to appear in IEEE Transactions on Software Engineering Special Issue on Formal Methods, September 1990

KIDS: A Semi-Automatic Program Development System

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304
4 February 1994

Abstract

The Kestrel Interactive Development System (KIDS) provides automated support for the development of correct and efficient programs from formal specifications. The system has components for performing algorithm design, deductive inference, program simplification, partial evaluation, finite differencing optimizations, data type refinement, compilation, and other development operations. Although their application is interactive, 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 system and we believe that KIDS could be developed to the point that it becomes economical to use for routine programming. To illustrate the use of KIDS, we trace the derivation of an algorithm for enumerating solutions to the k-queens problem. The initial algorithm that KIDS designs takes about sixty minutes on a SUN-4/110 to find all 92 solutions to the 8-queens problem instance. The final optimized version finds the same solutions in under one second.

Index Terms – formal specifications, program transformation, automated programming, automated deduction, algorithm design, program optimization, data type refinement.

Contents

List of Figures

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 (c.f. [2]). 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 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.

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. Our intent is to provide a base level of program transformations that can be composed via a tactic or metaprogramming language to yield higher-level or domain-specific transformations [19, 42, 18].

After briefly discussing the environment and inference system underlying KIDS, we step through the derivation of a program for enumerating all solutions to the k-queens problem. The steps are as follows. First we build up a domain theory in order to state and reason about the queens problem. Then, a well-structured but inefficient backtrack algorithm [37] is created that works by extending partial solutions. To improve efficiency we apply simplification and partial evaluation [6] (specialization [32]) operations. We also perform finite differencing [29] which results in the incremental maintenance of data structures recording the rows and diagonals that are currently unoccupied by queens in the partial solution. 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 translated to executable form by a conventional compiler.

The initial algorithm that KIDS designs takes about sixty minutes on a SUN-4/110 to find all 92 solutions to the 8-queens problem instance. The final optimized version finds the same solutions in under one second. 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.

In Section 2 we present an overview of the usage of KIDS and discuss some of the underlying system support and its graphical interface. In Section 3 we introduce some basic concepts of language, specification format, and deductive inference. The derivation of an algorithm for enumerating solutions to the k-queens problem is in Section 4. Discussions of the mathematical foundations of the various KIDS development operations are interleaved with the derivation steps. We conclude with a critique of KIDS with respect to various characteristics of a successful automated software design system and discussion of related work.

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, SUN-4, and SPARC workstations. It is built on top of REFINE [1], a commercial knowledge-based programming environment [1]. The REFINE environment 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.

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 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 basically 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.

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

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 from a menu and applies it to a specification. Currently KIDS has tactics for simple problem reduction (reducing a specification to a library routine) [35], divide-and-conquer [35], global search (binary search, backtrack, branch-and-bound) [37], and local search (hillclimbing) [22, 23].

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 conventonal 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

In this section we present the language used in this paper. We also describe "directed inference", a generalization of first-order theorem-proving that underlies many of the development operations of KIDS.

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 ($\wedge$, $\vee$, $\neg$, $\Longrightarrow$, $\forall, \exists$) with the exception that equality (=) is used for logical equivalence.

Sets

S: $set(Nat)$	example type declaration of a set of natural numbers
$\{\ \}$	the empty set
$\{1, 2, 4, 8\}$, $\{2..5\}$	literal set former; e.g. $\{2..5\} = \{2, 3, 4, 5\}$
$\{f(\vec{x}) \mid P(\vec{x})\}$	general set former
$=, \neq, \in, \notin, \subseteq$	comparison predicates: equality; membership; improper subset
$\cup, \uplus, \cap$	union, disjoint union, intersection
$reduce(bop, S)$	reduction of the set S by the associative and commutative binary operator bop; e.g., $reduce(\cup, \{\{1, 2\}, \{2\}, \{3\}\}) = \{1, 2, 3\}$
$S\ +\ x$	element addition
$S\ -\ x$	element deletion

Sequences

A: $seq(integer)$	type declaration
$[\ ]$	the empty sequence
$empty(A)$	$A = [\ ]$
$A(i)$	the ith element of A; e.g. $[4, 5, 6](2) = 5$
$=, \neq, \in, \notin$	comparison predicates: equality; membership, e.g. $3 \in [2, 3, 5, 3]$
$domain(A)$	the set of integers between 1 and $length(A)$ inclusive
$range(A)$	same as $\{A(i) \mid i \in domain(A)\}$
$length(A)$	$length([\ ]) = 0$, $length([3, 4]) = 2$
$first(A)$	same as $A(1)$
$rest(A)$	all but the first; e.g. $rest([4, 5, 6, 4]) = [5, 6, 4]$
$append(A, x)$	insert x at end of A; e.g. $append([1, 2], 3) = [1, 2, 3]$
$concat(A, B)$	concatenate sequences A and B.

REFINE is strongly typed and has a type inference system. We will freely insert or omit type specifications for the sake of readability. Other notation will be introduced as needed.

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

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

$$\forall(x : D)\ (\ I(x)\ \Longrightarrow\ F(x)\ =\ \{z \mid O(x, z)\}). \tag{1}$$

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*. In directed inference, a *source* term (or formula) is transformed into a *target* term (or formula) bearing a specified relationship to the first [34]. 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 (slightly simplified) form

Find *some* $(Target)$ $(A\ \Longrightarrow\ (Source(x_1, \ldots, x_m)\ \longrightarrow\ Target(y_1, \ldots, y_n)))$

where A is a conjunction of assumptions, $Source$ is the "source" term (or formula), and $\longrightarrow$ 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 $\{y_1, \ldots, y_n\}$ (a subset of the free variables $\{x_1, \ldots, x_m\}$ of $Source$) such that the relationship

$$Source(x_1, \ldots, x_m) \longrightarrow Target(y_1, \ldots, y_n)$$

holds under the given assumptions. Currently the inference direction can be specified to be one of the following.

forward inference	$\Longrightarrow$
backward inference	$\Longleftarrow$
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 over 500 rules for reasoning about REFINE program expressions. The rules have the general form of conditional rewrite rules:

$$C \implies (s \longrightarrow t)$$

where $\longrightarrow$ 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 $\Longrightarrow$ 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.

Figure 1: Formulating the k-queens problem

4 Derivation of a k-Queens Algorithm

KIDS has been used to design and optimize global search algorithms for several dozen problems including search of an ordered sequence (binary search), job scheduling according to a precedence relation, graph coloring, vertex covers, set covers, knapsack, traveling salesman tours, and others [37]. We use the classic k-queens problem to illustrate KIDS since it is well-known and the derivation exercises several interesting transformations.

The eight main steps of the derivation are presented in roughly independent subsections below. The most difficult step is the design of a backtrack algorithm in Section 4.2. The reader may profitably skim this section and continue with subsequent sections which present program optimizations and refinements.

