In *Automating Software Design*, M. Lowry & R. McCartney, Eds. AAAI/MIT Press, 1991, 483-514.

KIDS - A Knowledge-Based Software Development System

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
31 August 1990

Abstract

The Kestrel Interactive Development System (KIDS) provides knowledge-based support for the derivation of correct and efficient programs from formal specifications. We trace the use of KIDS in deriving an algorithm for solving a problem arising from the design of sonar and radar signals. This derivation illustrates algorithm design, a generalized form of deductive inference, program simplification, finite differencing optimization, partial evaluation, case analysis, and data type refinement. All of the KIDS operations are automatic except the algorithm design tactics which presently require some interaction. Dozens of programs have been derived using the KIDS environment and we believe that it could be developed to the point where it can be used for routine programming.

1. INTRODUCTION

The construction of a computer program is based on several kinds of knowledge: knowledge about the particular problem being solved, general knowledge about the application domain, programming knowledge peculiar to the domain, and general programming knowledge about algorithms, data structures, optimization techniques, performance analysis, etc. We report here on an ongoing effort to formalize and automate various sources of programming knowledge and to integrate them into a highly automated environment for developing formal specifications into correct and efficient programs (cf. Balzer 1983). The system, called KIDS (Kestrel Interactive Development System), provides tools for performing deductive inference, algorithm design, expression simplification, finite differencing, partial evaluation, data type refinement, and other transformations. The KIDS tools serve to raise the level of language from which the programmer can obtain correct and efficient executable code through the use of automated tools.

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 the Appendix). 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.

The unique features of KIDS include its algorithm design tactics and its use of a 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, fully automatic (except the algorithm design tactics which require some interaction at present) and perform significant, meaningful steps from the user's point of view. Dozens of programs have been derived using the system and we believe that KIDS could be developed to the point that it becomes economical to use for routine programming.

After briefly discussing the environment and inference system underlying KIDS, we step through the derivation of a program for enumerating all solutions to the Costas Array problem (Costas 1984), which is used to generate optimal sonar and radar signals. 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 (Smith 1987) is created that works by extending partial solutions. To improve efficiency we apply simplification and partial evaluation (Bjorner 1988, Scherlis 1981) operations. We also perform finite differencing (Paige 1982) 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.

The initial algorithm that KIDS designs takes about fifty minutes on a SUN Sparcstation to enumerate all solutions to the order 9 Costas Array problem. The final optimized version finds the same solutions in twelve seconds. We manually refined the optimized algorithm into C and this code was able to enumerate all order 17 solutions in about six days time, thereby duplicating all published results (Silverman et al. 1988).

2. Usage of KIDS

We present an overview of general characteristics of the KIDS system and how it is used. Currently, KIDS runs on Symbolics and SUN-4 workstations. It is built on top of Refine[1], a commercial knowledge-based programming environment (Abraido-Fandino 1987). The Refine environment provides

---

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

- 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.

The KIDS system 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 pretty printer lays out the text and sets up active regions on the screen so that by moving the mouse around, the system can compute the nearest subexpression in the text and highlight it.

KIDS is a program transformation system -- one applies a sequence of correctness-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 by defining appropriate types and functions. The user also provides laws that allow high-level reasoning about the defined functions. Our experience has been that distributive and monotonicity laws provide most of the laws that are needed to support design and optimization. Recently we have added a theory development component to KIDS that supports the automated derivation of distributive laws.

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 (method) from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction (reducing a specification to a

library routine) (Smith 1985), divide-and-conquer (Smith 1985), global search (binary search, backtrack, branch-and-bound) (Smith 1987), and local search (hillclimbing) (Lowry 1987, Lowry 1989).

4. *Apply optimizations* - The KIDS system allows the application of optimization techniques such as simplification, partial evaluation, finite differencing, and other transformations. 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.

5. *Apply data type refinements* - The user can select implementations for the high-level data types in the program. Data type refinement rules 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 Appendix 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 the simplifier. This ability to select arguments by pointing greatly enhances the usability of a program transformation system.

3. Preliminaries

3.1. Language

A functional specification/programming language augmented with set-theoretic data types will be used in this paper. The main type constructors and their operations (listed below) are based on those of the Refine language. The boolean type admits the usual operators and quantifiers of the predicate calculus ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$) with the exception that equality (=) is used for logical equivalence.

Finite Sets

S:set(Nat)	type declaration
{ }	empty set
{1,2,4,8}, *{2 ..5}*	literal set-former; *{2..5} = {2,3,4,5}*
$\{f(x) \mid P(x)\}$	general set-former

$\in\ \notin\ =\ \neq\ \subseteq$	comparison predicates: membership, equality, improper subset
$\cap$, $\cup$, union!	intersection, union, disjoint union
setdiff(S,R)	set difference; *setdiff({1,2,3},{2,4}) = {1,3}*
reduce(bop, S)	reduction of the set *S* by the associative and commutative operator *bop*; e.g., *reduce(*$\cup$*, {{1,2},{2},{3}}) = {1,2,3}*.

Finite Sequences

A:seq(integer)	type declaration
[]	empty sequence
[2,4,6,8]	literal sequence
empty(A)	*A = []*
length(A)	the length of *A*
A(i)	the *i*th element of *A;* e.g. *[4,5,6](2) = 5*
$\in\ \notin\ =\ \neq$	comparison predicates: membership, equality
domain(A)	the set of integers *{1 .. length(A)}*
range(A)	same as $\{A(i) \mid i \in domain(A)\}$
concat(A,B)	concatenate sequences *A* and *B*
append(A,x)	same as *concat(A,[x])*

Finite Maps

M:map(integer,integer)	type declaration
$\{\mid \mid\}$	empty map
$\{\mid x \rightarrow f(x) \mid P(x) \mid\}$	general map-former
domain(M), range(M)	domain and range of *M*
map-union(M,N)	*map-union(M,N)(x) = if* $x \in$ *domain(N) then N(x) else M(x)*
dom-shift(f,M)	$\{\mid f(x) \rightarrow M(x) \mid x \in domain(M) \mid\}$

3.2. Specifications

In this paper a formal specification serves to define the *problem* for which we desire an efficient computational solution. We define a problem by means of functional constraints on input/output behavior. A *specification* can be presented as a quadruple $F = \langle D,R,I,O \rangle$ where *D* is the input type restricted to those values satisfying $I{:}D \rightarrow boolean$, the *input condition* (also called *input assumptions*). The output type is R and the *output condition* $O{:}D \times R \rightarrow boolean$ defines the notion of acceptable or feasible solutions -- if *O(x,z)* then we say *z* is a *feasible solution* with respect to input *x*.

Specifications/programs will also be presented in a more program-like format:

Function *F (x:D)*
  Where *I(x)*
  Returns *{z:R | O(x,z)}*
  = *Body*

This program specification for problem *F* returns the set of *all* values *z* of type *R* that satisfy the output condition *O*. The expression *Body* (when present) is code that can be executed to compute *F*. A specification of this form is *consistent* if for all possible inputs satisfying the input condition, the body produces the same set as specified in the Returns expression; formally

$$\forall(x{:}D)(\, I(x) \Rightarrow F(x) = \{z \mid O(x,z)\}).$$

The KIDS interface shown in Appendix 1 separates the program body (left pane) from the input and output conditions, called the interface specification (right pane).

3.3. Directed Inference

Deductive inference is necessary for applying general knowledge to particular problems. We have built a system called RAINBOW II that performs a form of deduction called *directed inference* (Smith 1982). In directed inference, a *source* term (or formula) is transformed into a *target* term (or formula) bearing a specified relationship to the first. As special cases it can perform first-order theorem-proving and formula simplification. It also allows the inference of sufficient conditions (antecedents) or necessary conditions (consequents) of a formula. This flexibility allows us to formulate a variety of design and optimization problems as inference tasks. Directed inference can play a constructive role in design rather than simply verifying work done by the user or by some system.

Generally, inference tasks in this paper are specified in the following form (which is slightly simplified for the purposes of presentation)

Find some $(Target)\ (A \Rightarrow (Source(x_1, \ldots, x_m) \rightarrow Target(y_1, \ldots, y_m))$

where *A* is a conjunction of assumptions, *Source* is the "source" term (or formula), and $\rightarrow$ is a reflexive and transitive ordering relation between terms, called the *inference direction*. For notational simplicity all free variables are universally quantified. In words, we want to derive some term (or formula) Target expressed over the variables $\{y_1, .. ,y_m\}$ (a subset of the free variables $\{x_1, .. , x_m\}$ of *Source*) such that the relationship $Source(x_1, .. ,x_m) \rightarrow Target(y_1, .. ,y_m)$ holds under the given assumptions. Currently the inference direction can be specified to be one of the following.

forward inference	$\Rightarrow$
backward inference	$\Leftarrow$
simplification	$=$
deriving a lower bound	$\geq$
deriving an upper bound	$\leq$

The inference process involves applying a sequence of transformations to the source term. The transformations are restricted to those that preserve the specified inference direction.

