Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

Proposal for Thesis Research in Partial Fulfillment for the Degree of Doctor of Philosophy

Title: ViewForm: A Language for Building Program Transformation Systems

Submitted By: Luis H. Rodriguez Jr.
41 Ocean View Drive #44
Dorchester, MA 02125

(Signature of Author)

Date of Submission: July 26,1995

Expected Date of Completion: June 1997

Laboratory: Artificial Intelligence Laboratory

Brief Statement of the Problem:

A transformation system applies a set of rewrite rules to a source program to obtain a target program. Examples of existing transformation systems include compilers, optimizers, and systems that derive concrete programs from high-level specifications. A transformation system's behavior is determined by the semantic properties of both its rewrite rules and the way in which they are applied. One potential use for transformation systems is to construct user-customizable interactive tools for automating common programming tasks. These tasks include optimizing, reusing, generalizing, debugging, and parallelizing code. When performed manually, these tasks can be both time consuming and error prone, especially in large programs. Unfortunately, current transformation systems are not expressive, extensible, or fast enough to be used in this way. These problems arise because current transformation systems do not make adequate use of non-local program properties, they prohibit non-semantics-preserving rewrite rules, and they limit the degree of user participation in the transformation process. In return, transformation systems gain efficiency and correctness. I claim that a language for building transformation systems need not embrace these restrictions and can yet be general enough to yield efficient, safe, and user-extensible transformation systems. To demonstrate this claim, I propose to develop *ViewForm*, a language for building program transformation systems. ViewForm introduces two novel constructs; ***transforms*** and ***views***. Transforms are composable rewrite rules with independent control structures and associated ***explanations***. An explanation is a composable description of a transform's functionality. Views are composable abstractions over a program that provide access to non-local properties such as data-flow information. I present *Mini-ViewForm*, a working ViewForm prototype and a running example of its use. I also present an example dialog between a programmer and a user-extensible *Programmer's Assistant* that I will build to demonstrate my claim. The assistant's purpose is to automate complex programming tasks such as non-local program optimizations and modifications, debugging, parallelization, and verification of program properties. The assistant will be used to debug, modify, and optimize SCMUTILS (a tool for solving a rich and diverse set of mathematical expressions) in ways that would not otherwise be tenable by those unfamiliar with its implementation details.

1 Introduction

A *transformation system* is a program that applies a set of rewrite rules to a source program to obtain a target program. A *rewrite rule* or *transformation* is a mapping from source language constructs to target language constructs (e.g., $(a * b) + (a * c) \Rightarrow (a * (b + c))$). The source and target languages may or may not be the same. Examples of existing transformation systems include compilers, optimizers, debuggers, and systems that derive concrete programs from high-level specifications. The behavior of a transformation system is determined by the semantic properties of its rewrite rules and its control structure. A transformation system's *control structure* is the mechanism that determines how the rewrite rules are applied (e.g., "keep applying rules until no more rules are applicable").

One potential use for transformation systems is to construct user-customizable interactive tools for automating common programming tasks. These (currently) manual tasks can be both time consuming and error prone, especially in large programs. These tasks include building, augmenting, reusing, generalizing, debugging, and parallelizing code. Unfortunately, current transformation systems are not expressive, extensible, or fast enough to be used for this purpose.

For instance, suppose we are given the code in Figure 1-1 below, which is a Scheme [29] implementation of the puzzle `n-queens`. In this example, we would like to change the data structure returned by `n-queens` from a `list` to a `stream`*, so that the procedure will compute and return one solution before searching for any other solutions. Programmers perform data structure changes like this on a regular basis, and many times on code written by others. These kinds of changes can be tedious and error prone, even on small pieces of code. For example, to manually change the `n-queen` code a programmer must:

- learn how the code works
- perform a manual data-flow analysis on the code to find any internal call sites that must be changed
- ensure that external code can handle the change in representation

---

* A `stream` is a pair whose first element is an eagerly computed value and whose second element is a lazily computed value.

- contemplate whether the changes to the return-value data structure semantics are plausible to begin with†

It is a very helpful exercise for the reader to manually perform the prescribed change to the `n-queens` code, in order to better understand the potential difficulties involved.

```
(letrec ((make-queen cons)
         (queen-row  car)
         (queen-col  cdr)
         (consistent?
           (lambda (new-queen consistent-solution)
             (let ((queens-conflict?
                     (lambda (queen)
                       (or (= (queen-row new-queen) (queen-row queen))
                           (= (queen-col new-queen) (queen-col queen))
                           (= (abs (- (queen-row new-queen) (queen-row queen)))
                              (abs (- (queen-col new-queen) (queen-col queen)))
                              )))))
               (not (list-search-positive consistent-solution
                                          queens-conflict?)))))
         (find-new-queen-in-col
           (lambda (row col partial-solution)
             (let loop ((row row))
               (cond ((zero? row) '())
                     (#t
                      (let ((new-queen (make-queen row col)))
                        (if (consistent? new-queen partial-solution)
                            new-queen
                            (loop (-1+ row)))))))))
         (n-queens
          (lambda (n)
            (let loop ((col n)
                       (rows (list n))
                       (partial-solution '())
                       (solutions '()))
              (let* ((row (car rows))
                     (new-queen (find-new-queen-in-col row
                                                       col
                                                       partial-solution)))
                (cond ((and (= col n) (zero? row))
                       solutions)
                      ((null? new-queen)
                       (loop (1+ col)
                             (cdr rows)
                             (cdr partial-solution)
                             solutions))
                      ((= col 1)
                       (loop col
                             (cons (-1+ (queen-row new-queen)) (cdr rows))
                             partial-solution
                             (cons (cons new-queen partial-solution)
                                   solutions)))
                      (#t
                       (loop (-1+ col)
                             (cons* n
                                    (-1+ (queen-row new-queen))
                                    (cdr rows))
```

† In the `n-queens` code, there exists a procedure that accesses all of the elements of the resulting list. If the functionality of this procedure is not changed, the benefits of changing the return-value data structure may be lost.

```
                              (cons new-queen partial-solution)
                              solutions)))))))
          (display-solution
           (lambda (solution)
             (let* ((sort-by-row
                     (lambda (queen1 queen2)
                       (< (queen-row queen1) (queen-row queen2))))
                    (sorted
                     (sort solution sort-by-row))
                    (display-next-row
                     (lambda (queen)
                       (format #t "~A~%"
                                   (string-pad-right
                                    (string-pad-left "Q"
                                                     (queen-col queen)
                                                     #\-)
                                    (length sorted)
                                    #\-)))))
               (for-each display-next-row sorted))))
          (display-n-queens
            (lambda (n)
              (let* ((delimiter (make-string n #\*))
                     (solutions (n-queens n)))
                (format #t "~%~%~A~A~%" "Number of Solutions: "
                                        (length solutions))
                (for-each (lambda (soln)
                            (begin
                              (format #t "~%~A~%" delimiter)
                              (display-solution soln)
                              (format #t "~A~%" delimiter)))
                          solutions)
                solutions))))
  (display-n-queens 5))
```

Figure 1-1: n-queens

While the `n-queens` task can be characterized as a source-to-source transformation, it cannot be safely automated by current transformation systems. There are several limitations that cause this to be the case. One limitation is that current transformation systems do not typically allow changes to the semantics of the source program. In the `n-queens` example, by changing the data structure implementing the return value, we change the semantics of the source program. Another limitation is that rewrite rules cannot generally be based on data-flow information. In the `n-queens` example, we need data-flow information to find all call sites that must be changed to accommodate the change in the return-value's data structure. A third limitation is that transformation systems do not usually allow enabling conditions to be validated by user-provided assertions (since the assertions might be incorrect). In the `n-queens` example, in order to maintain the correctness of the code, the user must assert (i.e., validate the enabling condition) that external code is prepared to handle the change in the return-value data structure.

The *Programmer's Assistant* (PA) is a more involved example of a transformation system that also cannot be built using current techniques. The PA is a collection of transformations layered under a user interface. Ideally, the user communicates with the assistant via a natural language interface. The PA helps the user during program development and maintenance in tasks such as non-local optimizations, debugging, parallelization, and program verification. This is accomplished via the assistant's database of transformations. There is no need to assume that the database will be fixed: a user or even a program will be able to add more transformations via a simple interface. Such an interface will be simple because transformations in the assistant's database need not be strongly coupled. Since each view can be re-computed from scratch, the important practical aspect and difficulty in building the assistant is making it fast enough to be practical when working with large programs. An example session follows.

The session below consists of three aspects:

What the user types in,

`the Programmer's Assistant responses and comments,`

and commentary.

The natural language input interface is for expository purposes only, although the PA's design is not inconsistent with a natural language input language.

The user begins the dialog by instructing the Programmer's Assistant (PA) to load a system to work on. The system does not have to be finished code, though in this case it is. The gen-math system is a generic arithmetic package.

User: I want to work on the "gen-math" system. Please open the appropriate files.

```
PA: Working... Done.
```

We begin with a set of optimization examples that programmers typically do manually. The first optimization example is in-lining, which is an operation the PA knows how to do by default. At the start, the user is unsure where to in-line, so the user tries to in-line everywhere possible.

User: Inline the procedure matrix:* at call sites where it is invoked.

```
PA: Working...
```

The PA issues a warning because matrix: is passed as a first-class procedure (i.e., a procedural parameter), and generic dispatch makes it difficult to determine where matrix:* will actually be used at run time. The user will typically choose one of the options presented by the PA, but the user is not bound to accept any of the options.*

```
Warning: There exist data paths through which the procedure matrix:* can reach
         more than 50 call sites in the system, not including the recursive
         call sites from within matrix:*'s body.
```

```
    Options: <1> Inline matrix:* in all possible call sites
             <2> Inline matrix:* interactively
             <3> View the possible call sites
             <4> Abort current operation
```

The user can narrow the scope of an operation to some piece of the whole program in order to produce a more manageable effect. Here, the user eliminates uses where matrix: is passed as a value instead of being named explicitly.*

User: Narrow the scope of the in-line operation to uses where matrix:* is named explicitly.

This narrowed down the scope, but it still needs to be narrowed down further to avoid code bloat.

```
PA: Working...

Warning: matrix:* is called explicitly from 25 call sites in the system, not
         including the recursive call sites from within matrix:*'s body.

    Options: <1> Inline matrix:* in all call sites
             <2> Inline matrix:* interactively
             <3> View the call sites
             <4> Abort current operation
```

To narrow the scope down further, the user provides a predicate for deciding whether to in-line or not. This predicate can do something as simple as testing whether the name of the procedure in which matrix: appears is in a particular file or has a particular prefix in its name. Writing such a predicate is useful because it can then be re-used.*

User: Perform <2>, using the predicate matrix-subst? to determine whether to in-line.

```
PA: Working... Done. In-line Complete.
```

Below, the user wants to perform some simple optimizations on the in-lined procedure. Data structure references, in this case, are those that correspond to matrix: being able to accept arbitrary numbers of matrices to multiply (i.e., the matrix:* parameter list is a "dotted parameter list", also known as a restarg in Common Lisp). The PA applies some simple transformations to perform the user's request.*

User: Remove redundant, dead code, and data structure references inside the matrix:* loop.

```
PA: Working... Done.
```

In the scenario below, the user wants to substitute more efficient mathematical procedures in place of generic ones. This is because the user knows that certain input parameters will always be floating-point numbers for this particular use of the generic mathematical library. The transforms that accomplish this were written by the user.

For example, below, the user wants to substitute float:<op> for <op>, where <op> is a mathematical operator like +,, etc. The user begins by specifying the type of some input parameters given to a program at run time, and the PA traces them to the places at which an <op> is applied to them. The PA substitutes new mathematical operators at the call sites the values appear in. The PA is also responsible for modifying any arguments at those call sites that aren't floats.*

User: Load the transforms in the file "flonumize".

```
PA: Working... Done.
```

User: Assume that the arguments to the procedure make-probability-matrix are floating-point numbers, and flonumize the system.

Here, the PA finds false data paths. These false data paths can arise in several different situations. For example, they can result from the floating-point values being copied into non-homogeneous data structures, or from the values being arguments in generic procedure calls.

```
PA: Working...

 Warning: There exist data paths through which the arguments to
          make-probability-matrix may reach call sites that do not correspond
          to numerical operations. There are no matching transformations for
          these call sites.

    Options: <1> View the offending call sites
             <2> Ignore the offending call sites, and continue
             <3> Abort this operation
```

User: Perform <1>

```
PA: Working... Spawning Emacs viewer... The call sites are marked, and can be
    viewed using C-u C-<space>.
```

The user determines that the call sites are all false call sites. Note that by viewing these call sites, users may be able to find bugs in their code. Also, the programmer could have asked the PA to insert safety checks at the offending call sites to ensure correctness.

User: Perform <2>

Again, because of generic functions, the PA finds many data paths.

```
PA: Working...

    Warning: In the expression (* sum sum) inside the procedure square, there
             exist paths for which the type of sum can be any numerical type.

    Options: <1> Continue with the transformation on this expression.
             <2> View the expressions that can produce sum's value.
             <3> Convert all possible values of sum to floating-point
                 numbers, then continue with the transformation on this
                 expression.
             <4> Abort this operation
```

User: Perform <2>

```
PA: Working... Spawning Emacs viewer... The producer expressions are marked,
    and can be viewed using C-u C-<space>.
```

The user looks at the program, then decides to convert all values passed to square into floating-point numbers.

User: Perform <3>

Unfortunately, the user really doesn't want to taint the accuracy of the results with inexact numbers where exact ones are expected.

```
PA: Working...

    Warning: Some of the values to which sum can be bound are numerically
             exact. Converting them to floating-point numbers will result in
             numerically inexact results.

 Options: <1> Continue with the current operation anyway.
```

```
          <2> View the expressions that can produce sum's value.
          <3> Abort this operation
```

The user wants to minimize the amount of changes to the source code.

User: Perform <3>, and make a copy of the procedure square, call it float:square. Then, substitute a call to float:square anywhere square is called with a floating-point number as an argument. Insert run-time dispatches when necessary to maintain type safety.

```
PA: Working... Done.
```

Below, the PA implements synchronized accesses to a variable by first creating a global mutex. It then modifies all accesses to the variable so that a test and set is performed on the mutex before each variable access. A reset is performed on the mutex after the variable is accessed.

User: Load the system "transactions" and then synchronize all accesses to the global vector *process-queue*.

```
PA: Working... Done.
```

Here, the user is searching for a bug that manifests itself by causing all processes in the queue to disappear.

User: At each write to *process-queue*, insert a run-time check such that if the variable's new value satisfies the predicate null?, a debugging breakpoint is invoked. Mark these run-time checks to be deleted at a later time.

```
PA: Working...
    Warning: The predicate null? is already performed at the expression
             (if (null? *process-queue*) (loop))


 Options: <1> Insert the check anyway
          <2> Don't insert the check, and continue
          <3> Abort this operation
```

The breakpoint will already occur before this point is reached, so the user decides not to make the change.

User: Perform <2>

```
PA: Working... Done.
```

Here, the user is interested in replacing sites for which map is passed 1 list with a specialized version of map called map-1. Alternatively, the user could have requested that map be partial evaluated by a user-supplied partial evaluator.

User: Display all invocation sites of the procedure map.

```
PA: Working... Spawning Emacs viewer... The invocation-sites are marked, and
    can be viewed using C-u C-<space>.
```

User: Replace map with map-1 in all sites where map is called explicitly and with 1 list argument.

```
PA: Working... Done.
```

Here, the user wants to optimize usage of call/cc. The user knows that certain continuations created in the program are used in a catch/throw style. The compiler, however, won't be able to determine this usage in general, and may therefore allocate the continuations on the heap. The user hopes to avoid heap allocation by CPS converting the code between the catch and throw sites. This makes the continuations explicit and *makes it more likely that a reasonable compiler will realize that the explicit continuations (i.e., procedures) are stack allocable.*

An example of such a use of call/cc is in transaction systems. In such systems, a transaction may have to be aborted, and call/cc can be used to perform the non-local control transfer.

User: I want to perform CPS conversion between the points where continuations are created and where they are invoked.

