithms relies on a deductive inference system and a library of standard gs-theories for common domains. The steps of the tactic are (1) to select and specialize a standard gs-theory, (2) to infer various filters, (3) to infer a concrete program, and (4) to perform program optimizations and refinements.

We describe first how to specialize a gs-theory to a given problem theory. Let $\mathcal{G}_G$ be a gs-theory whose components are denoted $D_G, R_G, O_G,$ $Satisfies_G$, etc., and let $\mathcal{B}_F$ be a given problem theory with components D_F, R_F, I_F, O_F. The problem theory $\mathcal{B}_G$ *generalizes* $\mathcal{B}_F$ if for every input x to F there is an input y to G such that the set of feasible solutions $G(y)$ is a superset of $F(x)$; formally

$$\forall(x : D_F)\ \exists(y : D_G)\ \forall(z : R_F)(\ I(x)\ \Longrightarrow\ (R_F \subseteq R_G\ \wedge\ (O_F(x,z) \Rightarrow O_G(y,z)))\ ). \qquad (2)$$

Verifying (2) provides a substitution θ for the type parameters of the $\mathcal{B}_G$ (if any) and for input variables of $\mathcal{B}_G$ in terms of the input variables of F. The type and number of input variables can differ between $\mathcal{B}_G$ and $\mathcal{B}_F$, as in the example below. The gs-theory $\mathcal{G}_F$ is obtained by applying substitution θ across $\mathcal{G}_G$. To see that the axioms GS0 – GS3 hold for $\mathcal{G}_F$ note that we have replaced the input variables of $\mathcal{G}_G$ with terms which take on a subset of their previous values. Intuitively, the effect of verifying (2) is to reduce problem $\mathcal{B}_F$ to $\mathcal{B}_G$, so that a solution to $\mathcal{B}_G$ can be used to compose a solution to $\mathcal{B}_F$.

Example: Cyclic Difference Sets.

The gs-theory $\mathcal{G}_{subsets}$ generalizes the CDS specification. To see this, first instantiate (2)

$$\begin{array}{l}
\forall(\langle v,k,\ell\rangle : Nat \times Nat \times Nat) \\
\exists(S : set(Nat)) \\
\forall(Sub : set(Nat)) \\
\quad (\ 1 \le \ell \le k < v \\
\quad \wedge\ Sub \subseteq \{0..v-1\}\ \wedge\ size(sub) = k \\
\quad \wedge\ \forall(i)(\ i \in \{1..v-1\}\ \Longrightarrow\ self_overlap_under_rotation(i,v,sub) = \ell) \\
\quad \Longrightarrow\ (set(Nat) \subseteq set(\alpha)\ \wedge\ Sub \subseteq S)\ ).
\end{array}$$

The proof is trivial and yields the substitution

$$\theta = \{S \mapsto \{0..v-1\}, \alpha \mapsto Nat\}.$$

This substitution is a critical translation between the problem theory $\mathcal{B}_{subsets}$, which takes a single set-valued argument S, and $\mathcal{B}_{CDS}$, which takes three arguments v, k, ℓ. After applying these substitutions to $\mathcal{G}_{subsets}$, we obtain the following specialized gs-theory $\mathcal{G}_{CDS}$ for cyclic difference sets.

$$
\begin{array}{rcl}
D & \mapsto & Nat \times Nat \times Nat \\
I & \mapsto & \lambda\langle v,k,\ell\rangle.\ 1 \le \ell \le k < v \\
R & \mapsto & set(Nat) \\
O & \mapsto & \lambda\langle v,k,\ell\rangle, sub.\ Sub \subseteq \{0..v-1\} \\
 & & \quad \wedge\ size(Sub) = k \\
 & & \quad \wedge\ \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,sub) = \ell)\} \\
\hat{R} & \mapsto & set(\alpha) \times set(\alpha) \\
\hat{I} & \mapsto & \lambda\langle v,k,\ell\rangle, \langle U,V\rangle.\ U \uplus V \subseteq \{0..v-1\}\ \wedge\ U \cap V = \{\} \\
Satisfies & \mapsto & \lambda Sub, \langle U,V\rangle.\ U \subseteq Sub\ \ \wedge\ \ Sub \subseteq V \uplus U \\
\hat{r_0} & \mapsto & \lambda\langle v,k,\ell\rangle.\ \langle emptyset, \{0..v-1\}\rangle \\
Split & \mapsto & \lambda\langle v,k,\ell\rangle, \langle U,V\rangle, \langle U',V'\rangle.\ V \neq \{\ \}\ \wedge\ a = arb(V) \\
 & & \quad \wedge\ (\langle U',V'\rangle = \langle U, V-a\rangle\ \vee\ \langle U',V'\rangle = \langle U+a, V-a\rangle) \\
Extract & \mapsto & \lambda Sub, \langle U,V\rangle.\ empty(V)\ \ \wedge\ \ Sub = U
\end{array}
$$

The next step in constructing a global search theory is to derive filters. For this step we need an inference system capable of deriving necessary or sufficient conditions.