RAINBOW II relies on a library of theories comprising over 700 rules for reasoning about Refine program expressions. The rules have the general form of conditional rewrite rules:

$$C \Rightarrow (s \rightarrow t)$$

where $\rightarrow$ is an inference direction (as above), C is an applicability condition, s is the source expression, and t is the target expression. The rules are automatically compiled from first-order theorems and are indexed according to (1) the dominant operator symbol in s and (2) the inference direction. For example, when deriving a necessary condition on $P(f(x),g(x,y))$, RAINBOW II retrieves and tries to apply all rules whose dominant symbol is P and whose inference direction is either $\Rightarrow$ or $=$. RAINBOW II keeps track of how many inequations it has applied in deriving each target expression and uses this quantity to compute a measure of "semantic distance" of the source from the target. Semantic distance plus a heuristic measure of computational complexity is used to select an optimal solution from amongst the derived solutions.

Most of the development operations in this paper invoke RAINBOW II. Some of these tasks could be performed more efficiently by special-purpose inference systems, but we feel that the flexibility and conceptual economy allowed by using a common library of laws and a general-purpose inference system has resulted in a net productivity gain in our research.

RAINBOW II can be run in interactive or automatic modes, although in KIDS it is almost always treated as a subroutine that runs automatically and returns a result. It can be thought of as a transformational search engine that explores alternatives and selects solutions on the basis of a simple complexity measure (which can be user-supplied). The traditional problems with using general-purpose inference systems are treated by carefully structuring the deductive tasks that are fed to RAINBOW II so that solutions can be reached without deep search. Also, resource bounds are placed on the execution of RAINBOW II and it returns the best solution that it can find within the bounds.

4. A Session with KIDS

KIDS has been used to design and optimize global search algorithms for several dozen problems, some of which are listed below in Section 4.9. We use the Costas Array problem to illustrate KIDS.

					-1
				-3	1
			-4	-1	-2
		1	-2	-4	3
	-2	3	-5	1	2
2	4	1	6	5	3

Figure 1 - Costas Array of order 6 and its difference table

4.1. The Costas Array Problem

In (Costas 1984), Costas introduced a class of permutations that can be used to generate radar and sonar signals with ideal ambiguity functions. Since then there has been a flurry of work investigating various combinatorial properties of these permutations, now known as Costas arrays. No general construction has been found and the problem of enumerating Costas arrays has been explored by computer search (Silverman 1988). A *Costas Array* is defined as a permutation of the set *{1 .. n}* such that there are no repeated elements in any row of its difference table (See Figure 1). The first row of the difference table gives the difference of adjacent elements of the permutation, the second row gives the difference of every other element, and so on.

4.2. Domain Theory and Specification

Before a specification can be written, the relevant concepts, operations, relationship, 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* (or simply a *theory*) comprises a set of imported theories, new type definitions, function specifications with optional operational definitions, laws (axioms and theorems), and rules of inference (cf. Goguen 1988, Guttag 1986). The domain theory for the Costas Array problem is presented in Appendix 2 and is summarized below. A library of theories is maintained with importation providing hierarchic structure. As of mid-1990 about 30% of KIDS' domain knowledge is encapsulated in 25 domain theories. The rest of its domain knowledge is represented as an unstructured collection of definitions and rules.

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 theorems from definitions and axioms.

Just as there may be several ways to solve a given problem, there may be several ways to formalize a domain. Some formulations are provide a better foundation for subsequent development than others. We tried several formulations of the domain theory for Costas arrays, two of which are discussed below. A useful heuristic in constructing a domain theory is that the laws for reasoning about the domain concepts should be simple. A related notion is that over 80% of the laws needed to support design and optimization in KIDS are distributive laws - analogous to the familiar distributive laws of arithmetic:

$$a \times (b + c) = (a \times b) + (a \times c) \qquad \text{(distribute } \times \text{ over } +)$$
$$a + (b \times c) = (a + b) \times (a + c). \qquad \text{(distribute } + \text{ over } \times)$$

Consequently, our working methodology is to favor domain concepts that have simple distributive laws whenever possible. In addition, tools have been added to KIDS that support the derivation of distributive laws for user-designated functions.

Our first attempt at a Costas Array theory was based on the notion of the various rows of the difference table. The abstract data type of sequences over $\{1 .. n\}$ was used to represent solutions. We defined $diff(p,i)$ to be the bag of i th differences of the sequence p. The domain theory based on this concept was so complicated that the author had a hard time developing its laws. It turned out for example that *diff* distributes poorly over the operation of sequence concatenation. This suggested that there must be a simpler formulation.

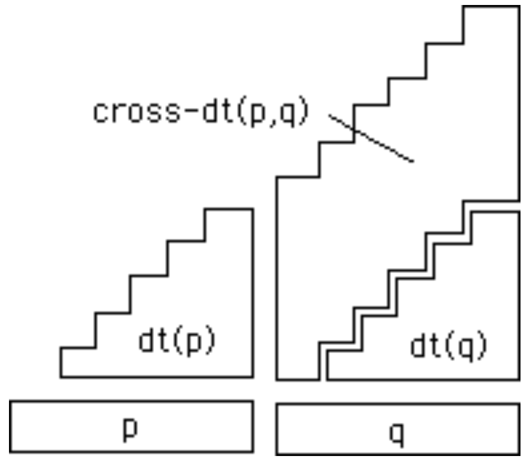

Figure 2 - Distributive law for difference tables

Our second attempt focused on the difference table itself as basic concept and this proved successful. The following function builds a difference table for a given sequence.

function *dt (p:seq(integer)) :map(tuple(integer,integer),integer)*
  $= \{|\ \langle i,j\rangle \rightarrow p(i) - p(i - j)\ |\ (i,j)\ i \in domain(p)\ \&\ j \in \{1\ ..\ i - 1\}\ |\}$

As can be seen in Figure 2, *dt* distributes nicely over concatenation of sequences:

$dt([]) = \{|\ |\}$

$dt([a]) = \{|\ |\}$

dt(concat(p,q)) = dt(p) map-union cross-dt(p,q) map-union dom-shift(lambda(x)x+size(p),dt(q))

where

function *cross-dt (p:seq(integer), q:seq(integer)) : map(tuple(integer,integer),integer)*
  $= \{|\ \langle i,j\rangle \rightarrow q(i - n) - p(i - j)\ |\ (i,j,n)$
    $n = size(p)$
    $\&\ i \in image(lambda(k)\ k + n, domain(q))$
    $\&\ j \in \{i - n\ ..\ i - 1\}\ |\}.$

The *dt* function entails the need for other functions: *dtrow(d,i)* returns the *i* th row of the difference table *d*; *nodups(s)* holds iff sequence (row) *s* does not contain duplicate occurrences of some element. Distributive laws for these concepts are straightforward. The concept that a sequence is a permutation is expressed by the notion of a bijection. This latter concept and associated laws for reasoning about it is imported with the theory called SEQUENCES-AS-MAPS.

function *injective (M:seq(integer), S:set(integer)):boolean*
$= range(M) \subseteq S$
$\& \ \forall\ (i, j)(i \in domain(M)\ \&\ j \in domain(M)\ \&\ i \neq j \Rightarrow M(i) \neq M(j))$

function *bijective (M:seq(integer), S:set(integer)):boolean*
$= injective(M,S)\ \&\ range(M)=S$

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.

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

$\forall(W,a,S)\ (\ injective(append(W,a),S) = (injective(W,S)\ \&\ a \in S\ \&\ a \notin range(W)))$

$\forall(W1,W2,S)\ (\ injective(concat(W1,W2),S)$
$= (injective(W1,S)\ \&\ injective(W2,S)\ \&\ (range(W1) \cap range(W2) = \{\})))$

The complete Costas Array theory used by KIDS is listed in Appendix 2.

