Automating the Design of Algorithms

Douglas R. Smith

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California 94304, USA

Abstract

This paper has two roughly independent parts. The first is devoted to the automation of program construction. The Kestrel Interactive Development System (KIDS) provides knowledge-based support for the derivation of correct and efficient programs from specifications. We trace the use of KIDS in deriving a scheduling algorithm. The derivation illustrates algorithm design, deductive inference, simplification, finite differencing, partial evaluation, data type refinement, and other techniques. All of the KIDS operations are automatic except the algorithm design tactics which require some interaction at present. Dozens of programs have been derived using the KIDS environment.

The second part discusses the theory of algorithm design used in KIDS. Concepts include problem theories, algorithm theories, program schemes as parameterized theories, design as interpretation between theories (theory morphisms), algorithm design tactics, and refinement hierarchies of algorithm theories and the incremental construction of algorithms.

1 Introduction

There are many researchers working towards the goal of an effective software engineering discipline. It is clear that sound mathematical foundations for such a discipline are required. Within the software engineering community there is far from universal concensus regarding what such foundations are and whether they are relevant. There is little agreement as to the direction of future progress. On the other hand, in the theoretical community, there seems to be a concensus that concepts from mathematical logic and algebra provide the necessary foundations and that programming should be treated as calculation within a suitable logic. A wide variety of logics and calculi are currently under investigation.

Several well-known program derivation methodologies, such as the deductive synthesis approach of Manna and Waldinger [8] or the calculus of Dijkstra [4], are based on inference rules for various programming language constructs - rules for inferring statement sequences, conditionals, loops, and so on. Our complementary approach can be viewed as providing inference rules for various problem-solving methods or algorithmic paradigms. Our view is that when we are solving a problem we don't think in terms of loops or case statements, but more typically we may think of a certain kind of algorithm that may be effective, such as divide-and-conquer or a greedy method.

This rest of this paper has two parts, which can be read more-or-less independently. In Section 2, we describe the KIDS (Kestrel Interactive Development System) approach to automating the construction of programs. KIDS supports the interactive development of correct, efficient programs from formal specifications [13]. Users can apply automated tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation, data type refinement, and other program transformations. After briefly discussing the environment underlying KIDS, we step through the derivation of a program for enumerating all solutions to a scheduling problem. The steps are as follows. First we build up a domain theory in order to state and reason about the problem. Then, a well-structured but inefficient backtrack algorithm is created that works by extending partial schedules. To improve efficiency we apply simplification and partial evaluation operations. We also perform finite differencing which results in the introduction of data structures. Next, high-level datatypes such as sets and sequences are refined into more machine-oriented types such as bit-vectors and linked lists. Finally, the resulting code is compiled.

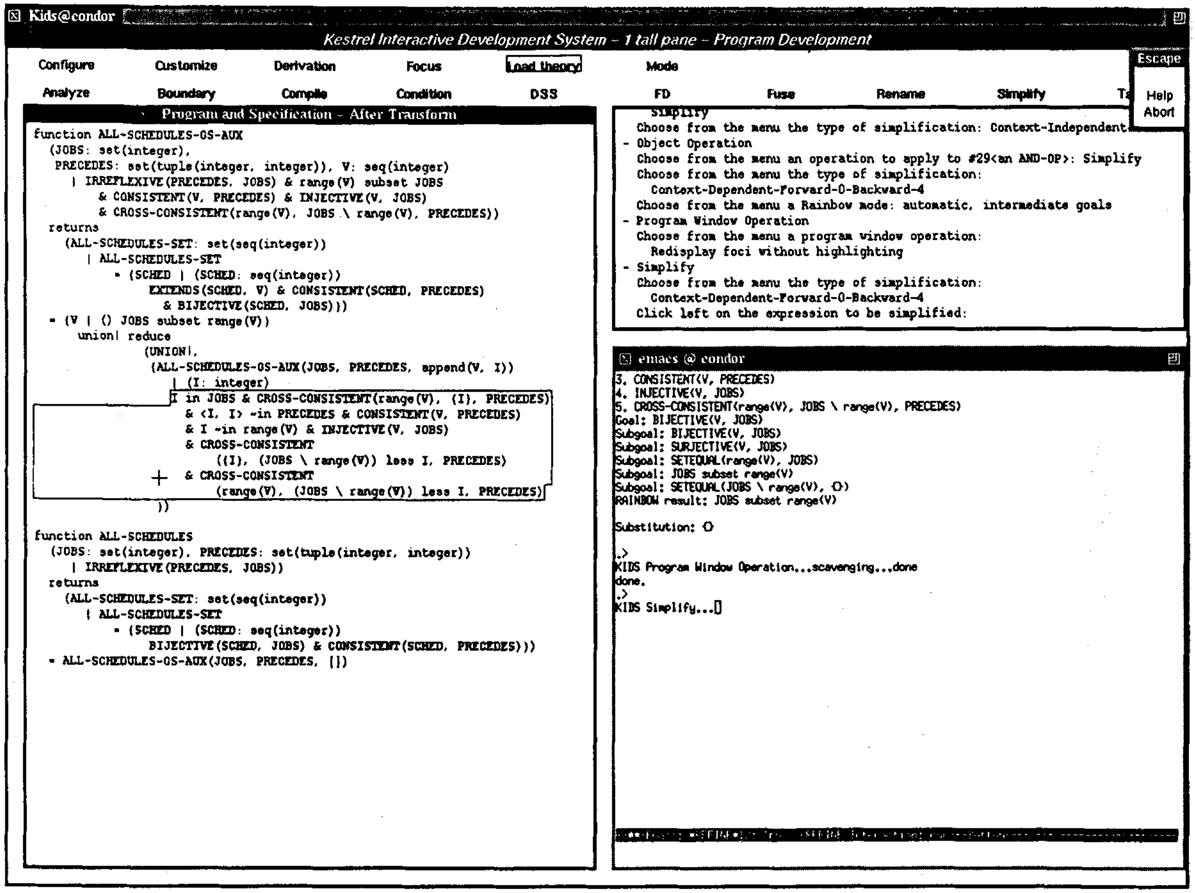
In Section 3, we describe a theory of algorithm design that is based on formalizing knowledge about various classes of algorithms. The essential structure of a class of algorithms is captured via a first-order theory presentation and the process of designing an instance of the class is construction of a theory morphism (interpretation between theories) [14]. For each class we have specialized design tactics for constructing instances of the class. The algorithm theories can be arranged in a refinement hierarchy and this hierarchy can be used to make the algorithm design incremental.

2 Automated Support for Program Construction

A computer program can be viewed as a composition of several kinds of knowledge: knowledge about the particular problem being solved, general knowledge about the application domain, and programming knowledge about architectures, algorithms, data structures, optimization techniques, performance analysis, etc. KIDS serves as a testbed for exploring automated support for this compositional approach to program construction.

A user of KIDS develops a formal specification into a program by interactively applying a sequence of high-level transformations. During development, the user views a partially implemented specification annotated with input assumptions, invariants, and output conditions (a snapshot of a typical screen appears in Figure 1). A mouse is used to select a transformation from a command menu and to apply it to a subexpression of the specification. In effect, the user makes high-level design decisions and the system carries them out.

Perhaps the most unique features of KIDS are its algorithm design tactics and its deductive inference component. Its other operations, such as simplification and finite differencing, are well-known, but have not been integrated before in one system. All of the KIDS transformations are correctness-preserving, automatic (except the algorithm design tactics which require some interaction at present) and perform significant, meaningful steps from the user's point of view.



```
function ALL-SCHEDULES-OS-AUX
  (JOBS: set(integer),
   PRECEDES: set(tuple(integer, integer)), V: seq(integer)
    | IRREFLEXIVE(PRECEDES, JOBS) & range(V) subset JOBS
        & CONSISTENT(V, PRECEDES) & INJECTIVE(V, JOBS)
        & CROSS-CONSISTENT(range(V), JOBS \ range(V), PRECEDES))
  returns
    (ALL-SCHEDULES-SET: set(seq(integer))
      | ALL-SCHEDULES-SET
          = {SCHED | (SCHED: seq(integer))
               EXTENDS(SCHED, V) & CONSISTENT(SCHED, PRECEDES)
                 & BIJECTIVE(SCHED, JOBS)})
  = {V | () JOBS subset range(V)}
      union| reduce
             (UNION|,
              {ALL-SCHEDULES-OS-AUX(JOBS, PRECEDES, append(V, I))
                | (I: integer)
                  I in JOBS & CROSS-CONSISTENT(range(V), {I}, PRECEDES)
                    & <I, I> ~in PRECEDES & CONSISTENT(V, PRECEDES)
                    & I ~in range(V) & INJECTIVE(V, JOBS)
                    & CROSS-CONSISTENT
                        ({I}, (JOBS \ range(V)) less I, PRECEDES)
                    & CROSS-CONSISTENT
                        (range(V), (JOBS \ range(V)) less I, PRECEDES)
              })

function ALL-SCHEDULES
  (JOBS: set(integer), PRECEDES: set(tuple(integer, integer))
    | IRREFLEXIVE(PRECEDES, JOBS))
  returns
    (ALL-SCHEDULES-SET: set(seq(integer))
      | ALL-SCHEDULES-SET
          = {SCHED | (SCHED: seq(integer))
               BIJECTIVE(SCHED, JOBS) & CONSISTENT(SCHED, PRECEDES)})
  = ALL-SCHEDULES-OS-AUX(JOBS, PRECEDES, [])
```

```
  Simplify
  Choose from the menu the type of simplification: Context-Independent
- Object Operation
  Choose from the menu an operation to apply to #29<an AND-OP>: Simplify
  Choose from the menu the type of simplification:
    Context-Dependent-Forward-0-Backward-4
  Choose from the menu a Rainbow mode: automatic, intermediate goals
- Program Window Operation
  Choose from the menu a program window operation:
    Redisplay foci without highlighting
- Simplify
  Choose from the menu the type of simplification:
    Context-Dependent-Forward-0-Backward-4
  Click left on the expression to be simplified:
```

```
3. CONSISTENT(V, PRECEDES)
4. INJECTIVE(V, JOBS)
5. CROSS-CONSISTENT(range(V), JOBS \ range(V), PRECEDES)
Goal: BIJECTIVE(V, JOBS)
Subgoal: BIJECTIVE(V, JOBS)
Subgoal: SURJECTIVE(V, JOBS)
Subgoal: SETEQUAL(range(V), JOBS)
Subgoal: JOBS subset range(V)
Subgoal: SETEQUAL(JOBS \ range(V), {})
RAINBOW result: JOBS subset range(V)

Substitution: {}

.>
KIDS Program Window Operation...scavenging...done
done.
.>
KIDS Simplify...
```