4.1 Domain Theory and Specification

Suppose that we want to find all ways to place k queens on a $k \times k$ chessboard so that no two queens are in the same row, column, or diagonal. A solution to the 4–*queens* problem is shown in Figure 1. In order to specify the queens problem we must first define some queens-related concepts and develop laws for reasoning about them – that is, we need to build up a *domain theory* for the queens problem. The output can be represented as a sequence $assign$ in which $assign(i)$ is the row placement of the queen in the ith column; e.g., $[3, 1, 2, 4]$ represents a solution to the 4-queens problem where queens are placed at coordinates (1,3), (2,1), (3,2), and (4,4) (see Figure 1). The constraint that no queens can appear in the same row or column is captured in the usual notion of a bijection.

$$injective(M : seq(integer), S : set(integer)) : boolean$$
$$= range(M) \subseteq S$$
$$\wedge\ \forall(i, j)(i \in domain(M)\ \wedge\ j \in domain(M)\ \wedge\ i \neq j\ \Longrightarrow\ M(i) \neq M(j))$$

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

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. We also create special definitions for the notion that no queens can appear in the same diagonal. Each "up" diagonal on the chessboard (arbitrarily from southwest to northeast) is uniquely characterized by the difference of the row number and column number of its squares (See Figure 1). Analogously, down diagonals are characterized by the sum of row and column numbers.

$$no\_two\_queens\_per\_up\_diagonal(S : seq(integer)) : boolean$$
$$= \forall(i, j)(i \in domain(S)\ \wedge\ j \in domain(S)\ \wedge\ \ i \neq j\ \Longrightarrow\ (S(i) - i \neq S(j) - j))$$

$$no\_two\_queens\_per\_down\_diagonal(S : seq(integer)) : boolean$$
$$= \forall(i, j)(i \in domain(S)\ \wedge\ j \in domain(S)\ \wedge\ \ i \neq j\ \Longrightarrow\ (S(i) + i \neq S(j) + j))$$

Distributive laws, monotonicity laws, and other inference rules are also developed in order to reason about the specification at the level of the defined terms rather than the level of their detailed definitions. While this is not logically necessary, it allows a reasoning process that is often simpler (because inference steps have a larger, more appropriate grainsize), better structured, and more understandable.

We are currently developing a configuration of KIDS that is used to develop domain theories. A theory is comprised of a list of imported theories, a set of introduced types, new operations and their definitions, laws, and rules (c.f. [16, 20]). A hierarchic library of theories is maintained with importation as the principle 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 and monotonicity laws.

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

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

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

$$\forall(W1, W2, S)\,(injective(concat(W1, W2), S) \\ \qquad = \; (injective(W1, S) \;\wedge\; injective(W2, S) \;\wedge\; range(W1) \bigcap range(W2) = \{\}))$$

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

We can now formulate a specification for the queens problem:

function $Queens\,(k :\; integer) :\; set(seq(integer))$
  where $1 \leq k$
  returns $\{assign \mid bijective(assign,\, \{1..k\})$
    $\wedge\; no\_two\_queens\_per\_up\_diagonal(assign)$
    $\wedge\; no\_two\_queens\_per\_down\_diagonal(assign)\}.$

This formulation incorporates the constraint that exactly one queen must be placed in each column. We could start with the more abstract specification that simply asks for a map from queens to the coordinates of their placement. Although we have not tried it, KIDS should be able to derive a similar algorithm.

4.2 Algorithm Design

The next step is to develop a correct, high-level algorithm for enumerating solutions to the queens problem. KIDS has specialized tactics for creating algorithms of various kinds such as divide-and-conquer [35], local search [22, 23], and global search [37]. The latter class generalizes binary search, backtracking, branch-and-bound, constraint satisfaction, and other algorithmic paradigms.

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*) $\mathcal{G} = \langle \mathcal{B}, \hat{R}, \hat{I}, \hat{r}_0, \mathit{Satisfies}, \mathit{Split}, \mathit{Extract}, \succ \rangle$ is presented as follows:

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

Operations

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

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

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

$\hat{r}_0 : D \to \hat{R}$

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

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

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

$\succ : \hat{R} \times \hat{R} \to \mathit{boolean}$

Axioms

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

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

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

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

GS4. Well-foundedness of $\succ$

where $\mathcal{B} = \langle D, R, I, O \rangle$ constitutes a problem specification, $\hat{R}$ is the type of space descriptors, $\hat{I}$ defines legal space descriptors, $\hat{r}$ and $\hat{s}$ vary over descriptors, $\hat{r}_0(x)$ is the descriptor of the initial set of candidate solutions, $\mathit{Satisfies}(z, \hat{r})$ means that z is in the set denoted by descriptor $\hat{r}$ or that z satisfies the constraints that $\hat{r}$ represents, $\mathit{Split}(x, \hat{r}, \hat{s})$ means that $\hat{s}$ is a subspace of $\hat{r}$ with respect to input x, and $\mathit{Extract}(z, \hat{r})$ means that z is directly extractable from $\hat{r}$. Axiom GS0 asserts that the initial descriptor $\hat{r}_0(x)$ is a legal descriptor. Axiom GS1 asserts that legal descriptors split into legal descriptors 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 $\hat{r}$ — an output object z is in the set denoted by $\hat{r}$ if and only if z can be extracted after finitely many applications of *Split* to $\hat{r}$ where

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

and

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

and for all natural numbers k

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

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

Example: Enumerating sequences of bounded length

Consider the problem of enumerating sequences of bounded length m over a given finite set S. A space is a set of sequences with common prefix *part_sol* and is represented as a pair $\langle part\_sol, m\rangle$ where $length(part\_sol) \leq m$. The descriptor for the initial space is just $\langle [], m\rangle$. Splitting is performed by appending an element from S onto the end of the common prefix *part_sol*. The sequence *part_sol* 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 [38] and a concrete gs-theory (technically this correspondence is known as theory interpretation or theory morphism).

$$\begin{array}{rcl}
F & \mapsto & gs_bounded_sequences_over_finite_set \\
D & \mapsto & set(\alpha) \times integer \\
I & \mapsto & \lambda\langle S, m\rangle.\ m \in integer \;\wedge\; 0 \leq m \\
R & \mapsto & seq(\alpha) \\
O & \mapsto & \lambda\langle S, m\rangle, q.\ range(q) \subseteq S \;\wedge\; length(q) \leq m \\
\hat{R} & \mapsto & seq(\alpha) \\
\hat{I} & \mapsto & \lambda\langle S, m\rangle, part_sol.\ length(part_sol) \leq m \;\wedge\; range(part_sol) \subseteq S \\
Satisfies & \mapsto & \lambda q, part_sol.\ \exists(r)\ (q \;=\; concat(part_sol, r)) \\
\hat{r_0} & \mapsto & \lambda\langle S, m\rangle.\ [\,] \\
Split & \mapsto & \lambda\langle S, m\rangle, part_sol, part_sol'.\ length(part_sol) < m \\
 & & \qquad \wedge\ \exists(i : S)\ (part_sol' = append(part_sol, i)) \\
Extract & \mapsto & \lambda q, part_sol.\ q = part_sol
\end{array}$$

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* $\psi : D \times \hat{R} \rightarrow boolean$ 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,

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

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

$$\exists(z : R)\ (\ \mathit{Satisfies}(z, \hat{r})\ \wedge\ O(x, z)\ )\ \Longrightarrow\ \Phi(x, \hat{r}). \tag{3}$$

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 [37]. 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 4.1 *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 $\{z \mid \text{Satisfies}(z, \hat{r})\ \wedge\ O(x, z)\}$
  $=\ \{z \mid \mathit{Extract}(z, \hat{r})\ \wedge\ O(x, z)\}$
    $\cup\, \mathit{reduce}(\cup,\ \{\ F\_gs(x, \hat{s}) \mid \mathit{Split}(x, \hat{r}, \hat{s})\ \wedge\ \Phi(x, \hat{s})\}).$

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 4.1 to *gs_bounded_sequences_over_finite_set* then we would get an algorithm to generate all sequences of length at most m over the input set S. The backtrack generator of this set could be pictured as in Figure 2.