We can now formulate a specification for the Costas Array problem:

function costas (*n*:*integer*)
where $1 \leq n$
returns $\{\ p \mid (p{:}seq(integer))$
$bijective(p,\ \{1\ ..\ n\})$
$\&\ \forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(p),j)))\ \}$

4.3. Algorithm Design

The next step is to develop a correct, high-level algorithm for enumerating Costas Arrays. We select a global search tactic in order to design a backtrack algorithm. The basic idea of global search (Smith 1987).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 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* . In addition to the extraction and splitting operations mentioned above, the abstract data 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) $\mathbf{G} = \langle \mathbf{B}, \mathbf{S}, \mathbf{J}, \mathbf{s_0}, \mathbf{Satisfies}, \mathbf{Split}, \mathbf{Extract} \rangle$ is presented as follows:

Sorts **D**, **R**, **S**

Operations

$\mathbf{I} : \mathbf{D} \to$ boolean

$\mathbf{O} : \mathbf{D} \times \mathbf{R} \to$ boolean

$\mathbf{J} : \mathbf{D} \times \mathbf{S} \to$ boolean

$\mathbf{s_0} : \mathbf{D} \to \mathbf{S}$

$\mathbf{Satisfies} : \mathbf{R} \times \mathbf{S} \to$ boolean

$\mathbf{Split} : \mathbf{D} \times \mathbf{S} \times \mathbf{S} \to$ boolean

$\mathbf{Extract} : \mathbf{R} \times \mathbf{S} \to$ boolean

Axioms

GS0. $\mathbf{I(x)} \Rightarrow \mathbf{J(x,s_0(x))}$

GS1. $\mathbf{I(x)}\ \&\ \mathbf{J(x,s)}\ \&\ \mathbf{Split(x,s,t)} \Rightarrow \mathbf{J(x,t)}$

GS2. $\mathbf{I(x)\ \&\ O(x,z)\ \Rightarrow\ Satisfies(z,\ s_0(x))}$

GS3. $\mathbf{I(x)\ \&\ J(x,s)\ \Rightarrow\ (Satisfies(z,s)\ =\ (\exists t)(Split^*(x,s,t)\ \&\ Extract\ (z,t)))}$

where **B = <D,R,I,O>** constitutes a problem specification, **S** is the type of space descriptors, **J** defines legal space descriptors, **s** and **t** vary over descriptors, $\mathbf{s_0(x)}$ is the descriptor of the initial set of candidate solutions, **Satisfies**(z,s) means that **z** is in the set denoted by descriptor **s** or that **z** satisfies the constraints that **s** represents, **Split(x,s,t)** means that **t** is a subspace of **s** with respect to input **x**, and **Extract (z,s)** means that **z** is directly extractable from **s**. Axiom GS0 asserts that the initial descriptor $\mathbf{s_0(x)}$ is a legal descriptor. Axiom GS1 asserts that legal descriptors split into legal descriptors and that **Split** 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 **s** -- an output object **z** is in the set denoted by **s** if and only if **z** can be extracted after finitely many applications of **Split** to **s** where

$$\mathbf{Split^*(x,s,t)} = \exists(\mathrm{k{:}Nat})(\mathbf{Split^k(x\ s,t)})$$

and

$$\mathbf{Split^0(x,s,r)} = (\mathbf{s} = \mathbf{r})$$

and for all natural numbers k

$$\mathbf{Split^{k+1}(x,s,r)\ =\ \exists\ (t{:}S)\ (Split(x,s,t)\ \&\ Split^k(x,t,r)).}$$

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

Example: Enumerating sequences over a given set

Consider the problem of enumerating sequences over a given finite set *S*. A space is a set of sequences with common prefix *V* and is represented as a sequence *V*. The descriptor for the initial space is just *[]*. Splitting is performed by appending an element from *S* onto the end of the common prefix *V*. The sequence *V* itself is directly extractable from the space. This global search theory, called *gs_sequences_over_finite_set,* for enumerating sequences can be presented via a correspondence between the components of abstract gs-theory (Smith 1989) and a concrete gs-theory (technically this correspondence is known as theory interpretation or theory morphism).

D $\rightarrow$ $set(\alpha)\times integer$

I $\rightarrow \lambda\ S.\ true$

R $\rightarrow$ $seq(\alpha)$

O $\rightarrow \lambda\ S,\ q.\ \ range(q) \subseteq S$

S $\rightarrow$ $seq(\alpha)$

J	$\rightarrow \lambda\, S,\, V.\;\; range(V) \subseteq S$
Satisfies	$\rightarrow \lambda\;\, q,\, V.\;\; \exists\, (r)\; (q = concat(V,\, r))$
$\mathbf{s_0}$	$\rightarrow\; \lambda\, S.\;\; []$
Split	$\rightarrow\; \lambda\, S,\;\; V,\, V'.\; \exists\, (i{:}S)\; (V' = append(V,\, i))$
Extract	$\rightarrow \lambda\, q,\, V.\;\; q{=}V$

In words, the abstract input domain symbol **D** is interpreted here to be $set(\alpha)\times integer$ where α is a type variable, allowing the theory to be polymorphic; the input condition **I** is interpreted to be the trivial predicate $\lambda\, S.\; true$ that always returns *true;* the **split** operation of global search theory is interpreted as $\lambda\, S,\;\; V,\, V'.\; \exists\, (i{:}S)\; (V' = append(V,\, i))$, and so on. This correspondance is used later to instantiate abstract programs (program schemas).

In addition to the above components of global search theory, there are various other operations which 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* is used to eliminate spaces from further processing. The *ideal filter* decides the question "Does there exist a feasible solution in space **s** ?", or, formally,

$$\exists\, (\mathbf{z{:}R})\; (\mathbf{Satisfies}(\mathbf{z},\mathbf{s})\;\&\;\mathbf{O}(\mathbf{x},\mathbf{z})). \tag{2}$$

However, to use (2) directly as a filter would usually be too expensive, so instead we use an approximation to it. A *necessary filter* Φ satisfies

$$\exists\, (\mathbf{z{:}R})\; (\mathbf{Satisfies}\;(\mathbf{z},\mathbf{s})\;\;\&\;\;\mathbf{O}(\mathbf{x},\mathbf{z})) \Rightarrow \Phi\;(\mathbf{x},\mathbf{s}). \tag{3}$$

By the contrapositive of (3), if $\Phi(\mathbf{x},\mathbf{s})$ is false for some space **s** then there does not exist a solution in **s**. Thus necessary filters can be used to eliminate spaces that do not contain solutions. We derive Φ by using directed inference to infer a necessary condition on Formula (2) that depends only on the variables **x** and **s**.

The design tactic for global search in KIDS is based on the following theorems. The proofs may be found in (Smith 1987). 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 **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 | O(x,z)}**
= If Φ**(x,s$_0$ (x))**
Then **F_gs(x,s$_0$(x))**
Else { }

function **F_gs (x:D, s:S) : set(R)**
where **I (x) & J(x,s) & Φ(x,s)**
returns **{z | Satisfies (z,s) & O(x,z)}**
= **{z | Extract (z, s) & O(x,z)}**
$\cup$ **reduce($\cup$, { F_gs(x,t) | Split(x,s,t) & Φ(x,t)}).**

In words, the abstract global search program works as follows. On input **x** the program F calls F_gs with the initial space **s_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 **s**, and (2) the union of all solutions found recursively in spaces **t** that are obtained by splitting **s** 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 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 $B_A = < D_A, R_A, I_A, O_A >$ *completely reduces* to specification $B_B = <D_B, R_B, I_B, O_B>$ if

$$R_A = R_B \ \& \ \forall (x:D_A) \exists (y:D_B) \forall (z:R_A)(I_A(x) \ \& \ O_A(x,z) \Rightarrow O_B(y,z)). \tag{4}$$

B_A *completely reduces to* B_B *with substitution* θ if $\theta(y) = t(x)$ and $R_A = R_B\theta$ and

$$\forall (x{:}D_A)\ \forall (z{:}R_A)(\ I_A(x)\ \&\ O_A(x,z) \Rightarrow O_B(t(x),z))$$

Theorem 2. Let G_B = <B_B, **S**, **J**, $\mathbf{s_0}$, **Satisfies**, **Split**,**Extract**> be a global search theory, and let B_A be a specification that completely reduces to B_B with substitution θ, then the structure G_A = <B_A, **S**θ, **J**θ, **Satisfies**θ, $\mathbf{s_0}$θ, **Split**θ, **Extract** θ> is a global search theory.

A simplified tactic for designing global search algorithms has three steps.

1. Select a global search theory 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 B_A completely reduces to B_B by verifying (4). Apply Theorem 2 to create a specialized global search theory G_A .