In the transaction system, there are multiple uses of continuations, not all of which use the continuation in a catch/throw style.

```
PA: Working...

    Warning: Liveness analysis shows that there exist several
             continuations that may not and several that do not have
             dynamic extent.

    Options: <1> Perform full CPS conversion.
             <2> CPS convert only for continuations that may have
                 dynamic extent.
             <3> Interactively decide which continuations to consider.
             <4> Abort this operation.
```

The user has written a procedure that can identify the appropriate continuations from contextual information (e.g., by knowing that the call/cc's of interest are in a particular set of procedures or that the continuations of interest have a particular prefix in their name). This procedure is used by the PA to determine when to perform CPS conversion

User: Perform <3>, using the procedure cont-OK? as a predicate for determining which sites will be CPS converted.

```
PA: Working... CPS conversion complete.
```

1.1 A New Language for Building Transformation Systems

In order to be successful, the PA and a transformation system that can carry out the prescribed changes to the `n-queens` code must have all three of the following properties:

I. The ability to interact with the user.

II. The ability to handle transformations that alter a program's semantics.

III. The ability to perform non-local program analyses.

Current transformation systems do not have one or more of these properties for reasons including:

- maintaining adequate performance
- maintaining correctness
- preventing the user from making mistakes or accidentally introducing errors
- keeping the system simple and comprehensible

I claim that a language for building transformation systems can have all three properties (I-III) and can yet be simple, general and practical enough to yield efficient and user-extensible transformation systems. To demonstrate this claim, I propose to develop *ViewForm*, a language for building program transformation systems. ViewForm introduces two novel constructs; *transforms* and *views*. Transforms are composable rewrite rules with independent control structures and associated *explanations*. An explanation is a composable description of a transform's functionality. Views are composable abstractions over a program, and can be used to compute and access non-local program properties such as data-flow and control-flow information. I also present *Mini-ViewForm*, a working ViewForm prototype as well as a running example that automates the transformation of the `n-queens` code in Figure 1-1. ViewForm will be used to implement the Programmer's Assistant. The assistant will be used to perform the kinds of transformations in the sample session above on SCMUTILS, a tool for solving a rich and diverse set of complex mathematical expressions. The assistant will be used to optimize, debug, and modify SCMUTILS in ways that would not otherwise be tenable by those unfamiliar with its implementation details. The results of applying the assistant to SCMUTILS will be used to substantiate my claims regarding ViewForm.

In Section 2 below, I present a context and some terminology on transformation systems and I discuss the three restrictions commonly found among transformation systems. In Section 3, I discuss current and past research on transformation systems and automated programming assistants. In Section 4, I present and discuss ViewForm's design. In Section 5, I present the Mini-ViewForm prototype and a running example of its use on the `n-queens` code. In Section 6, I discuss the issues that remain to be resolved. I conclude in Section 7 with other directions for this research and I summarize in Section 8. The Appendices contain sample code, the Mini-ViewForm session on the `n-queens` example, and the Mini-ViewForm interface.

2 Transformation Systems

This section describes a vanilla transformation system and defines general terms related to transformation systems. These terms will be used throughout this proposal. This section also discusses the three properties that are needed both to build the PA and to solve the `n-queens` example.

2.1 Context and Definitions

Program transformation systems have been developed in an attempt to automate a variety of common programming tasks. These tasks include:

- Optimization of clearly written, high-level programs
- Producing concrete programs from high-level descriptions
- Verification of program properties and of mathematical proofs
- Addition of new functionality to existing code
- Debugging both sequential and parallel programs
- Discovering potential bugs (e.g., “linting” a program)
- Developing new code
- Porting Programs
- Compiling Programs
- Parallelizing Programs
- Teaching Programming Skills

A vanilla program transformation system consists of a set of *rewrite rules* (or *transformations*) and a *control structure* that controls how they are applied to a piece of source code. A rewrite rule or transformation consists of a *source pattern*, a *target pattern*, and an *enabling condition*, as shown in Figure 2-2 below. This rule implements β reduction, and is a correctness-preserving rule for an applicative-order language. A rule’s source pattern is an expression that can contain pieces of program text, meta variables (that represent arbitrary pieces of source code), and other directives that help specify the set of program expressions that the source pattern matches. In the rule below, the source pattern matches a one-parameter lambda expression that is being applied to an argument. The target pattern is also an expression that can contain pieces of program text and meta variables. It is a potential replacement for a piece of source code, once the source pattern has been unified to the source code and the meta variables have been instantiated. In the rule in Figure 2-2, the target pattern is the body of the original lambda, with the actual argument substituted for the formal parameter. The enabling condition is a predicate that must be true in order for the rule to be applicable. In general, enabling conditions do not depend on arbitrary and global program properties. In the rule in Figure 2-2, the enabling condition ensures that no naming conflicts occur and that the actual argument is free of side effects.

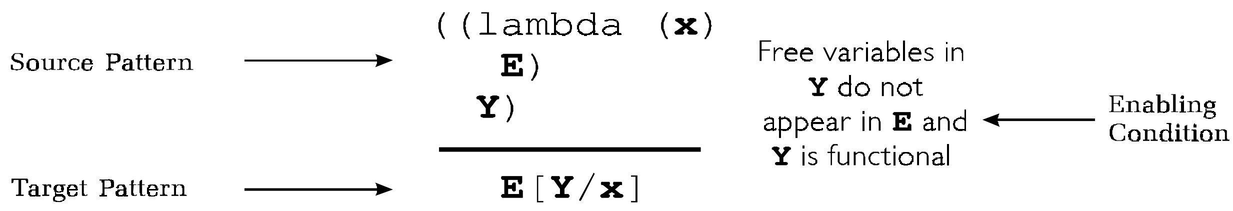

Figure 2-2: A Sample Rewrite Rule (or Transformation)

As illustrated in Figure 2-3, a vanilla transformation system's control structure takes a source program and a set of rules as input, parses the source program, transforms the source program, and then unparses and outputs a target program. During the program transformation process, the control structure scans a program for text that can be unified to some rule's source pattern. Once such a rule is found, the control structure checks the rule's enabling condition to validate its applicability. If the enabling condition is valid, the control structure instantiates the target pattern using the unification computed for the source pattern, and replaces the original piece of unified program text with the instantiated target pattern. If the enabling condition is invalid, the control structure does not use the rule to change the original piece of program text.

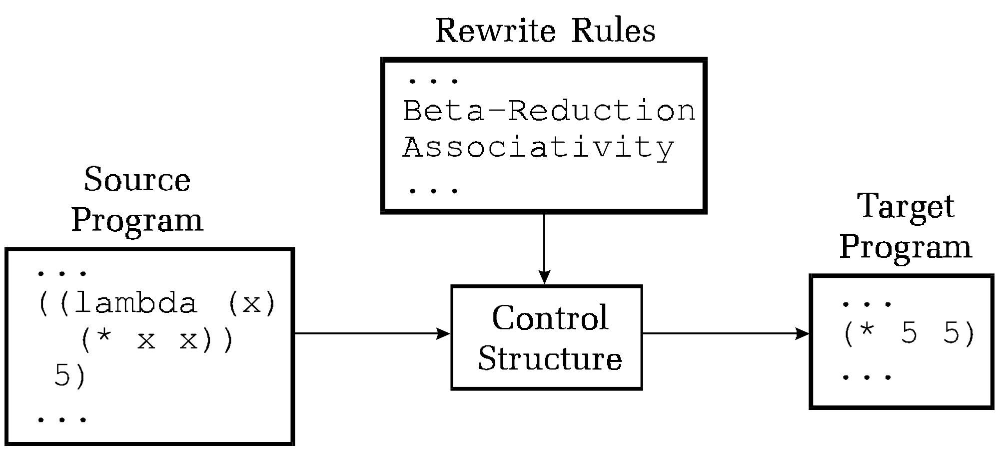

Figure 2-3: A Program Transformation System

A transformation system's control structure must make a variety of difficult decisions during the transformation process including:

- How should the source program be searched for matching source patterns?
- Which rule should be used if more than one is applicable or if source patterns overlap?
- Should rules be reapplied? Under what conditions?
- Should rules be applied recursively?
- When is the right time to stop applying rules?

- When should enabling conditions be computed and re-computed?

Some of these decisions are not practical to compute optimally and some are not even generally decidable. The control structures of current transformation systems typically address these issues in a fixed and non-extensible manner. An *extensible* control structure allows the user to control the decisions (such as those above) made by the control structure. Extensibility is an important property that allows a transformation system's behavior to be customized to meet the demands of particular users. This is important because in practice, a user or some external entity may know or may be able to determine the answer to the questions above for a specific scenario. In addition, some of the questions above can be better answered by developing systems to experiment with various approaches.

A transformation system is said to be *correctness preserving* if it produces correct target programs when given correct source programs. The criteria for whether or not a source program is correct can vary, but usually includes the condition that a program meets a set of concrete, well-understood specifications for its behavior. Furthermore, the notion of correctness preserving implicitly assumes that the source program is complete and operational, whereas programmers do not always work with complete or operational programs. Therefore, I introduce the notion of a *semantics-preserving* transformation system. A transformation system is said to be semantics preserving if it does not alter the functionality or interface of a piece of syntactically correct code. Note that a non-semantics-preserving change can be a correctness-preserving change, if correctness is based on external functionality and observations. Current transformation systems strive to be correctness preserving, do not permit observable changes to a program's functionality, but do allow changes to a program's performance.

2.2 Restrictions Found in Current Transformation Systems

Most program transformation systems are restrictive in at least one of three basic aspects. These restrictions prevent the implementation of the PA and also prevent current systems from transforming the `n-queens` example. For each aspect, I discuss why the restriction exists. I also discuss the tradeoffs involved in lifting the restrictions and I argue that the benefits derived by imposing each restriction do not outweigh the drawbacks.

2.2.1 User Interaction

The first restrictive aspect is not allowing user interaction in the transformation process. User interaction is useful and sometimes necessary in order to carry out transformations (e.g., the PA and the `n-queens` example). There are several forms of useful user interactions that I will focus on:

- determining the validity of enabling conditions
- guiding or controlling the decisions made by the control structure
- narrowing the extent of the code where a rewrite rule is applied
- providing non-local program information that is not generally statically decidable
- reducing the amount and precision of non-local program information that must be computed

One benefit to prohibiting user interaction, however, is the elimination of three undesirable effects; *inconsistent information, incomprehensible code,* and *over interaction.* Inconsistent information refers to invalid information that is used to validate enabling conditions. If a transformation system relies on the user to help determine whether an enabling condition holds, then the system's behavior will be correct only to the extent of the correctness of the user-provided information. Incomprehensible code refers to intermediate code presented to the user that is no longer readily understandable. When working with large amounts of source code, a user may be required to understand hundreds of lines of transformed code in order to interact effectively. Since transformed code can be very difficult to understand, a user will not typically work well under such conditions because of the high intermediate code complexity. Over interaction occurs when the transformation system demands more interaction than a user is willing to provide. A system that repeatedly asks the user questions about irrelevant parts of the program is over interactive.

I discuss each of these undesirable effects below. My fundamental argument in favor of user interaction is that without it, programmers must manually perform transformations that could otherwise be partially or fully automated.

2.2.1.1 Inconsistent information

User-provided information can be useful and sometimes essential. For example, a user may be able to help verify an enabling condition, such as one that requires that the arguments to an add operation in a loop always be integers between `0` and $(2^{30}-1)$. A user may also be able to provide non-local program information that cannot be statically computed. For instance, the user may know that some potential control paths discovered using static analysis may never be exercised at run time. This kind of knowledge could allow the system to infer the absence of side effects in the execution of an expression. These kinds of very specific user knowledge can be vital in optimizing a program's hot spots.

One problem with user-provided information is determining whether or not the information is consistent with the input program. If it is inconsistent, the transformation system is said to have *inconsistent information* and may produce erroneous output. For example, suppose a user is allowed to "vouch" for the validity of an enabling condition, such as whether or not the arguments to a function are always floating-point numbers. In this case, by supplying a wrong answer, a user can unintentionally compromise the integrity of the results and thus introduce a bug that could be very difficult to track down.

Since many transformation systems promise correctness-preserving behavior, they do not allow the user to validate enabling conditions with non-verifiable assertions. While prohibiting such user interaction does lead to correct information on the part of the system (in the absence of bugs!), it also limits the language of enabling conditions to one that is decidable in a reasonable amount of time. ViewForm does not limit user interaction, and instead encourages its use when appropriate.

For instance, in the `n-queens` example, we cannot correctly change the return value's `list` data structure without knowledge of all of the return value's destructors. Since the return value is returned at the top level, we cannot correctly change the return type in a non-closed world.[‡] In a non-closed world, the user must validate an enabling condition that states that all destructors of the return value must be prepared to accept a `stream` data structure. In a traditional semantics-preserving transformation system without user interaction, this kind of enabling condition is not expressible. Furthermore, by automating the changes that a programmer would otherwise perform manually, we can eliminate some common sources of errors. These errors include introducing syntax errors and overlooking constructs that must be changed. Since inconsistent information can thus be present in the absence of user interaction, inconsistent information is not a good reason to forbid user interaction. Rather, a transformation

[‡] A closed world is one where the entire program is statically available.

system that supports user interaction can help to reduce the risk of introducing inconsistent information.

2.2.1.2 Incomprehensible Code

A user can also provide beneficial guidance to a transformation system's control structure. This usually requires the user to inspect a piece of intermediate code and then to make a beneficial decision. For example, a user can pick a rewrite rule when more than one applies to a piece of intermediate code. A major problem with user guidance of the control structure is that intermediate code can be difficult to understand. This problem is the result of successive applications of rewrite rules that rename variables with unique but non-descriptive names, introduce new variables, eliminate dead code, in-line procedures, change the interface to procedures, or perform any of a number of other standard changes. These common changes can render code incomprehensible, even in a small program.

A high degree of intermediate code complexity can thwart a user's efforts to comprehend the code or to make useful control decisions. Only a few transformation systems even expect the user to see intermediate code. In some systems, the intermediate code is in a non-textual data structure (e.g., an attributed tree) that is not readily convertible to a user-comprehensible form. Furthermore, many systems strive to be completely automatic and thus have no need to render the intermediate code to the user.

Nevertheless, even in the instances when it is untenable for the user to interact directly with the transformation process, there remain opportunities for the user to interact indirectly by supplying an *agent*. An agent is a program, usually configured or written by the user, that interacts directly with the transformation system on behalf of the user. An agent may also interact with the user, if necessary. Such an agent can analyze and potentially recognize the parts of the program that the system is working with that are relevant to the user. For example, suppose a user writes Scheme code that uses continuations in a catch/throw style. In an environment where continuation creation is expensive, the cost of using continuations in a catch/throw style can be reduced by converting the code dynamically between the catch and throw to Continuation Passing Style (CPS). A transformation that performs such a conversion must be able to recognize which continuations are used in catch/throw style and which are not. This problem is not decidable in general. Fortunately, if the user wrote the catch/throw continuations in an automatically recognizable style (e.g., by distinctively naming them), the user can write an agent that recognizes them. This agent can then direct the transformation system's control structure to use the CPS rewrite rule when it encounters a catch/throw-style continuation.

Unfortunately, most current systems do not support this type of indirect interaction via user-supplied agents. ViewForm will support interaction with agents, to allow users to write programs that can make salient the relevant or interesting properties of pieces of intermediate code. These properties can then be presented to the user in a more comprehensible way.

2.2.1.3 Over Interaction

Two important uses for user interaction are narrowing the extent of the code where a rewrite rule should be applied and reducing the amount and precision of non-local program information that must be computed. The former is important in cases where rewrite rules need only be applied to specific parts of a program, such as a program's hot spot. The latter is important because non-local program analyses are expensive to compute. A user may only want accurate data-flow information computed among a small, fixed number of procedures, such as those that are part of a hot spot or a data abstraction. The performance of current transformation systems that use non-local program information is severely degraded due to the high cost of the applying the program analyses globally, even if global information is not needed. User interaction can help to reduce or eliminate those high costs.

A transformation system that embraces user interaction, however, may expect too much from the user. For example, because a program's hot spot is not generally statically decidable, a transformation system may ask the user whether to apply hot spot rewrite rules at many different points in the program. Also, a transformation system may repeatedly ask the user how much non-local program analysis to perform for various parts of a source program. Thus, a programmer may be reluctantly forced to provide information that will not result in veritable gains.

This problem can again be approached using agents. An agent can be programmed to give a certain standard reply, except in interesting cases such as program hot spots. For hot spots, the user can configure the agent to pass the transformation system's questions directly to the user. From this perspective, an agent is a filter for the questions that the transformation system asks. Only the truly interesting and relevant questions are allowed to filter through to the user.

2.2.2 Non-Local Program Information

The second restrictive aspect found in current transformation systems is that they do not make adequate use of non-local information about the program being transformed. Non-local information is required to carry out many kinds of transformations. For example, in the `n-queens` example, data-flow analysis is needed to discover which call sites need to be modified to accommodate the changes to the return-value data structure. I focus on the following kinds of non-local program information, although ViewForm will not be limited to them:

- Syntactic Program Properties
  - Variable Naming
- Dynamic Program Properties
  - Data Flow
  - Control Flow
  - Side Effects
  - Liveness

There are three burdensome problems that current transformation systems avoid by forbidding non-local information: *performance*, *maintenance*, and *semantics*. Performance is the problem that non-local information can be expensive to compute, especially if it is done so globally. Maintenance is the problem of keeping the non-local information up to date, in the presence of program modifications. Since program modifications can occur at a high rate, fine-grained or incremental updates can lead to additional complexity for the transformation system implementor (and user, for user-defined non-local program analyses). Semantics is the difficulty of formalizing the non-local information, especially if it is not globally computed or in a non-closed world. Such formalizations are used to prove a transformation's correctness; a property that is very important for some users.