Figure 2: Generator of bounded length sequences

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

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

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

$$\forall(x : D_A)\ \forall(z : R_A)\ (I_A(x) \ \wedge \ O_A(x, z) \Rightarrow O_B(t(x), z)). \tag{5}$$

Theorem 4.2 *Let* $\mathcal{G}_B = \langle \mathcal{B}_B, \hat{R}, \hat{I}, \hat{r}_0, \text{Satisfies}, \mathit{Split}, \mathit{Extract} \rangle$ *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* $\mathcal{G}_A = \langle \mathcal{B}_A, \hat{R}\theta, \hat{I}\theta, \text{Satisfies}\theta, \hat{r}_0\theta, \mathit{Split}\theta, \mathit{Extract}\theta \rangle$ *is a global search theory.*

A proof may be found in [37].

A simplified tactic for designing global search algorithms has three 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 (4). Apply Theorem 4.2 to create a specialized global search theory $\mathcal{G}_A$.
3. Derive a necessary filter Φ via formula (3). That is, use directed inference to derive a necessary condition of formula (2) expressed over the variables $\{x, \hat{r}\}$. Apply Theorem 4.1 to create a global search program.

The tactic is sound and thus only generates correct programs. The interested reader should consult [37] 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 queens problem we select from a library a standard global search theory for enumerating sequences over a finite domain – *gs_bounded_sequences_over_finite_set*. In accord with step 2, the following inference specification is created.

$$\begin{array}{l}
set(integer) = set(\alpha) \wedge \\
\forall(k : integer)\ \exists(S : set(integer),\ m : integer)\ \forall(assign : seq(integer)) \\
\quad (bijective(assign, \{1..k\}) \\
\qquad \wedge\ no_two_queens_per_up_diagonal(assign) \\
\qquad \wedge\ no_two_queens_per_down_diagonal(assign)) \\
\quad \Rightarrow \\
\qquad range(assign) \subseteq S \ \wedge \ length(assign) \leq m).
\end{array}$$

The derivation is simple and proceeds as follows: The types are unified yielding substitution $\{\alpha \mapsto integer\}$. By forward inference from $bijective(assign, \{1..k\})$ KIDS derives $injective(assign, \{1..k\})$ and $range(assign) = \{1..k\}$, then

$range(assign) \subseteq S$

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

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

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

$true$ with substitution $\{S \mapsto \{1..k\}\}$.

From $bijective(assign, \{1..k\})$ KIDS infers $length(assign) = k$ then

$length(assign) \leq m$

$=$ % applying $length(assign) = k$

$k \leq m$

$=$ % unifying with the reflexivity law $\forall(i)(i \leq i)$

$true$ with substitution $\{ m \mapsto k \}$.

Thus, altogether the queens problem completely reduces to *gs_bounded_sequences_over_finite_set* with substitution $\{\alpha \mapsto integer, S \mapsto \{1..k\}, m \mapsto k\}$. The construction in Theorem 4.2 yields the following global search theory.

$$
\begin{array}{rcl}
F & \mapsto & queens \\
D & \mapsto & integer \\
I & \mapsto & 1 \leq k \\
R & \mapsto & set(seq(integer)) \\
O & \mapsto & \lambda k, assign.\ bijective(assign,\ \{1..k\}) \\
 & & \quad \wedge\ no\_two\_queens\_per\_up\_diagonal(assign) \\
 & & \quad \wedge\ no\_two\_queens\_per\_down\_diagonal(assign)\} \\
\hat{R} & \mapsto & seq(integer) \\
\hat{I} & \mapsto & \lambda k, part\_sol.\ length(part\_sol) \leq k\ \wedge\ range(part\_sol) \subseteq \{1..k\} \\
Satisfies & \mapsto & \lambda assign, part\_sol.\ \exists(r)(assign, concat(part\_sol, r)) \\
\hat{r_0} & \mapsto & [\,] \\
Split & \mapsto & \lambda k, part\_sol, part\_sol'.\ length(part\_sol) < k \\
 & & \quad \wedge\ \exists(i : integer)\ (i \in \{1..k\}\ \wedge\ part\_sol' = append(part\_sol, i)) \\
Extract & \mapsto & \lambda assign, part\_sol.\ assign = part\_sol
\end{array}
$$

The analogue to Figure 2 for this new specialized theory is shown in Figure 3. This generator enumerates a superset of queens solutions. Notice that, for example, the space descriptor

Figure 3: Generator of sequences over the set $\{1..k\}$

[2,2] cannot lead to any queens solutions since it denotes the placement of two queens in row 2. 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.

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

find *some* (Φ)

$$(1 \leq k \implies \exists(assign)\ (\ \exists(r)\ (assign = concat(part\_sol, r)$$
$$\wedge\ bijective(assign, \{1..k\})$$
$$\wedge\ no\_two\_queens\_per\_up\_diagonal(assign)$$
$$\wedge\ no\_two\_queens\_per\_down\_diagonal(assign))$$
$$\implies \Phi(k, part\_sol)).$$

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 *Queens* constraints. The derivations proceed as follows.

$bijective(assign, \{1..k\})$

$=$ % by definition of $bijective$

$injective(assign, \{1..k\}) \ \wedge \ range(assign) \ = \ \{1..k\}$

$\Longrightarrow$ % applying $assign = concat(part\_sol, r)$ to the first conjunct

$injective(concat(part\_sol, r), \{1..k\})$

$=$ % distributing $injective$ over $concat$

$injective(part\_sol, \{1..k\}) \ \wedge \ injective(r, \{1..k\})$
$\wedge \ range(part\_sol) \ \bigcap \ range(r) = \{\})$

$\Longrightarrow$ % dropping conjuncts

$injective(part\_sol, \{1..k\})$.

Also

$no\_two\_queens\_per\_down\_diagonal(assign)$

$=$ % applying $assign = concat(part\_sol, r)$

$no\_two\_queens\_per\_down\_diagonal(concat(part\_sol, r))$

$=$ % distributing $no\_two\_queens\_per\_down\_diagonal$ over $concat$

$no\_two\_queens\_per\_down\_diagonal(part\_sol) \ \wedge \ no\_two\_queens\_per\_down\_diagonal(r) \ \wedge$

$\Longrightarrow$ % dropping conjuncts

$no\_two\_queens\_per\_down\_diagonal(part\_sol)$.

An analogous derivation yields the necessary condition $no\_two\_queens\_per\_up\_diagonal(part\_sol)$.

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. The following choices are the strongest and most useful. (It is possible to automate this selection process using dependency tracking but we have not done so at this writing).

$no\_two\_queens\_per\_down\_diagonal(part\_sol)$
$\wedge \ no\_two\_queens\_per\_up\_diagonal(part\_sol)$
$\wedge \ injective(part\_sol, \{1..k\})$

In words, the partial solution must itself satisfy the constraints that there are no two queens per diagonal and no two queens per row. An important property of this inference task is that the system can spend as much or as little resource as desired in search of necessary conditions. The constant *true* is immediately available (though when used as a filter it prunes no branches). Spending more resource at design time typically results in stronger filters.