Fig. 1. KIDS Interface

2.1 General Characteristics of KIDS

KIDS is built on top of REFINE[1], a commercial knowledge-based programming environment which provides

- an object-attribute-style database that is used to represent software-related objects via annotated abstract syntax trees;
- grammar-based parser/unparsers that translate between text and abstract syntax;
- a very-high-level language (also called REFINE) and compiler. The language supports first-order logic, set-theoretic data types and operations, transformation and pattern constructs that support the creation of rules. The compiler generates CommonLisp code.

KIDS is almost entirely written in REFINE and all of its operations work on the annotated abstract syntax tree representation of specifications in the REFINE database. A key feature of the unparsers/pretty-printers is the option for mouse-sensitive syntax — the user can refer to an expression on the screen by pointing to it.

KIDS is a program transformation system – one applies a sequence of consistency-preserving transformations to an initial specification and achieves a correct and hopefully efficient program. The system emphasizes the application of complex high-level transformations that perform significant and meaningful actions. From the user's point of view the system allows the user to make high-level design decisions like, "design a divide-and-conquer algorithm for that specification" or "simplify that expression in context". We hope that decisions at this level will be both intuitive to the user and be high-level enough that useful programs can be derived within a reasonable number of steps.

The user typically goes through the following steps in using KIDS for program development.

1. *Develop a domain theory* – The user builds up a domain theory that defines the basic concepts and operations of the domain and the laws for reasoning about them. Currently, KIDS has over 100 theories in its library, ranging from basic theories (e.g. for booleans, natural numbers, linear orders, and finite sequences) to problem-specific theories. Support for theory development in KIDS includes mechanisms to import theories from the library and some automated support for consistently extending theories by deriving laws from definitions. However, users typically must provide most of the problem-specific information in a domain theory.
2. *Create a specification* – The user enters a specification stated in terms of the underlying domain theory.
3. *Apply a design tactic* – The user selects an algorithm design tactic from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction (reducing a specification to a library routine) [10], divide-and-conquer [10], global search (binary search, backtrack, branch-and-bound) [12], problem reduction generators (dynamic programming, generalized branch-and-bound, game-tree search) [15], local search (hillclimbing) [7], and others.

[1] REFINE is a trademark of Reasoning Systems, Inc., Palo Alto, California.

4. *Apply optimizations* – The KIDS system allows the application of optimization techniques such as simplification, partial evaluation, finite differencing, case analysis, abstraction, unfold, and other transformations. The user selects an optimization method from a menu and applies it by pointing to a program expression. Each of the optimization methods are fully automatic and take only a few seconds, with the exception of simplification (which is arbitrarily hard).
5. *Apply data type refinements* – The user can select implementations for the high-level data types in the program. Data type refinement rules automatically carry out the details of constructing the implementation.
6. *Compile* – The resulting code is compiled to executable form. In a sense, KIDS can be regarded as a front-end to a conventional compiler.

Actually, the user is free to apply any subset of the KIDS operations in any order – the above sequence is typical of our experiments in algorithm design and is followed in this paper. The screen dump in Figure 1 shows the interface at the point after algorithm design when the user has just selected the Simplify operation on the command menu at the top and is pointing to an expression as the argument to simplify.

KIDS is supported by a deductive inference system called RAINBOW II. All of the laws used by RAINBOW II during program development are supplied via the problem domain theory. That is, the current version of KIDS has no built-in knowledge — the first step in performing a derivation is building and loading its domain theory. Loading a theory has the effect of installing definitions, laws, and inference rules in the Refine object base in way that is accessible to RAINBOW II.

2.2 Domain Theory and Specification for Scheduling

Suppose that we wish to schedule a set of jobs on a processor subject to a precedence relation that constrains the order in which jobs can run. Further suppose that each job completes in unit time, that each job has a deadline, and that we wish to minimize the number of jobs that fail to complete before their deadlines. If we define a *schedule* to be an ordering of a given set of jobs that is consistent with a given precedence relation, then this is an optimization problem where the feasible space is the set of schedules, and the cost function is the number of jobs in a schedule that fail to complete before their deadline.

Before a specification can be written, the relevant concepts, operations, relationships, and properties of the problem must be defined. Thus the first, and often the hardest, step in deriving an algorithm for solving a problem is the formalization of its domain theory.

KIDS provides rudimentary support for the development of domain theories. A theory presentation is comprised of sets of imported theories, type definitions, function specifications with optional operational definitions, laws (axioms and theorems), and rules of inference. A hierarchic library of theories is maintained with importation as the principal link. Users can enter definitions of new functions or create new definitions by abstraction on existing expressions. The inference system can be used to verify common properties such as associativity, commutativity, or idempotence. More interestingly, we have used RAINBOW II to automatically derive distributive, monotonicity, and other kinds of laws. For some problems, we have derived almost all of the laws needed to support design and optimization (see for example [17]). The scheduling problem reported here was performed before these theory development tools were available and the domain theory was entered entirely by hand.

The scheduling domain theory is summarized below. The concept that a schedule is a linear arrangement of a set of jobs can be expressed in terms of a bijection.

$$\begin{aligned}&Injective(M : seq(integer), S : set(integer)) : boolean\\ &\quad = range(M) \subseteq S\\ &\qquad \wedge\ \forall(i,j)(i \in domain(M)\ \wedge\ j \in domain(M)\ \wedge\ i \neq j \implies M(i) \neq M(j))\end{aligned}$$

$$\begin{aligned}&Bijective(M : seq(integer), S : set(integer)) : boolean\\ &\quad = Injective(M,S)\ \wedge\ range(M) = S\end{aligned}$$

That is, a sequence M is injective into a set S if all elements of M are in S and no element of M occurs twice. A sequence M is bijective into a set S if it is injective and each element of S occurs in M.

Distributive laws for the *Injective* predicate are as follows.

$$\forall(S)(Injective([\,], S)\ =\ true)$$

$$\forall(W,a,S)\,(Injective(append(W,a),S)\ =\ (Injective(W,S)\ \wedge\ a \in S\ \wedge\ a \notin range(W)))$$

$$\begin{aligned}&\forall(W1,W2,S)\,(Injective(concat(W1,W2),S)\\ &\qquad = (Injective(W1,S)\ \wedge\ Injective(W2,S)\ \wedge\ range(W1) \bigcap range(W2) = \{\}))\end{aligned}$$

The concept that a schedule must be consistent with the given precedence relation is captured in the following definition and associated laws:

$$\begin{aligned}&Consistent(S : seq(JOB), P : binrel(JOB,JOB)) : boolean\\ &\quad = \forall(i,j)(i \in range(S)\ \wedge\ j \in range(S)\ \wedge\ \langle i,j\rangle \in P\\ &\qquad\quad \implies Index(i,S) < Index(j,S))\end{aligned}$$

where $Index(i,S)$ returns the index of element i in sequence S.

$$\forall(P)(Consistent([], P)\ =\ true)$$

$$\forall(a,P)\,(Consistent([a], P)\ =\ true)$$

$$\begin{aligned}&\forall(S1,S2,P)\,(Consistent(concat(S1,S2),P)\\ &\qquad = (Consistent(S1,P)\ \wedge\ Consistent(S2,P)\\ &\qquad\quad \wedge\ Cross\text{–}Consistent(range(S1), range(S2), P)))\end{aligned}$$

where

$$\begin{aligned}&Cross\text{–}Consistent(R1 : set(JOB), R2 : set(JOB), P : binrel(JOB,JOB)) : boolean\\ &\quad = \forall(I,J)(I \in R1\ \wedge\ J \in R2 \implies \langle J,I\rangle \notin P)\end{aligned}$$

Formally, the problem of enumerating schedules can be specified as follows.

$Schedules(Jobs : set(JOB), Precedes : binrel(JOB, JOB))$
where $Irreflexive(Precedes, Jobs)$
returns $\{S : seq(JOB) \mid Bijective(S, Jobs) \ \land \ Consistent(S, Precedes)\}$

Here *Jobs* is the set of jobs that we wish to schedule. The parameter *Precedes* is a binary relation over *Jobs* and satisfies the *input condition* that it is irreflexive (with respect to the set *Jobs*). The output is specified to be the set of all sequences S of *Jobs* that are bijective with respect to *Jobs* and are consistent with the *Precedes* relation. The constraint $Bijective(S, Jobs) \ \land \ Consistent(S, Precedes)$ is called the *output condition*. The following is a specification of the schedule optimization problem, called *SWD* (Scheduling With Deadlines).