Despite the potential problems associated with non-local information, it is still essential for many kinds of transformations. Programmers regularly make changes that can be classified as source-to-source transformations and that require manual computation of non-local program properties. For example, a user who would like to add an extra parameter to a first-class procedure must at least determine all of the procedure's call sites. A programmer who doesn't make an accurate determination of the call sites can introduce bugs into his or her code. If the region of the program from which the first-class procedure can be called is small, the cost of performing the analysis can be reduced. In addition, the time cost of such a non-local analysis can be less than both the time cost for a programmer to manually do the same analysis *and* the additional time cost for finding and fixing bugs introduced as a result of a programmer error in the manually computed analysis. Automatically providing non-local program information to the transformation system allows the automation of many common, time consuming, error prone, and mundane programming tasks.

It is worthy to note that optimizing compilers are the best example of transformation systems that use non-local information. Optimizing compilers, however, are themselves optimized to reduce the expense of computing and maintaining non-local information. In addition, it is expected that optimizations used by the compilers have been manually verified for correctness. Compilers are not generally extensible, however, and they are correctness preserving. This makes them unlikely candidates for solving the `n-queens` example.

2.2.3 Non-Semantics-Preserving Transformations

The third restrictive aspect is that many transformation systems do not allow non-semantics-preserving transformations. Non-semantics-preserving transformations are essential for a variety of purposes. For instance, the `n-queens` example involves changing the semantics of the data structure returned by a procedure. Some important purposes for non-semantics-preserving transformations include:

- Modification of Existing Programs
  - Optimizing programs
  - Customizing programs
  - Adding new functionality
- Developing Software
  - Debugging
  - Altering interfaces
  - Changing data structures
  - Generalizing functionality
  - Introducing data and procedural abstractions

On the other hand, a system that does not use non-semantics-preserving transformations has the potent benefit that under most circumstances,[§] it will be correctness-preserving. Preservation of correctness is useful in certain cases, such as when the user is interested only in safe optimizations. Even in the optimization case, however, correctness-preserving rules can be limiting. This is because some optimizations are not decidably correctness-preserving. For instance, programmers may want to replace generic mathematical operations with their type-specific and more efficient analogs. In a system that requires transformations to be correctness preserving, such optimizations are often not possible without user-provided assertions.

Preservation of correctness or semantics is also a hindrance in other situations, such as during program development, maintenance, and debugging. One example is adding an extra parameter to a first-class procedure, as discussed in Section 2.2.2. Another example is converting a given set of callers and callees from call by value to call by need to introduce lazily evaluated data structures. Such a rule can be used to optimize programs so that values that are rarely used are computed on demand. A third example is a rule that inserts debugging code into a program for tracking reads and writes to a variable. While these rules are certainly not semantics preserving, they correspond to useful and common transformations that are not expressible in correctness-preserving transformation systems. ViewForm, on the other hand, will allow the expression of non-semantics-preserving transformations.

One argument against non-semantics-preserving transformations is that they make it easier for programmers to introduce bugs into otherwise correct code. Nevertheless, many programmers already resort to manually making non-semantics-preserving changes, typically with a character-based text editor. By using a transformation system to partially or fully automate non-semantics-preserving changes, programmers can continue to operate as usual, but without unnecessarily taking the risk of making any of a variety of common mistakes.

2.3 ViewForm

ViewForm will not fundamentally impose any of the three common limitations discussed above. The benefits of removing the limitations are synergistic in three important ways:

[§] Not all semantics-preserving rewrite rules (or even purported correctness-preserving rules) will preserve correctness. For example, it is easy to write a rewrite rule that bloats the amount of code in a real-time program to the point that the program no longer adheres to its real-time constraints. Such code-bloat is a common problem in partial evaluation.

- User interaction enables the cost of providing non-local program analyses to be reduced.
- Non-local program analysis enables the automation of a wide variety of common programming tasks using non-semantics-preserving transformations.
- Non-semantics-preserving transformations enable the validation of enabling conditions via user interaction.

At the same time, building transformation systems with ViewForm will remain simple, structured, and practical. ViewForm will support the construction of higher-level sets of transformations that may be less general and more restrictive for circumstances when the user would gain some benefit from the restrictions. ViewForm will provide the ability to construct, abstract, combine, explain, undo, and externally guide transformations.

3 Related Work

Since ViewForm is a language for building transformation systems, there exists a large body of research that is partially related to ViewForm. We discuss the more closely related work below.

3.1 Views

In general, *views* refer to different ways of presenting the same information. For example, a point in two dimensional space can be represented in Cartesian or polar coordinates. Each of these representations corresponds to a different view on the point.

In this section, I describe systems that use or provide views. ViewForm provides its own kind of views as a vehicle for computing, updating, accessing, and abstracting non-local program information. The systems below demonstrate the utility and variety of views in current transformation systems and programming environments. In addition, the work below provides examples of the kinds of views that are useful in practice.

3.1.1 The View Oriented Model

The View Oriented Model (VOM) [23] is the design of an environment of cooperating tools that can share data via a collection of views. Views in this system are different interfaces to data shared between tools. One of the main thrusts of this work is to show how views can be used to share data among independently written tools. Consistency is maintained by procedures associated with specific views and data. These procedures are activated when the data is modified. The system is implemented in a strongly-typed, object-oriented style.

VOM treats the data in its environment as a database made up of data-representation descriptions that are provided by the tool implementors. The descriptions are specified as combinations of a base set of primitive data types provided by the VOM. The descriptions are used to statically and automatically generate an implementation of the database and its concrete data representations. New views can be added to the database if necessary.

VOM is restricted by a limitation in the data types that can be combined to form objects with views. In later work [27], this limitation is overcome. On the other hand, maintaining consistency between arbitrary views in a transformation system is not practical for at least two reasons. The first reason is that local changes in one view can engender global changes in others, and computing these changes can be computationally intensive. The second reason is that transformations may not need fine-grained consistency between all active views (if any), in which case maintaining consistency leads to wasteful computation. In addition, maintaining consistent multiple views can lead to incomprehensible code. This can happen, for example, if pervasive and complex changes are made to the data-flow view and are then automatically propagated back to the text view.

3.1.2 PECAN

PECAN [48] is a programming environment that renders multiple views of a program for a user. The objective of PECAN is to show that modern computational power and graphics technology are capable of providing multiple views of a program in development that are updated consistently in real time. PECAN supports three kinds of views; *syntactic*, *semantic*, and *execution*. All of the concrete views are rendered graphically to the user of the PECAN programming environment.

PECAN's syntactic view is rendered through an editor that supports a mix of a traditional character-based view and a structure-oriented view. Four of the supported semantic views include the symbol table view, the data-type view, the expression view, and the flow view. The symbol table view is a hierarchical representation of the environment structure from the point of view of a particular expression. The data-type view is a representation of a data type being edited by the user. The expression view is a representation of the abstract data tree for an expression. The flow view displays control-flow information on the program in a form based on Nassi-Shneiderman flow charts. Other views that were not implemented are a data-flow view, a view that focuses on module-level abstraction, and a declaration view.

The third kind of view in PECAN is the execution view. Execution views are representations of the execution of a program. The two execution views supported by PECAN are the feedback and stack views. The former shows the statement being executed and the latter shows the execution stack as a series of stack frames.

When users make changes in some of the views, PECAN automatically propagates the changes to other appropriate views. The default views implemented in PECAN can all be updated incrementally, thereby making the programming environment more responsive. In addition, PECAN is flexible enough to allow new views to be added in a modular way. PECAN's views are designed to be graphically rendered and efficiently updatable. PECAN also depends on the existence of a method for computing the incremental changes on all appropriate views given a small change to each specific view. These dependencies and requirements are non trivial to comply with for arbitrary non-local program information. In addition, compliance isn't always useful or necessary.

3.1.3 Semantic Program Graphs

A Semantic Program Graph (SPG) [40] is a canonical program representation that allows various views of a program to be easily derived and incrementally modified when a change is made in one of the views. The purpose of the SPG design is to show how a general program representation can also be used to accommodate views. SPG's are designed to allow the representation of different language paradigms (imperative, functional, and data flow), parallel constructs, non-deterministic computation, strict and non-strict parameter passing, control-flow and data-flow information, and naming scopes. In an SPG, nodes represent operations and edges represent paths along which tokens can flow. Edges can have fan in or fan out, and tokens can represent data or control flow. A predicate is associated with each edge to regulate token flow.

Nodes in an SPG can also be treated as black boxes that perform some arbitrary computation. This property is essential for representing programs in languages such as Prolog [25,57], that can't be easily represented using the default notion of a node. Annotations can also be attached to nodes to record arbitrary properties [47]. Nodes can be grouped to more easily allow annotations to refer to sets of nodes.

SPG's are designed to allow the computation of control-flow views, data-flow views, execution views, and "abstract views of system structures"**. In addition, SPG's are designed to allow views to be automatically updated every time a change is made to one of them. ViewForm is likely to benefit from an internal representation that can represent as wide a diversity of programming languages as SPG's. ViewForm's internal representation, however, must remain convertible into a form that is as comprehensible as possible to users.

3.1.4 Documenting Programs Through Views

The Chapter and Verse Program Description [58] is a method of documenting a program from a variety of views. The views are organized hierarchically and vertically, from a high-level overview of the program's purpose to the lower-level implementation details. Unlike ViewForm, the Chapter and Verse system is a methodology and does not provide automation. For example, it does not provide any automation for recomputing the documentation when changes are made to the program.

3.2 Special-Purpose Languages via Transformations

In several systems, special-purpose or higher-level languages are "created" by virtue of a transformation system that translates the new languages to a base concrete language. These systems assume that the source program is complete and correct, and then apply a set of correctness-preserving transformations to "implement" the source program in the target language. The source and target languages may or may not be similar. The systems described below serve as good examples of the kinds of tools that can be built using transformation systems.

3.2.1 TXL

TXL, the Turing eXtender Language [10], is a transformation system designed to easily allow new language constructs to be added to an existing base language. It has been used to provide object-oriented extensions to a language [11] and to automatically generate the code

** Abstract views of system structures are ways of viewing the overall system from a higher-level point of view.

necessary to call a C graphics library from Prolog [12]. In addition, TXL has an associated denotational semantics [38]. The TXL system consists of three parts; a parser, a transformation engine, and an unparser. The syntax of the source language is described in a notation similar to BNF. The parser takes a source program, converts it into a typed tree, and then gives it to the TXL transformation engine. The transformations are recursive, context-sensitive tree rewrite rules that can call arbitrary external procedures. A rewrite rule can only replace nodes in the subtree it is given. The engine applies a single, main rule to the tree. This rule is a composition of user-defined sub-rules. After the main rule has been applied, the unparser can be invoked to output the target program. TXL doesn't provide multiple views of the source program and it doesn't have a need for user interaction since it is a completely automatic transformation tool. It can't therefore be used to solve the `n-queens` example.

3.2.2 ASCENT

ASCENT [24] is a system for building special-purpose languages out of general-purpose languages. ASCENT assumes the existence of a base programming language and environment. The new language is built by modifying the set of production rules that describe the base language's syntax. ASCENT then partially automates the production of a set of transformation rules that map constructs from the new language to the original base language. As the program is being transformed, ASCENT associates the new, target-language program expressions with the source program expressions from which they were derived. ASCENT is not a general program transformation language; ASCENT is designed to provide a way to specify a new language that incrementally differs from an existing base language. In addition, ASCENT doesn't support user interaction or multiple views. It therefore cannot solve the `n-queens` example.

3.2.3 Proxac

Proxac [59] is a user interface for applying transformations to a source program. It displays the system-provided transformations and the program they are being applied to. The user selects the next transformation to be applied and can then see the results. While Proxac can be a useful tool for testing individual rewrite rules, it is highly susceptible to the problem of incomprehensible code. Proxac transformations are also textually local and correctness preserving. It therefore cannot be used to solve the `n-queens` example.

3.2.4 Elaborations

Elaborations [21] are a mechanism for developing programs top down, starting with a base high-level description. Elaborations are progressively lower-level descriptions (i.e., implementations) of the higher-level descriptions. Elaborations also provide a single vertical and hierarchical view of a program, and don't provide composable explanations for rewrite rules. Since elaborations are correctness preserving, they cannot be used to solve the `n-queens` example.

3.2.5 Darlington's User-Interactive Transformation System

In the transformation system described by Darlington [13], a clearly written source program is transformed into a more efficient program (in the source language) via a set of transformations. While the system is shown to successfully transform some small programs, the system relies heavily on user interaction for deciding which transformations to apply. For example, the user is shown a program that has been transformed and is asked whether it is adequate. This presents a significant difficulty to a user who may not be able to recognize whether a certain transformation has produced an effect that will lead to the desired target program. The degree of this difficulty is large for any reasonably sized program or set of transformations. This kind of difficulty arises when the objective of the transformations is to heavily optimize a program. This system suffers from both over interaction and incomprehensible code. Furthermore, since it doesn't support non-local program information, it cannot be used to solve the `n-queens` example.

3.2.6 PECOS and LIBRA

PECOS [2] is a knowledge-based system that takes a high-level specification and implements it by transforming it into a concrete program. PECOS draws transformations from a database of about 400 rules of which 75% are not directly related to the target language. When several rules are applicable, PECOS can call on LIBRA [30], an external program that estimates which of the applicable rules will lead to a more efficient program. The system can also interact with the user for assistance. This system is similar to ViewForm in that it provides a mechanism for reducing the amount of user interaction by relying on an external program designed to make reasonable decisions. PECOS, however supports neither multiple views nor arbitrary non-local program information. This makes PECOS unable to solve the `n-queens` example.

3.2.7 Cheatham's Transformation System for Reuse

Cheatham proposes a transformation system that supports the reuse of abstract programs [8]. The source programs are written in EL1, a high-level language with the means for syntactic extension. The source program is transformed into a lower-level concrete language by sets of transformations and a control structure. The transformations are pattern based, but a semantic pre-condition can be associated with each pattern. The transformation system performs a limited amount of semantic analysis of the source program. Cheatham describes two interesting forms of reuse: *Rapid Prototyping* and *Custom Tailoring*. The former involves the rapid development of a working source program without significant regard for performance. The latter is the derivation of an efficient concrete program from a piece of working source code. In this system, there is no support for views or for user interaction. It is therefore unable to solve the `n-queens` example.

3.2.8 CIP

The CIP system [3] was designed to investigate the construction of concrete programs from a formal specification via a transformation library. The source language is CIP-L, a wide-spectrum language that is extensible via schemas. The developers of CIP claim that user interaction, especially for higher-level decisions, will likely be necessary for large, non-toy applications. They do concede, however, that straightforward, mechanical, lower-level tasks could be done automatically. CIP supports only correctness-preserving transformations and CIP does not support multiple views of the source program. It can't therefore solve the `n-queens` example.

3.2.9 The *transform* Language Construct

In [26], two language constructs for implementing coordinate transformations [14] are introduced. The constructs are the *transform* for describing coordinate transforms and the *transform directive* for directing their use. These constructs are assumed to be correctness preserving and they work with variables but not values. Thus, these constructs cannot be used to solve the `n-queens` example.

3.3 *Open Implementations*

One way of building transformation systems via *open implementation* [31,45,46] is using Metaobject Protocols (MOPs) [33,32]. A MOP opens an implementation by allowing a user to control important aspects of an implementation's decision-making process. This control is provided by documenting a set of metaobjects and a protocol that the implementation adheres to. A metaobject is an object that controls some aspect of an implementation's behavior, usually by making a decision. The protocol describes the interaction between the metaobjects themselves and between the metaobjects and other parts of the implementation. MOPs are designed to provide the user with both locality and incrementality, among other benefits.

ViewForm does not employ a MOP because the important decisions made by the control structure are distributed among the transforms and because the important decisions can be made salient via user interaction. When building a transformation system using ViewForm, an implementor can incorporate user interaction directly into the decision-making process. In addition, since ViewForm "users" can be programs (i.e., agents), user interaction need not require the presence of a human to guide each and every decision. ViewForm decouples the ability of the user to direct the decision-making process from constraints on the way the decision-making process is implemented.