Finally the recursive REFINE program in Figure 4 is produced by applying Theorem 4.1. That is, the correspondence between the symbols of abstract gs-theory and concrete expressions is used to instantiate the program scheme in Theorem 4.1. Note that the filter derived above is tested prior to each call to the backtracking function *Queens_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 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.

Other derivations of backtrack algorithms for the queens problem appear in [3, 12, 43]. Related approaches to deriving backtrack algorithms are presented in [9, 26].

4.3 Simplification

KIDS provides two expression simplifiers. The simplest and fastest, called the *Context-Independent Simplier* (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 Queens theory as rewrite rules: e.g.

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

and

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

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

if $injective([\,], \{1..k\})$
$\quad \wedge\ no\_two\_queens\_per\_up\_diagonal([])$
$\quad \wedge\ no\_two\_queens\_per\_down\_diagonal([])$
then $Queens\_gs(k, [\,])$
else $\{\,\}$

$$
\begin{array}{l}
\textbf{function}\ \mathit{Queens}\,(k) \\
\quad \textbf{where}\ 1 \leq k \\
\quad \textbf{returns}\ \{\mathit{assign} \mid \mathit{bijective}(\mathit{assign},\ \{1..k\}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_up\_diagonal}(\mathit{assign}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_down\_diagonal}(\mathit{assign})\} \\
\quad =\ \textbf{if}\ \mathit{injective}([\,],\ \{1..k\}) \\
\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_up\_diagonal}([]) \\
\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_down\_diagonal}([]) \\
\qquad\ \textbf{then}\ \mathit{Queens\_gs}(k,\ [\,]) \\
\qquad\ \textbf{else}\ \{\,\} \\
\\
\quad \textbf{function}\ \mathit{Queens\_gs}\,(k,\ \mathit{part\_sol} :\ \mathit{seq}(\mathit{integer})) \\
\qquad \textbf{where}\ 1 \leq k \\
\qquad\qquad \wedge\ \mathit{range}(\mathit{part\_sol}) \subseteq \{1..k\} \\
\qquad\qquad \wedge\ \mathit{length}(\mathit{part\_sol}) \leq k \\
\qquad\qquad \wedge\ \mathit{injective}(\mathit{part\_sol}, \{1..k\}) \\
\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_up\_diagonal}(\mathit{part\_sol}) \\
\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_down\_diagonal}(\mathit{part\_sol}) \\
\qquad \textbf{returns}\ \{\mathit{assign} \mid \exists(r)\ (\mathit{assign}\ =\ \mathit{concat}(\mathit{part\_sol}, r)) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_down\_diagonal}(\mathit{assign}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_up\_diagonal}(\mathit{assign}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{bijective}(\mathit{assign},\ \{1..k\})\} \\
\quad \%\ \text{This set-former gathers the solutions extracted from each node of the search tree} \\
\qquad =\ \{\mathit{assign} \mid \mathit{no\_two\_queens\_per\_down\_diagonal}(\mathit{assign}) \\
\qquad\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_up\_diagonal}(\mathit{assign}) \\
\qquad\qquad\qquad \wedge\ \mathit{bijective}(\mathit{assign}, \{1..k\}) \\
\qquad\qquad\qquad \wedge\ \mathit{assign} = \mathit{part\_sol}\} \\
\quad \%\ \text{This reduction collects the solutions returned from child nodes of the search tree} \\
\qquad \cup\, \mathit{reduce}(\cup, \{\mathit{Queens\_gs}\ (k,\ \mathit{new\_part\_sol}) \mid \\
\qquad\qquad\qquad\qquad \mathit{no\_two\_queens\_per\_down\_diagonal}(\mathit{new\_part\_sol}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{no\_two\_queens\_per\_up\_diagonal}(\mathit{new\_part\_sol}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{injective}(\mathit{new\_part\_sol}, \{1..k\}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{length}(\mathit{part\_sol}) < k \\
\qquad\qquad\qquad\qquad \wedge\ \exists(i)\ (i \in \{1..k\}\ \wedge\ \mathit{new\_part\_sol} = \mathit{append}(\mathit{part\_sol}, i))\}
\end{array}
$$

Figure 4: Global search algorithm for the k-queens problem

function $Queens\,(k)$
where $1 \leq k$
returns ...
$= \; Queens\_gs(k,\, [\,])$

function $Queens\_gs\,(k,\, part\_sol : seq(integer))$
where $1 \leq k$
$\wedge\; range(part\_sol) \subseteq \{1..k\}$
$\wedge\; length(part\_sol) \leq k$
$\wedge\; injective(part\_sol, \{1..k\})$
$\wedge\; no\_two\_queens\_per\_up\_diagonal(part\_sol)$
$\wedge\; no\_two\_queens\_per\_down\_diagonal(part\_sol)$
returns ...
$= \; \{part\_sol \mid no\_two\_queens\_per\_down\_diagonal(part\_sol)$
$\wedge\; no\_two\_queens\_per\_up\_diagonal(part\_sol)$
$\wedge\; bijective(part\_sol, \{1..k\})\}$
$\cup\, reduce(\cup, \{Queens\_gs\,(k,\, append(part\_sol, i)) \mid$
$no\_two\_queens\_per\_down\_diagonal(part\_sol)$
$\wedge\; cross\_ntqpdd(part\_sol, [i])$
$\wedge\, no\_two\_queens\_per\_up\_diagonal(part\_sol)$
$\wedge\; cross\_ntqpud(part\_sol, [i])$
$\wedge\; injective(part\_sol, \{1..k\}) \;\wedge\; i \in \{1..k\} \;\wedge\; i \notin range(part\_sol)$
$\wedge\; length(part\_sol) < k$
$\wedge\; i \in \{1..k\}\})$.

Figure 5: Queens code after context-independent simplification

simplifies to $Queens\_gs(k,\, [\,])$.

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_part_sol* by $append(part\_sol, i)$ everywhere in $Queens\_gs$. This replacement in turn triggers the application of the laws for distributing *no_two_queens_per_up_diagonal*, *no_two_queens_per_down_diagonal*, and *injective* over *append*. These distributive laws (see Appendix 2) results in the introduction of the terms $cross\_ntqpud(part\_sol, [i])$ and $cross\_ntqpdd(part\_sol, [i])$. The result of applying CI-Simplify to the bodies of *Queens* and $Queens\_gs$ 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 in both

set-formers of $Queens\_gs$ the predicate

$$no\_two\_queens\_per\_up\_diagonal(part\_sol)$$

is being tested, 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.

In applying CD-Simplify to the predicate of the first set-former in $Queens\_gs$, the following inference task is setup:

find *some* (*simplified_wff*)
$(no\_two\_queens\_per\_down\_diagonal(part\_sol)$
$\wedge\ no\_two\_queens\_per\_up\_diagonal(part\_sol)$
$\wedge\ injective(part\_sol, \{1..k\})$
$\wedge\ length(part\_sol) \leq k$
$\wedge\ range(part\_sol) \subseteq \{1..k\}$
$\wedge\ 1 \leq k$
$\Longrightarrow\ (no\_two\_queens\_per\_down\_diagonal(part\_sol)$
$\wedge\ no\_two\_queens\_per\_up\_diagonal(part\_sol)$
$\wedge\ bijective(part\_sol, \{1..k\}))$
$=\ simplified\_wff\,(part\_sol, k))$.