Example: Cyclic Difference Sets.
We can obtain a feasibility filter for *CDS* by deriving a necessary condition of the expression

$$
\begin{array}{l}
U \subseteq Sub \subseteq V \uplus U \\
\wedge\ Sub \subseteq \{0..v-1\}\ \wedge\ size(Sub) = k \\
\wedge\ \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i,v,sub) = \ell)
\end{array}
$$

which is obtained by instantiating (1). Furthermore we are only interested in necessary conditions expressed over the variables $\{v,k,\ell,U,V\}$. The inference process may exploit the assumptions $1 \le \ell \le k < v$ and $U \uplus V \subseteq \{0..v-1\}$ (the input conditions I and the data type invariants $\hat{I}$ of $\hat{R}$).

The derivation of the filters exploits two monotonicity laws. One,

$$S \subseteq T \implies size(S) \le size(T)$$

is from the domain of set theory, and the other

$$S \subseteq T \implies self\_overlap\_under\_rotation(i,v,S) \le self\_overlap\_under\_rotation(i,v,T)$$

is from the domain theory of the CDS problem. Inference proceeds as follows.

$$
\begin{array}{l}
U \subseteq Sub \subseteq V \uplus U \\
\qquad \implies \qquad \text{(by monotonicity of } size\text{)} \\
\qquad size(U) \le size(Sub) \le size(V \uplus U) \\
\qquad \iff \qquad \text{(using } k = size(Sub)\text{)} \\
\qquad size(U) \le k \le size(V) + size(U).
\end{array}
$$

Thus we obtain $size(U) \leq k \leq size(V) + size(U)$ as one necessary feasibility filter. In words, the partial set being incrementally constructed (U) must have at most k elements, but there must be at least k elements between U and the pool of remaining elements V. Another filter can be derived as follows.

$U \subseteq Sub \subseteq V \uplus U$

$\implies$ (by monotonicity of *self_overlap_under_rotation*)

$$\forall(i)(\ i \in \{1..v-1\} \implies \\ self\_overlap\_under\_rotation(i, v, U) \\ \leq self\_overlap\_under\_rotation(i, v, Sub) \\ \leq self\_overlap\_under\_rotation(i, v, V \uplus U))$$

$\iff$ (using $self\_overlap\_under\_rotation(i, v, Sub) = \ell$)

$$\forall(i)(\ i \in \{1..v-1\} \implies \\ self\_overlap\_under\_rotation(i, v, U) \\ \leq \ell \leq self\_overlap\_under\_rotation(i, v, V \uplus U))$$

Thus we obtain

$$\forall(i)(\ i \in \{1..v-1\} \implies \\ self\_overlap\_under\_rotation(i, v, U) \\ \leq \ell \leq self\_overlap\_under\_rotation(i, v, V \uplus U))$$

as another necessary feasibility filter. In words, the partial solution U must have a self-overlap of at most ℓ and the combined set $U \uplus V$ must have a self-overlap of at least ℓ.

The result of applying Theorem 4..1 is the consistent program specification in Figure 2. Note that this program specification includes not only the input and output conditions of the function, but also invariants that characterize the meaning of all data structures. These invariants are crucial to later optimizations.
End of example.

The filter Φ will often dramatically reduce the amount of work needed to enumerate the feasible space. One feature of necessary filters is that one, *true*, is immediately available; stronger filters are obtained with more investment of computational resource at design-time.