3.3.1 Intrigue

Intrigue [37] is a MOP for a Scheme compiler. The Intrigue architecture consists of two parts; a general data-flow engine (see Section 3.3.3) and an abstraction capable of representing Intrigue's intermediate languages. Intrigue's MOP allows users to control the compilation process via annotations on the source program, via user-defined object classes that localize and represent interesting program elements, and via user-defined methods that override or amend the compiler's default actions on those program elements. Intrigue's data-flow engine performs non-local analyses on its input program and provides support for user-defined, iterative data flows. ViewForm differs from Intrigue in several different aspects. Primarily, ViewForm is not a compiler. In addition, ViewForm provides explanations for its transformations and supports direct or indirect user interaction during the transformation process. ViewForm can work on sub-program pieces, whereas Intrigue does not support separate compilation.

3.3.2 Anibus

Anibus [51,52] is a MOP for controlling the coarse-grained parallelization of a program. Anibus's source language is sequential and its target language is explicitly parallel. A source language program is parallelized via annotations on program expressions. The annotations specify the program's parallelization and are used by Anibus to transform the program into the target language. A user can add new parallelizing transformations or modify existing ones by using Anibus's MOP. Anibus's transformation system is divided into three phases that implement the distribution of the program, the distribution of the data, and synchronization. The Anibus architecture without a MOP can be easily implemented as a transformation system using ViewForm. The ability to add or modify transformations is a natural consequence of being able to recombine the transformations used to construct Anibus.

3.3.3 Data Path Macros

Data Path Macros (DPM) [34] are macros that may expand in arbitrary parts of a program. The DPM engine takes a source program and converts it into a program tree. In this tree, the nodes are objects representing syntactic program elements and the edges are objects representing the ways in which those elements are connected via s-expressions. A DPM is a set of generic functions on objects in the program tree. These generic functions have access to data-flow information computed by the DPM engine. Syntactic annotations on the source program denote the places where a DPM is to be applied. The annotations make the corresponding program tree nodes easily recognizable to specific DPMs.

A DPM is created by writing a set of generic functions that perform transformations on the program tree and by defining a set of objects that will be used to annotate program tree nodes. One important property of DPM's is the ability to specify new user-defined data-flow computations. Such a specification describes initial data-flow values, how values propagate throughout the program tree, how values are combined at join points, and how equality between flow values is determined. This specification is procedural and takes the form of generic function methods. By using object-oriented inheritance, the amount of code that must be written to define a new DPM or a new data flow is significantly reduced. Some of the notions underlying ViewForm have their roots in DPM's. Three of these notions are the utility of arbitrary non-local program analyses, of non-semantics-preserving modifications, and of extensibility in a transformation system.

ViewForm differs from DPM's in that ViewForm supports direct user interaction, explanations for transformations, a more general notion of non-local program information, and a set of primitives for defining, manipulating, combining and abstracting transformations. In addition, ViewForm is not object oriented, although an efficient generic dispatch system [35] would certainly be quite useful in its implementation.

3.4 Optimizing Programs via Transformations

Several approaches for building or designing optimizers are based on transformation systems. These systems are concerned with high-level source programs and low-level target programs. While the domain of these systems is more specific than ViewForm's domain, they do address some of the same issues that ViewForm faces.

3.4.1 Dora

Dora [18,17] is a transformation system for experimenting with compiler optimizations. Dora's intermediate language (IL) is DILS. DILS is based on Scheme [29] and is extensible by virtue of being a schema. Instead of employing a traditional attribute grammar, Dora uses *attribution pattern sets* [19] to provide a more flexible framework for attributing the nodes of the IL tree. The same tree-pattern language is used to express both the attribution equations and the transformation pattern-matching language. This tree-pattern language is implemented via an efficient automata-based algorithm [28,5]. In Dora, the way in which transformations are applied can be specified from a fixed set of options. For example, the IL tree can be traversed from top to bottom, left to right, or the reverse of either option. Transformations can also be applied once or repeatedly. Another option is whether a transform's applicable sites should be recomputed every time a change is made to the IL tree or whether all of the applicable sites should be computed once, before the transformed is applied.

Dora transforms have access to non-local program information and can call arbitrary Lisp forms when a pattern is matched. The non-local attributes are computed automatically from a set of attribute definitions. Dora, however, suffers from some of the problems discussed in Section 2.2. For example, Dora does not have support for user interaction. In addition, Dora suffers from performance problems in the computation of non-local program information because the information is computed globally.

3.4.2 GENesis

GENesis [61] is an optimizer generator whose specification language is GOSpeL. GENesis takes a GOSpeL specification and automatically generates an optimizer. User interaction can be part of a GOSpeL specification in cases where several optimizing transformations are applicable. An optimizing transformation consists of a pattern, an enabling condition, and an action. The pattern is syntactic and the enabling condition can be in terms of non-local semantic program properties. When the pattern is matched and the pre-condition is met, the corresponding action is invoked. GOSpeL's action language consists of five primitives that allow program elements to be deleted, copied, moved, added (i.e., synthesized), and modified. GENesis patterns are not arbitrarily general, arbitrary multiple views are not supported, and the kinds of optimizations permitted by the transformation language are limited.

3.4.3 Program Optimization and Derivation via Transformations

In [4], transformations are investigated as a form of optimizing or deriving programs by transforming and connecting axioms. One interesting aspect of this work is that transformations can assert information about parts of the program that they create. For example, a transformation that reifies an implicit loop can assert a type and numerical range on the loop variable. These kinds of assertions are consistent with ViewForm's design, since one of ViewForm's aims is to support interaction with entities that can provide more accurate and more available knowledge about pieces of the program.

3.5 Interactive Program Design and Construction

A few transformational systems are designed with the end goal of assisting users in the design and construction of software. As described in [50], the vision discussed in [22] is one of the earliest examples of a Programmer's Assistant. These systems are related to the PA that will be built using ViewForm.

3.5.1 KBEmacs

KBEmacs [60] the Knowledge-Based Editor in Emacs, is an editor that partially acts as a Programmer's Apprentice [49] during the construction of a program. It uses a knowledge base of *clichés*. A cliché is an algorithmic fragment about which both KBEmacs and the programmer have knowledge. KBEmacs is designed to complement a software developer's skills by automatically handling certain details of a program development such as checking for errors, filling in pieces of source code, and rearranging the code when necessary. The Programmer's Assistant (PA) that will be built using ViewForm differs from KBEmacs in that the PA will be designed to expect no prior knowledge about its input program and will not attempt to understand the input program as a programmer would. In addition, the PA will be extensible, to allow user-defined transforms and views to be incorporated into its environment.

3.5.2 AGAVE

AGAVE [44] is a computing environment for the development of interactive, structure-oriented editors. The editors can check the syntax and semantics of program structures entered by a programmer. The error checking is performed via an action. While actions are arbitrary procedures, they do not have access to the entire program or to arbitrary non-local analyses. In addition, AGAVE structure-oriented editors aren't transformation languages and they don't provide multiple views on a program.

3.5.3 A Program Verifier Assistant

The Designer/Verifier's Assistant [43] is a system designed to help programmers verify the designs of their programs. The user gives the system a specification of the verification conditions for a program. The system uses this specification to verify a program, to verify changes to a program, to provide explanations for proposed changes, and to answer queries from the user. For example, the system can speculate on the effects that a proposed change will have with respect to a program's verification specification. The Assistant does not, however, support arbitrary non-local program analyses.

3.5.4 CCEL

CCEL [15,9] is a language for describing user-defined constraints on a C++ program. A user can describe design, implementation, and stylistic constraints. When a program is compared to its CCEL specification, any inconsistencies are reported via default system error handlers. The default method of reporting an inconsistency is to display the constraint that was violated. A user can optionally associate a more descriptive message that is displayed when a constraint is violated. CCEL is specific to C++ and cannot perform transformations on the source program.

3.6 Parallelization via Transformations

We have previously discussed one system (Section 3.3.2) for parallelizing programs via transformations. In general, automatically generating an optimal parallelization for a program is not decidable. Some systems, such as the one described below, are moving towards providing the user with some control over the parallelizing transformation process.

3.6.1 Crystal

Crystal [7,6] and Meta-Crystal [62,63] are languages used to parallelize source programs via transformations. These languages are lexically scoped, functional, and are based on a lambda calculus extended with provisions for conditionals, recursion, and local variable definitions. Meta-Crystal is used to define transformations when the default transformations do not result in adequate performance. The user is responsible for statically knowing which transformations will result in adequate performance. Meta-Crystal does not support interactive, non-semantics-preserving transformations or arbitrary non-local program information.

4 ViewForm's Design

This section discusses the proposed ViewForm design. The ViewForm language includes several types of primitive objects, including *Transforms*, *Views*, and *Explanations* (see Figure 4-1). A ViewForm Transformation System (VTS) is built by combining these primitives. A VTS parses the source program, applies a main transformation to the program, computes any requested views, commits the resulting program modifications, then unparses and outputs the program. We discuss and relate the ViewForm design aspects below.

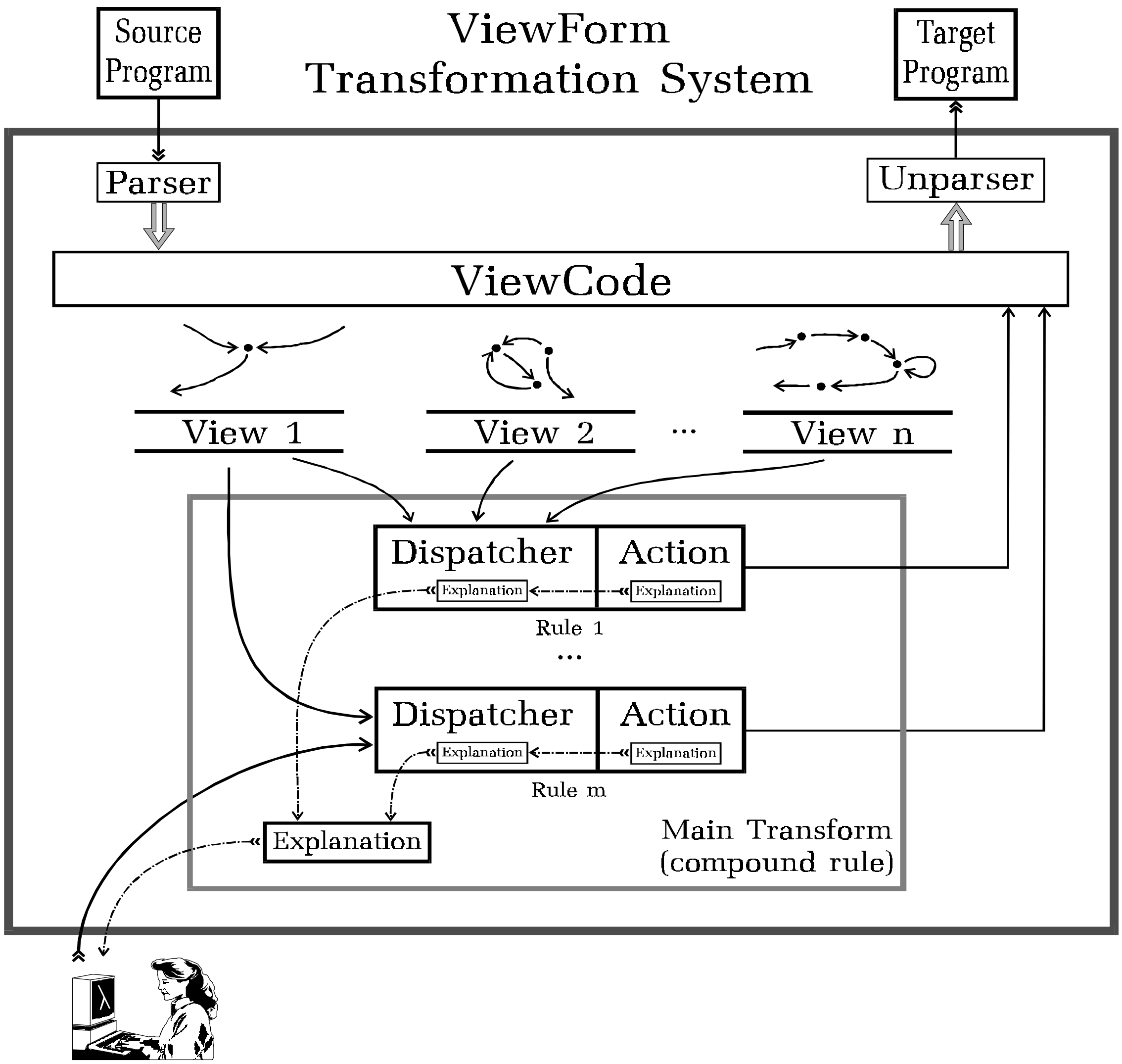

Figure 4-1: A ViewForm Transformation System

We begin with a high-level description of a VTS. As shown in Figure 4-1, a VTS takes a source program, makes changes to it, then outputs the resulting target program. Internally, a VTS contains a parser, an unparser, a main transform, and ViewCode. The parser converts the source program into ViewCode, which is ViewForm's internal representation for intermediate code. The unparser does the reverse; it converts ViewCode into a program in the target language. The main transform plays a role somewhat like that of the control structure in a vanilla transformation system–it makes changes to the ViewCode based on a set of rules. The changes can be made speculatively, which means that they will not be incorporated into the ViewCode unless they are committed by the transform.

The VTS works by taking a source program and converting it to ViewCode via the parser. The ViewCode is then modified by the main transform. The main transform can use views to request and access various non-local program properties. The main transform can also interact with the user. This interaction includes providing the user with explanations of what the transform does. Once the desired changes are complete, they are committed by the transform and the resulting ViewCode is then passed to the unparser for translation to the target language. In the sections that follow, I discuss the ViewForm primitives and their means of combination in more detail.

4.1 Transforms

One of ViewForm's primitive types is the transform. The purpose of a transform is to make changes to a piece of ViewCode. A transform is constructed by combining a *control structure* and an *explanation*. A control structure controls the behavior of a transform and is implemented as a piece of code that takes transforms as parameters. A transform is considered *activated* once its control structure has been passed a set of transforms. An activated transform carries out its work when it is supplied with a program expression. It is possible to construct a control structure that takes no transforms as parameters: this property is required in order to construct base transforms that can be combined to form more complex transforms.

An explanation is a description of what the transform does. This description is used during user interaction to provide the user (or agent) with a useful way of evaluating the transform. The description can be in a natural language or it can be in a machine-readable language. The latter case is useful for a variety of circumstances, including for use with agents that test transforms for correctness or that search transform databases. Explanations could also be used by an intelligent agent that builds higher-level transforms from lower-level transforms.

An explanation takes other explanations as parameters and needs the same number of explanations as its associated control structure needs transforms. An explanation uses the explanations it is passed to construct a description of what its associated control structure does. As is the case with control structures, it is possible to construct an explanation that takes no explanations as parameters, in order to be able to construct the base explanations from which more complex explanations can be built.

A variety of transforms can be constructed using this approach to building and composing transforms. Below, I identify and provide terminology for the kinds of transforms that the ViewForm implementation will provide special support for. This support will come as an interface for easily manipulating the transforms and an efficient implementation of their use. The terminology introduced below will be used in the remainder of this document.

An *action* is an activated transform that performs direct modifications on the program. An action does no work other than unconditionally performing the program modification itself. A *dispatcher* is a transform whose control structure takes a single action. An activated dispatcher is called a *rule*. A rule takes a program expression, decides whether the expression satisfies some enabling condition, and, if so, applies the action to the expression. A dispatcher does not modify the program directly: it uses its action to perform program modifications. A *compound rule* is a transform whose control structure takes rules as parameters, and then applies the rules in some fashion. For example, the rules can be applied in sequence until one succeeds. Alternatively, the rules can be applied non-deterministically, until a special goal rule succeeds. Separating the control structure from the rules provides the flexibility to reuse control structures, to have different control structures in the same system, and to experiment with control structures when the "right" control structure is not known ahead of time. The role of these transforms and their relationships are illustrated in Figure 4-1.

ViewForm transforms can be constructed, combined, abstracted, explained, and are Turing Complete. Transforms can also access arbitrary non-local program information, can be directed externally, and can be applied speculatively (i.e., can have their effects undone).