The first two conjuncts of the source expression immediately unify with assumptions and thus simplify to *true*. For the third conjunct KIDS infers

$bijective(part\_sol, \{1..k\})$

$=$ % by definition of $bijective$

$injective(part\_sol, \{1..k\})\ \wedge\ range(part\_sol) = \{1..k\}$

$=$ % matching the first conjunct with assumption

$range(part\_sol) = \{1..k\}$

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

$range(part\_sol) \subseteq \{1..k\}\ \wedge\ \{1..k\} \subseteq range(part\_sol)$

$=$ % matching the first conjunct with assumption

$\{1..k\} \subseteq range(part\_sol)$

function $Queens\,(k)$
$= Queens\_gs\,(k,\,[\,])$

function $Queens\_gs\,(k,\,part\_sol)$
$= \{part\_sol \mid \{1..k\} \subseteq range(part\_sol)\}$
$\cup\, reduce(\cup,\, \{Queens\_gs\,(k,\, append(part\_sol,\, i))$
$\mid i \notin range(part\_sol) \;\wedge\; i \in \{1..k\}$
$\wedge\; length(part\_sol) < k$
$\wedge\; cross\_ntqpud(part\_sol,\, [i])$
$\wedge\; cross\_ntqpdd(part\_sol,\, [i])\})$

Figure 6: Queens code after context-dependent simplification

The resulting simplified expression is

$$\{1..k\} \subseteq range(part\_sol).$$

After applying CD-Simplify to the predicates of both set-formers in $Queens\_gs$ we obtain the code in Figure 6.

4.4 Partial Evaluation/Specialization

Next we notice that the calls to *cross_ntqpud* and *cross_ntqpdd* have arguments of a restricted form — singleton sequences. This suggests the application of partial evaluation [6] or specialization [32]. KIDS has the classic UNFOLD transformation [10] that replaces a function call by its definition (with arguments replacing parameters). Partial evaluation proceeds by first UNFOLDing then simplifying.

UNFOLDing $cross\_ntqpud(part\_sol, [i])$, we obtain

$$\forall(a,b)(a \in domain(part\_sol) \;\wedge\; b \in domain([i])$$
$$\Longrightarrow\; part\_sol(a) - a \neq [i](b) - (length(part\_sol) + b)).$$

The following rules in the KIDS rule base

$$domain([x]) \;=\; \{1\}$$
$$x \in \{a\} \;=\; (x = a)$$
$$\forall(x, y_1, \ldots, y_n)(Q(x) \;\wedge\; x = e \;\Longrightarrow\; P(x)) \;=\; \forall(y_1, \ldots, y_n)(Q(e) \;\Longrightarrow\; P(e)).$$

and others are used by CI-Simplify resulting in

function $Queens\_gs\,(k,\ part\_sol)$
$= \{part\_sol \mid \{1..k\} \subseteq range(part\_sol)\}$
$\cup\, reduce(\cup,\ \{Queens\_gs\,(k,\ append(part\_sol,\ i))$
$\mid i \notin range(part\_sol)\ \ \wedge\ \ i \in \{1..k\}$
$\wedge\ length(part\_sol) < k$
$\wedge\ \forall(a)(a \in domain(part\_sol)$
$\Longrightarrow\ part\_sol(a) - a \neq i - (length(part\_sol) + 1))$
$\wedge\ \forall(a)(a \in domain(part\_sol)$
$\Longrightarrow\ part\_sol(a) + a \neq i + (length(part\_sol) + 1))\}).$

Figure 7: Queens code after partial evaluation/specialization

$\forall(a)(a \in domain(part\_sol)$
$\Longrightarrow\ part\_sol(a) - a \neq i - (length(part\_sol) + 1)).$

After applying similar operations on $cross\_ntqpdd(part\_sol, [i])$ KIDS produces the code in Figure 7.

4.5 Finite Differencing

Notice that the expression $range(part\_sol)$ in Figure 7 is computed each time $Queens\_gs$ is invoked and that the parameter $part\_sol$ changes in a regular way. This suggests that we create a new variable whose value is maintained equal to $range(part\_sol)$ and which allows for incremental computation – a significant speedup. This transformation is known as finite differencing [29]. 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 8. 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 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 8, 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 $U\prime(E(x))$ and then $U\prime(c)$. 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 $E(x)$.

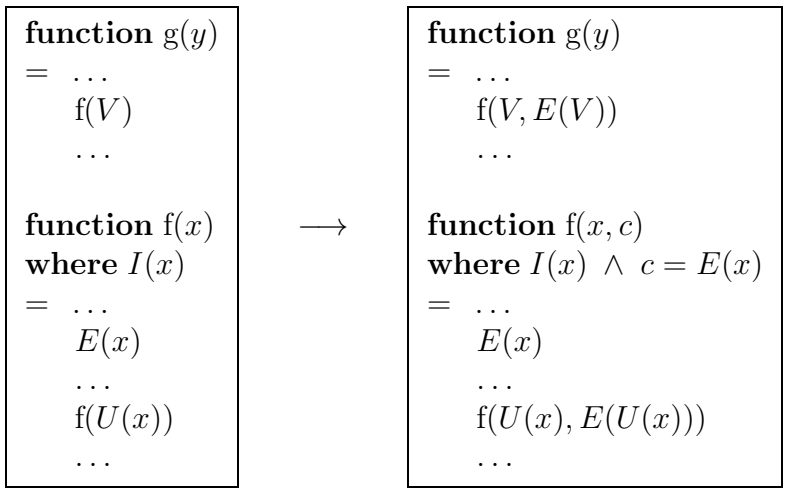

Figure 8: Abstraction operation underlying the finite differencing optimization

Our approach to finite differencing differs from that in Paige's RAPTS system [29] 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 RAINBOW II to perform simplifications whereas the RAPTS approach is specialized and more efficient. Also, Paige has analyzed various set-theoretic expressions in order to ascertain when finite differencing would result in a net improvement with respect to a simple performance model. We have also implemented a table-lookup finite differencing system and are working to integrate these two systems to get the advantages of both.