$SWD(Jobs : set(JOB), Precedes : binrel(JOB, JOB), Deadline : map(JOB, Nat))$
where $Irreflexive(Precedes, Jobs) \ \land \ domain(Deadline) = Jobs$
returns $extremum(\lambda(S_1, S_2)\ SWD\text{–}Cost(S_1, Deadline) \leq SWD\text{–}Cost(S_2, Deadline),$
$\{S \mid Bijective(S, Jobs) \ \land \ Consistent(S, Precedes)\})$

$SWD\text{–}Cost(S : seq(JOB), Deadline : map(JOB, Nat)) : Nat$
$= size(\{j \mid j \in range(S) \ \land \ Deadline(j) < index(S, j)\})$

The input *Deadline* is a mapping from jobs to deadline times (represented as natural numbers). The (nondeterministic) function *extremum* returns an extremal element:

$$extremum(f, C) = some(x)(x \in C \ \land \ \forall(y)(y \in C \implies f(x) \leq f(y))).$$

The function *SWD-Cost* computes the number of jobs that miss their deadline. Thus this specification seeks a schedule that minimizes the number of missed deadlines.

2.3 Algorithm Design

The next step is to develop a correct, high-level algorithm for enumerating schedules. KIDS has specialized tactics for creating algorithms of various kinds such as divide-and-conquer [10], local search [7], and global search [12]. The latter class (which will be applied here) generalizes binary search, backtracking, branch-and-bound, constraint satisfaction, and other algorithmic paradigms. The algorithm design tactics are discussed further in Section 3.

The basic idea of global search is to represent and manipulate sets of candidate solutions. The principal operations are to *extract* candidate solutions from a set and 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 solutions, 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 global search algorithms are based on an abstract data type of intensional representations called *space descriptors* (denoted by hatted symbols). In addition to the extraction and splitting operations mentioned above, the type also includes a predicate *satisfies* that determines when a candidate solution is in the set denoted by a descriptor.

The various operations in the abstract data type of space descriptors together with problem specification can be packaged together as a theory. Formally, abstract *global search theory* (or simply *gs–theory*) $\mathcal{G}$ is presented as follows:

Sorts

D	*input domain*
R	*output domain*
$\hat{R}$	*subspace descriptors*

Operations

$I : D \rightarrow boolean$	*input condition*
$O : D \times R \rightarrow boolean$	*input/output condition*
$\hat{I} : D \times \hat{R} \rightarrow boolean$	*subspace descriptors condition*
$\hat{r}_0 : D \rightarrow \hat{R}$	*initial space*
$Satisfies : R \times \hat{R} \rightarrow boolean$	*denotation of descriptors*
$Splits\text{–}into : D \times \hat{R} \times \hat{R} \rightarrow boolean$	*split relation*
$Extractable : R \times \hat{R} \rightarrow boolean$	*extractor of solutions from spaces*

Axioms

GS0. $I(x) \implies \hat{I}(x, \hat{r}_0(x))$

GS1. $I(x) \wedge \hat{I}(x,\hat{r}) \wedge Splits\text{–}into(x,\hat{r},\hat{s}) \implies \hat{I}(x,\hat{s})$

GS2. $I(x) \wedge O(x,z) \implies Satisfies(z, \hat{r}_0(x))$

GS3. $I(x) \wedge \hat{I}(x,\hat{r})$
$\implies (Satisfies(z,\hat{r}) = \exists(\hat{s})\,(\, Splits\text{–}into^*(x,\hat{r},\hat{s}) \wedge Extractable(z,\hat{s})))$

where D is the input domain, R is the output domain, I is the input condition, O is the output condition, $\hat{R}$ 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, $Satisfies(z,\hat{r})$ means that z is in the set denoted by descriptor $\hat{r}$ or that z satisfies the constraints that $\hat{r}$ represents, $Splits\text{–}into(x,\hat{r},\hat{s})$ means that $\hat{s}$ is a subspace of $\hat{r}$ with respect to input x, and $Extractable(z,\hat{r})$ 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 and that *Splits–into* induces a well-founded ordering on spaces. Axiom GS2 constrains 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 *Splits–into* to $\hat{r}$ where

$$Splits\text{–}into^*(x,\hat{r},\hat{s}) \iff \exists(k : Nat)\,(\, Splits\text{–}into^k(x,\hat{r},\hat{s})\,)$$

and

$$Splits\text{–}into^0(x,\hat{r},\hat{t}) \iff \hat{r} = \hat{t}$$

and for all natural numbers k

$$Splits\text{-}into^{k+1}(x, \hat{r}, \hat{t}) \iff \exists(\hat{s} : \hat{R}) \, (\, Splits\text{-}into(x, \hat{r}, \hat{s}) \wedge Splits\text{-}into^{k}(x, \hat{s}, \hat{t})).$$

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

Example: Enumerating sequences

Consider the problem of enumerating sequences over a given finite set S. A space is a set of sequences with common prefix ps. The descriptor for the initial space is just $[]$. Splitting is performed by appending an element from S onto the end of the common prefix ps. The sequence ps itself is directly extractable from the space. This global search theory for enumerating sequences can be presented via a correspondence between the components of abstract gs-theory and a concrete gs-theory (technically this correspondence is known as a theory morphism or interpretation between theories).

$$\begin{array}{rcl} D & \mapsto & set(\alpha) \times integer \\ I & \mapsto & \lambda(S) \, true \\ R & \mapsto & seq(\alpha) \\ O & \mapsto & \lambda(S, q) \, range(q) \subseteq S \\ \hat{R} & \mapsto & seq(\alpha) \\ \hat{I} & \mapsto & \lambda(S, ps) \, range(ps) \subseteq S \\ Satisfies & \mapsto & \lambda(q, ps) \, \exists(r) \, (q = concat(ps, r)) \\ \hat{r}_0 & \mapsto & \lambda(S) \, [\,] \\ Splits\text{-}into & \mapsto & \lambda(S, ps, ps') \, \exists(i) \, (i \in S \wedge ps' = append(ps, i)) \\ Extractable & \mapsto & \lambda(q, ps) \, q = ps \end{array}$$

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 *filter* $\psi : D \times \hat{R} \to boolean$ is used to eliminate spaces from further processing. The *ideal filter* decides the question "Does there exist a feasible solution in space $\hat{r}$?", or, formally,

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

However, to use (1) directly as a filter would usually be too expensive, so instead we use an approximation to it. A *necessary filter* Φ satisfies

$$\exists(z : R) \, (\, Satisfies(z, \hat{r}) \wedge O(x, z) \,) \implies \Phi(x, \hat{r}). \tag{2}$$

By the contrapositive of this definition, if $\Phi(x, \hat{r})$ is false for some space $\hat{r}$ then there does not exist a solution in $\hat{r}$. Thus necessary filters can be used to eliminate spaces that do not contain solutions.

The design tactic for global search in KIDS is based on the following theorems. The proofs may be found in [12]. The first shows how to produce a correct program from a given global search theory. Consequently, construction of a correct global search program reduces to the problem of constructing a global search theory. The second theorem tells us how to obtain a global search theory for a given problem by specializing an existing global search theory. This theorem suggests that we set up a library of global search theories for the various data types of our language and simply select and specialize these library theories.