3. Derive a necessary filter Φ via Formula 2. That is, use directed inference to derive a necessary condition of Formula 2 expressed over the variables {**x**,**s**}. Apply Theorem 1 to create a global search program.

The tactic is sound and thus only generates correct programs. The interested reader should consult (Smith 1987) 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 Costas Array 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.

$$\begin{aligned}
&set(integer) = set(\alpha)\ \& \\
&\forall (n{:}\ integer)\ \exists (S{:}\ set(integer))\ \ \forall (assign{:}\ seq(integer)) \\
&\quad (bijective(assign, \{1..n\})\ \&\ \forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(p),j))) \Rightarrow range(assign) \subseteq S).
\end{aligned}$$

The derivation is simple and proceeds as follows: The types are unified yielding substitution $\{|\ \alpha \rightarrow\ integer\ |\}$. By forward inference from *bijective(assign, {1..n})* KIDS derives

$$injective(assign, \{1..n\}) \text{ and } range(assign) = \{1..n\},$$

then

$range(assign) \subseteq S$

$=$ applying $range(assign) = \{1..n\}$

$\{1..n\} \subseteq S$

$=$ unifying with the reflexivity law $\forall(R)(R \subseteq R)$

$true$ with substitution $\{| S \rightarrow \{1 .. n\} |\}$.

Thus, altogether the Costas Array problem completely reduces to *gs_sequences_over_finite_set* with substitution $\theta = \{| \alpha \rightarrow integer, S \rightarrow \{1 .. n\} |\}$. The construction in Theorem 2 yields the following global search theory.

D $\rightarrow$ *integer*
I $\rightarrow \lambda\, n.\ 1 \leq n$
R $\rightarrow$ *seq(integer)*
O $\rightarrow \lambda n,p.\ bijective(p, \{1..n\})\ \&\ \forall\, (j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(p),j)))$
S $\rightarrow$ *seq(integer)*
J $\rightarrow \lambda\ n,V.\quad range(V) \subseteq \{1 .. n\}$
Satisfies $\rightarrow \lambda\, p,V.\ \ \exists(r)(p,concat(V,r))$
$\mathbf{s_0} \rightarrow\ []$
Split $\rightarrow \lambda$ n, $V, V'.\ \exists(integer)(i \in \{1..n\}\ \&\ V' = append(V,i))$
Extract $\rightarrow \lambda\ p,V.\quad p=V$

This new specialized theory corresponds to the generator shown in Figure 3. This generator enumerates a superset of Costas Arrays. 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 more problem-specific information into the generator in order to improve efficiency.

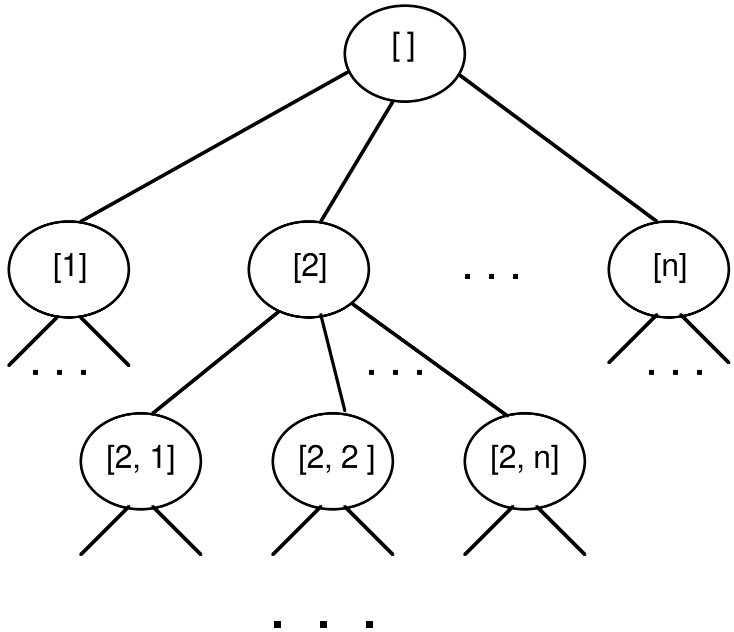

Figure 3 - Generator of sequences over the set *{1 .. n}*

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

$$
\begin{aligned}
\text{find some } (\Phi)\ (1 \le n \Rightarrow \exists (p)\ (\ \exists (r)\ (&p = concat(V,r)) \\
&\&\ bijective(p,\ \{1..n\}) \\
&\&\ \forall (j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(p),j))) \\
\Rightarrow\ &\Phi(n,V)).
\end{aligned}
$$

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 Costas Array constraints. The derivations proceed as follows.

bijective(p, {1..n})

= by definition of *bijective*

injective(p, {1..n}) & range(p) = {1 .. n}

$\Rightarrow$ applying $p = concat(V, r)$ to the first conjunct

$injective(concat(V, r), \{1..n\})$

$=$ distributing *injective* over *concat*

$injective(V,\{1..n\})\ \&\ injective(r,\{1..n\})\ \&\ range(V) \cap range(r)=\{\})$

$\Rightarrow$ dropping conjuncts

$injective(V, \{1..n\}).$

Also

$\forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(p),j)))$

$=$ applying $p = concat(V, r)$

$\forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(concat(V, r)),j)))$

$=$ distributing *dt* over *concat*

$\forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(map\text{-}disjoint\text{-}union(dt(V),$
$map\text{-}disjoint\text{-}union(cross\text{-}dt(V,r), dt(r))),j)))$

$=$ distributing *dtrow* over *map-disjoint-union*

$\forall(j)(j \in domain(p) \Rightarrow nodups(concat(dtrow(dt(V),j),$
$dtrow(cross\text{-}dt(V,r),j),$
$dtrow(dt(r),j))))$

$=$ distributing *nodups* over *concat*

$\forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(V),j))$
$\&\ nodups(dtrow(cross\text{-}dt(V,r),j))$
$\&\ nodups(dtrow(dt(r),j)))$

$\Rightarrow$ dropping conjuncts

$\forall(j)(j \in domain(p) \Rightarrow nodups(dtrow(dt(V),j)))$

$=$ applying $p = concat(V, r)$

$\forall(j)(j \in domain(concat(V, r)) \Rightarrow nodups(dtrow(dt(V),j)))$

$=$ distributing *domain* over *concat*

$\forall(j)(j \in (domain(V)\ union\ j\ in\ domain(r)) \Rightarrow nodups(dtrow(dt(V),j)))$

$\Rightarrow$ distributing and dropping conjuncts

$\forall(j)(j \in domain(V) \Rightarrow nodups(dtrow(dt(V),j))).$

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. We obtain

$$\forall(j)(j \in domain(V) \Rightarrow nodups(dtrow(dt(V),j))) \;\;\&\;\; injective(V, \{1..n\})$$

In words, the partial solution must itself satisfy the constraints that there are no duplicates in the partial solution and no duplicates in any row of its difference table. It is possible to automate the selection of filters using dependency tracking but we have not done so at this writing.

Finally the recursive Refine program in Figure 4 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. The first set-former in COSTAS-GS-AUX simply checks whether the partial solution *V* is a complete solution and if so returns *{V}*. The expression *reduce(union!, ...)* explores all possible extensions to the partial solution *V* that survive the filter. The program specification COSTAS initializes the backtrack search after checking that the initial partial solution *[]* survives the filter. Note that the filter is displayed as an input invariant in COSTAS-GS-AUX since it is tested prior to each call to the backtracking function.