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

$$i \notin range(part\_sol) \ \wedge \ i \in \{1..k\}$$

to

$$i \in setdiff(\{1..k\}, range(part\_sol)).$$

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 *unoccupied_rows*, and its invariant to *Queens_gs*, (2) all occurrences of the term $setdiff(\{1..k\}, range(part\_sol))$ in *Queens_gs* are replaced by *unoccupied_rows*, (3) appropriate arguments are created and simplified for all calls to the function *Queens_gs*. The initial call to *Queens_gs* becomes

$$Queens\_gs(k, [\,], setdiff(\{1..k\}, range([])))$$

function $Queens\,(k)$
$= Queens\_gs(k,\ [\,],\ \{1..k\})$

function $Queens\_gs\ (k,\ part\_sol,\ unoccupied\_rows)$
where ... $\wedge$ $unoccupied\_rows\ =\ setdiff\,(\{1..k\},\ range(part\_sol))$
$= \{part\_sol \mid empty\,(unoccupied\_rows)\}$
$\cup\, reduce\,(\cup,\ \{Queens\_gs\,(k,\ append(part\_sol,\ i),$
$unoccupied\_rows\ -\ \{i\})\mid$
$i \in unoccupied\_rows$
$\wedge\ \forall(a)(a \in domain(part\_sol)$
$\Longrightarrow\ part\_sol(a) - a \neq i - (length(part\_sol) + 1))$
$\wedge\ \forall(a)(a \in domain(part\_sol)$
$\Longrightarrow\ part\_sol(a) + a \neq i + (length(part\_sol) + 1))$
$\wedge\ length(part\_sol) < k\})$

Figure 9: Queens algorithm after one finite differencing step

which CI-Simplifies to

$$Queens\_gs(k,\ [\,],\ \{1..k\}).$$

The recursive call to $Queens\_gs$ becomes

$$Queens\_gs\,(k,\ append(part\_sol,\ i),\ setdiff\,(\{1..k\}, range(append(part\_sol,\ i))))$$

which CI-Simplifies to

$$Queens\_gs\,(k,\ append(part\_sol,\ i),\ unoccupied\_rows\ -\ \{i\}).$$

The resulting code is shown in Figure 9.

Notice how finite differencing introduces a meaningful data structure at this point. The concept of which rows are currently unoccupied would naturally occur to many programmers who are developing a queens 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.

Next we condition

$$\forall(a)(a \in domain(part\_sol)\ \Longrightarrow\ part\_sol(a) - a \neq i - (length(part\_sol) + 1)).$$

for finite differencing yielding

function $Queens\,(k)$
$= Queens\_gs\,(k,\,[\,],\,\{1..k\},\,\{\,\},\,\{\,\},\,0)$

function $Queens\_gs\,(k,\,part\_sol,$
$unoccupied\_rows,$
$occupied\_up\_diagonals,$
$occupied\_down\_diagonals,$
$part\_sol\_length)$
where ...
$\wedge\ occupied\_down\_diagonals\ =\ \{part\_sol(j)+j \mid j \in domain(part\_sol)\}$
$\wedge\ occupied\_up\_diagonals\ =\ \{part\_sol(j)-j \mid j \in domain(part\_sol)\}$
$\wedge\ part\_sol\_length\ =\ length(part\_sol)$
$= \{part\_sol \mid empty(unoccupied\_rows)\}$
$\cup\, reduce\,(\cup,\,\{Queens\_gs\,(k,\,append(part\_sol,\,i),$
$unoccupied\_rows\ -\ \{i\},$
$occupied\_up\_diagonals\ \cup\ \{i-(part\_sol\_length+1)\},$
$occupied\_down\_diagonals\ \cup\ \{i+part\_sol\_length+1)\},$
$part\_sol\_length+1)$
$\mid i \in unoccupied\_rows$
$\wedge\ i-(part\_sol\_length+1) \notin occupied\_up\_diagonals$
$\wedge\ 1+part\_sol\_length+i \notin occupied\_down\_diagonals$
$\wedge\ part\_sol\_length < k\})$

Figure 10: Queens algorithm after finite differencing

$$i-(length(part\_sol)+1) \notin \{part\_sol(a)-a \mid a \in domain(part\_sol)\}.$$

After selecting

$$\{part\_sol(a)-a \mid a \in domain(part\_sol)\},$$
$$\{part\_sol(a)+a \mid a \in domain(part\_sol)\},$$

and

$$length(part\_sol)$$

for incremental maintenance (and naming them $occupied\_up\_diagonals$, $occupied\_down\_diagonals$, and $part\_sol\_length$ respectively), KIDS produces the code in Figure 10.

```
function Queens_gs (k, part_sol,
                    unoccupied_rows,
                    occupied_up_diagonals,
                    occupied_down_diagonals,
                    part_sol_length)
    = if empty(unoccupied_rows)
      then {part_sol}
      else reduce ( ∪, {Queens_gs (k, append(part_sol, i),
                                   unoccupied_rows − {i},
                                   occupied_up_diagonals ∪ {i − (part_sol_length + 1)},
                                   occupied_down_diagonals ∪ {i + part_sol_length + 1)},
                                   part_sol_length + 1)
                    | i ∈ unoccupied_rows
                     ∧ i − (part_sol_length + 1) ∉ occupied_up_diagonals
                     ∧ 1 + part_sol_length + i ∉ occupied_down_diagonals})
```

Figure 11: Queens algorithm after case analysis

4.6 Case Analysis

The Queens_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 Queens_gs we select the whole body as e and use $empty(unoccupied\_rows)$ as the case analysis predicate. After simplification we get the code in Figure 11.

4.7 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 [5, 33].

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 *part_sol* 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.

Operation	*Frequency*
$part\_sol \leftarrow [\,]$	once
$append(part\_sol, a)$	often
$solution \leftarrow part\_sol$	occasionally

A standard representation for sequences is linked lists; however, this representation is expensive for *part_sol* because it entails copying *part_sol* every time the *append* operation is performed. A better representation is shown in Figure 12 where alternative versions of *part_sol* coexist and share common structure. The data structure *part_sol* is simply a pointer to the last element of the sequence. In this reversed list representation, initialization and *append* take constant time, and the assignment operation takes time linear in the length of *part_sol* (by tracing upwards from the element pointed to by *part_sol*).

Consider next the set-valued parameter *occupied_up_diagonals*. It is initialized to the empty set once, element membership is applied often, and union with a singleton set is performed often. If we can show that the set is bounded, then a bit vector representation would be applicable and efficient. The data type refinement for bit vectors sets up the inference task of deriving upper and lower bounds. The inference specification for deriving an upper bound is as follows. As assumptions we gather all input conditions for *Queens_gs*. The goal is to derive an upper bound on the maximum value of the variable *occupied_up_diagonals* (which we replace by its definition).

Figure 12: A structure-sharing representation of sequences

$$
\begin{aligned}
\text{find } \mathit{some}\ (\mathit{ub})(1 \leq k \\
&\wedge\ \mathit{range}(\mathit{part\_sol}) \subseteq \{1..k\} \\
&\wedge\ \mathit{length}(\mathit{part\_sol}) \leq k \\
&\wedge\ \mathit{injective}(\mathit{part\_sol}, \{1..k\}) \\
&\wedge\ \mathit{no\_two\_queens\_per\_up\_diagonal}(\mathit{part\_sol}) \\
&\wedge\ \mathit{no\_two\_queens\_per\_down\_diagonal}(\mathit{part\_sol}) \\
&\wedge\ \mathit{unoccupied\_rows} = \mathit{setdiff}(\{1..k\}, \mathit{range}(\mathit{part\_sol})) \\
&\wedge\ \mathit{occupied\_down\_diagonals} = \{\mathit{part\_sol}(j) + j \mid j \in \mathit{domain}(\mathit{part\_sol})\} \\
&\wedge\ \mathit{occupied\_up\_diagonals} = \{\mathit{part\_sol}(j) - j \mid j \in \mathit{domain}(\mathit{part\_sol})\} \\
&\wedge\ \mathit{part\_sol\_length} = \mathit{length}(\mathit{part\_sol}) \\
\Longrightarrow\ & \mathit{max}\{\mathit{part\_sol}(j) - j \mid j \in \mathit{domain}(\mathit{part\_sol})\} \leq \mathit{ub}(k)).
\end{aligned}
$$

The derivation of an upper bound proceeds as follows.