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

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

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 $\{z \mid \text{Satisfies}(z, \hat{r}) \wedge O(x, z)\}$
    $= \{z \mid Extractable(z, \hat{r}) \wedge O(x, z)\}$
        $\cup\ reduce(\cup, \{ F\_gs(x, \hat{s}) \mid Splits\text{--}into(x, \hat{r}, \hat{s}) \wedge \Phi(x, \hat{s})\})$.
```

In this abstract program and later programs we mainly use conventional notations from first-order logic and set theory. Our notation for reduction is $reduce(\cup, SS)$ which could be written $\bigcup_{S \in SS} S$ or $\cup/SS$.

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. The program F_gs 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. In terms of the search tree model, F_gs unions together the solutions found at the current node with the solutions found at descendants. Note that Φ is an input invariant in F_gs.

If we were to apply Theorem 1 to *gs_sequences_over_finite_set* then we would get a generator of all sequences over the input set S.

The following definition gives conditions under which an algorithm for solving problem B can be used to enumerate all solutions to A. Specification $\mathcal{B}_A = \langle D_A, R_A, I_A, O_A\rangle$ *completely reduces* to specification $\mathcal{B}_B = \langle D_B, R_B, I_B, O_B\rangle$ if

$$(R_A = R_B) \wedge \forall(x : D_A)\, \exists(y : D_B)\, \forall(z : R_A)\, (I_A(x) \wedge O_A(x, z) \Rightarrow O_B(y, z)). \qquad (3)$$

$\mathcal{B}_A$ *completely reduces* to $\mathcal{B}_B$ *with substitution* θ if $\theta(y) = t(x)$ and $R_A = R_B\theta$

$$\forall(x : D_A)\, \forall(z : R_A)\, (I_A(x) \wedge O_A(x, z) \Rightarrow O_B(t(x), z)). \qquad (4)$$

Theorem 2. *Let $\mathcal{G}_B = \langle \mathcal{B}_B, \hat{R}, \hat{I}, \hat{r}_0, \text{Satisfies}, Splits\text{--}into, Extractable\rangle$ be a global search theory, and let $\mathcal{B}_A$ be a specification that completely reduces to $\mathcal{B}_B$ with substitution θ, then the structure $\mathcal{G}_A = \langle \mathcal{B}_A, \hat{R}\theta, \hat{I}\theta, \text{Satisfies}\theta, \hat{r}_0\theta, Splits\text{--}into\theta, Extractable\theta\rangle$ is a global search theory.*

In this theorem $P\theta$ denotes the application of substitution θ to expression P.
A simplified tactic for designing global search algorithms has four steps.

1. Select a global search theory $\mathcal{G}_B$ from a library which solves the problem of enumerating the output type for the given problem A.
2. Find a substitution θ whereby $\mathcal{B}_A$ completely reduces to $\mathcal{B}_B$ by verifying Formula (3). Apply Theorem 2 to create a specialized global search theory $\mathcal{G}_A$.
3. Derive a necessary filter Φ via Formula (2). That is, use directed inference to derive a necessary condition of Formula (1) expressed over the variables $\{x, \hat{r}\}$.
4. Apply Theorem 1 to create a global search program.

The tactic is sound and thus only generates correct programs. The interested reader should consult [12] for the full generality of the global search model and design tactic.

The KIDS library currently contains global search theories for a number of problem domains, such as enumerating sets, sequences, maps, and integers. For the Scheduling problem we select from a library a standard global search theory for enumerating sequences over a finite domain – *gs_sequences_over_finite_set.* In accord with step 2, the following inference specification is created.

$$\begin{array}{l}
set(JOB) = set(\alpha) \land \\
\forall(Jobs : set(JOB), Precedes : binrel(JOB, JOB)) \\
\exists(S : set(integer)) \\
\forall(assign : seq(integer)) \\
\quad (Bijective(assign, Jobs) \;\land\; Consistent(assign, Precedes) \\
\quad \Longrightarrow\; range(assign) \subseteq S).
\end{array}$$

The proof process is simple and proceeds as follows: The types are unified yielding substitution $\{\alpha \mapsto JOB\}$. By forward inference from $Bijective(assign, Jobs)$ the inference system derives $Injective(assign, Jobs)$ and $range(assign) = Jobs$, then

$$\begin{array}{lll}
range(assign) \subseteq S & & \\
\quad \Longleftrightarrow & & \text{applying } range(assign) = Jobs \\
 & Jobs \subseteq S & \\
\quad \Longleftrightarrow & & \text{unifying with the reflexivity law } \forall(R)(R \subseteq R) \\
 & true & \text{with substitution } \{S \mapsto Jobs\}.
\end{array}$$

Thus, altogether the Scheduling problem completely reduces to *gs_sequences_over_finite_set* with substitution $\{\alpha \mapsto integer, S \mapsto Jobs\}$. The construction in Theorem 2 yields the following global search theory for scheduling.

$$\begin{array}{rcl}
D & \mapsto & set(JOB) \times binrel(JOB, JOB) \\
I & \mapsto & \lambda(Jobs, Precedes)\ Irreflexive(Precedes, Jobs) \\
R & \mapsto & seq(integer) \\
O & \mapsto & \lambda(Jobs, Precedes, p)\ Bijective(p, Jobs) \wedge Consistent(p, Precedes) \\
\hat{R} & \mapsto & seq(integer) \\
\hat{I} & \mapsto & \lambda(Jobs, Precedes, ps)\ range(ps) \subseteq Jobs \\
Satisfies & \mapsto & \lambda(p, ps)\ \exists(r)(p = concat(ps, r)) \\
\hat{r}_0 & \mapsto & [\,] \\
Splits\text{–}into & \mapsto & \lambda(Jobs, Precedes, ps, ps')\ \exists(i)\ (i \in Jobs\ \wedge\ ps' = append(ps, i)) \\
Extractable & \mapsto & \lambda(p, ps)\ p = ps
\end{array}$$

This theory defines a generator of partial schedules. Some of these partial schedules cannot possibly be extended to complete schedules. For example, if $a \in Jobs$ then partial schedules $[a, a]$, $[a, a, a]$, and so on, would be generated. The next design step is to derive mechanisms for pruning away such useless nodes of the search tree. The effect of this step is to incorporate problem-specific information into the generator in order to improve efficiency.

To derive a necessary filter for the Scheduling problem, the inference system is directed to produce necessary conditions on the existence of an extension of a partial solution *ps* that satisfies all the Scheduling constraints; formally

$$\begin{array}{l}
\text{find } some\ (\Phi) \\
\quad (\exists(p)\ (\ \exists(r)(p = concat(ps, r)) \\
\qquad\qquad\quad \wedge\ Bijective(p, Jobs) \\
\qquad\qquad\quad \wedge\ Consistent(p, Precedes)) \\
\quad \Longrightarrow \Phi(Jobs, Precedes, ps)).
\end{array}$$

Any such Φ serves as a filter since if Φ does not hold for some partial solution, then by the contrapositive of the implication there does not exist an extension that satisfies the Scheduling constraints. The derivations proceed as follows.

$Bijective(p, Jobs)$

$\Longleftrightarrow$ by definition of *Bijective*

$Injective(p, Jobs)\ \wedge\ range(p) = Jobs$

$\Longrightarrow$ applying $p = concat(ps, r)$ to the first conjunct

$Injective(concat(ps, r), Jobs)$

$\Longleftrightarrow$ distributing *Injective* over *concat*

$Injective(ps, Jobs)\ \wedge\ Injective(r, Jobs)$
$\wedge\ range(ps) \bigcap range(r) = \{\}$

$\Longrightarrow$ dropping conjuncts

$Injective(ps, Jobs).$

Also
Consistent(p,Precedes)

$\Longleftrightarrow$ applying $p = concat(ps, r)$

$Consistent(concat(ps, r), Precedes)$

$\Longleftrightarrow$ distributing *Consistent@rover@concat*

$Consistent(ps, Precedes)$
$\wedge\ Cross\text{–}Consistent(range(ps), range(r), Precedes)$
$\wedge\ Consistent(r, Precedes)$

$\Longrightarrow$ *@rusing@range(r)* $= Jobs \setminus range(ps)$

$Consistent(ps, Precedes)$
$\wedge\ Cross\text{–}Consistent(range(ps), Jobs \setminus range(ps), Precedes)$
¿From among the many derived consequents RAINBOW discards useless ones and presents a menu of possibilities for the user to choose from. The conjunction of any subset will result in a correct algorithm. It is possible to automate the selection of filters using dependency tracking but we have not done so at this writing.

$Injective(ps, Jobs)$
$\wedge\ Consistent(ps, Precedes)$
$\wedge\ Cross\text{–}Consistent(ps, Jobs \setminus range(ps), Precedes)$

In words, the partial solution must itself be Consistent with *Precedes*, contain no duplicate elements, and satisfy the cross-consistency condition – no element in the partial solution can be preceded by an uncommitted JOB.

Finally the recursive REFINE program in Figure 2 is produced by applying Theorem 1. That is, the correspondence between the symbols of abstract gs-theory and concrete expressions is used to instantiate the program scheme in Theorem 1. Note that the filter derived above is tested prior to each call to the backtracking function *SCHEDULES-GS* and thus the filter is displayed as an input invariant. Being produced as an instance of a program abstraction, this algorithm obviously has some inefficiencies, even though it is correct. The goal of a design tactic is to produce a correct, very-high-level, well-structured algorithm. Subsequent refinement and optimization is necessary in order to realize the potential of the algorithm.

2.4 Simplification

KIDS provides two expression simplifiers. The simplest and fastest, called the *Context-Independent Simplier* (CI-SIMPLIFY), applies rewrite rules that are fired exhaustively. Only those laws from the domain theory that are treated as rewrite rules are fired by CI-SIMPLIFY. Some typical equations used as rewrite rules are

$$length([]) = 0$$

```
function $SCHEDULES\text{–}GS$
  $(JOBS : set(JOB),$
   $Precedes : binrel(JOB, JOB),$
   $ps : seq(integer))$
where $Irreflexive(Precedes, Jobs) \land range(ps) \subseteq Jobs$
        $\land\ Consistent(ps, Precedes)$
        $\land\ Injective(ps, Jobs)$
        $\land\ @iCross - Consistent@(range(ps), Jobs \setminus range(ps), Precedes)$
returns $\{SCHED \mid$
                      $Extends(SCHED, ps) \land Consistent(SCHED, Precedes)$
                        $\land\ Bijective(SCHED, Jobs)\}$
  $= \{SCHED \mid Consistent(SCHED, Precedes)$
                       $\land\ Bijective(SCHED, Jobs) \land SCHED = ps\}$
   $\cup\ reduce(\cup,$
            $\{SCHEDULES\text{–}GS(Jobs, Precedes, @iNew - ps@) \mid$
                 $Consistent(@iNew - ps@, Precedes)$
                 $\land\ Injective(@iNew - ps@, Jobs)$
                 $\land\ @iCross - Consistent@(range(@iNew - ps@),$
                                          $Jobs \setminus range(@iNew - ps@), Precedes)$
                 $\land\ \exists(I)\ (@iNew - ps@ = append(ps, I) \land I \in Jobs)$
            $\})$

function $@iSCHEDULES@(Jobs : set(JOB), Precedes : binrel(@iJOB@, @iJOB@))$
where $Irreflexive(Precedes, Jobs))$
returns $\{SCHED \mid Bijective(SCHED, Jobs)$
                           $\land\ Consistent(SCHED, Precedes)\})$
= $if\ @iCross - Consistent@(range([\,]), Jobs \setminus range([\,]), Precedes)$
       $\land\ Injective([\,], Jobs) \land Consistent([\,], Precedes)$
   $then\ SCHEDULES\text{–}GS(Jobs, Precedes, [\,])$
   $else\ \{\}$
```

Fig. 2. Global search algorithm for the scheduling problem

(from sequence theory) and

$$if\ true\ then\ P\ else\ Q\ =\ P$$

(from boolean theory). We also treat the distributive laws in Scheduling theory as rewrite rules: e.g.

$$Injective([\,], S) \iff true$$

and

$$Injective(append(W, a), S) \iff (Injective(W, S) \land a \in S \land a \notin range(W)).$$

We apply CI-Simplify to the body of all newly derived programs. As a result, the conditional

```
if $@iCross - Consistent@(range([\,]), Jobs \setminus range([\,]), Precedes)$
        $\wedge$ $Injective([\,], Jobs) \wedge Consistent([\,], Precedes)$
then $SCHEDULES\text{–}GS(Jobs, Precedes, [\,])$
else $\{\}$
```

simplifies to $SCHEDULES - GS(Jobs, Precedes, [\,])$.

Another rule modifies a set former by replacing all occurrences of a local variable that is defined by an equality:

$$\{\, C(x) \mid x = e \wedge P(x) \,\} = \{\, C(e) \mid P(e) \,\}.$$

For example, this rule will replace *New-ps* by *append(ps, i)* everywhere in *SCHEDULES-GS*. This replacement in turn triggers the application of the laws for distributing *Consistent*, *Injective* and so on, over *append*.

The result of applying CI-Simplify to the bodies of *SCHEDULES* and *SCHEDULES-GS* is shown in Figure 3. (For brevity we will sometimes omit or use ellipsis in place of expressions that remain unchanged after a transformation).