Being produced as an instance of a program abstraction, this algorithm obviously has some inefficiencies, even though it is correct. The intent of the design tactics is to produce correct, very-high-level, well-structured algorithms. Subsequent refinement and optimization is necessary in order to realize the potential of the algorithm.

```
function COSTAS-GS-AUX (var N: integer, var V: seq(integer))
  where 1 ≤ N & range(V) ⊆ {1 .. N}
           & injective(V, {1 .. N})
           & ∀ (J)(J ∈ domain(V) ⇒ nodups(DTROW(DT(V), J)))
  returns {P | extends(P, V)
                      & ∀ (J)(J ∈ domain(P) ⇒ nodups(DTROW(DT(P), J)))
                      & bijective(P, {1 .. N})})
  = {P | ∀ (J: integer)(J ∈ domain(P) ⇒ nodups(DTROW(DT(P), J)))
           & bijective(P, {1 .. N})
           & P = V}
      union! reduce(union!,
               {COSTAS-GS-AUX(N, NEW-V) | injective(NEW-V, {1 .. N})
                                          & ∀ (J)(J ∈ domain(NEW-V) ⇒ nodups(DTROW(DT(NEW-V), J)))
                                          & ∃ (I: integer)(NEW-V = append(V, I) & I ∈ {1 .. N})})

function COSTAS (var N: integer | 1 ≤ N)
  returns {P | bijective(P, {1 .. N})
                      & ∀ (J)(J ∈ domain(P) ⇒ nodups(DTROW(DT(P), J)))}
  = if ∀ (J)(J ∈ domain([]) ⇒ nodups(DTROW(DT([]), J)))
         & injective([], {1 .. N})
      then COSTAS-GS-AUX(N, [])
     else {}
```

Figure 4: Global search algorithm for the Costas Array problem

----

4.4. Simplification

KIDS provides two expression simplifiers. The simplest and fastest, called the Context-Independent Simplifier (CI-SIMPLIFY), is a set of equations treated as left-to-right rewrite rules that are fired exhaustively until none apply. Some typical equations used as rewrite rules are

$$length([]) = 0$$

and

$$(if\ \ true\ \ then\ \ P\ else\ Q) = P.$$

We also treat the distributive laws in Costas Array theory as rewrite rules: e.g.

$$injective([],S) = true$$

and

$$injective(append(W,a),S) = (injective(W,S)\ \&\ a \in S\ \&\ a \notin range(W)).$$

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

if $\forall (J)(J \in domain([]) \Rightarrow nodups(dtrow(dt([]), j)))$
    $\&\ injective([], \{1\ ..\ n\})$
  then COSTAS-GS-AUX(n, [])
 else {}

simplifies to *COSTAS-GS-AUX(n, []).*

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

$$\{\ C(x)\ |\ x{=}e\ \&\ P(x)\ \} = \{\ C(e)\ |\ P(e)\ \}.$$

For example, this rule will replace *new_V* by *append(V,i)* everywhere in COSTAS-GS-AUX. This replacement in turn triggers the application of the laws for distributing *dt, dtrow, nodups,* and *injective* over *append*.

The result of applying CI-Simplify to the bodies of Costas and COSTAS-GS-AUX is shown in Figure 5. (For brevity we will sometimes omit or use ellipsis in place of expressions that remain unchanged after a transformation).

There are other simplification opportunities in this code. For example, notice that the predicate *injective(v, {1 .. n})* is being tested in COSTAS-GS-AUX, 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.

```
function COSTAS-GS-AUX (var N: integer, var V: seq(integer))
  where 1 ≤ N & range(V) ⊆ {1 .. N}
          & injective(V, {1 .. N})
          & ∀ (J)(J ∈ domain(V) ⇒ nodups(DTROW(DT(V), J)))
  returns ...
  = {V | ∀ (J: integer)(J ∈ domain(V) ⇒ nodups(DTROW(DT(V), J)))
            & bijective(V, {1 .. N})}
     union! reduce(union!,
              {COSTAS-GS-AUX(N, append(V, I)) |
                   I ∈ {1 .. N} & I ∉ range(V)
                 & injective(V, {1 .. N})
                 & ∀ (J)(J ∈ domain(V) ⇒ nodups(DTROW(DT(V), J)))
                 & nodups(DTROW(DT(V), 1 + size(V)))
                 & ∀ (J)(J ∈ domain(V)
                         ⇒ cross-nodups(DTROW(DT(V), J), [I - V(1 + size(V) - J) | J ≤ size(V)]))
                 & ∀ (J)(J ∈ domain(V) ⇒ nodups([I - V(1 + size(V) - J)| J ≤ size(V)]))})

function COSTAS (var N: integer | 1 ≤ N)
  returns ...
  = COSTAS-GS-AUX(N, [])
```

Figure 5. Costas-Array code after context-independent simplification

---

In applying CD-Simplify to the predicate of the first set-former in COSTAS-GS-AUX, the following inference task is setup:

find some (*simplified_wff*)

$$(1 \le N \;\&\; range(V) \subseteq \{1 .. N\} \;\&\; injective(V, \{1 .. N\}) \;\&\; \forall (J)(J \in domain(V) \Rightarrow nodups(DTROW(DT(V), J)))$$
$$\Rightarrow (\forall (J{:}\ integer)(J \in domain(V) \Rightarrow nodups(DTROW(DT(V), J))) \;\&\; bijective(V, \{1..n\}))$$
$$= simplified\_wff(V,n)).$$

The first conjunct of the source expression immediately unifies with an assumption and thus simplifies to *true*. For the second conjunct KIDS infers

$bijective(V, \{1..n\})$

= by definition of *bijective*

$injective(V, \{1..n\}) \;\&\; range(V) = \{1 .. n\}$

= matching the first conjunct with assumption

$range(V) = \{1 .. n\}$

= by definition of set equality: $(S{=}T) = (S \subseteq T \;\&\; T \subseteq S)$

$range(V) \subseteq \{1 .. n\} \;\&\; \{1 .. n\} \subseteq range(V)$

= matching the first conjunct with assumption

$\{1 .. n\} \subseteq range(V)$.

The resulting simplified expression is

$$\{1 .. n\} \subseteq range(V).$$

After applying CD-Simplify to the predicates of both set-formers in COSTAS-GS-AUX
we obtain the code in Figure 6.

```
function COSTAS-GS-AUX (N, V)
  = {V | {1 .. N} ⊆ range(V)}
    union! reduce(union!, {COSTAS-GS-AUX(N, append(V, I)) |
                            I ∈ {1 .. N} & I ∉ range(V)
                            & ∀(J)(J ∈ domain(V)
                                  ⇒ cross-nodups(DTROW(DT(V), J),[I - V(1 + size(V) - J)]))})
```

Figure 6. COSTAS-GS-AUX after context-dependent simplification

---

4.5. Partial Evaluation/Specialization

Next we notice that the call to *cross-nodups* has an argument of a restricted form -- a singleton sequence. This suggests the application of partial evaluation (Bjorner 1988). KIDS has the classic UNFOLD transformation (Burstall 1977) that replaces a function call by its definition (with arguments replacing parameters). The definition of *cross-nodups* is part of the domain theory for Costas arrays and may be found in Appendix 2. Partial evaluation proceeds by first UNFOLDing then simplifying.

UNFOLDing *cross_nodups(V, [I])*, we obtain

$$
\begin{aligned}
&\forall (J)\ (J \in domain(V) \\
&\qquad \Rightarrow \forall (I1{:}\ integer,\ J1{:}\ integer) \\
&\qquad\qquad (I1 \in domain(DTROW(DT(V), J))\ \&\ J1 \in domain([I - V(1 + size(V) - J)]) \\
&\qquad\qquad \Rightarrow DTROW(DT(V), J)(I1) \neq [I - V(1 + size(V) - J)](J1)))
\end{aligned}
$$

The following rules in the KIDS rule base

$$domain([x]) = \{1\}$$

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

$$\forall (x,y)(Q(x)\ \&\ x=e \Rightarrow P(x)) = \forall (y)(Q(e) \Rightarrow P(e)).$$

and others are used by CI-Simplify resulting in

$$\forall (J)\ (J \in domain(V) \Rightarrow I - V(1 + size(V) - J) \notin range(DTROW(DT(V), J)))$$

KIDS produces the code in Figure 7.

```
function COSTAS-GS-AUX (N, V)
  = {V | {1 .. N} ⊆ range(V)}
     union! reduce (union!, {COSTAS-GS-AUX(N, append(V, I)) |
                            I ∈ {1 .. N} & I ∉ range(V)
                            & ∀ (J) (J ∈ domain(V) ⇒ I - V(1 + size(V) - J) ∉ range(DTROW(DT(V), J)))})
```

Figure 7. COSTAS-GS-AUX after partial evaluation/specialization

---

4.6. Finite Differencing

Notice that the expression *range(V)* in Figure 7 is computed each time COSTAS-GS-AUX is invoked and that the parameter *V* changes in a regular way - with each recursive call, *V* has a single element appended to it. This suggests that we create a new variable whose value is maintained equal to *range(V)* and which allows for incremental computation - a significant speedup. This transformation is known as finite differencing (Paige 1982). 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. Abstraction of function *f(x)* 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 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 - invocation *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. 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)*.