Following the production of a concrete global search program, there are typically many opportunities for program optimization. These tend to follow a stereotypic order: simplifying the body of programs with respect to their input assumptions/data structure invariants,

function $CDS(v, k, \ell)$
where $1 \leq \ell \leq k < v$
returns $\{sub \mid sub \subseteq \{0..v-1\} \ \wedge \ size(sub) = k$
$\wedge \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, sub) = \ell)\}$
$= \ \{sub \mid size(\{\ \}) \leq k \leq size(\{\ \}) + size(\{0..v-1\})$
$\wedge \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, \{\ \}) \leq \ell$
$\wedge \ \ell \leq self\_overlap\_under\_rotation(i, v, \{\ \} \uplus \{0..v-1\}))$
$\wedge \ sub \in CDS\_gs(v, k, \ell, \{\ \}, \{0..v-1\})\}$

function $CDS\_gs(v, k, \ell, U, V)$
where $1 \leq \ell \leq k < v \ \wedge \ V \uplus U \subseteq \{0..v-1\}$
$\wedge \ size(U) \leq k \leq size(U) + size(V)$
$\wedge \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, U) \leq \ell$
$\wedge \ \ell \leq self\_overlap\_under\_rotation(i, v, U \uplus V))$
returns $\{\ sub \mid U \subseteq sub \subseteq V \uplus U \ \wedge \ size(sub) = k$
$\wedge \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, sub) = \ell)\}$
$= \ \{sub \mid empty(V) \ \wedge \ sub = U$
$\wedge \ sub \subseteq \{0..v-1\}$
$\wedge \ size(sub) = k$
$\wedge \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, sub) = \ell)\}$
$\cup \ reduce(\cup, \ \{CDS\_gs(v, k, \ell, U', V') \mid$
$V \neq \{\ \} \ \wedge \ a = arb(V)$
$\wedge \ (\langle U', V' \rangle = \langle U, V - a \rangle \ \vee \ \langle U', V' \rangle = \langle U + a, V - a \rangle)$
$\wedge \ size(U') \leq k \leq size(U') + size(V')$
$\wedge \forall(i)(\ i \in \{1..v-1\} \implies self\_overlap\_under\_rotation(i, v, U') \leq \ell$
$\wedge \ \ell \leq self\_overlap\_under\_rotation(i, v, U' \uplus V'))\}).$

Figure 2: Cyclic Difference Set Algorithm

applying partial evaluation, finite differencing, data structure refinement, and compilation. For example, in *CDS* the expression

$$self\_overlap\_under\_rotation(i, v, \{\,\}) \leq \ell \leq self\_overlap\_under\_rotation(i, v, \{0..v-1\})$$

simplifies to *true* using the laws

$$self\_overlap\_under\_rotation(i, v, \{\,\}) = 0$$

and

$$self\_overlap\_under\_rotation(i, v, \{0..v-1\}) = v$$

and the input condition $1 \leq \ell \leq v$. For another example, in *CDS_gs* the expressions

$$self\_overlap\_under\_rotation(i, v, U')$$

and

$$self\_overlap\_under\_rotation(i, v, U' \uplus V')$$

can be maintained incrementally using the technique of finite differencing [13]. The effect is to introduce map data structures that represent lower and upper bounds on the value of *self_overlap_under_rotation* on the current partial solution.

6. KIDS

We have implemented the global search design tactic and various optimization techniques in the KIDS/CYPRESS system [18, 19]. Implementation serves to check on the effectiveness of our derivations and to uncover issues that might be overlooked when doing derivations on paper. We have been able to derive a wide variety of algorithms without having to rely on the programmer supplying invariants, generalizations, or other such "eureka" steps. All interaction with the system during design and optimization involves the use of a mouse to select program expressions or items from a machine-generated menu. We hope to demonstrate that interaction at this level can be natural enough to pose successful explanations of the derivation process and that the tactics can be made both efficient enough and comprehensive enough to be useful for routine programming.

The user goes through the following steps in using KIDS for algorithm design.

1. *Define terms* – The user builds up a domain theory by defining appropriate terms. In this paper we defined the term *self_overlap_under_rotation*.
2. *Provide laws* – The user currently must provide derived laws that allow reasoning about the derived terms at a high level. Our experience has been that distributive and monotonicity laws provide most of the laws that are needed to support design and optimization.
3. *Create a specification* – The user enters a specification in a general but stylized format that is easily converted to a problem theory.

4. *Apply a design tactic* – The user selects a design tactic from a menu and applies it to a specification S by pointing to S with a mouse. Subsequent steps describe the global search design tactic.

   (a) *Specialize a known gs-theory* – The system presents a menu of gs-theories that are currently in its library and the user selects one. It is automatically matched (by instantiating and verifying (2)) and specialized to the given problem.