```
function $SCHEDULES\text{–}GS$
  $(Jobs : set(JOB), Precedes : binrel(JOB, JOB), ps : seq(integer))$
where $Irreflexive(Precedes, Jobs) \wedge range(ps) \subseteq Jobs$
        $\wedge$ $Consistent(ps, Precedes)$
        $\wedge$ $Injective(ps, Jobs)$
        $\wedge$ $@iCross - Consistent@(range(ps), Jobs \setminus range(ps), Precedes)$
returns ...
= $\{ps \mid Consistent(ps, Precedes) \wedge Bijective(ps, Jobs)\}$
  $\cup$
  $reduce(\cup, \{SCHEDULES\text{–}GS(Jobs, Precedes, append(ps, I)) \mid$
                $I \in Jobs \wedge I \notin range(ps) \wedge \langle I, I\rangle \notin Precedes$
                $\wedge$ $@iCross - Consistent@(range(ps), \{I\}, Precedes)$
                $\wedge$ $Consistent(ps, Precedes)$
                $\wedge$ $Injective(ps, Jobs)$
                $\wedge$ $@iCross - Consistent@(\{I\}, (Jobs \setminus range(ps)) \setminus \{I\}, Precedes)$
                $\wedge$ $@iCross - Consistent@(range(ps), (Jobs \setminus range(ps)) \setminus \{I\}, Precedes)\})$

function $SCHEDULES(Jobs : set(JOB), Precedes : binrel(JOB, JOB))$
where $Irreflexive(Precedes, Jobs)$
returns ...
= $SCHEDULES\text{–}GS(Jobs, Precedes, [\,])$
```

Fig. 3. Scheduling code after context-independent simplification

There are other simplification opportunities in this code. For example, notice that the predicate *Injective(ps, Jobs)* is being tested in *SCHEDULES-GS*, but it is already true because it is an input invariant. The second expression simplifier, *Context-Dependent Simplify* (CD-Simplify), is designed to simplify a given expression with respect to its context. CD-Simplify gathers all predicates that hold in the context of the expression by walking up the abstract syntax tree gathering the test of encompassing conditionals, sibling conjuncts in the condition of a set-former, etc. and ultimately the input conditions of the encompassing function. The expression is then simplified with respect to this rich assumption set and the laws in the underlying domain theory. Technically, the inference system infers a variety of equivalent forms of the given expression, selecting one of these that minimizes a built-in heuristic measure of complexity.

In applying CD-Simplify to the predicate of the first set-former in *SCHEDULES-GS*, we analyze the context to find all properties that hold when the expression is evaluated:

$$\begin{aligned}&\mathit{Irreflexive}(\mathit{Precedes},\ \mathit{Jobs})\\ &\wedge\ \mathit{range}(ps) \subseteq \mathit{Jobs}\\ &\wedge\ \mathit{Consistent}(ps,\ \mathit{Precedes})\\ &\wedge\ \mathit{Injective}(ps,\ \mathit{Jobs})\\ &\wedge\ @i\mathit{Cross}-\mathit{Consistent}@(\mathit{range}(ps),\ \mathit{Jobs} \setminus \mathit{range}(ps),\ \mathit{Precedes})\end{aligned}$$

When we attempt to simplify the expression

$$\mathit{Consistent}(ps, \mathit{Precedes})\ \wedge\ \mathit{Bijective}(ps, \mathit{Jobs})$$

the first conjunct immediately unifies with an assumption and thus simplifies to *true*. For the second conjunct:

$\mathit{Bijective}(ps, \mathit{Jobs})$

$\Longleftrightarrow$ by definition of *Bijective*

$\mathit{Injective}(ps, \mathit{Jobs})\ \wedge\ \mathit{range}(ps) = \mathit{Jobs}$

$\Longleftrightarrow$ unifying the first conjunct with an assumption

$\mathit{range}(ps) = \mathit{Jobs}$

$\Longleftrightarrow$ by definition of set equality : $(S = T) = (S \subseteq T\ \wedge\ T \subseteq S)$

$\mathit{range}(ps) \subseteq \mathit{Jobs}\ \wedge\ \mathit{Jobs} \subseteq \mathit{range}(ps)$

$\Longleftrightarrow$ unifying the first conjunct with an assumption

$\mathit{Jobs} \subseteq \mathit{range}(ps)$.

The resulting simplified expression is $\mathit{Jobs} \subseteq \mathit{range}(ps)$. After applying CD-Simplify to the predicates of both set-formers in *SCHEDULES-GS* we obtain the code in Figure 4.

2.5 Partial Evaluation

Next we notice that the call to *Cross-Consistent* has an argument of a restricted form — a singleton set. This suggests the application of partial evaluation [1]. KIDS has the classic

```
function SCHEDULES–GS
  (Jobs : set(JOB), Precedes : binrel(JOB, JOB), ps : seq(integer))
where Irreflexive(Precedes, Jobs)
        ∧ range(ps) ⊆ Jobs
        ∧ Consistent(ps, Precedes)
        ∧ Injective(ps, Jobs)
        ∧ @iCross – Consistent@(range(ps), Jobs \ range(ps), Precedes)
= {V | Jobs ⊆ range(ps)}
  ∪ reduce(∪,
        {SCHEDULES–GS(Jobs, Precedes, append(ps, I)) |
              I ∈ Jobs ∧ I ∉ range(ps)
               ∧ @iCross – Consistent@({I}, (Jobs \ range(ps)) \ {I}, Precedes)
              })
```

Fig. 4. Scheduling Algorithm after CD-Simplify

UNFOLD transformation that replaces a function call by its definition (with arguments replacing parameters). Partial evaluation proceeds by first UNFOLDing then simplifying.

UNFOLDing *@iCross – Consistent@*$(\{I\}, (Jobs \setminus range(ps)) \setminus \{I\}, Precedes)$, we obtain

$$\forall(I1 : @iJOB@,\ J : @iJOB@)$$
$$(I1 \in \{I\} \ \land\ J \in (Jobs \setminus range(ps)) \setminus \{I\} \implies \neg\langle J,\ I1\rangle \in Precedes)$$

The rules

$$x \in \{a\} \iff (x = a)$$

$$\forall(x, y_1, \ldots, y_n)(Q(x) \ \land\ x = e \implies P(x)) \iff \forall(y_1, \ldots, y_n)(Q(e) \implies P(e)).$$

(imported with finite set theory and boolean theory) and others are used by CI-Simplify resulting in

$$\forall(J)(J \in Jobs \setminus range(ps) \ \land\ J \neq I \implies \langle J,\ I\rangle \notin Precedes).$$

2.6 Finite Differencing

Notice that the expression *range(ps)* in Figure 4 is computed each time *SCHEDULES-GS* is invoked and that the parameter *ps* changes in a regular way. This suggests that we create a new variable whose value is maintained equal to *range(ps)* and which allows for incremental computation – a significant speedup. This transformation is known as strength reduction or finite differencing [9] (see also [6]). We have developed and implemented a version of finite differencing for functional programs.

Finite differencing can be decomposed into two more basic operations: abstraction followed by simplification. The abstraction operation is presented informally in Figure 5. Abstraction of function f with respect to expression $E(x)$ adds a new parameter c to f's parameter list (now $f(x,c)$) and adds $c = E(x)$ as a new input invariant to f. Any

```
function @rg@(y)
= ...
   @rf@(V)
   ...

function @rf@(x)
where I(x)
= ...
   E(x)
   ...
   @rf@(U(x))
   ...
```

$\longrightarrow$

```
function @rg@(y)
= ...
   @rf@(V, E(V))
   ...