4.2 Views

Another of ViewForm's primitive types is the *view*. Views are used to control the computation of non-local program information and to access the results. Conceptually, a view is an abstraction over a program; see Figure 4-2. For example, the `text` view is a tree whose nodes are program objects such as numbers, symbols, and s-expressions, and whose edges connect compound expressions (e.g., combinations) to their components. The `call-graph` view is a graph with call sites and procedures as nodes, and with undirected edges between call sites and procedures they may invoke. The `data-flow` view (from which the `call-graph` view can be derived) is a graph whose nodes are expressions in the program and where an edge exists between two nodes if a value can flow from one node's corresponding expression to another. Figure 4-2 shows the view of the parameter `f` through the `dfa` view (data-flow view). It shows that `f` may be assigned one of two procedure values or any values that are values of the variable `proc`. It also shows that `f`'s value may be applied to the value of the variable `current`. Note that this figure does not illustrate the fact that as a parameter to a top-level procedure, `f` may be bound to any denotable value.

```
(define repeated
  (lambda (n f )
    (lambda (arg)
      (let loop ((n n)
                 (current arg))
        (if (= n 0)
            current
            (loop (- n 1)
                  ( f current)))))))

(define make-repeated-proc
  (lambda ( proc )
    (lambda (n arg)
      ((repeated n proc ) arg))))

(define nth-cdr
  (make-repeated-proc cdr ))

(define square-n-times
  (make-repeated-proc (lambda (x)
                        (* x x)) ))

(cons (nth-cdr 3 '(1 2 3 4))
      (square-n-times 2 5))
```

Figure 4-2: Partial Data-Flow Analysis View on `f`.

A view is made up of two parts. The first part is code that performs non-local program analysis over a piece of given ViewCode. The second part implements an interface to request and access the non-local program information. This interface is documented for use by transform writers.

Some important properties of views are:

- Views simplify the construction of transforms by abstracting away details of a program that are unimportant and by making salient the program properties desired by a transform writer.
- Each view provides an interface for accessing the program information it computes and for controlling when such information is computed, recomputed, or updated.
- Views are composable and can themselves be abstracted.
- Users can define their own views; user-defined views are integrated by means of a simple, documented interface.
- Transforms can control how and when views are applied.

- Dispatchers can access a program via views to determine whether or not their associated actions are applicable.

It may be desirable for views to also provide explanations for the information they compute. A high-level description would be helpful, at the very least. Unfortunately, some view algorithms may not be designed in such a way that makes them easily explainable. By using the same explanation mechanism that is used for transforms, however, the degree of detail provided by a view explanation can be made to depend on the nature of the view computation algorithm.

One final but important property of views is that they can be used to provide a more descriptive form of abstraction. Under traditional abstraction, the interface to a procedure consists of a procedure name, a specification on the number and types of its arguments, and a description of the procedure's functionality and return value. In particular, implementation details remain hidden (i.e., abstracted). Traditional abstraction, however, is not always desirable [46,45]. With views, other aspects of a procedure's functionality can be made visible, but without over constraining the procedure's possible implementations. For example, a view over an abstracted procedure may provide just enough information to enable an optimization rule to determine that an argument to the procedure will not be side effected by the procedure or that the argument will not be alive after a call to the procedure. Since views are already conservative, reducing the amount or accuracy of information provided by a view does not require any additional support. Furthermore, if the abstracted procedure's implementation changes, so will its views. This provides an automatic way of checking whether a previously applied optimization is still applicable after changes to the procedure's implementation. In addition, since the information provided by views can be constructed by hand, a programmer now has a specification language with which to describe non-traditional but important properties of a procedure that should be part of the procedure's interface.

4.3 ViewForm Transformation Systems

A ViewForm Transformation Systems (VTS) is a program that performs various tasks:

- Parses the source program into ViewCode
- Runs a main transform on the ViewCode
- Computes, updates, and maintains views (i.e., non-local program information)
- Handles changes to ViewCode (e.g., commits actions)

- Unparses ViewCode into the target language

A VTS can be told to invoke a parser other than its default parser so long as the result is in the form that ViewForm expects. For example, a VTS could be given a parser that is parameterized on a BNF specification. The default parser performs some simple manipulations necessary to bootstrap the transformation process, such as ensuring that the leaves of the ViewCode tree have identity. One goal of these manipulations is to strive to keep the internal representation of the program as close to the original as possible. This keeps program expression manipulations easy to write and maintains any prior familiarity the programmer may have had with the code.

When the program is ready for output, a VTS invokes a set of unparsers. These unparsers are also invoked to display program expressions as needed for user interaction. As in some of the systems presented in Section 3, users will be allowed to substitute their own unparsers in place of the default unparsers. Unparsers will also be used by systems such as the PA to render program expression in different forms (e.g., graphically).

A VTS carries out the program transformation process by applying a single, *main transform* to an expression in the program. Since transforms can be combined, this approach does not result in any loss of generality. In practice, this main transform will be a compound transform. A transform can carry out its task in a variety of ways. It can request views in order to find interesting program expressions and it can selectively apply rules to those expressions.

The application of a rule begins with the application of its dispatcher to a program expression. A dispatcher is free to consult the user during its execution. A dispatcher can bind variables whose values can be extracted by the action. This allows a dispatcher to easily pass interesting program expressions to the action. Once the dispatcher tests and verifies the rule's applicability, the action can be applied. Otherwise, the action cannot be applied. An action that can be applied performs its changes once it has been *committed*. A transform decides the order in which to commit valid actions. Some transforms may decide to queue the valid actions until no more rules are applicable. This gives the user a chance to abort the transform before any changes are made. Alternatively, a transform may decide to commit actions as soon as they are valid so that a view can be updated correctly.

The VTS carries out view computation and performs the actual modification of the program. In view computation, the initial program (or some designated portion of it) is analyzed. The VTS ensures that any requested view is not currently valid before computing it. The information gathered by the analysis generally takes the form of a graph and is available via a documented interface for each view.

For example, the `dfa` view information is made available via documented procedures such as `producers-consumers`.[††] The program analysis aspect of the transformation process is carried out by a view, but transforms control which program parts the view is applied to and also control when the view is updated. Ideally, views are updated at a fine grain; the finest grain being every time the program is modified. From a practical standpoint, this is not possible for moderately sized programs, where gratuitous view computation may consume an unbearable amount of resources such as time and space. In addition, not all program modifications leave the program in a consistent state. For these reasons, ViewForm allows transforms to control when views are updated.

Program modification is carried out via simple procedures for replacing, copying, deleting, and constructing[‡‡] program expressions. These procedures are called by actions and replace existing program expressions by normal s-expressions. The VTS automatically parses the normal s-expressions into ViewCode.

4.4 User Interaction

Transforms can interact with external entities such as the user, through an interface supported by the VTS. This interface will allow access to a transform's explanation. Transforms involved in user interaction have explanations that are machine readable so that agents can use them. A VTS will have a protocol for interacting both with users and agents. This interaction protocol's semantics will depend on the particular functionality of the transforms in a VTS. This property is important, for it decouples the need for user interaction from the details of a given transformation system's implementation. In essence, it is possible to interact with a ViewForm transformation system in a way that does not depend on the particular control structures or view computation techniques that the system employs.

[††] This procedure takes one argument, `<expression>`, and returns two sets of expressions. The first set may produce values returned by `<expression>` and the second set may consume the values produced by `<expression>` (see Appendix I).

[‡‡] Deconstruction of program expressions is already available via the `text-view`.

4.5 *Discussion*

Transforms are composable and Turing Complete. With a mechanism such as translucent procedures [54], combinations of transforms could even be dynamically decomposed and re-ordered. Through a standard interface to views, a transform can explicitly control when a view is updated or more generally, when it is re-computed from scratch. This control is important when view computations and non-semantics-preserving program modifications are interleaved, since the ViewCode needs to be in a consistent state for a view to be updated correctly. This control is also important for performance. For example, a set of transforms or dispatchers that each use the `dfa` view but do not alter it can be combined without the need to recompute the view each time another transform or dispatcher in the set is invoked. ViewForm allows transforms that do not alter the information contained in a view to explicitly state this when initially requesting a view, in order to prevent unnecessary view re-computation the next time the transform is invoked.

In ViewForm, the computation of interesting program properties is conceptually separate from both the identification of relevant points in a source program and from the modification of the program itself. Thus, views can be shared by many different transforms. In addition, when transforms and views are written in the source language that ViewForm operates on, they can be self applied for example, to reduce the potential performance cost of decoupling views from transforms.

5 Solving the n–queens example in Mini-ViewForm

Mini-ViewForm is a ViewForm prototype. It was implemented to test the viability of a ViewForm-like language. Based upon experience with Mini-ViewForm, significant changes have been made to the ViewForm design. The previous sections describe that newer design. While the prototype described below is not a complete or an exact implementation of the full ViewForm design, it is faithful to the important aspects that make ViewForm unique. Mini-ViewForm supports user interaction, non-semantics-preserving transforms, and non-local program information in the form of multiple views. Mini-ViewForm also implements the functionality of transforms, rules, dispatchers, and actions as well as a means to combine them. Mini-ViewForm does not have a formal notion of a transformation system; rather, Mini-ViewForm has a *run time system* that has the same functionality.

Mini-ViewForm takes source programs written in a subset of Scheme. This subset contains all of the special forms needed to run the `n-queens` example. In the interest of brevity, when I refer to Scheme for the discussion in this section, I mean the subset of Scheme that Mini-ViewForm accepts.

Mini-ViewForm provides three views: `text`, which is an s-expression-like representation, `alpha`, which is an alpha conversion, and `dfa`, which is a data-flow analysis. Mini-ViewForm provides rudimentary procedures for interacting with the user but does not support explanations or undoing of actions (though these could be added without difficult changes to the implementation). Mini-ViewForm implements actions, dispatchers, rules, and compound rules as separate types of primitives. It was in the process of examining Mini-ViewForm's resulting design and implementation that it became obvious that conceptually, these were all one in the same. Mini-ViewForm treats rules as data structures composed of a dispatcher and an action. Thus, the rules described below are not executable.

Appendix I documents Mini-ViewForm. Appendix II contains transcripts of a run of Mini-ViewForm code on the `n-queens` example. Appendix III contains the Mini-ViewForm code discussed below in its entirety.

5.1 The Data-Type Conversion Transform for **`n-queens`**

We create a transform for changing the return type of a Scheme procedure from an eagerly evaluated data structure to a lazily evaluated one. To motivate the example, we look at the class of procedures that perform a search to find all possible solutions to a problem. Specifically, our procedure returns a set of all solutions to the `n-queens` problem using depth-first search (see Figure 1-1). In this example, we are interested in problem instances with very large search spaces. Therefore, we want to use Mini-ViewForm to modify the `n-queens` code so that instead of computing all possible answers at once, we compute the possible answers each on demand. In particular, we want to convert the return type of the `n-queens` procedure from a `list` to a `stream` . We implement this transform in two steps. First, we define a parameterized transform that identifies and changes any expressions that produce a given expression's return value, including non-local expressions. This parameterized transform is the kind that would normally be found in a library of ViewForm transforms. Then, we use this parameterized transform to define a specific transform for changing the return type of an expression from a `list` to a `stream`. This specific transform is then bundled up with the Mini-ViewForm run time system and run on the `n-queens` example.

We start with the `n-queens` implementation in Figure 1-1. This code is in a form that can be passed directly to Mini-ViewForm; a fully developed ViewForm will handle top-level `define`s directly. The output of this code can be found in Appendix II. It is a very helpful exercise for the reader to manually perform this change on the `n-queens` code, then later compare the changes and effort to the automation provided by the Mini-ViewForm transform.

5.1.1 A Higher-Level Transform

The construction of our transform begins with the piece of code in Figure 5-1. This code implements the parameterized transform described above and is likely to be found in many ViewForm transform libraries due to its high degree of utility. Using the `with-views` Mini-ViewForm construct, `make-change-return-type-xform` returns a transform that first ensures that the `alpha` and `dfa` view's are available and valid on `<program>`. `<program>` is an expression that contains the sub-expression to which the transform is applied. The transform then uses `apply-rules!` to apply a set of rules to `<expression>`'s ***producers,*** which are all expressions that can produce the value returned by `<expression>`. Subsequently, the transform applies a set of rules to `<expression>`'s ***consumers***, which are all expressions that can consume the value returned by `<expression>`. Once the rules have been successfully applied, any applicable actions are committed using `commit-queued-actions!`. The interface to the Mini-ViewForm procedures used by the code is documented in Appendix I.

A subtle technicality exists in computing the producers and consumers of the return value returned by `<expression>`. For example, in our `n-queens` example, we could have had another procedure, `sample-board-config`, in the top-level `letrec` expression that created queen configurations and passed them to `display-n-queens`. For reference, I call these types of producers and consumers ***secondary***. In this case, in order to modify `display-n-queens` to work with `streams`, we would need to make more program alterations in or around `sample-board-config`. For example, we could convert `sample-board-config` to return streams, we could insert a type coercion between the call from `sample-board-config` to `display-n-queens`, or we could make a copy of `display-n-queens` that works with lists and is called by `sample-board-config`.

```
(define (make-change-return-type-xform program
                                       producer-rules
                                       consumer-rules)
  (make-xform
   (lambda (expression)
     (let ((xform-thunk
            (lambda ()
              (let* ((prod-con (producers-consumers expression))
                     (producer-sites (car prod-con))
                     (consumer-sites (cdr prod-con)))
                (apply-rules! producer-rules producer-sites)
                (apply-rules! consumer-rules consumer-sites)
                (commit-queued-actions!)
                #t))))
       (with-views '((alpha invalidate) (dfa invalidate))
                   program
                   xform-thunk)))))
```

Figure 5-1: A Parameterized Transform

Our transform instead uses the procedure `producers-consumers`, which returns both primary and secondary producers and consumers. The transform makes both primary and secondary producers and consumer candidates for change. In our example, it is up to the set of rules to determine what choices are available in such cases. An alternative implementation choice is to identify and treat secondary producers and consumers with a different set of rules. Such an alternative transform could be written by a Mini-ViewForm user or could also be available as part of a library of transforms.

5.1.2 A Specific Transform

We now create our specific transform using the code in Figure 5-2. In this code, `<program>` will eventually correspond to the `n-queens` program in Figure 1-1. `construct-list->stream-rules` returns a list of rules that implement replacements such as `cons-stream` for `cons` and `stream-car` for `car`. We shall examine those rules below. `add-escape-rule` adds a rule that determines whether a value can escape the scope of program and consults with the user when it finds such an escape value. The rule depends on the `alpha` and `dfa` views being valid. The producer rules are constructed by combining each of our `list` to `stream` rules with the escape rule. This ensures that the user is warned about any value that may escape before a producer expression in the source program is modified. In a program with top-level `define`s, the consumers would require an analogous rule for external values being passed in.

```
(define (make-list->stream-xform program)
  (let ((rules (construct-list->stream-rules)))
    (make-change-return-type-xform program
                                   (map add-escape-rule rules)
                                   rules)))
```

Figure 5-2: A Specific Transform

Figure 5-3 contains the implementation of `construct-list->stream-rules`. These rules replace the set of `list` producing and consuming primitives with the available corresponding `stream` primitives. The first subset of rules automatically performs standard replacements, such as `cons` with `cons-stream`. The second subset of rules makes replacements only upon user confirmation. This is because users may not necessarily want to take the length of an infinite stream, for example. The third rule converts all two-argument calls to `map` and `for-each` into calls to `stream-map`. If a call to `map` or `for-each` with more than two arguments is encountered, the transform aborts. If we had not limited the replacement procedures to Scheme primitives, `map` and `for-each` could have been replaced by user-defined `stream` procedures that would have better approximated their behavior. The fourth rule converts constructions of the empty `list` into those of the empty `stream`. Finally, we end with a rule that discards all queued actions to the program if a producer or consumer isn't handled by another rule. A more comprehensive set of rules (and a better `dfa` view computation) would also include a form of user interaction for handling cases when the procedures producing the return value are extracted from data structures. This situation can arise, for example, when dealing with generic procedures that maintain a dispatch table.

The transform performs its changes automatically unless one of the following situations arises:

1. One of the list values constructed by a primary or secondary producer escapes the scope of the code being analyzed through some route other than the starting expression. This is handled by `escape-rule`.
2. A producer site is a procedure call where a first-class procedure is being called and at least one of the possible procedure values does not match a producer rule. This case is handled by `ask-user-partial-match`, which is combined as a dispatcher into primitive matching by both `conjunctive-prim-match` and `make-prim-subst-rule`.
3. Either `list-ref` or `length` is called on a list being converted to a stream. This alerts the user to a possibly undesirable situation.