   (b) *Derive filters* – The tactic then automatically derives filters by exhaustively searching to a fixed inference depth all necessary conditions of (1). The user is then presented with a menu of candidate filters and must select a subset (any subset will yield a correct algorithm). The user also has the option of having the search for necessary conditions continue to greater depths. Generally the stronger the filter the better, although one has to trade off filtering power with the cost of executing the filter. Currently this step takes the bulk of the design time – about 10 minutes for the CDS problem. Mechanisms for automating the selection of strongest filters are known and would make use of dependency-tracking in the inference system.

   (c) *Instantiate a program theory* – The user is presented with a menu of program theories (schemes) that embody different control strategies and possibly different languages. Having selected one, the resulting programs and their problem theories and invariants are displayed.

5. *Apply program optimizations and data type refinements* - The KIDS system allows the application of optimization techniques such as simplification, partial evaluation, finite differencing, loop fusion, and others. The user selects an optimization method from a menu and applies it by pointing at a program expression. Each of the optimization methods are fully automatic and, with the exception of simplification (which is arbitrarily hard), take only a few seconds. Analogously, the user can select different implementations for the abstract types in the program specification (e.g. sets can be implemented via linked lists, bit vectors, or others).

6. *Compile* – Finally, the resulting code is treated by a conventional compiler.

Examples of derivations that present the above steps in more detail may be found in [18, 21, 19].

We have run the partially optimized CDS program on a number of inputs, and were surprised to discover a cyclic difference set that was previously unknown (according to [1]): the $\langle 13, 4, 1\rangle$-CDS: {0,1,4,6}. Other problems treated by specializing the subset theory $\mathcal{G}_{subsets}$ include set covers, binary knapsack, k-clique, vertex covers, and k-subset problems. As of early 1989 over two dozen global search algorithms have been designed and optimized using the KIDS system.

7. Other Algorithm Theories

We have studied a number of algorithm theories and their design tactics, as shown in Figure 1. Given a problem theory, it is possible to create a generate-and-test algorithm which simply enumerates the output domain checking for feasible solutions. Because generate-and-test requires no additional structure than problem theory it can be viewed as a most general algorithm paradigm.

Local structure results from the imposition of a discrete neighborhood structure (graph) on the output domain. Local search algorithms start with a candidate solution and then iteratively traverse from candidate to neighboring candidate until a feasible (or optimal) solution is found. Examples of local search algorithms include steepest ascent algorithms, simulated annealing, closure algorithms, and many network flow algorithms. A theory of local search and a design tactic based on it are presented in [8, 9]. The implemented tactic has been used to derive a variant of the classic simplex algorithm for linear programming.

Problem reduction involves the reduction of a problem to a structure of subproblems. Solutions to the subproblems are composed to form a solution to the initial problem. A simple example is the reduction of a given problem to the problem solved by a library subroutine. For example, in Section 5 we reduced the problem of enumerating cyclic difference sets to the problem of enumerating subsets of a given set.

Complementation structure is useful when it is easier to enumerate infeasible solutions than feasible solutions. The initial problem is reduced to two subproblems: (1) enumerate a superset of feasible solutions and (2) enumerate infeasible solutions. The feasible solutions can then be found by set subtraction. Sieve algorithms are based on complementation structure. Typically the superset of feasible solutions is explicitly represented and set subtraction is interleaved with the enumeration of infeasible solutions.

And-reduction (divide-and-conquer) involves the reduction of a problem to a structure of subproblems all of whose solutions are required in order to compose a solution to the initial problem. The subproblems typically include an instance of the initial problem so that the reduction is recursive.

A divide-and-conquer algorithm can be treated as a homomorphism from a decomposition algebra on the input domain of a problem to a composition algebra on its output domain [15]. One tactic for designing a divide-and-conquer algorithm involves selecting a simple or standard decomposition algebra from a library and then using the homomorphism condition to derive a specification for the corresponding composition algebra on the output domain. Another useful tactic is to select a simple composition algebra and derive a specification for a decomposition algebra on the input domain. We have implemented these tactics and used them to derive dozens of algorithms [16], including recently one that was previously unknown and asymptotically faster than previously known algorithms [17].

Or-reduction (global search) involves the reduction of a problem to a structure of subproblems at least one of whose solutions are required in order to obtain a solution to the initial problem. Solutions to the initial problem are obtained by selecting solutions to subproblems [18].