The evolving algorithm is prepared for finite differencing by subjecting it to conditioning transformations. In this case they transform the two conjuncts

$$I \notin range(V) \;\&\; I \in \{1..n\}$$

to

$$I \in setdiff(\{1..n\},range(V)).$$

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 *pool*, and its invariant to COSTAS-GS-AUX, (2) all occurrences of the term *setdiff({1..n}, range(V))* in COSTAS-GS-AUX are replaced by *pool*, (3) appropriate arguments are created and simplified for all calls to the function COSTAS-GS-AUX. The initial call to COSTAS-GS-AUX (See Figure 5) becomes

COSTAS-GS-AUX(n, [], setdiff({1..n}, range([])))

which CI-Simplifies to

COSTAS-GS-AUX(n, [], {1..n}).

The recursive call to COSTAS-GS-AUX becomes

COSTAS-GS-AUX (n, append(V, I), setdiff({1..n}, range(append(V, I)))

which CI-Simplifies to

COSTAS-GS-AUX (n, append(V, I), pool - {I}).

The resulting code is shown in Figure 8.

```
function COSTAS-GS-AUX (N, V,POOL)
  where  POOL = set-diff({1 .. N},  range(V)) & ..
  = {V | empty(POOL)}
     union! reduce(union!,  {COSTAS-GS-AUX(N, append(V, I), POOL less I)  |
                            I ∈ POOL
                            & ∀(J)(J∈domain(V)  ⇒ I - V(1 + size(V) - J) ∉  range(DTROW(DT(V), J))) })

function COSTAS (N)
  = COSTAS-GS-AUX(N, [], {1 .. N})
```

Figure 8. Costas-Array algorithm after one finite differencing step

---

Notice how finite differencing introduces a meaningful data structure at this point. The concept of which elements of *{1..n}* have not yet been added to the partial solution *V* would naturally occur to many programmers who are developing a Costas Array 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.

We proceed as above by selecting

range(DTROW(DT(V), J))

then

1 + length(V)

for incremental maintenance (and naming them *DT-RANGE* and *VSIZE1* respectively), KIDS produces the code in Figure 9.

```
function COSTAS-GS-AUX (N, V, POOL, DT-RANGE: map(integer, set(integer)), VSIZE1 :integer)
  where  VSIZE1  = 1 + size(V)
         & DT-RANGE = {| J → DTROW(DT(V), J) |  J ∈ domain(V) |}) ..
 = {V | empty(POOL)}
    union!  reduce(union!, {COSTAS-GS-AUX(N,
                              append(V, I), POOL less I,
                              map-union({| VSIZE1  → {} |},
                                        {| J → {I - V(VSIZE1  - J)} ∪ DT-RANGE (J)  |  J ∈ domain(V) |}),
                              VSIZE1  + 1) |
                                   I ∈ POOL
                                   & ∀(J) (J ∈ domain(V) ⇒ I - V(VSIZE1  - J) ∉  DT-RANGE (J))})

function COSTAS (N)
  = COSTAS-GS-AUX(N, [], {1 .. N}, { }, 1)
```

Figure 9. Costas Array algorithm after finite differencing

4.7. Case Analysis

The COSTAS-GS-AUX algorithm is a union of two set-valued expressions. 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 COSTAS-GS-AUX we select the whole body as e and select *empty(pool)* as the case analysis predicate. After simplification we get the code in Figure 10 .

```
function COSTAS-GS-AUX (N, V, POOL, DT-RANGE , VSIZE1 )
 = if empty(POOL) then {V}
   else reduce(union!,  {COSTAS-GS-AUX(N,
                          append(V, I), POOL less I,
                          map-union({| VSIZE1  → {} |},
                                  {| J → {I - V(VSIZE1  - J)} ∪ DT-RANGE (J) | J ∈ domain(V) |}),
                          VSIZE1  + 1) |
                               I ∈ POOL
                               & ∀ (J) (J ∈ domain(V) ⇒ I - V(VSIZE1  - J) ∉  DT-RANGE(J))})
```

Figure 10 . COSTAS-GS-AUX after case analysis

4.8. 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 being 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 occurrences of an abstract type. Work on data structure selection and refinement for very-high-level languages attempts to deal with these problems (Barstow 1979, Schonberg 1981). We are currently integrating a data type refinement system (called DTRE and built by Lee Blaine) with KIDS. DTRE allows interactive specification of implementation annotations for data types in programs. It also provides machinery for stating data type refinements (as theory interpretations) and a modified compiler that handles the translation of 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 *V* 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 *V* because it entails copying *V* every time the *append* operation is performed. A better representation is to allow alternative versions of *V* to coexist and share common structure. The data structure *V* is simply a reversed sequence with a pointer to the last element. In this reversed list representation, initialization and append take constant time, and the assignment operation takes time linear in the length of *V* (by tracing upwards from the element pointed to by *V*).

Consider next the parameter *DT-RANGE* which is map from integers to sets. It is initialized to the empty set once, element membership is applied often to the component sets, and union with a singleton set is performed often on the component sets. If we can deduce bounds on the component sets, then a bit vector representation would be applicable and efficient; that is, *DT-RANGE* is implemented as an array of bit-vectors.

4.9. Results and Summary

The Costas Array algorithm produced by the global search tactic has been optimized, refined, and compiled. It is interesting to note the differences between the initial specification and the final code in Figure 10. The code contains no mention of the basic concepts of the domain theory. Instead it contains derived concepts, such as *DT-RANGE* and *POOL,* which combine concepts from the domain theory and global search theory. This illustrates the phenomenon that a good descriptive theory of a problem may be quite different from a good computational theory.

The unoptimized global search algorithm takes just under 50 minutes on a SUN-4/160 to find all 760 Costas Arrays of size 9. The final optimized version finds all 760 solutions in about 5 minutes. By manually performing the datatype refinements discussed in Section 4.8, we found the same 760 solutions 12 seconds. Incorporating a simple isomorph rejection mechanism further cut the time in half. We then hand implemented the resulting algorithm in C and were able to enumerate all 18,276 Costas Arrays of size 17 in about 6 days time. This duplicates and confirms the results of Silverman et al. (Silverman 1988) who also enumerated solutions up to order 17 (the runtime of their code was not described). [1]

---

[1] Note added after publication: In 1991 we used this C code to extend previous work and determine that there are 15096, 10240, and 6464 Costas Arrays of size 18, 19, and 20 respectively. The last result was obtained by running about a dozen SUN Sparcstations in parallel background mode over a period of about 2 months.

The derivation as presented above took place over a weeks time and most of the time was spent developing the domain theory. The actual derivation and variations of it took less than a day. For the Costas Array derivation, the user makes a total of 11 high-level decisions some of which involve subsidiary decisions. It would be easy to cut this number significantly by automatically applying CI-Simplify after every operation (this is not done at present). Each decision involves either selecting from a machine-generated menu, pointing to an expression, or typing a name into a text buffer. The high-level development operators encapsulate the firing of hundreds of low-level transformation rules. Excluding the time spent setting up the Costas Array domain theory, the total time for the derivation is about 25 minutes on a SUN-4/160.

There are several opportunities for automating the selection and application of KIDS operations. The steps of the Costas Array derivation are typical of almost all the global search algorithms that we have derived. After algorithm design the program bodies are fully simplified, partial evaluation is applied, followed by finite differencing, and data type refinement. It is conceivable that the entire Costas Array derivation could be performed automatically.

We have used KIDS to design and optimize algorithms for over fifty problems. Examples include optimal job scheduling (Smith 1988), enumerating cyclic difference sets (Smith 1989), finding graph colorings, bin packing, binary search, vertex covers, graph coloring, set covers, knapsack, traveling salesman tours, linear programming (Lowry 1987), maximal segment sum, and sorting (Smith 1985). On several occasions we have been able to perform new derivations before an audience.

5. Related Work

In addition to KIDS, a number of experimental interactive transformational systems have been developed, a few of which are mentioned below. For a survey of early systems see (Partsch 1983). Feather's ZAP system (Feather 1982) built on the basic fold/unfold method (Burstall 1977) by introducing tactics - metaprograms to control the application of basic transformations. More recently Darlington has been developing a system which provides a uniform functional and transformational programming environment (Darlington 1989). This project is also exploring the use of functional metaprogramming and high-level transformations such as function inversion and memoizing. The TI (Transformational Implementation) system at the Information Sciences Institute (Balzer 1981) had a large library of transformations for implementing the GIST specification language. The GLITTER system (Fickas 1985) was built on TI to provide a higher-level of transformational activity to the user. It used a problem-solving model where the user supplies development goals and occasionally some formal reasoning steps and manual editing. A library of methods were applied to solve goals and selection rules were used to prune and order the search. The DRACO system (Neighbors 1984) emphasizes domain-specific modeling and program transformation. LOPS (Bibel 1986) has strategies that are similar in some respects to KIDS algorithm design tactics. Their intent is to transform logic specifications into a specialized form solvable by known algorithmic methods, such as conditionals and recursion. The NuPrl system (Constable 1986) supports program construction as a by-product of the interactive development of a constructive mathematical proof. The RAPTS system (Paige 1987) emphasizes the optimization of set-theoretic programs. RAPTS achieves a high level of automation by restricting its specification language.