function @rf@(x, c)
where I(x) ∧ c = E(x)
= ...
   E(x)
   ...
   f(U(x), E(U(x)))
   ...
```

Fig. 5. **Abstraction operation underlying the finite differencing optimization**

call to f, whether a recursive call within f or an external call, must now be changed to supply the appropriate new argument that satisfies the invariant – $f(U)$ is changed to $f(U, E(U))$.

It now becomes possible to simplify various expressions within f and calls to f. In the KIDS implementation, CI-Simplify is applied to the new argument in all external calls. In terms of Figure 5, within f we temporarily add the invariant $E(x) = c$ as a rule and apply CI-Simplify to the body of f. This replaces all occurrences of $E(x)$ by c. Often, distributive laws apply to $E(U(x))$ yielding an expression of the form $U'(E(x))$ and then $U'(c)$. The real benefit of this optimization comes from the last step, because this is where the new value of the expression $E(U(x))$ is computed in terms of the old value $E(x)$.

Our approach to finite differencing differs from that in Paige's RAPTS system [9] in several respects. KIDS can incrementally maintain expressions containing user-defined terms as long as appropriate distributive laws are available. Also the initialization and update codes are performed in parallel with the modification to the dependent variable. Also there is considerable flexibility gained by relying on a common knowledge-base of laws rather than a specialized format as in RAPTS. On the other hand our functional approach relies on inference to perform simplifications whereas the RAPTS approach is specialized. Also, Paige has analyzed various set-theoretic expressions in order to ascertain sufficient conditions under which finite differencing results in a net improvement with respect to a simple performance model.

The evolving algorithm is prepared for finite differencing by subjecting it to a collection og built-in conditioning transformations. In this case they transform the two conjuncts

$$I \notin range(ps) \ \wedge \ I \in Jobs$$

to

$$I \in Jobs \setminus range(ps).$$

The rationale is to group together information concerning a local variable.

We select the set difference as an expression to maintain incrementally. The changes include (1) the addition of a new input parameter, named *Unscheduled–Jobs*, and its invariant to *SCHEDULES-GS*, (2) all occurrences of the term $Jobs \setminus range(ps)$ in *SCHEDULES-GS* are replaced by *Unscheduled–Jobs*, (3) appropriate arguments are created and simplified for all calls to the function *SCHEDULES-GS*. The initial call to *SCHEDULES-GS* becomes

$$SCHEDULES-GS(Jobs, Precedes,\ [\,],\ Jobs \setminus range([]))$$

which simplifies to

$$SCHEDULES-GS(Jobs, Precedes,\ [\,],\ Jobs).$$

The recursive call to *SCHEDULES-GS* becomes

$$SCHEDULES-GS\,(Jobs, Precedes,\ append(ps,\ I),\ Jobs \setminus range(append(ps,\ I)))$$

which simplifies to

$$SCHEDULES-GS\,(Jobs, Precedes,\ append(ps,\ I),\ Unscheduled\text{–}Jobs \setminus \{I\}).$$

The resulting code is shown in Figure 6.

```
function SCHEDULES–GS
  (Jobs : set(JOB), Precedes : binrel(JOB, JOB), ps : seq(integer),
   Unscheduled–Jobs : set(JOB))
where Unscheduled–Jobs = Jobs \ range(ps) ∧ ...
  = {ps | empty(Unscheduled–Jobs)}
    ∪ reduce(∪,
    {SCHEDULES–GS(Jobs, Precedes, append(ps, I), Unscheduled–Jobs \ {I}) |
          I ∈ Unscheduled–Jobs
          ∧ size({J | ⟨J, I⟩ ∈ Precedes ∧ J ∈ Unscheduled–Jobs ∧ J ≠ I}) = @m0@})
```

Fig. 6. Scheduling algorithm after one finite differencing step

Notice how finite differencing introduces a meaningful data structure at this point. The concept of which elements of *Jobs* have not yet been added to the partial solution *ps* would naturally occur to many programmers who are developing a scheduling algorithm. Here it is introduced by a problem-independent transformation technique. Not only is the concept natural in the context of the problem, but its incremental computation dramatically improves the efficiency of the algorithm. Note also the need for a software database – this transformation needs global access to all invocations of a function in order to consistently modify its interface.

Notice also that a minor simplification can be applied at this point; the expression $J \neq I$ is redundant in

$$\{J | \langle J, I\rangle \in Precedes \ \wedge\ J \in Unscheduled\text{–}Jobs \ \wedge\ J \neq I\}.$$

CD-Simplify reduces $J \neq I$ to *true* since *irreflexive(Precedes,Jobs)* and $\langle J, I\rangle \in Precedes$ imply $J \neq I$.

After this simplification we select

$$size(\{J \mid \langle J,I\rangle \in Precedes \land J \in Unscheduled\text{--}Jobs\})$$

for incremental maintenance. This finite differencing operation is somewhat more complex since the target expression involves a local variable I. The FD operation must analyze the context of the expression to determine a finite set of values that I ranges over. This finite bound becomes the domain of a map data structure. Here the analysis is relatively easy, since the expression $I \in Unscheduled\text{-}Jobs$ occurs in the immediate context. Thus KIDS produces a map data structure which we name *NUM-PREDS* (number of predecessors):

$$\{|I \rightarrow size(\{J|\langle J,I\rangle \in Precedes \land J \in Unscheduled\text{-}Jobs\}) \mid I \in Unscheduled\text{-}Jobs|\}).$$

KIDS produces the code in Figure 7.

```
function SCHEDULES–GS
  (Jobs : set(JOB), Precedes : binrel(JOB, JOB), ps : seq(integer),
   Unscheduled–Jobs : set(JOB),
   Numpreds : map(JOB, integer))
where Numpreds = {| I → size({J | ⟨J, I⟩ ∈ Precedes ∧ J ∈ Unscheduled–Jobs})
                          | I ∈ Unscheduled–Jobs |}
        ∧ Unscheduled–Jobs = Jobs \ range(ps) ∧ ...
  = {ps | empty(Unscheduled–Jobs)}
    ∪ reduce(∪, {SCHEDULES – GS
                          (Jobs, Precedes,
                          append(ps, I), Unscheduled–Jobs \ {I},
                          {| I1 → if ⟨I, I1⟩ ∈ Precedes
                                  then Numpreds(I1) – 1
                                  else Numpreds(I1)
                              | I1 ≠ I ∧ I1 ∈ Unscheduled–Jobs |})
                    | I ∈ Unscheduled–Jobs ∧ Numpreds(I) = @m0@})
```

Fig. 7. Scheduling algorithm after finite differencing

2.7 Case Analysis

The SCHEDULES-GS algorithm is a union of two set-valued expressions. Notice that these two sets treat disjoint cases – when one is nonempty the other is empty. This suggests the use of case-analysis to clarify and simplify the code. The idea of the case analysis transformation in KIDS is simple: an expression e is replaced with the expression *if* P *then* e *else* e, where P is a predicate whose variables are all bound in e's context. The payoff from this transformation rule comes from applying CD-simplification to the branches of the conditional. For *SCHEDULES-GS* we select the whole body as e and use *empty(Unscheduled-Jobs)* as the case analysis predicate. After simplification we get the code in Figure 8.