When one of the above situations occurs, the dispatcher presents the user with one of three options: to ignore the expression, to perform the action anyway, or to discard all actions queued by the transform and abort.

```
(define (construct-list->stream-rules)
 `(,@(map make-prim-subst-rule
         '(cons         car        cdr        pair?        null?)
         '(cons-stream  stream-car stream-cdr stream-pair? stream-null?))

   ,@(map make-prim-subst-rule
         '( length                 list-ref                )
         '( stream-length          stream-ref              )
         `(,ask-user-about-length ,ask-user-about-list-ref))

   ,(make-rule (combine-dispatchers-conjunctively
                (conjunctive-prim-match 'map 'for-each)
                (combine-dispatchers-disjunctively ensure-two-arg-comb
                                                   discard-changes))
               (combine-actions (make-replacer replace-operator! 'stream-map)
                                reverse-args!))

   ,(make-rule null-list?
               (make-replacer replace-entire-exp! '(stream)))

   ,(make-rule discard-changes continue)
   ))
```

Figure 5-3: Rules

One important aspect of the `n-queens` program is that it implements a depth-first search. If it had implemented a breath-first search, the transform would not have produced the intended result, since the purpose of changing the return data structure from a `list` to a `stream` is to delay the computation of part of the search process. In a breadth-first search for the `n-queens` problem, once one solution is found, most of the work for finding all the other solutions is also complete. The premise of the example, however, is that we want to compute only one answer while doing minimal work towards the computation of the others. Thus, we need a depth-first algorithm in order for our transform to achieve the intended results. The point is that while ViewForm can be used to build powerful tools that automate programming tasks, it does not solve the undecidable problem of automatically identifying and replacing arbitrary algorithms in arbitrary programs.

The results of running the specific transform on the `n-queens` program can be found in Appendix II and the transform code can be found in Appendix III in its entirety. This transform code uses Mini-ViewForm primitives for creating and combining dispatchers, actions, and rules, and for gathering information collected by views.

5.2 *Discussion - Alternate Forms of User Interaction*

There are several ways of handling cases 1, 2, and 3 above in addition to what the example transform does. Three of these options are given below:

A. For situation 1, place a type coercion at each escape site and at each site where either a transformed value or non-transformed value may reach.

B. For situations 1 or 2, duplicate some of the code, run the transform on one of the duplicates, then insert a run-time dispatch to invoke the appropriate piece of code.

C. For any of situations 1, 2, or 3, ask the user for help in making a better decision.

The benefit of Option A is correctness; or at least it may seem that way. By coercing each `stream` back to a `list` at escape sites other than the starting expression, we preserve the way any procedures outside the scope of the view see the procedures we have transformed. If, however, the value of the starting expression can escape through the starting expression, then we will have two types of values escaping. This is not correct in general. The correctness of Option A depends on what the outside code does, but it is nevertheless an option that is useful in cases where the user has good knowledge of what the outside code does.

In Option B, we produce two different pieces of code; one that is used by code that has been transformed, and the other for use by non-transformed code. This solution suffers from the same problem as Option A but, like A, it is useful in some cases. In addition, we can use partial evaluation to help optimize the resulting, copied pieces of code.

Option C is useful in at least two different aspects. User interaction can help the transform decide what kind of program transformation to apply (as in Options A or B). Just as important, it can also help the transform to more accurately refine the information it has been given by a view. For example, the set of producers and consumers may be too large if the set is based on one or more control paths that the user knows are never taken at run time. When interacting with the user, a transform might ask for help in a variety of different ways, including:

I. <u>Query Replace</u>–The user interactively decides where to apply a specific transformation.

II. <u>Alternate Action</u>–The user mandates that the transform apply an alternate action on the code.

III. <u>Refine a View's Analysis</u>–This is similar to I, except that the user helps the transform decide where it is applicable by providing a better approximation to a view's static program analysis.

The transform presented for the `n-queens` example allows users to interact via Method I. Agents can also interact with the example transform by mimicking the user, though this is not the best form of interaction for this transform.

5.3 Other Example Sets of Transforms

Another transform that can be built using ViewForm introduces parallelism into code. For example, Anibus [51] can be implemented using ViewForm. In Anibus, the target language is a superset of the base language. The target language contains primitives for distributing the computation, distributing the data, and for performing synchronization. These primitives correspond to lower-level procedures and are thus syntactically and semantically consistent with procedures in the base language. In Section 6.7.2, I discuss how ViewForm can be used with intermediate or target languages that are similar to the source language.

ViewForm can be used to write sets of transformations that aid in the debugging a program. For example, when debugging a program that has side effects, it may be necessary to monitor reads and writes to variables. Instead of using a text editor to find occurrences of a variable to be manually replaced with monitoring code, a ViewForm transform can be used to more robustly perform the search and replace with less programmer effort. As a further example, a case study on debugging parallel code that also discusses views is described in [36].

Another interesting use for ViewForm is in working with numerical code. One use is to change the representation of a set of input values to a numerical library. This can enable the library to suitably handle a different range of input values. For example, suppose that a user is given a mathematical library that takes a set of input parameters whose values, for the user's purposes, are always going to be small numbers that are multiplied together (e.g., probabilities).[§§] Suppose that the user prefers to work with the `log` of the numbers, in order to prevent numerical underflow. We can write a transform that follows the parameter values to the call sites where they are operated on and performs appropriate substitutions of the mathematical operations.[***] Unfortunately, static program analysis cannot accurately track values through arbitrary data structures or first-class closures. In these cases, the transform may choose to ask the user for help or may defer the decision until run time by tagging the data as it goes in and

---

[§§] This example is taken from [34].

[***] In fact, we could use `make-change-return-type-xform` to define this transform in Mini-ViewForm.

out of data structures and closures. While the latter case may lead to a decrease in run-time performance, manually modifying the mathematical library may not be feasible or may lead to bugs that a correct and tested transform would not introduce. Thus, an increase in running time due to a transform (if any) must be weighed against the alternative increase in overall time to perform, test, and debug manual changes to the numerical library.

Other examples for which ViewForm is helpful are: introducing or enforcing data abstractions in non-abstracted code, in-lining and loop-unrolling optimizations, user-assisted partial evaluation, introducing and debugging non-determinism, and understanding the way a program works.

6 Progress and Pending Issues

Currently, a prototype of ViewForm, Mini-ViewForm has been implemented in MIT Scheme. Mini-ViewForm supports the `text`, `alpha` and `dfa` views. The `n-queens` example in Figure 1-1was run using Mini-ViewForm. Mini-ViewForm is currently unoptimized and has not been integrated into the MIT Scheme environment. There are several important issues that need to be addressed by a fully developed ViewForm implementation. I discuss some important ones below.

6.1 User Interaction in Fully Developed ViewForm

In the fully developed ViewForm, several user interaction issues will be addressed. The first is an interface for accessing machine-readable explanations. These explanations will be the basis upon which an agent makes decisions. The language used to construct these explanations will be developed for ViewForm and will likely include a way to query the explanation about a default set of properties (e.g., correctness preserving, optimizes for space or time, etc.). The use of agents is important because there will be many times when a transform needs a user's help and when the user's answers for many interactions will be predictable with computable accuracy. In these cases, agents are useful for assisting the user with ViewForm interactions. Some agents may attempt to learn the user's responses and replay them, while others will have some responses hard coded. Some may always choose a semantics-preserving response and some may make suggestions to the user.

While an agent can be used to simulate a user's interaction with a particular transform, it may also act on behalf of the user in deciding which transforms to apply. For example, an agent can try one of several different transforms if the current one does not make acceptable progress. In essence, the purpose of an agent is to further automate the transformation process. A user should need to interact with ViewForm only when the interactions are interesting or non-trivial.

One other user interaction issue involves explanations. For example, a transform should be able to explain *why* it wants to apply a given action. The approach used by ViewForm is to construct explanations the same way that transforms are constructed. ViewForm constructs complex explanations by combining more simple explanations. For instance, a primitive dispatcher can provide a concrete reason for why it returned a true or false value when applied to a particular expression. The procedures for combining transforms can also combine these explanations. For example, if two dispatchers are combined conjunctively, the explanation provided by the compound dispatcher is that the explanations provided by both of the original dispatchers are true.

It is important to allow explanations to be abstracted. A user may not want to see all of the details of the lower-level explanations from which a complex, compound explanation is composed. Instead, a user may want a summary and the ability to call up the details if necessary. This can be implemented with ViewForm because users can define their own explanation-construction mechanisms that appropriately construct explanations more abstractly and accurately than what the default mechanism provides.

6.2 Performance

Not surprisingly, Mini-ViewForm spends most of its time computing views. This is not uncharacteristic of a transformation system that computes global program information [20]. A significant amount of effort will be needed to ensure that ViewForm is responsive enough to be useful in an interactive setting. Three approaches to this problem include increasing the performance of view computations, providing views with different performance characteristics for use in different situations, and allowing view computation to be restricted to smaller regions of a program.

We begin with a discussion of the first two approaches. In Mini-ViewForm, the `dfa` view implementation is heavily based on LIAR [53]. While our particular implementation is slow, it was not written with performance as a significant concern. While other data-flow algorithms could be used, the program representation employed by the algorithm must be amenable to ViewCode. For example, the type of algorithms presented in [55] would not be useful in ViewForm because they require that the ViewCode be CPS converted into a subset of Scheme. Some algorithms are better suited for use in an interactive environment or in an environment where the analysis need not be performed everywhere in the program. One example of such an algorithm is demand-driven data flow [16]. This algorithm can be used to compute interprocedural data-flow analyses that use a finite domain set. For distributive flow functions, this algorithm provides precise answers. For non-distributive, monotone flow functions, however, the answers are approximate.

The third approach to increasing performance is directing views to be computed on smaller regions of the program. This can be done via a suitable interface to the view computation procedures. The smaller regions could be selected by either a transform, the user, or an agent. Another way to implement this idea is to use a less accurate algorithm over a larger portion of the program and use a more accurate algorithm in places where transforms or users require greater accuracy. In addition, views could use information provided by module or environment specifications when such information is available in the source program.

6.3 Graphically Viewing the Program Graph

There is evidence that programmers are more productive when allowed to use tools that graphically render different views on a program [41]. ViewForm will be capable of graphically displaying different views of a program. It is likely that ViewForm will use browsers currently under development for a similar purpose. For example, Adams has constructed an HTML interface for viewing Scheme programs as structures [1]. This interface can be adapted for browsing Scheme programs under a variety of views. Interestingly enough, such a model allows users to create their own favourite way of viewing a Scheme program and allows anyone with internet access to view the Scheme programs remotely.

A vanilla transformation browser is also being developed by Stark [56]. This browser allows users to graphically examine changes made by a single transformation phase in the MIT Scheme compiler. Current work is aimed at incorporating explanations for transformations into the browser. Such a browser would be a useful interface for ViewForm users.

6.4 *Applicability to Other Languages*

The ViewForm design is not coupled with and is not dependent on any particular programming language. It is applicable to a variety of programming languages. It is also applicable to non-programming language scenarios where source-to-source transformations are useful, such as verifying or deriving mathematical proofs. It is not likely that any particular implementation of ViewForm will be trivially portable to different source languages. This is because a language with a different syntax and semantics will need different ways of parsing source programs and computing views. We discuss a related issue in Section 6.7.2

6.5 *Reliability of the User*

One concern raised in the context of user interaction is the reliability of the user. There are instances when the user is not trusted to provide a correct assumption or to "vouch" for a condition.[†††] ViewForm does not attempt to solve this problem. In some cases, however, it is possible to ensure safety even if the user is wrong. For example, ViewForm might be able to insert runtime checks to ensure the accuracy of information provided by the user. These runtime checks could be removed when the degree of confidence in a user's assertions are high enough. Under other circumstances, ViewForm might also be able to verify a user's assertions. The reliability of the user can thus be taken into consideration when developing VTS's

6.6 *Modification Histories*

Since the source program is modified through procedures implemented in the ViewForm runtime, it is possible to maintain a history of every program modification. More practically, it is possible to dynamically trace the modifications. These histories and traces can be invaluable when debugging transforms or as data for agents designed to learn to interact with ViewForm as a real person would. A program modification history also makes it possible to undo program modifications, thereby allowing speculative modifications. For example, suppose that an agent has the task of making a particular program property true by virtue of choosing the "right" rule to apply. Using speculative modifications, an agent can take a reasonable guess and then check the resulting program for the given property once the rule has been applied. If the property is

[†††] For example, register variable declarations in C are ignored by many highly-optimizing compilers.

true, the agent is done. Otherwise, the agent undoes the program modifications and tries the next rule. Such an agent could also be used to verify a rule's correctness.

6.7 ViewForm's Limitations

ViewForm is not the "perfect" transformation language. It is not certain that a perfect language exists. We discuss the limitations inherent in ViewForm below.

6.7.1 A View's Accuracy

One of ViewForm's limitations is that the power and applicability of a transform can be limited by the inherent conservative nature of static analyses performed by views such as `dfa`. There are two aspects to this problem. One is the need to improve the accuracy of a view in cases when the user has knowledge about a program that is not generally decidable or realistically computable. The other aspect is the user's need to predict the effects of a transform. Usually, a desire for the former is a consequence of a need for the latter. Given that ViewForm allows user interaction and agents, these problems can be addressed by developing an interface that provides some form of predictability.

6.7.1.1 Type Systems and Annotations

The accuracy of a view can be enhanced by a language's type system, even if it is one supplied by the user that is not necessarily part of the language. An example of a strongly-typed language is ML [42], in which user-provided type information is used to statically ensure that a program is "type safe". ViewForm for ML would use the type information to produce more accurate views.

In a non-typed language like Scheme, a user may decide to introduce a new syntax for types, which can then be used by a view computation in order to produce more accurate view information. The types need not necessarily have meaning with respect to the language. For example, the user-defined types may be assertions used by user-defined views. These types can take the form of annotations, as used by Intrigue or Anibus. In order to use user-defined types or annotations, it is necessary to modify the implementation of views. Modifying ViewForm's implementation to understand various source program syntaxes and semantics is discussed in Section 6.7.2.

The utility of annotations is great enough to warrant that the fully developed ViewForm provide an interface to support them. For example, annotations can be used to name specific expressions in a program that are passed to transforms. This issue will be addressed during ViewForm's development.

6.7.1.2 Interactive View Computation

Another way to improve the accuracy of a view is to make the view computation itself interactive. While this option may provide a plethora of useful information, in practice it can be problematic. For example, it is not likely that the user would want to be bothered with answering questions about the program each time a view is computed. Another problem is that there may be a prohibitively large set of questions that a view computation may want to ask the user and it is difficult to determine which of these questions are actually beneficial to ask.

One alternative is to use agents to answer a view's queries. An agent can be pre-programmed with more detailed knowledge about an important part of the program and may have little or no information about the rest. This would result in more accuracy for some part of the program and the normal, default accuracy for the rest. Alternatively, the agent could instruct a view to use a more accurate but time-consuming analysis method on an important part of the program and the normal, default method on the rest.

Another way to make view computation interactive is to have views interact with transforms. A transform that has an interest in a specific part of the program may ask a view to perform a more locally accurate analysis. The view may then decide to ask the user questions to help it compute such an analysis. The advantage of this approach is that it allows a view to use a faster but less accurate analysis initially and a slower, but more accurate analysis if a transform actually needs the information. We will study both of these routes in the fully developed ViewForm.

6.7.2 Syntax and Semantics

ViewCode and the target program returned by ViewForm are limited in some ways. First, the default views use the semantics of the source language for view computation. This means that it may be difficult to reuse the implementation of default views for different languages. There are several approaches to alleviating this problem. For example, views based on data-flow analyses could be built from general or open data-flow engines. In fact, several of the systems discussed in Section 3 utilize this approach. The same limitation can exist for the implementation of the parser and unparser. Again, general parsers and unparsers can be used to alleviate this problem.

While ViewForm has been presented thus far in a source-to-source context, there is nothing inherent in the design that prevents it from being used otherwise. For example, the output of one set of transforms may result in a program that can be easily (e.g., locally) unparsed or translated to a target language other than the source language. This target program can then be acted upon by a conceptually different set of transforms and views. While a compiler may be built using this approach, it will cause strain to the user interaction aspect of the system. This is because the code and the decisions being made will become increasingly foreign to a human user as the code becomes less abstract. User interaction under these circumstances will likely require the use of agents to be useful.

6.8 A Variety of Transform Types

ViewForm is a language in the sense that it has a few simple primitive objects, namely views, transforms, and explanations. Although few in number, primitives can be combined to form many different sets of transforms, each with particular properties. For example, one can build a set of transforms that are to be used only for debugging a finished program. In this case, the transforms will likely be semantics-preserving and will depend on user interaction. Alternatively, a different set of transforms may be used to alter the semantics of the base program. For example, a transform can change the calling convention of a set of callers and callees from call by value to call by name or call by need. A transform can also help implement non-deterministic additions to a program, by recognizing special forms such as the non-deterministic operator `amb` [39,64]. Additionally, ViewForm can be used to implement traditional macros with different semantics, including varieties of hygienic and non-hygienic macros. These macros can have syntaxes and semantics based on personal preference and would be portable to varieties of base languages with different macro conventions.