Other automated program development systems are described in (Barstow 1988, Lubars 1987, McCartney 1987, Steier 1989) and in the current volume. A comparative study of published algorithm derivations is given in (Steier and Anderson 1989).

6. Concluding Remarks

The final Costas Array algorithm is apparently not very complicated, however we see that it is an intricate combination of knowledge of the Costas Array 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 supports a knowledge-based approach to formal program development that is fairly natural to use. The extent to which a KIDS-like system can evolve to the breakeven point for routine programming will depend on formalizing enough programming knowledge at an appropriate level of abstraction. Properties of well-formalized programming knowledge include (1) wide applicability, (2) automatic or near-automatic application, and (3) accomplishing a significant and readily understandable design step.

KIDS is unique among systems of its kind for having been used to design, optimize, and refine dozens of programs. Applications areas have included scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, routing for VLSI, and linear programming. We have had good success in using KIDS to account for the structure of many well-known algorithms. In order to demonstrate the practicality of automated knowledge-based support for formal methods, we are working toward the goal of using KIDS for its own development.

Acknowledgements

Drs. Marc Lippmann and Ralph Wachter of the Office of Naval Research presented the Costas Array problem to me as a challenge problem for KIDS. I would like to thank Wolfgang Bibel, Mike Lowry, Mitch Lubars, and Tom Pressburger for their comments on drafts of this paper. Lee Blaine, Li-Mei Gilham, Allen Goldberg, Mike Lowry, Tom Pressburger, Xiaolei Qian, and Stephen Westfold have all contributed to the KIDS system. This research was supported in part by the Office of Naval Research under Contract N00014-87-K-0550, the Rome Air Development Center under contracts F30602-86-C-0026 and F30602-88-C-0127, and the Air Force Office of Scientific Research under Contract F49620-88-C-0033.

References

Abraido-Fandino, L. 1987.
An overview of Refine 2.0.
In *Proceedings of the Second International Symposium on Knowledge Engineering*,
Madrid, Spain, April 8-10, 1987.

Balzer, R., Cheatham, T.E., and Green, C. 1983.
Software technology in the 1990's: using a new paradigm.
IEEE Computer 16, 11 (November 1983), 39-45.

Balzer, R.M. 1981.
Transformational implementation: an example.
IEEE Transactions on Software Engineering SE-7, 1 (1981), 3-14.

Barstow, D.R. 1979.
Knowledge-Based Program Construction.
North-Holland, New York, 1979.

Barstow, D.R.1988.
Automatic Programming for Streams II: Transformational Implementation,
ICSE-10, Singapore, 1988, 439-447.

Bibel, W. and Hörnig, K. 1984.
LOPS - A System based on a Strategical Approach to Program Synthesis,
in *Automatic Program Construction Techniques,* Eds. Biermann, Guiho, and Kodratoff,
Macmillan, New York, 1984.

Bjorner, D., Ershov, A.P., and Jones, N.D., eds. 1988.
Partial Evaluation and Mixed Computation.
North-Holland, Amsterdam, 1988.

Blaine, L., Goldberg, A., Pressburger, T., Qian, X., Roberts, T., and Westfold, S. 1988.
Progress on the KBSA Performance Estimation Assistant.
also Tech. Rep.KES.U.88.11, Kestrel Institute, May 1988.

Burstall, R.M., and Darlington, J. 1977.
A transformation system for developing recursive programs.
Journal of the ACM 24, 1 (January 1977), 44-67.

Constable, R.L. 1986.
Implementing Mathematics with the NuPrl Proof Development System.
Prentice-Hall, New York, 1986.

Costas, J. 1984.
A study of a class of detection waveforms having nearly ideal range - doppler amibiuity properties.
Proceedings of the IEEE (1984), pp.996-1009.

Darlington, J.D. et al. 1989.
A functional programming environment supporting execution, partial execution and transformation.
In *PARLE 89: Parallel Architectures \& Languages Europe, Vol. I: Parallel Architectures*,
E.Odijk, M.Rem, and J.Syre, Eds., Springer-Verlag, New York, 1989, pp.286-305.
Lecture Notes in Computer Science, Vol. 365.

Feather, M. 1982.
A system for transformationally deriving programs.
ACM Transactions on Programming Languages and Systems 4, 1(January 1982), 1-21.

1985.
Fickas, S.F.
Automating the transformational development of software.
IEEE Transactions on Software Engineering SE-11, 11 (November 1985), 1268-1278.

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

Goldberg, A. 1989.
Reusing Software Developments.
Tech. Rep., Kestrel Institute, July 1989.
in *Proceedings of the KBSA Workshop*, Rome Air Development Center, Utica, NY, August 1989.

Gordon, M.J., Milner, A.J., and Wadsworth, C.P. 1979.
Edinburgh LCF: A Mechanised Logic of Computation.
Springer-Verlag, Berlin, 1979.
Lecture Notes in Computer Science, Vol. 78.

Guttag, J., and Horning, J. 1986.
Report on the larch shared language.
Science of Computer Programming 6, 2 (1986), 103-157.

Lowry, M.R. 1987.
Algorithm synthesis through problem reformulation.
In *Proceedings of the 1987 National Conference on Artificial Intelligence*
(Seattle, WA, July 13-17, 1987).
also Technical Report KES.U.87.10, Kestrel Institute, August 1987.

Lowry, M.R. 1989.
Algorithm Synthesis Through Problem Reformulation.
Ph.D. thesis, Computer Science Department, Stanford University, 1989.

Lubars, M., and Harandi, M. 1987.
Knowledge-based software design using design schemas.
In *Proceedings of the Ninth International Conference on Software Engineering*
(Monterey, California, 1987), pp.253-262.

McCartney, R.D. 1987.
Synthesizing algorithms with performance constraints.
In *Proceedings of the 1987 National Conference on Artificial Intelligence*
(Seattle, WA, July 13-17, 1987), pp.149-154.

Mostow, J.D. 1983.
Machine transformation of advice into a heuristic search procedure.
In *Machine Learning: An Artificial Intelligence Approach,*
R.S.Michalski, Ed., Tioga Press, Palo Alto, CA, 1983, pp.367-404.

Neighbors, J.M. 1984.
The Draco approach to constructing software from reusable components.
IEEE Transactions on Software Engineering SE-10, 5 (September 1984), 564-574.

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

Paige, R., and Henglein, F. 1987.
Mechanical Translation of Set-theoretic Problem Specifications in to Efficient RAM Code,
Journal of Symbolic Computation 4,2(1987), 207-232.

Partsch, H. 1983.
The CIP transformation system.
In *Program Transformation and Programming Environments*,
P.Pepper, Ed., Springer-Verlag, New York, 1983, pp.305-322.

Partsch, H., and Steinbrueggen, R. 1983.
Program transformation systems.
ACM Computing Surveys 15, 3 (September 1983), 199-236.

Scherlis, W. 1981.
Program improvement by internal specialization.
In *Eighth ACM Symposium on Principles of Programming Languages*
(Williamsburg, VA, January 1981), ACM, pp.41-49.

Schonberg, E., Schwartz, J., and Sharir, M. 1981.
An automatic technique for the selection of data representations in SETL programs.
ACM Transactions on Programming Languages and Systems 3, 2(April 1981), 126-143.

Silverman, J., Vickers, V., and Mooney, J. 1988.
On the number of costas arrays as a function of array size.
In *Proceedings of the IEEE* 76,7(1988), pp.851-853.

Smith, D.R. 1982.
Derived preconditions and their use in program synthesis, LNCS 138.
In *Sixth Conference on Automated Deduction* (Berlin, 1982),
D.W.Loveland, Ed., Springer-Verlag, pp.172-193.

Smith, D.R. 1985.
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.).

Smith, D.R. 1987.
Structure and Design of Global Search Algorithms.
Tech. Rep.KES.U.87.12, Kestrel Institute, November 1987.
to appear in Acta Informatica.