$max(\{part\_sol(j) - j \mid j \in domain(part\_sol)\})$

$\leq$ % distributing max over subtraction expression in a setformer

$max(\{part\_sol(j) \mid j \in domain(part\_sol)\}) - min(\{j \mid j \in domain(part\_sol)\})$

$=$ % by definition of *domain* and *range* of a sequence

$max(range(part\_sol)) - min(domain(part\_sol))$

$\leq$ % using assumption $range(part\_sol) \subseteq \{1..k\}$ and simplifying

$max(\{1..k\}) - 1$

$=$ % applying $max(\{m..n\}) = n$ and simplifying

$k - 1$.

Analogously, RAINBOW II derives $1 - k$ as a lower bound on *occupied_up_diagonals*. Analogous bounds are derived for *occupied_down_diagonals* and *unoccupied_rows*.

The user must supply a fixed upper bound on k thus restricting the initial specification. The result of applying data type refinements is that various annotations are added to the type declarations of variables. The resulting code is shown in Figure 13. These annotations are used by the DTRE compiler to construct the detailed implementation.

4.8 Compilation

The COMPILE operation translates the program into Common Lisp and then into machine code.

4.9 Results and Summary

The Queens algorithm produced by the global search tactic has been optimized, refined, and compiled. The initial specification cannot be compiled by REFINE. The unoptimized global search algorithm with default implementation spends $O(k^2)$ time per node (disregarding the cost of unioning solutions together). The unoptimized global search algorithm takes just under 60 minutes to find all 92 solutions to the 8-queens problem instance on a SUN-4/110. The final optimized version spends O(1) time per node and finds all 92 solutions in less than 1 second.

function $Queens\,(k)$
$= Queens\_gs\,(k,\,[\,],\,\{1..k\},\,\{\,\},\,\{\,\},\,0)$

function $Queens\_gs\,(k : integer,$
$part\_sol : seq(integer)$
implemented as $reverse\_list(std\_integer)$,
$unoccupied\_rows : set(integer)$
implemented as $int\_bit\_vector(std\_integer)$,
$occupied\_up\_diagonals : set(integer)$
implemented as $int\_bit\_vector(std\_integer)$,
$occupied\_down\_diagonals : set(integer)$
implemented as $int\_bit\_vector(std\_integer)$,
$part\_sol\_length : integer)$
where ...
returns ...
$=$ ...

Figure 13: Queens algorithm after data type selection

For the queens derivation, the user makes a total of 16 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 queens domain theory, the total time for the derivation is about 15 minutes on a SUN-4/110.

There are several opportunities for automating the selection and application of KIDS operations. The steps of the queens 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 encouraging to note the degree to which Paige has automated the application of finite diferencing in RAPTS [28]. It is conceivable that the entire queens derivation could be performed automatically.

We have used KIDS to design and optimize algorithms for over fifty problems. Examples include optimal job scheduling [39], enumerating cyclic difference sets [38], finding graph colorings, bin packing, binary search, vertex covers of a graph, linear programming (derived a variant of the simplex algorithm [22]), maximal segment sum [36], and sorting [35]. On several occasions we have been able to perform new derivations before an audience.

5 Critique

To become more widely used, we believe that formal methods will require automated support. Listed below are several characteristics that we feel are important in order for a system to successfully support formal software development. We comment on the extent to which the current KIDS system does and does not meet them.

1. *Automated Support* – Formal design methods bring with them a heavy burden of detail. To be useful it must be possible to provide machine support for expressing and reasoning about this detail.

2. *Soundness* – A central tenet of formal methods is that changes preserve some specified semantics. Typically one desires that transformations preserve correctness or consistency.

3. *Coverage* – The system should be competent to generate the detailed structure of conventional software.

4. *Explanatory Power* – Derivations performed by the system should provide a unified and compelling explanation of the structure of known algorithms and data structures. Such explanation is useful both during the design process itself and for later documentation. It seems that a useful explanation must combine general concepts of programming with domain knowledge and the details of problem structure.

5. *Conceptual Coherence* – Techniques for dealing with various aspects of software design should be mutually coherent. They should share underlying concepts and tools and not require difficult translations when moving from one aspect to another during design.

6. *Effectiveness* – The application of development methods to particular problems must be effective and reasonably efficient.

7. *Quality of the designed code* – The resulting code must not only meet functional requirements, but also satisfy any constraints on resource utilization, interfaces, reliability, fault-tolerance, security, architecture, etc.

8. *Usability and the Control of Decision-making* – Given the theoretical complexity of the problem of program derivation, there is little hope of producing a fully automatic program development system in the short term – interaction with a skilled user seems to be required. Thus a successful theory of software design must carefully cater to the needs of interactive control. Users need a relatively small understandable collection of automatic tools that they can apply to do their programming tasks. Other general principles of interactive systems apply: significant decisions should be made first; users should not be forced to make too many low-level decisions; users should be able to understand the ongoing design process and be able to supply what they know when appropriate, etc.

9. *Acquisition of domain theories and specifications* – The availability of automated software development tools will result in a greater proportion of the effort in software engineering being placed on the building up of domain theories and specifications.

10. *Evolution* – The dominant cost in the software lifecycle is for error-correction and enhancement (evolution). An automated formal system should minimize or eliminate error-correction costs (see "Soundness" above), leaving evolution as the dominant activity of the future. Ultimately the effectiveness of a theory of software design will depend on the degree to which it facilitates the adaptation of software to changing human needs and changing environments.

As we have tried to indicate in this paper, KIDS provides a high level of automated support and all of its operations are sound. We believe that a derivation that results from applying general algorithmic, optimization, and data refinement concepts will be compelling to programmers and provide a satisfying explanation of the code for documentation purposes.

The conceptual coherence of KIDS derivations depends partly on the large "grain-size" of the KIDS operations and their high level of automation (effectiveness). Directed inference provides a technical unifying foundation. Term simplification is naturally performed as the search for a minimal complexity equivalent term. Finite differencing can be decomposed into an abstraction operation followed by simplification of some subterms. Partial evaluation and specialization are both decomposed into an unfold step followed by simplification. Algorithm design tactics make repeated use of directed inference – for example, the global search tactic requires the derivation of a necessary condition in order to obtain a search tree pruning mechanism. Data type refinement uses inference to check applicability conditions by deriving properties such as upper and lower bounds of sets. The coherence of this view of the various development steps stems from the common underlying set of rules (axioms) used by the inference system. Furthermore, all of these development operations mainly depend on the existence of distributive, monotonicity, and other laws concerning the preservation of structure under change.