7 Other Directions

One important aspect of ViewForm can be generalized: ViewForm's concept of the ViewCode. ViewForm conceptually treats ViewCode as a set of nodes connected by a variety of different interconnection networks. In the case of Scheme, the nodes are the objects passed to ViewForm during parsing and the interconnection networks are determined by Scheme's syntax and semantics. For example, in the `text` view, the program is a graph whose nodes are objects such as symbols, numbers, and s-expressions, and their interconnection network is a tree. In the `alpha` view, the nodes are variables and the interconnection network is a set of bipartite graphs between variables that are assigned values and variable references.

We can therefore generalize ViewForm as a language for describing computations on a substrate composed of many processors that communicate via a variety of interconnection networks with time-varying topologies. In this generalization, computing a view is analogous to building (either physically or virtually) an interconnection network between nodes and re-computing a view is analogous to updating the links in an interconnection network. Changing views is analogous to moving to a different kind of interconnection network between nodes. A rule is analogous to a task to be performed by a set of nodes. Applying a rule is analogous to performing work on a node or set of nodes. A dispatcher is analogous to a decision-making process that determines at which nodes the work is going to be performed and an action is analogous to the work itself. Implementing a version of ViewForm along these lines will be pursued only if time permits.

8 Conclusion

Current transformation systems do not make adequate use of non-local program properties, they prohibit non-semantics-preserving transformations, and they limit the degree of user participation in the transformation process. This precludes them from being used to build a Programmer's Assistant and from solving the `n-queens` example. I claim that a language for building transformation systems need not embrace these restrictions and can yet be simple, general, and practical enough to yield efficient, safe, and user-extensible transformation systems. To demonstrate this claim, I propose to construct ViewForm, a language for building and experimenting with transformation systems. ViewForm supports non-semantics-preserving transforms, non-local program properties as enabling conditions, and direct or indirect user interaction. ViewForm also introduces two novel constructs; *transforms* and *views*. Transforms are composable rewrite rules with independent control structures and associated *explanations*. An explanation is a composable description of a transform's functionality. Views are composable abstractions over a program and can be used to abstract non-local program properties such as data-flow and control-flow information. To demonstrate my claim, I propose to use ViewForm to construct the Programmer's Assistant, a powerful tool for partially or fully automating complex programming tasks. The assistant will be used to debug, modify, and optimize SCMUTILS (a tool for solving a rich and diverse set of mathematical expressions) in ways that would not otherwise be tenable by those unfamiliar with its implementation details.

9 Acknowledgments

Michael Blair took extra care in reading this proposal and provided detailed comments and suggestions. Kleanthes Koniaris and Rajeev Surati also read this proposal and provided ample feedback. Guillermo Rozas is a great person to discuss ideas with. He suggested some of the examples for which ViewForm would be useful.

John Lamping provided insightful guidance in the development of the ideas presented in this proposal, including the suggestion to make views more predictable via annotations. Chris Hanson is working on SCMUTILS and helped me to explore different ways in which the Programmer's Assistant could be applicable. Hal Abelson provided guidance in the development and presentation of the ideas in this proposal. Gerry Sussman read the proposal and discussed the ideas and directions with me.

Christine Moore provided lots of personal support throughout the different stages of this document's development; her encouragement helped me to keep many things in perspective.

My Mom has continually provided tireless support and encouragement, and my late Father's memory will always provide me with the highest standard by which to live.

9.1 NSF Disclaimer

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Appendix I: Mini-ViewForm Interface

We present the interface to Mini-ViewForm. We begin with a set of higher-level procedures that are utilized by the code in Appendix III. We then present the base Mini-ViewForm primitives. Since Mini-ViewForm is a prototype implementation, there are some significant changes to this interface that have been made in ViewForm. We describe the nature of those changes where appropriate.

1 Higher-Level Procedures

```
(with-views '((<view> <validation>) ...) <prog-exp> <thunk>)
```

Ensures the validity of a set of views on the program expression <prog-exp>, and then invokes the thunk, <thunk>. <view> is the name of a view, and <validation> is either valid or invalid. invalid means that <thunk> invalidates its associated view, and valid means that <thunk> maintains the validity of its associated view. with-views returns <thunk>'s return value, unless <thunk> is aborted as described below.

<thunk> invokes one or more transforms on an expression in <prog-exp>. with-views sets up an environment where any applicable actions are queued until one of three procedures is called. If clear-queued-actions! is called, all queued actions are discarded. If commit-queued-actions! is called, the queued actions are committed. If discard-queued-actions! is called, the queued actions are discarded, each <view> is kept in its previous state of validity, and with-views's continuation is immediately invoked on the value #f. If <thunk> returns normally, with-views assigns the <view>s their given <validation> status.

In addition, with-views supports automatic view validation checking. For example, any <view> that was previously valid will not be recomputed before <thunk> is invoked. Furthermore, if a view is requested from within <thunk>, the view will be recomputed on <prog-exp> only if the view is invalid.

In Mini-ViewForm, with-views is used to apply a single transform on an expression within a program. The transform provides the user with the option to abort without changing the program during user interaction. ViewForm will provide several higher-level procedures like with-views for applying transforms under various conditions.

(producers-consumers <exp>)

Returns a pair whose first element is a list of primary and secondary expressions that can produce `<exp>`'s value. The pair's second element is a list of primary and secondary expressions that can consume `<exp>`'s value. `producers-consumers` consolidates raw information produced in the `dfa` view. The raw information is consolidated into two sets containing every program expression visible to the `dfa` view that might be affected if `<exp>`'s return value is changed. In ViewForm, most users will access view information through this kind of higher-level procedure.

(value-escapes? <exp>)

A dispatcher that returns `#t` if `<exp>`'s value can escape either through the top-level or through a maximally lexically dominant expression processed by the `dfa` view. Returns `#f` otherwise.

2 Views

Mini-ViewForm supports a subset of Scheme, as do the views below. Mini-ViewForm provides ways of defining and invoking views and of testing and setting their validity status. Views are implemented as simple data structures that are referenced via symbolic names. In addition, the Mini-ViewForm validation mechanism is very coarse grained; either a view is valid or it is not.

In ViewForm, views will be first-class objects that can be combined and abstracted. In addition, a VTS will provide finer-grained view validation checking. ViewForm views will also support user-provided explanations for the information they compute. These explanations will be accessible by transforms.

(def-view <view-name> <view-proc>)

Defines a view and makes it invalid. `<view-name>` is a symbol that will be used to refer to the view. `<view-proc>` is the view's working procedure. `<view-proc>` takes a program expression as an argument, and when invoked, computes the view's program information on the expression.

(ensure-views! <view-list> <exp>)

Computes each view in the list `<view-list>` on `<exp>` if the view is invalid.

(view/valid? <view-name>)

Returns `#t` if the view `<view-name>` is valid. Otherwise returns `#f`.

(view/validate! <view-name>)

Makes the view `<view-name>` valid.

(view/invalidate! <view-name>)

Makes the view `<view-name>` invalid.

(all-views)

Returns a list of all view names.

3 Text View

The text view is maintained in a valid state and is computed automatically on any program expression incorporated into ViewCode. The text view is always active. In this view, the source program is accessible as a `list`. The source program can be traversed in two directions; lexically down (i.e., using `car` and `cdr`) and lexically up (using `enclosing-sexp`, below). Upward traversal is an indispensable tool for the transform writer. For example, upward traversal allows transforms that are passed only a variable reference to determine the s-expression that contains the reference.

One exception exists in the list representation described above; certain s-expressions are converted to Mini-ViewForm *constants*. Constants include expressions that do not have identity or that are self evaluating. In Mini-ViewForm, constants are null lists, numbers, characters, booleans, symbols, quoted datum, and variables. Mini-ViewForm provides ways of testing and extracting a constant's value.

The Mini-ViewForm text view provides a way of replacing s-expressions with other s-expressions. Reserved symbols denoting special forms such as `define` and `if`, however, cannot be replaced. This restriction is an implementation artifact and will not be present in ViewForm. In addition, the VTS will interact with the expression replacement mechanisms to provide finer-grained control over view validation checking. ViewForm will also provide a mechanism for copying s-expressions. The mechanisms for explaining program modifications will interact with both program expression replacement and copying.

In order to assist a transform writer, ViewForm will also provide a variety of predicates and procedures in addition to those below that differentiate between and that parse the various kinds of Scheme expressions. In addition, ViewForm will provide various program walkers for traversing programs and unparsers for ViewCode.

`(combination? <exp>)`

Returns `#t` if `<exp>` is a combination (and not a special form) and `#f` otherwise.

`(special-form? <exp>)`

Returns `#t` if `<exp>` is a special form and `#f` otherwise.

`(variable-ref? <exp>)`

Returns `#t` if `<exp>` is a variable reference and `#f` otherwise.

`(constant? <exp>)`

Returns `#t` if `<exp>` is a constant and `#f` otherwise.

`(constant/value <const>)`

Returns the actual Scheme value of the Mini-ViewForm constant `<const>`. The return value can be tested with standard scheme primitives such as `symbol?`, `quote?`, and `null?`.

`(enclosing-sexp <exp>)`

Returns the s-expression that immediately contains `<exp>` (i.e., the s-expression that immediately lexically dominates `<exp>`). If `<exp>` is a top-level form, returns `<exp>`.

`(replace-exp! <orig-exp> <new-exp>)`

Replaces `<orig-exp>` with `<new-exp>`. `<orig-exp>` is an expression in the source program. `<new-exp>` is an s-expression that may be composed from program expressions and from newly constructed s-expressions. Mini-ViewForm constant-converts `<new-exp>` before substituting it in for `<orig-exp>`. Neither `<orig-exp>` nor `<new-exp>` may be a symbol representing a special form, such as `define` or `if`.

4 *Alpha View*

The alpha view performs an alpha conversion. Scheme primitive procedures are treated as variables bound to Mini-ViewForm objects that represent internal Scheme procedures. The user is given access to these objects.

(lookup-scheme-prim <scheme-prim>)

Returns a Mini-ViewForm representation of the internal procedure that <scheme-prim> is associated with. <scheme-prim> is a symbol (e.g., 'cons).

(primitive/name <prim>)

<prim> is the Mini-ViewForm representation of a Scheme primitive procedure. Returns the name (e.g., a symbol) of the primitive that <prim> represents.

(primitive? <object>)

Returns #t if <object> is the Mini-ViewForm representation of a Scheme primitive procedure. Returns #f otherwise.

(var-ref->var-bindings <var-ref>)

<var-ref> is a variable reference. Returns a list of variable bindings of <var-ref>. A variable reference is a variable in a position such that its value will be returned upon evaluation. A variable binding is a variable in an expression that defines or binds it.

(var-binding->var-refs <var-binding>)

<var-binding> is a variable binding. Returns a list of variable references to <var-binding>. A variable reference is a variable in a position such that its value will be returned upon evaluation. A variable binding is a variable in an expression that defines or binds it.

5 Data-Flow Analysis View

In the dfa view, the following information is collected on each program s-expression:

1. A set of expressions whose return value may be the value of the given expression.
2. A set of expressions whose return value may be the value returned by the given expression.

The expressions in (1) are separated into the following categories:

- A set of procedures that can potentially be the value of the given expression.
- A set of constants that can potentially be the value of the given expression.
- A set of value-type objects.

A value type object consists of a type and an expression that produces a value of that type. The expression is either a special form or a combination whose operator is a primitive.

```
(exp->consumers <exp>)
```

Returns a list of expressions that consume <exp>'s return value. A consumer has three properties: it refers to <exp>'s return value directly, it does not directly return <exp>'s value, and it is either a special form or a primitive combination. Examples of consumers include pair and procedure constructors, and arithmetic operations.

```
(expression->return-value-exps <exp>)
```

Returns a list of expressions whose return value can be <exp>'s return value.

```
(dfa/exp->procs <exp>)
```

Returns a list of the procedure expressions that can be <exp>'s value.

```
(dfa/exp->constants <exp>)
```

Returns a list of the constant expressions that can be <exp>'s value.

```
(dfa/exp->val-types <exp>)
```

Returns a list of value types that can be <exp>'s value.

```
(value-type/exp <value-type>)
```

Returns the expression that produces <value-type>'s value. A value-type represents both a value-producing expression and the type of value it produces.

Mini-ViewForm also supports the value-type predicates below. ViewForm will support a more complete set of value types.

```
(value-type/pair? <value-type>)

(value-type/stream? <value-type>)

(value-type/boolean? <value-type>)

(value-type/unspecified? <value-type>)

(value-type/type-unknown? <value-type>)
```

6 Transforms

Mini-ViewForm transforms are implemented as procedures that apply a series of rules. Thus, transforms can be combined and abstracted as procedures. As I have previously discussed, transforms in ViewForm will take on a different character. For example, they will contain an associated explanation.

(make-xform <proc>)

Returns a transform. <proc> is a procedure of one argument, a program expression, that applies a set of rules to a program expression(s). <proc> returns #t if successful, and #f otherwise.

(apply-xform <xform> <expression>)

Applies <xform> to <expression>. Returns #t if <xform> was successfully applied, and #f otherwise.

(combine-xforms-conjunctively <xform-1> <xform-2> ... <xform-n>)

Returns a compound transform that applies <xform-1> to a program expression. If <xform-1> is successful, <xform-2> is applied, and so on. If any <xform-i> is not successful, no further transforms are applied, and the compound transform returns #f. Otherwise, the compound transform returns the value returned by <xform-n>.

(combine-xforms-disjunctively <xform-1> <xform-2> ... <xform-n>)

Returns a compound transform that applies <xform-1> to a program expression. If <xform-1> is successful, the compound transform immediately returns <xform-1>'s return value. Otherwise, <xform-2> is applied, and so on. If no <xform-i> is successful, the compound transform returns #f.

7 Rules

Unlike ViewForm rules, which are simply transforms, Mini-ViewForm rules are data structures composed of two parts: a dispatcher and an action. A Mini-ViewForm transform applies rules to program expressions.

(make-rule <dispatcher> <action>)

Returns a rule composed of <dispatcher> and <action>.

(combine-rules-conjunctively <rule-1> <rule-2> ... <rule-n>)

Combines the rules <rule-1>, <rule-2>, ... <rule-n> by conjunctively combining their dispatchers (see combine-dispatchers-conjunctively) and by combining their actions.

(combine-rules-disjunctively <rule-1> <rule-2> ... <rule-n>)

Combines the rules <rule-1>, <rule-2>, ... <rule-n> by disjunctively combining their dispatchers (see combine-dispatchers-disjunctively) and by combining their actions.

(apply-rule <rule> <exp>)

Applies <rule> to the s-expression <exp>. Returns #t if the rule application was successful, and #f otherwise.

(apply-rules! <rules> <exps>)

<rules> is a list of rules, and <exps> is a list of s-expressions. For each s-expression, sequentially applies each rule until one is successful. Upon a successful rule, immediately begins the same process for the next s-expression, starting with the first rule.

8 *Dispatchers*

Unlike ViewForm dispatchers, which are just transforms, Mini-ViewForm dispatchers are different kinds of objects. Like ViewForm dispatchers, a Mini-ViewForm dispatcher decides whether or not an action is applicable at a given program expression.

(make-dispatcher <proc>)

Returns a dispatcher. <proc> is a procedure of one argument, a program expression. <proc> returns a non-#f value if the dispatcher accepts the program expression and #f otherwise.

(apply-dispatcher <dispatcher> <exp>)

Applies <dispatcher> to <exp> and returns <dispatcher>'s return value if the dispatcher accepts <exp>. Otherwise, returns #f.

(combine-dispatchers-conjunctively <disp-1> <disp-2> ... <disp-n>)

Returns a compound dispatcher that sequentially applies <disp-1>, <disp-2>, ... <disp-n> to a program expression. If any <disp-i> returns #f, the compound dispatcher immediately returns #f. Otherwise, the compound dispatcher returns the value returned by <disp-n>.

(combine-dispatchers-disjunctively <disp-1> <disp-2> ... <disp-n>)

Returns a compound dispatcher that sequentially applies <disp-1>, <disp-2>, ... <disp-n> to a program expression. The compound dispatcher returns #f if all <disp-i>'s return #f. Otherwise, the return value of the compound dispatcher is the value of the first <disp-i> to return a non-#f value.

9 Actions

Again, unlike ViewForm actions, which are just transforms, Mini-ViewForm actions are different kinds of objects. Like ViewForm actions, a Mini-ViewForm action unconditionally applies a program modification at whatever program expression it is given. Also, Mini-ViewForm actions take as an argument the result returned by its dispatcher. In ViewForm, a more flexible mechanism is provided for passing values from one transform to another.

`(make-action <action-proc>)`

Returns an action. `<action-proc>` is a procedure that takes two arguments; an s-expression, and a dispatcher result. `<action-proc>` modifies the program based on its arguments.