Smith, D.R., and Pressburger, T.T. 1988.
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).

Smith, D.R., and Lowry, M.R. 1989.
Algorithm theories and design tactics.
In *Proceedings of the International Conference on Mathematics of Program Construction*,
LNCS 375, L.van de Snepscheut, Ed., Springer-Verlag, Berlin, 1989, pp.379-398.
(extended version to appear in Science of Computer Programming).

Steier, D. 1989.
Automatic Algorithm Design within a General Architecture for Intelligence.
PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, April 1989.

Steier, D.M., and Anderson, A.P. 1989.
Algorithm Synthesis: A Comparative Study.
Springer-Verlag, New York, 1989.

Wile, D.S. 1983.
Program developments: formal explanations of implementations.
Communications of the ACM 26, 11 (November 1983), 902-911.

Appendix: Domain Theory for the Costas Array Problem

The Costas Array derivation presented in this paper depends on the definitions, specifications, and inference rules contained in the Costas Array domain theory. It also depends implicitly on theories for various datatypes (sets, sequences, integers, etc.). Building a domain theory is the first and often the hardest step in performing a derivation on KIDS. It involves defining the concepts that can be used to formally specify the problem at hand and giving inference rules for reasoning about those concepts.

The following domain theory is typical of the dozens in the KIDS library. One aspect is worth pointing out. All of the inference rules are based on distributive laws and are treated as left-to-right rewrite rules (i.e. they fire without provision for backup). Of the roughly 700 rules currently in the KIDS knowledge-base, about 80% are based on distributive laws. This is encouraging because by concentrating on deriving distributive laws (and other laws concerned with structure-preservation) before performing a derivation, we have greater confidence that the rule base is complete enough to allow a derivation to go through. It can happen that deeper laws about the problem are required in order to derive certain algorithms or perform certain optimizations. However we have been able to derive the essential structure of many well-known algorithms using the general techniques described in the paper and using mainly distributive laws.

The domain theory for the Costas Array problem is listed below. The compiler directive "rb-compile-simplification-equality" in the THEORY-RULES section tells the compiler to translate a quantified equation as a left-to-right rewrite rule.

```
THEORY COSTAS-ARRAY-THEORY

THEORY-IMPORTS {SEQUENCES-AS-MAPS}

THEORY-OPERATIONS

function costas (n:integer )
  where  1 ≤  n
  returns { p | bijective(p, {1 .. n})  & ∀ (j)(j ∈ domain(p) ⇒ nodups(dtrow(dt(p),j))) })

function dt (p:seq(integer)) :map(tuple(integer,integer),integer)
  = {| <i,j> → p(i) - p(i - j) | (i,j) i∈ domain(p) & j∈ {1 .. i - 1} |}

function cross-dt (p:seq(integer),q:seq(integer)) :map(tuple(integer,integer),integer)
  = {| <i,j> → q(i - n) - p(i - j) | (i,j,n)
             n = size(p)
           & i ∈ image(lambda(k) k + n ,domain(q))
           & j∈ {i - n .. i - 1} |}

function dtrow (d:map(tuple(integer,integer),integer), j:integer): seq(integer)
  = [ d(i,j) | (i) <i,j> ∈ domain(d) ]

function nodups (p:seq(integer)):boolean
  = ∀ (i,j)(i∈ domain(p) & j∈ domain(p) & i ≠ j ⇒ p(i) ≠ p(j))

function cross-nodups (p:seq(integer),q:seq(integer)):boolean
  = ∀ (i,j)(i∈ domain(p) & j∈ domain(q) ⇒ p(i) ≠ q(j))
```

```
%-----------------------------------------------------------------------
THEORY-LAWS

assert distribute-dt-over-emptyseq
  ∀ ()( dt([]) = {||})

assert distribute-dt-over-singleton-seq
  ∀ (a)( dt([a]) = {||})

assert distribute-dt-over-prepend
  ∀ (S,a)( dt(prepend(S,a)) = map-disjoint-union( dt(S), cross-dt(S,[a])))

assert distribute-dt-over-append
  ∀ (S,a)( dt(append(S,a)) = map-disjoint-union( dt(S), cross-dt(S,[a])))

assert distribute-dt-over-concat
  ∀ (R,S)( dt(concat(R,S))
            = map-disjoint-union( dt(R),  map-disjoint-union( cross-dt(R,S),
                                                  dom-shift(lambda(x)x+size(S), dt(S))))

assert trivial-upper-rows-of-dt
  ∀ (l,p)( size(p) ≤  1 ⇒ (dtrow(dt(p),l) = []))

assert trivial-upper-rows-of-cross-dt
  ∀ (l,p,q)( size(p) + size(q) ≤  1 ⇒ (dtrow(cross-dt(p,q),l) = []))

assert distribute-dtrow-over-emptymap
  ∀ ()( dtrow( {||}, j) = [])

assert distribute-dtrow-over-map-disjoint-union
  ∀ (d,e,j)( dtrow(map-disjoint-union(d,e),j)  = concat(dtrow(d,j), dtrow(e,j)))

assert dtrow-of-incremental-cross-dt
  ∀ (p,a,j)(dtrow(cross-dt(p,[a]),j) =  [a - p(size(p) + 1 - j) |  j ≤  size(p)])

assert distribute-nodups-over-emptyseq
  ∀ ()( nodups([]) = true)

assert distribute-nodups-over-singleton-seq
  ∀ (a)( nodups([a]) = true)

assert distribute-nodups-over-prepend
  ∀ (S,a)( nodups(prepend(S,a)) = (nodups(S) & cross-nodups(S,a)))

assert distribute-nodups-over-append
  ∀ (S,a)( nodups(append(S,a)) = (nodups(S) & cross-nodups(S,a)))
```

assert distribute-nodups-over-concat
  $\forall$ (R,S)(nodups(concat(R,S)) = (nodups(R) & cross-nodups(R,S) & nodups(S)))

assert distribute-cross-nodups-over-emptyseq1
  $\forall$ (S)(cross-nodups([],S) = true)

assert distribute-cross-nodups-over-emptyseq2
  $\forall$ (S)(cross-nodups(S,[]) = true)

%--
THEORY-RULES

function costas-rule-distribute-dt-over-emptyseq ()
  rb-compile-simplification-equality
  distribute-dt-over-emptyseq

function costas-rule-distribute-dt-over-singleton-seq ()
  rb-compile-simplification-equality
  distribute-dt-over-singleton-seq

function costas-rule-distribute-dt-over-prepend ()
  rb-compile-simplification-equality
  distribute-dt-over-prepend

function costas-rule-distribute-dt-over-append ()
  rb-compile-simplification-equality
  distribute-dt-over-append

function costas-rule-distribute-dt-over-concat ()
  rb-compile-simplification-equality
  distribute-dt-over-concat

function costas-rule-trivial-upper-rows-of-dt ()
  rb-compile-equality
  trivial-upper-rows-of-dt

function costas-rule-trivial-upper-rows-of-cross-dt ()
  rb-compile-equality
  trivial-upper-rows-of-cross-dt

function costas-rule-distribute-dtrow-over-emptymap ()
  rb-compile-simplification-equality
  distribute-dtrow-over-emptymap

function costas-rule-distribute-dtrow-over-map-disjoint-union ()
  rb-compile-simplification-equality
  distribute-dtrow-over-map-disjoint-union

function costas-rule-dtrow-of-incremental-cross-dt ()
  rb-compile-simplification-equality
  dtrow-of-incremental-cross-dt

```
function costas-rule-distribute-nodups-over-emptyseq ()
  rb-compile-simplification-equality
  distribute-nodups-over-emptyseq

function costas-rule-distribute-nodups-over-singleton-seq ()
  rb-compile-simplification-equality
  distribute-nodups-over-singleton-seq

function costas-rule-distribute-nodups-over-prepend ()
  rb-compile-simplification-equality
  distribute-nodups-over-prepend

function costas-rule-distribute-nodups-over-append ()
  rb-compile-simplification-equality
  distribute-nodups-over-append

function costas-rule-distribute-nodups-over-concat ()
  rb-compile-simplification-equality
  distribute-nodups-over-concat

function costas-rule-distribute-cross-nodups-over-emptyseq1 ()
  rb-compile-simplification-equality
  distribute-cross-nodups-over-emptyseq1

function costas-rule-distribute-cross-nodups-over-emptyseq2 ()
  rb-compile-simplification-equality
  distribute-cross-nodups-over-emptyseq2

%------------------------------------------------------------------------
end-theory
```