The effectiveness of KIDS is limited by several problems. One is its reliance on general-purpose deductive inference. We have sought to postpone the combinatorial explosion by only using deductive inference in highly constrained contexts where solutions to problems can usually be found without deep inference. A KIDS derivation involves a sequence of relatively short deductions – the queens derivation is typical in this regard. T.C. Wang at Kestrel Institute is currently constructing a new directed inference system that is significantly faster than RAINBOW II. Another problem facing the KIDS user is incompleteness of the library of inference rules. Our experience has been that distributive laws comprise the bulk of the rule set. We have sought to address the problem of incompleteness by providing distributive laws for the REFINE operators and by constructing a theory development system which supports the automatic derivation of distributive and monotonicity laws. Users build up a domain theory by entering initial information into a text buffer, such as definitions of relevant concepts and laws for reasoning about them. The system has been used to automatically derive some of the needed laws, especially distributive laws. The theory development system should encourage a more systematic approach to building a pragmatically complete rule

library. Another problem is that deriving highly optimized programs can become a long and intricate process. See for example the derivation of a topological sort in [7].

The code generated by KIDS can be quite efficient and comparable to that produced manually by a competent programmer. Furthermore, the interface specification provides a complete characterization of the semantics of all data structures via the invariants. The invariants plus the derivation itself (which constitutes a proof of consistency between specification and code) provide documentation of the program.

Our experience has been that users find the interface easy to learn and enjoyable to use. The high-level operations of KIDS tend to be readily understandable with some training, although gaining a sense of their applicability conditions seems to require some practice. A history mechanism allows the user to switch to earlier design states or to switch to an entirely different development path – that is, the user is supported in exploring a tree of derivations.

KIDS is weakest in the areas of coverage and support for evolution. Much more work needs to be done in order to give KIDS enough general knowledge to support routine programming – for example, to use KIDS in developing KIDS. One of our goals has been to develop the competence in KIDS to derive algorithms from standard graduate texts on algorithm design. We feel that KIDS is well on its way towards that goal. For these kinds of algorithms it seems that KIDS is nearing the breakeven point where it becomes more economical to use the formal approach than to use manual approaches.

However, many people believe that software often has little algorithmic content, leading to concerns that a KIDS-like system would play a minor role in conventional programming. The question seems to reduce to the difference in information content between the specification and program that implements it. For the Queens problem there is a great difference which has been supplied by bringing to bear representations of domain knowledge and general programming knowledge. For some kinds of software, such as report generators, there may be little difference. Our approach has been to use an executable wide-spectrum language (an extension of REFINE) that allows users to write specifications at an appropriate level of abstraction from implementation detail. As the KIDS system becomes more robust and provides greater coverage, users will need to supply less detail. Some subset of KIDS operations will be used to transform specifications down to the level at which the compiler produces acceptably efficient code. Our intent is to demonstrate this incremental introduction of KIDS-like support through the use of KIDS for its own development.

Evolution is supported in KIDS the same as in any transformation system – one modifies the specification and then reimplements it. The specification is easier to modify than the detailed code that is finally produced by KIDS. Furthermore KIDS has a rudimentary ability to replay design decisions after a specification change [17].

6 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 [31]. Feather's ZAP system [14] built on the basic fold/unfold method [10] 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 [13]. 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 [3] had a large library of transformations for implementing the GIST specification language. The GLITTER system [15] 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 goals and selection rules were used to prune and order the search. The DRACO system [27] emphasizes domain-specific modeling and program transformation. The NuPRL system [11] supports program construction as a by-product of the interactive development of a constructive mathematical proof. The RAPTS system [28] emphasizes the optimization of set-theoretic programs. RAPTS achieves a high level of automation by restricting its specification language. The CIP project in Munich emphasizes wide-spectrum languages, algebraic specifications, and program transformation. An interactive system called CIP-S has been developed for student use [30]. Proof obligations are built up during development and must be discharged by the user. The ERGO [21] and CENTAUR [8] systems emphasize language-generic programming environments. Both use abstract syntax representations of programs and aim to provide tools to support program transformation. Other automated program development systems are described in [4, 24, 25, 40]. A comparative study of published algorithm derivations is given in [41].

7 Concluding Remarks

The final queens algorithm is apparently not very complicated, however we see that it is an intricate combination of knowledge of the *Queens* 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 formal approach to 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 support for formal methods, we are working toward the goal of using KIDS for its own development.

Acknowledgements

I would like to thank Mike Lowry 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, Personal Services Agreement No. LC90G0250F from Lockheed Corporation, and the Air Force Office of Scientific Research under Contract F49620-88-C-0033.

Appendix 1: KIDS Interface

Appendix 2: Domain Theory for the Queens Problem

References

[1] Abraido-Fandiño, L. An overview of REFINE^{TM} 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering* (Madrid, Spain, April 8–10, 1987).

[2] Balzer, R., Cheatham, T. E., and Green, C. Software technology in the 1990's: Using a new paradigm. *IEEE Computer 16*, 11 (November 1983), 39–45.

[3] Balzer, R. M. Transformational implementation: An example. *IEEE Transactions on Software Engineering SE-7*, 1 (1981), 3–14.

[4] Barstow, D. Automatic programming for streams II: Transformational implementation. In *10th International Conference on Software Engineering* (Singapore, 1988), pp. 439–447.

[5] Barstow, D. R. *Knowledge-Based Program Construction*. North-Holland, New York, 1979.

[6] Bjørner, D., Ershov, A. P., and Jones, N. D., Eds. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.

[7] 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.

[8] Borras P., et al. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Boston, Massachusetts, November 1988), ACM, pp. 14–24. (*ACM SIGPLAN Notices*, 24(2), November 1988).

[9] Broy, M., and Wirsing, M. Program development: From enumeration to backtracking. *Information Processing Letters 10*, 4 (July 1980), 193–197.

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

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

[12] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. *Structured Programming*. Academic Press, London, 1972.

[13] Darlington, J.D. et al. 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.

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

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

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

[17] Goldberg, A. Reusing software developments. In *Proceedings of the ACM SIGSOFT 4th Symposium on Software Development Environments* (Irvine, CA, December 6–8, 1990), pp. 107–119.

[18] Goldberg, A., Blaine, L., Pressburger, T., Qian, X., Roberts, T., , and Westfold, S. KBSA Performance Estimation Assistant, Final TR. Tech. Rep. RADC-TR-89-98, Rome Air Development Center, August 1989.

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

[20] Guttag, J. V., and Horning, J. J. Report on the larch shared language. *Science of Computer Programming 6*, 2 (1986), 103–157.

[21] Lee, P., Pfenning, F., Rollins, G., and Scott, D. The ERGO support system: An integrated set of tools for prototyping integrated environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (November 1988), pp. 25–34.

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

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

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

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

[26] Mostow, J. D. 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.

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

[28] Paige, R., and Henglein, F. Mechanical translation of set theoretic problem specifications into efficient RAM code – a case study. *Journal of Symbolic Computation 4*, 2 (1987), 207–232.

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

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

[31] Partsch, H., and Steinbrüggen, R. Program transformation systems. *ACM Computing Surveys 15*, 3 (September 1983), 199–236.

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

[33] Schonberg, E., Schwartz, J., and Sharir, M. 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.

[34] Smith, D. R. 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.

[35] 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.).

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

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

[38] Smith, D. R., and Lowry, M. R. 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. (reprinted in *Science of Computer Programming*, 14(2-3), October 1990, pp. 305–321).

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

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

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

[42] Wile, D. S. Program developments: Formal explanations of implementations. *Communications of the ACM 26*, 11 (November 1983), 902–911.

[43] Wirth, N. Program development by stepwise refinement. *Communications of the ACM 14*, 4 (April 1971), 221–227.