```
function SCHEDULES-GS (...)
= if empty(Unscheduled-Jobs)
    then {ps}
    else reduce(∪,
                 {@iSCHEDULES - GS@
                     (Jobs, Precedes,
                      append(ps, I),
                      Unscheduled-Jobs \ {I},
                      {| I1 → if ⟨I, I1⟩ ∈ Precedes
                                 then Numpreds(I1) − 1
                                 else Numpreds(I1)
                            | I1 ≠ I ∧ I1 ∈ Unscheduled-Jobs |})
                 | I ∈ Unscheduled-Jobs ∧ Numpreds(I) = @m0@})
```

Fig. 8. Scheduling algorithm after case analysis

2.8 Data Type Refinement

Our next step is to choose implementations for the abstract data types in the algorithm. Compilers typically provide a standard implementation for each type in their programming language. However as the level of the language rises, and higher-level data types, such as sets, sequences, and mappings, are included in the language, or as users specify their own abstract data types, standard implementations cease to be satisfactory. The difficulty is that the higher-level datatypes can be implemented in many different ways; e.g. sets may be implemented as lists, arrays, trees, etc. Depending on the mix of operations, their relative frequency of invocation, size information, and dataflow considerations, one implementation may be much better than another. Thus no single default implementation will give good performance for all instances of an abstract type.

We are currently integrating a data type refinement system, called DTRE [2], with KIDS. DTRE allows interactive specification of implementation annotations for data types in programs. It also provides machinery for specifying data type refinements as theory morphisms and a modified compiler that automatically translates high-level types to low-level implementations. The following refinements have been performed using DTRE, but required some manual transformation in order to deal with special assumptions in the current version. We continue the derivation as if DTRE and KIDS were smoothly integrated. We see no fundamental impediment to this integration.

Consider the sequence-valued parameter *ps* which denotes a partial schedule: it is initialized to the empty sequence once, the operation *append* is applied many times, and occasionally it is copied to the output. A standard representation for sequences is linked lists; however, this representation is expensive for *ps* because it entails copying *ps* every time the *@iappend@* operation is performed. A better representation is shown in Figure 9 where alternative versions of *ps* coexist and share common structure. The data structure *ps* is simply a pointer to the last element of the sequence. In this reversed list representation, initialization and *@iappend@* take constant time, and the assignment operation takes time linear in the length of *ps* (by tracing upwards from the element pointed to by *ps*).

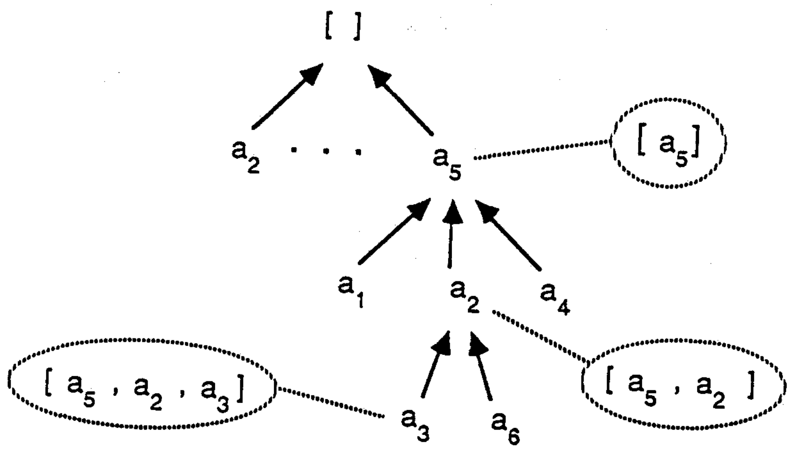


Fig. 9. A Structure-Sharing Representation of Sequences

The parameter *Numpreds* is map from *JOB* to integers. The further refinement and implementation of this data structure as a stack is detailed in [3].

2.9 Exploiting the Cost Function

We have derived a generator of schedules. The next task is to exploit the cost function *SWD-cost* (see Section 2.2) so that we can efficiently find minimal cost schedules. In Section 2.3 we showed how to use necessary conditions to prune away search paths that cannot lead to *feasible* solutions. Similarly we now use necessary conditions to prune away search paths that cannot lead to *minimal cost* solutions – we derive a necessary condition on the existence of optimal solutions. In [12] we show that several common pruning techniques such as lower bound pruning and dominance relations can be derived in this way.

Lower bound pruning works in the following way. Suppose that $lb(x,\hat{r})$ computes a lower bound on the cost of any solution in the space $\hat{r}$; i.e.

$$\forall(x:D,\hat{r}:\hat{R},z:R)\,(I(x)\wedge\hat{I}(x,\hat{r})\wedge Satisfies(z,\hat{r})\wedge O(x,z)\implies lb(x,\hat{r})\le f(x,z)) \tag{5}$$

where $f(x,z)$ denotes the cost of solution z. Suppose that during search we have a schedule of cost c. Any space $\hat{r}$ whose lower bound exceeds c cannot contain a better schedule than the one we have, so $\hat{r}$ can be eliminated from further consideration. Pruning via dominance relations generalizes lower bounds and is discussed in detail in [12].

Here we focus on the derivation of a lower bound function for SWD and how it can be used to improve the search proces. Interpreting (5), we assume

$$\begin{aligned}&Irreflexive(Precedes, Jobs)\\ &\wedge S = concat(ps, qs)\\ &\wedge Bijective(S, Jobs)\\ &\wedge Consistent(S, Precedes)\end{aligned}$$

and then derive a lower bound on the scheduling cost function

$$size(\{J \mid J \in range(S) \ \wedge \ Deadline(J) < Index(S, J)\}).$$

The derivation in Figure 10 results in a bound that is the number of Jobs in the partial schedule ps that have already missed their deadlines. Other terms which we have derived manually (but not presented in the figure) would measure (1) the number of Jobs not in ps that have already missed their deadlines (i.e. their deadline lies between 1 and $size(ps)$) and (2) the number of jobs J not yet in ps that could not possibly meet their deadline (because too many predecessors must be executed before J's earliest possible execution time). This additional term would improve the bound and thus improve performance of the search procedure. The lower bound function can be computed incrementally via finite differencing.

Exploiting the lower bound function gives us a classic branch-and-bound algorithm. It records the best schedule found so far in the search process and its cost ub, and deletes from consideration any subspace whose lower bound is not less than ub. Program schemes that incorporate lower bound functions and dominance relations are given in [12].

2.10 Summary

The final scheduling algorithm is apparently not very complicated, however we see that it is an intricate combination of knowledge of the scheduling problem, the global search algorithm paradigm, various program optimization techniques and data structure refinement. The derivation has left us not only with an efficient, correct program but also assertions that characterize the meaning of all data structures and subprograms. These invariants together with the derivation itself serve to explain and justify the structure of the program. The explicit nature of the derivation process allows us to formally capture all design decisions and reuse them for purposes of documenting the derivation and helping to evolve the specifications and code as the user's needs change.

KIDS is unique among systems of its kind for having been used to design, optimize, and refine programs for over 50 problems. Applications areas have included scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, and linear programming. We have had good success in using KIDS to account for the structure of many well-known algorithms and have often derived algorithms that were as good or better than previously known algorithms (see for example [11]). As an example, recently we have been working with the U.S. Transportation Command on transportation scheduling [18]. The transportation scheduling problem under study is much richer than the problem solved in this paper. We derived a global search algorithm that exploits constraint propagation. We compared our algorithm with a well-known scheduler

$$
\begin{aligned}
& \mathit{size}(\{J \mid J \in \mathit{range}(S) \ \wedge \ \mathit{Deadline}(J) < \mathit{index}(S, J)\}) \\
= \quad & @\mathit{rbyassumption}@S = \mathit{concat}(ps, qs) \\
& \mathit{size}(\{J \mid J \in \mathit{range}(\mathit{concat}(ps, qs)) \\
& \qquad \wedge \ \mathit{Deadline}(J) < \mathit{index}(\mathit{concat}(ps, qs), J)\}) \\
= \quad & @\mathit{rdistributing}@\mathit{range}@\mathit{rover}@\mathit{concat} \\
& \mathit{size}(\{J \mid (J \in \mathit{range}(ps) \ \vee \ J \in \mathit{range}(qs)) \\
& \qquad \wedge \ \mathit{Deadline}(J) < \mathit{index}(\mathit{concat}(ps, qs), J)\}) \\
= \quad & @\mathit{rdistributing}@ \\
& \mathit{size}(\{J \mid J \in \mathit{range}(ps) \\
& \qquad \wedge \ \mathit{Deadline}(J) < \mathit{index}(\mathit{concat}(ps, qs), J)\} \\
& \quad \bigcup \{J \mid J \in \mathit{range}(qs) \\
& \qquad \wedge \ \mathit{Deadline}(J) < \mathit{index}(\mathit{concat}(ps, qs), J)\}) \\
\geq \quad & @\mathit{rbymontonicitypropertiesofsets}@ \\
& \mathit{size}(\{J \mid J \in \mathit{range}(ps) \\
& \qquad \wedge \ \mathit{Deadline}(J) < \mathit{index}(\mathit{concat}(ps, qs), J)\} \\
= \quad & @\mathit{rsimplifying}@ \\
& \mathit{size}(\{J \mid J \in \mathit{range}(ps) \ \wedge \ \mathit{Deadline}(J) < \mathit{index}(ps, J)\}).
\end{aligned}
$$

Fig. 10. Deriving a lower bound function

(*OPIS*). On one data set with about 500 transportation requirements, *OPIS* takes over 30 minutes and finds a near-feasible schedule. Our derived scheduler finds a complete feasible schedule in less than .5 seconds, a factor of over 3600 times faster.

3 Theory of Algorithm Design

The discussion of algorithm design in Section 2.3 was limited to the class of global search algorithms. In this section we discuss a general theory of algorithm design. Our approach is based on representing the essential structure of various classes of algorithms as algorithm theories. An algorithm design tactic based on such algorithm theories provides a formal method for constructing instances of the class from a problem specification. 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.

3.1 Problem Theories

A *first-order theory presentation* (or more simply a *theory*) is a triple $\langle S, \Sigma, A\rangle$ consisting of sorts S, operations over those sorts Σ, and axioms A to constrain the meaning of the operations. A *theory morphism* (*interpretation between theories*) 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 $A_1 \subseteq A_2$.

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 $z : R$ is a *feasible solution* with respect to input $x : D$. 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.

For example, the problem of finding feasible schedules can be presented as a problem theory via a theory morphism:

$$\begin{array}{lll} D & \mapsto & set(JOB) \times binrel(JOB, JOB) \\ I & \mapsto & \lambda(Jobs, Precedes)\ Irreflexive(Precedes, Jobs) \\ R & \mapsto & seq(JOB) \\ O & \mapsto & \lambda(Jobs, Precedes, S)\ (S, Jobs)\ \wedge\ Consistent(S, Precedes) \end{array}$$

3.2 Algorithm Theories

An *algorithm theory* represents the essential structure of a certain class of algorithms A. 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 $\mathcal{B}$. A theory morphism from the algorithm theory into some problem domain theory provides the problem-specific concepts needed to construct an instance of an A algorithm.

For example, gs-theory presented in Section 2.3 extends problem theory with the basic concepts of backtracking: subspace descriptors, initial space, the splitting and extraction operations, filters, and so on. A divide-and-conquer theory would extend problem theory with concepts such as decomposition operators and composition operators [10, 15].

3.3 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. 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.

Operations $F : D \rightarrow set(R)$

$\ldots$

Axioms $\forall(x : D)(\ I(x) \implies F(x) = \{z \mid O(x, z)\}\ )$
$\forall(x : D)(\ I(x) \implies F(x) = Body(x)\ )$

$\ldots$

where *Body* is code that can be executed to compute F. In order to express *Body* it is generally necessary to import the theory of a programming 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.

For example, Theorem 1 asserts the consistency of one particular program theory for global search algorithms. This program uses recursion and is defined in the Refine language.

3.4 Refinement Hierarchy of Algorithm Theories

The algorithm theories that we have studied can be arranged in a refinement hierarchy as in Figure 11. Below each algorithm theory in this hierarchy are listed various well-known classes of algorithms or computational paradigms that are based on it. The refinement relation between theories is expressed as a theory morphism.

Starting at the root of the hierarchy, we briefly describe the various algorithm theories. 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 [7]. 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.

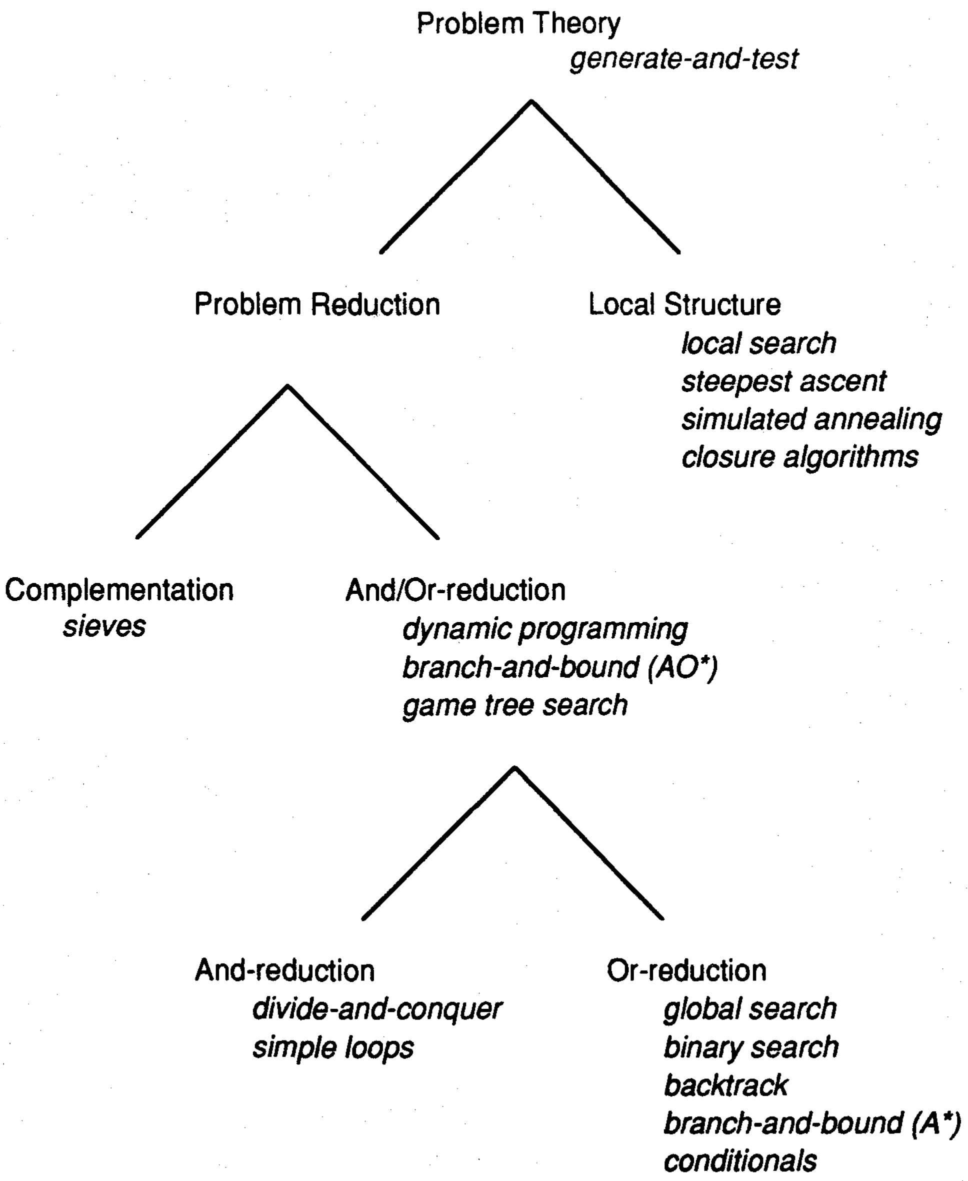


Fig. 11. Refinement Hierarchy of Algorithm Theories

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 [10].

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 [12].

And/or-reduction involves a combination of And- and Or-reductions resulting in alternative ways to decompose an initial problem [15]. This theory supports the design of dynamic programming, general branch-and-bound, and game tree search.

3.5 Design Tactics

Once we have characterized a class of algorithms A via an algorithm theory, and developed at least one program theory, the problem of constructing an A algorithm for a problem P is reduced to the construction of a theory morphism from the algorithm theory into the domain theory for P.

We have developed specialized design tactics for several algorithm theories. An $\mathcal{A}$-design tactic constructs an $\mathcal{A}$-algorithm for a given problem theory. Each tactic uses various techniques for constructing a theory morphism from $\mathcal{A}$ into the problem domain theory, and then instantiates the parameter of a program theory to produce a concrete program.

The first three steps of the global search design tactic construct the theory morphism (Φ is regarded as a defined function in gs-theory) and the last applies a program theory.

The global search tactic relies on a preexisting library of global search theories (for enumerating sets, sequences, maps, and so) and constructs a "connection" between the library theory and the problem domain theory [16]. In the scheduling problem we constructed a global search theory of scheduling via a connection from the gs-theory for sequences and scheduling theory.

The various steps of a tactic typically involve inference tasks. The highly structured context of these inference tasks tends to keep them relatively simple and thus tractable.

3.6 Classification-Based Design

A key problem in algorithm design is the choice of an appropriate algorithm theory. The refinement hierarchy provides a framework for solving this problem. The stronger a theory is, the more problem structure can be exploited in a program theory. Consequently, we want to construct a morphism from the deepest possible theories in the hierarchy to the given problem domain theory. This suggests the following procedure for accessing into the library. First, construct a morphism from the root theory of the hierarchy – this is simply a matter of viewing a specification as a problem (as we did above for scheduling). Next, once we have a morphism from some algorithm theory, then we attempt to incrementally construct an morphism from the children theories. If several succeed, then we can select one or keep several and repeat the process. If none succeed, then we know that (with respect to this classification hierarchy of algorithms) the current algorithm theory exploits as much of the problem structure as possible and the corresponding program theories should yield a fairly efficient program.

The process of incrementally constructing a morphism is illustrated in the "ladder construction" diagram on the left:

$$\begin{tikzcd}
\text{Problem Theory} \arrow[r, "m_0"] \arrow[d] & Spec_0 \arrow[d] \\
\text{Constraint Satisfaction} \arrow[r, "m_1"] \arrow[d] & Spec_1 \arrow[d] \\
\text{Linear Programming} \arrow[r, "m_3"] \arrow[d] & Spec_3 \arrow[d, dashed] \\
\text{Transshipment Problem} \arrow[r, dashed] & ?
\end{tikzcd}$$

$$\begin{tikzcd}
P_i \arrow[r, "m_i"] \arrow[d, "I_i"'] & S_i \arrow[d] & \\
P_{i+1} \arrow[r, "m'_{i+1}"] \arrow[rrd, "m_{i+1}"'] & P_{i+1} \oplus S_i \arrow[rd, "m''_{i+1}"] & \\
 & & S_{i+1}
\end{tikzcd}$$

The left-hand side of the ladder is a path in a refinement hierarchy of algorithm theories starting at the root (Problem Theory). $Spec_0$ is a given specification theory of a problem. The ladder is constructed a rung at a time from the top down. The initial arrow (theory morphism) from problem theory to $Spec_0$ is trivial. Subsequent rungs are constructed abstractly as in the diagram on the right above, where $P_{i+1} \oplus S_i$ is the pushout theory (in the category of theories and theory morphisms) and S_{i+1} is an extension of S_i determined by constructing the theory morphism m''_{i+1}. The morphism m_{i+1} is determined by composition.

The creative step is the construction of m''_{i+1}. Recently, we analyzed the algorithm design tactics in KIDS and abstracted out four general mechanisms for completing the construction of theory morphisms: verification, composition, unskolemization, and connections between specifications (see [16] for details). These are being implemented in KIDS and used to support algorithm design directly from algorithm theories.

3.7 Concluding Remarks

The preceeding approach to algorithm design raises several issues for further research. First, it builds on logical concepts of theories and interpretations between theories. Most of our work has been in first-order logic. However many problems have features of time, state, exceptions, uncertainty, and so on that suggest the need for a variety of richer logics and inference mechanisms. How do the concepts of algorithm theory generalize to these new logics? A second issue concerns the very notion of a classification hierarchy of algorithms. How much coverage could such a hierarchy provide? This depends of course on the extent of the hierarchy and how fine the refinement relationships are. Some researchers believe that only a domain-specific hierarchy can provide the kind of coverage necessary to be economically useful. A related issue is how hard is would be to populate a useful hierarchy. The theories discussed above have required significant effort to develop. Are there effective techniques for identifying and formalizing new classes of algorithms?

Acknowledgements

I would like to thank Jim McDonald and the referees for their valuable comments on a draft of this paper. This research was supported in part by the Office of Naval Research under Contracts N00014-87-K-0550 and N00014-90-J-1733, the Rome Air Development Center under contract F30602-88-C-0127, and the Air Force Office of Scientific Research under Contracts F49620-88-C-0033 and F30602-91-C-0043.

References

1. Bjørner, D., Ershov, A. P., and Jones, N. D., Eds. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.
2. Blaine, L., and Goldberg, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.
3. Blaine, L., Goldberg, A., Pressburger, T., Qian, X., Roberts, T., and Westfold, S. Progress on the KBSA Performance Estimation Assistant. Tech. Rep. KES.U.88.11, Kestrel Institute, May 1988. Appeared in 3rd Annual RADC KBSA Conference, August 2–4, 1988, Utica, New York.
4. Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
5. Goguen, J. A., and Winkler, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.
6. Gries, D. *The Science of Programming*. Springer-Verlag, New York, 1981.
7. Lowry, M. R. Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence* (Seattle, WA, July 13–17, 1987). Technical Report KES.U.87.10, Kestrel Institute, August 1987.
8. Manna, Z., and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems 2*, 1 (January 1980), 90–121.
9. Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402–454.
10. 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.).
11. Smith, D. R. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming 8*, 3 (June 1987), 213–229.
12. Smith, D. R. Structure and design of global search algorithms. Tech. Rep. KES.U.87.12, Kestrel Institute, November 1987.

13. 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 (September 1990), 1024–1043.
14. SMITH, D. R., AND LOWRY, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.
15. 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.
16. SMITH, D. R. Constructing specification morphisms. Tech. Rep. KES.U.92.1, Kestrel Institute, January 1992. to appear in Journal of Symbolic Computation, Special Issue on Automatic Programming, 1993.
17. SMITH, D. R. Track assignment in an air traffic control system: A rational reconstruction of system design. In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference* (McLean, VA, September 1992), pp. 60–68.
18. SMITH, D. R. Transformational approach to scheduling. Tech. Rep. KES.U.92.2, Kestrel Institute, November 1992.