And/or-reduction involves a combination of And- and Or-reductions resulting in alternative ways to decompose an initial problem [20]. This theory supports the design of dynamic programming, general branch-and-bound, game tree search, and greedy algorithms.

Other examples of theories that relate problem structure to algorithm structure can be found in the literature, although only recently has there been much interest in using these theories as a basis for the formal derivation of algorithms as opposed to analysis or verification. Bellman's principle of optimality, if suitably formalized, is a sufficient condition for solution by dynamic programming [7]. Various theories of branch-and-bound algorithms have been presented [6, 11, 12]. Matroids [4] provide sufficient structure for an optimization problem to be solved by a greedy algorithm. Bird [2] presents algorithm theories for special cases of greedy and dynamic programming algorithms and applies them to a coding problem.

8. Concluding Remarks

Algorithm design can be treated as the construction of an algorithm theory that extends a problem theory with the structure of a certain class of algorithms. We presented one special design tactic for constructing global search theories and used it to derive an algorithm for enumerating cyclic difference sets. Theorem 4..1 and its analogues mediate the transition from an algorithm theory to a concrete program by factoring in commitments to control strategy and target language.

Our specialized design tactics should be useful in many formal approaches to programming. For those algorithms that can be derived via tactics the resulting derivations are shorter, simpler, and more motivated than they would be if derived from first principles or in a more general-purpose calculus of programs. We cannot now claim that all algorithms are naturally derivable as instances of various well-known classes of algorithms. Therefore one might want to embed the tactics in a more general derivation methodology.

Acknowledgements

This paper benefited from helpful comments by Richard Bird on an earlier draft and from discussions with Joseph Goguen. This research was supported in part by the Office of Naval Research under Contract N00014-87-K-0550 and in part by the Air Force Office of Scientific Research under Contract F49620-88-C-0033.

References

[1] Baumert, L. D. *Cyclic Difference Sets*. Springer-Verlag, Berlin, 1971. Lecture Notes in Mathematics, Vol. 182.

[2] Bird, R. A calculus of functions for program derivation. Tech. Rep. PRG-64, Oxford University, Programming Research Group, December 1987.

[3] Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[4] Edmonds, J. Matroids and the greedy algorithm. *Mathematical Programming 1* (1971), 127–136.

[5] Goguen, J. A., and Winkler, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.

[6] Ibaraki, T. Branch-and-bound procedures and state space representation of combinatorial optimization problems. *Information and Control 36* (1978), 1–36.

[7] Karp, R., and Held, M. Finite state processes and dynamic programming. *SIAM Journal of Applied Mathematics 15*, 3 (May 1967), 693–718.

[8] Lowry, M. R. Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence* (Seattle, WA, July 13–17, 1987).

[9] Lowry, M. R. *Algorithm Synthesis Through Problem Reformulation.* PhD thesis, Computer Science Department, Stanford University, 1989.

[10] Manna, Z., and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems 2*, 1 (January 1980), 90–121.

[11] Mitten, L. G., and Warburton, A. R. Implicit enumeration procedures. Tech. Rep. Working Paper 251, University of British Columbia, 1973.

[12] Nau, D., Kumar, V., and Kanal, L. General branch and bound and its relation to A* and AO*. *Artificial Intelligence 23*, 1 (May 1984), 29–58.

[13] Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402–454.

[14] Skinner, G. K. X-ray imaging with coded masks. *Scientific American 259*, 2 (August 1988), 84–89.

[15] Smith, D. R. The structure of divide-and-conquer algorithms. Tech. Rep. NPS52-83-002, Naval Postgraduate School, Monterey, CA, March 1983.

[16] Smith, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43–96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

[17] Smith, D. R. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming 8*, 3 (June 1987), 213–229.

[18] Smith, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

[19] Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (1990), 1024–1043.

[20] Smith, D. R. Structure and design of problem reduction generators. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 91–124.

[21] Smith, D. R., and Pressburger, T. T. Knowledge-based software development tools. In *Software Engineering Environments*, P. Brereton, Ed. Ellis Horwood Ltd., Chichester, 1988, pp. 79–103. (also Technical Report KES.U.87.6, Kestrel Institute, May 1987).

[22] Veloso, P. A. Problem solving by interpretation of theories. In *Contemporary Mathematics*, vol. 69. American Mathematical Society, Providence, Rhode Island, 1988, pp. 241–250.