`(apply-action! <action> <exp> <dispatcher-result>)`

Performs `<action>`'s program modification given a program expression and the result from `<action>`'s associated dispatcher. The actions may be automatically queued to be committed or discarded later. If queued and then committed, the actions will be committed in the same order as they would have been had they not been queued.

`(queue-actions!)`

Queues action applications.

`(dont-queue-actions!)`

Stops queuing action applications.

`(clear-queued-actions!)`

Discards any queued action applications.

`(commit-queued-actions!)`

Commits then discards each queued action application.

`(combine-actions <action-1> <action-2> ... <action-n>)`

Returns a compound action that performs all the modifications, in sequence, performed by `<action-1>` `<action-2>` ... `<action-n>`.

10 Mini-ViewForm Run-Time System

ViewForm will provide an extensible protocol for interacting with a user or an agent. Transforms will be able to provide more specific interaction protocols. An agent will be able to fully interact with a transform if it understands the transform's interaction protocol.

(discard-ignore-continue-options <exp> <mesg> <arg-1> <arg-2> ... <arg-n>)
Presents the user with a set of options to choose from. The options are: whether to discard all changes made, whether to ignore the expression <exp>, or whether to perform an action explained in <mesg>. <mesg> is a format-based string and the <arg-i>'s are the format-based arguments for <mesg>.

(ask-user <options> <message> <arg-1> <arg-2> ... <arg-n>)
Outputs a message, presents the user with a set of options, and returns the option selected by the user. <options> is a list of options (see make-option). <message> and <arg-1>, <arg-2> ... <arg-2> are a format-based message and arguments, respectively.

(make-option <message> <arg-1> <arg-2> ... <arg-n>)
Returns a user interaction option. <message> and <arg-1>, <arg-2> ... <arg-n> are a format-based message and arguments, respectively that are displayed when the option is presented to the user.

(initialize-Mini-ViewForm!)
Initializes Mini-ViewForm.

(output <program>)
Unparses and outputs an executable representation of <program> in the source language.

Appendix II: Results with Transforming n-queens

The code for `n-queens` is given in Figure 1-1. First, I show the results of running the original program, referred to as `n-queens-test`. Next, I apply the transform and then I run the target program. At the end, I show the transformed program.

```
(run n-queens-test)  ;this is evaluated in Scheme.  The output is below.

Number of Solutions: 10

*****
----Q
-Q---
---Q-
Q----
--Q--
*****

*****
----Q
--Q--
Q----
---Q-
-Q---
*****

*****
--Q--
----Q
-Q---
---Q-
Q----
*****

*****
-Q---
----Q
--Q--
Q----
---Q-
*****

*****
---Q-
-Q---
----Q
--Q--
Q----
*****

*****
Q----
--Q--
----Q
-Q---
---Q-
*****
```

```
*****
---Q-
Q----
--Q--
----Q
-Q---
*****

*****
Q----
---Q-
-Q---
----Q
--Q--
*****

*****
-Q---
---Q-
Q----
--Q--
----Q
*****

*****
--Q--
Q----
---Q-
-Q---
----Q
*****
;Value 209: (((4 . 1) (2 . 2) (5 . 3) (3 . 4) (1 . 5)) ((3 . 1) (5 . 2) (2 .
3) (4 . 4) (1 . 5)) ((5 . 1) (3 . 2) (1 . 3) (4 . 4) (2 . 5)) ((4 . 1) (1 . 2)
(3 . 3) (5 . 4) (2 . 5)) ((5 . 1) (2 . 2) (4 . 3) (1 . 4) (3 . 5)) ((1 . 1) (4
. 2) (2 . 3) (5 . 4) (3 . 5)) ((2 . 1) (5 . 2) (3 . 3) (1 . 4) (4 . 5)) ((1 .
1) (3 . 2) (5 . 3) (2 . 4) (4 . 5)) ((3 . 1) (1 . 2) (4 . 3) (2 . 4) (5 . 5))
((2 . 1) (4 . 2) (1 . 3) (3 . 4) (5 . 5)))
;;;
;;; Now, let's run the transform
;;;
(test n-queens-test)                 ;Apply the transform to the n-queens code
Initializing...
Enclosing code...
Rewriting code...
Computing alpha View...
Computing dfa View...
Applying XFORM...
The value being changed is passed out of the program scope. If you choose to
continue, it is possible that external procedures may be passed an argument
type they are not expecting.
 Options:
1) Discard all changes made and abort.
2) Ignore expression (cons380 (cons380 new-queen127 partial-solution122)
solutions121) and continue.
3) Perform the replacement
3                                    ;User types this in
The value being changed is passed out of the program scope. If you choose to
continue, it is possible that external procedures may be passed an argument
type they are not expecting.
 Options:
1) Discard all changes made and abort.
2) Ignore expression '() and continue.
```

```
3) Perform the replacement
3                                   ;User types this in
A call to length has been found on the list being converted to a stream type.
Taking the length of a stream may not be desirable.
 Options:
1) Discard all changes made and abort.
2) Ignore expression (length376 solutions111) and continue.
3) Perform the replacement
3                                   ;User types this in, though the user
Invalidating alpha View...          ;should change the code to not use length
Invalidating dfa View...
;No value

(run n-queens-test)                 ;Let's run our result

Number of Solutions: 10

*****
----Q
-Q---
---Q-
Q----
--Q--
*****
;Value 210: (((4 . 1) (2 . 2) (5 . 3) (3 . 4) (1 . 5)) . #[promise 211])
;;;
;;; This is the transformed code.
;;;
(pp n-queens-test)                  ;pretty print out the results
(letrec ((make-queen108 cons302)
         (queen-row107 car278)
         (queen-col106 cdr269)
         (consistent?105
          (lambda (new-queen135 consistent-solution134)
            (let ((queens-conflict?136
                   (lambda (queen137)
                     (or (=275 (queen-row107 new-queen135)
                               (queen-row107 queen137))
                         (=275 (queen-col106 new-queen135)
                               (queen-col106 queen137))
                         (=275 (abs283 (-299 (queen-row107 new-queen135)
                                             (queen-row107 queen137)))
                               (abs283 (-299 (queen-col106 new-queen135)
                                             (queen-col106 queen137))))))))
              (not295 (list-search-positive281 consistent-solution134
                                               queens-conflict?136)))))
         (find-new-queen-in-col104
          (lambda (row130 col129 partial-solution128)
            (let loop132
              ((row131 row130))
              (cond ((zero?303 row131) '())
                    (#t
                     (let ((new-queen133 (make-queen108 row131 col129)))
                       (if (consistent?105 new-queen133 partial-solution128)
                           new-queen133
                           (loop132 (-1+288 row131)))))))))
         (n-queens103
          (lambda (n120)
            (let loop125 ((col124 n120)
                          (rows123 (list271 n120))
                          (partial-solution122 '())
                          (solutions121 (stream)))              ;changed
              (let* ((row126 (car278 rows123))
```

```
                  (new-queen127 (find-new-queen-in-col104
                                  row126
                                  col124
                                  partial-solution122)))
             (cond ((and (=275 col124 n120) (zero?303 row126))
                    solutions121)
                   ((null?296 new-queen127)
                    (loop125 (1+308 col124)
                             (cdr269 rows123)
                             (cdr269 partial-solution122)
                             solutions121))
                   ((=275 col124 1)
                    (loop125 col124
                             (cons302 (-1+288 (queen-row107 new-queen127))
                                      (cdr269 rows123))
                             partial-solution122
                             (cons-stream (cons302 new-queen127 ;changed
                                                   partial-solution122)
                                          solutions121)))
                   (#t
                    (loop125 (-1+288 col124)
                             (cons*301 n120
                                       (-1+288 (queen-row107 new-queen127))
                                       (cdr269 rows123))
                             (cons302 new-queen127 partial-solution122)
                             solutions121)))))))
       (display-solution102
        (lambda (solution113)
          (let* ((sort-by-row116
                  (lambda (queen1115 queen2114)
                    (<266 (queen-row107 queen1115)
                          (queen-row107 queen2114))))
                 (sorted117 (sort273 solution113 sort-by-row116))
                 (display-next-row119
                  (lambda (queen118)
                    (format287 #t "~A~%"
                     (string-pad-right272
                      (string-pad-left305 "Q" (queen-col106 queen118) #\-)
                      (length279 sorted117)
                      #\-)))))
            (for-each268 display-next-row119 sorted117))))
       (display-n-queens101
        (lambda (n109)
          (let* ((delimiter110 (make-string293 n109 #\*))
                 (solutions111 (n-queens103 n109)))
            (begin
             (format287 #t "~%~%~A~A~%"
                           "Number of Solutions: "
                           (stream-length solutions111))         ;changed
             (stream-map                                          ;changed
              solutions111
              (lambda (soln112)
                (begin (format287 #t "~%~A~%" delimiter110)
                       (display-solution102 soln112)
                       (format287 #t "~A~%" delimiter110))))
             solutions111)))))
  (display-n-queens101 5))
;No value
```

Appendix III: Transform Code

```
;;;
;;; The Mini-ViewForm code implementing a transform that changes the
;;; return type of a program expression.
;;;
;;; By: Luis H. Rodriguez Jr.
;;; Copyright 1994, 1995  All Rights Reserved
;;;

(declare (usual-integrations))

;;;
;;; We use test to apply the list->stream transform to an expression.
;;;

(define (test exp)
  (initialize-Mini-ViewForm!)
  (let ((xform (make-list->stream-xform exp)))
    (apply-xform xform exp)))

;;;
;;; Here, I make our transform.  It can be applied to either
;;; <program> or to one of <program>'s sub-expressions.  The transform
;;; consults the user when it encounters a value that escapes from the
;;; scope of the dfa view.
;;;

(define (make-list->stream-xform program)
  (let ((rules (construct-list->stream-rules)))
    (make-change-return-type-xform program
                                   (map add-escape-rule rules)
                                   rules)))

;;;
;;; make-change-return-type-xform takes a program, a set of rules to
;;; apply to producers, and a set to apply to consumers.  It returns
;;; a transform that applies the producer and consumer rules to primary
;;; and secondary producers and consumers of <expression>,
;;; respectively.  It also automatically queues applicable actions to
;;; provide the user with an opportunity to abort without having made
;;; any changes to the program.
;;;

(define (make-change-return-type-xform program
                                       producer-rules
                                       consumer-rules)
  (make-xform
   (lambda (expression)
     (let ((xform-thunk
            (lambda ()
              (let* ((prod-con (producers-consumers expression))
                     (producer-sites (car prod-con))
                     (consumer-sites (cdr prod-con)))
                (apply-rules! producer-rules producer-sites)
                (apply-rules! consumer-rules consumer-sites)
                (commit-queued-actions!)
                #t))))
       (with-views '((alpha invalidate) (dfa invalidate))
                   program
                   xform-thunk)))))
```

```
;;;
;;; Here are the rules for the list->stream transform.
;;;

(define (construct-list->stream-rules)
  `(,@(map
       make-prim-subst-rule
       '(cons         car        cdr        pair?        null?)
       '(cons-stream  stream-car stream-cdr stream-pair? stream-null?))

    ,@(map make-prim-subst-rule
           '( length                 list-ref                )
           '( stream-length          stream-ref              )
           `(,ask-user-about-length ,ask-user-about-list-ref))

    ,(make-rule (combine-dispatchers-conjunctively
                 (conjunctive-prim-match 'map 'for-each)
                 (combine-dispatchers-disjunctively ensure-two-arg-comb
                                                    discard-changes))
                (combine-actions
                 (make-replacer replace-operator! 'stream-map)
                 reverse-args!))

    ,(make-rule null-list?
                (make-replacer replace-entire-exp! '(stream)))

    ,(make-rule discard-changes continue) ;includes reverse and append

    ))

;;;
;;; add-escape-rule takes a rule, and combines it with another that
;;; ensures that an expression's value does not escape out of the scope
;;; of what the dfa knows about.  If it does, the user is prompted.
;;;

(define (add-escape-rule rule)
  (combine-rules-conjunctively
   rule
   (make-rule
    (combine-dispatchers-conjunctively value-escapes?
                                       ask-user-about-passing-out)
    continue)))

;;;
;;; We combine these two dispatchers and two actions with others.
;;;

(define discard-changes
  (make-dispatcher
   (lambda (expression)
     (format #t "~%Changes not committed: No rule for handling: ~S"
             expression)
     (discard-queued-actions!)
     #f)))

(define ensure-two-arg-comb
  (make-dispatcher
   (lambda (expression)
     (let ((match-exp (->comb-expression expression)))
       (and (combination? match-exp)
            (= 2 (length (combination/args match-exp))))))))
```

```
(define continue
  (make-action
   (lambda (expression matcher-result)
     matcher-result                        ;ignored
     expression                            ;ignored
     #f)))

(define reverse-args!
  (make-action
   (lambda (expression matcher-result)
     matcher-result                        ;ignored
     (let ((match-exp (->comb-expression expression)))
       (if (combination? match-exp)
           (replace-exp! match-exp
                         `(,(combination/operator match-exp)
                           ,@(reverse (combination/args match-exp))))
           (error "~S not a combination: reverse-args! "
                  expression))))))

;;;
;;; make-prim-subst-rule makes rules that replace calls to primitive
;;; procedures with calls to other procedures.  other-dispatcher
;;; is a way to introduce other conditions such as user interaction
;;; into the rule's dispatcher's decision-making process.
;;;

(define (make-prim-subst-rule orig-proc new-proc . other-dispatcher)
  (make-rule (apply combine-dispatchers-conjunctively
                    (conjunctive-prim-match orig-proc)
                    other-dispatcher)
             (make-replacer replace-operator! new-proc)))

;;;
;;; conjunctive-prim-match makes a dispatcher that returns a non-#f
;;; value under one of two conditions; the first is that the only
;;; procedures that can reach a combination site's operator can be
;;; found in the list of primitives <prims> (represented as symbols),
;;; and the second is that the user agrees to proceed if the first
;;; condition does not hold.
;;;

(define (conjunctive-prim-match . prims)
  (make-conjunctive-prim-matcher ask-user-partial-match prims))

;;;
;;; make-conjunctive-prim-matcher returns a dispatcher that is applied to a
;;; combination or to one of its variable arguments. The dispatcher returns #t
;;; if all the primitives listed in <prims> reach the combination's operator.
;;; <partial-match> called when some but not all of the primitives in <prims>
;;; reach a combination's operator.
;;;

(define (make-conjunctive-prim-matcher partial-match prims)
  (let ((prim-procs (map lookup-scheme-prim prims)))
    (make-dispatcher
     (lambda (expression)
       (let ((match (match-prims expression prim-procs)))
         (cond ((or (not match)
                    (null? prims)
                    (null? (car match))) ;no matching prims found
                #f)
```

```
              ((not (null? (cdr match))) ;not all prims matched
               (partial-match (cdr match) expression prims))
              (else #t)))))))

;;;
;;; match-prims looks at the dfa view's information to obtain a list of
;;; procedures that can be the values of a combination's operator.  It
;;; takes these procedures and separates them into two lists.  It
;;; returns a pair of the two lists.  The first list contains the
;;; procedures that are eq? to any procedure in <prim-procs>, and the
;;; second consists of those procedures that aren't in the first list.
;;;

(define (match-prims expression prim-procs)
  (let ((matches-prims? (lambda (proc)
                          (memq proc prim-procs)))
        (comb-exp       (->comb-expression expression)))
    (cond ((combination? comb-exp)
           (let ((proc-vals (dfa/exp->procs
                             (combination/operator comb-exp))))
             (cons (list-transform-positive proc-vals matches-prims?)
                   (list-transform-negative proc-vals matches-prims?)
                   )))
          (else #f))))

;;;
;;; These are procedures for making replacers.  A replacer is an action
;;; that replaces a program expression. <replacer!> controls the scope
;;; of the expression to be replaced. replace-entire-exp! replaces an
;;; entire expression, whereas replace-operator! replaces the operator
;;; in a combination it is given.  The latter may also be passed, as
;;; the <old-exp>, a variable that is an argument to the combination.
;;;

(define (make-replacer replacer! replacement-expression)
  (make-action
   (lambda (expression matcher-result)
     matcher-result                     ;ignored
     (replacer! expression replacement-expression))))

(define (replace-entire-exp! old-exp new-exp)
  (replace-exp! old-exp new-exp))

(define (replace-operator! exp new-operator)
  (let ((comb (->comb-expression exp)))
    (replace-exp! (combination/operator comb) new-operator)))

;;;
;;; User interaction dispatchers
;;;

(define (ask-user-about-proc warning)
  (make-dispatcher
   (lambda (expression)
     (discard-ignore-continue-options (->comb-expression expression)
                                      warning))))

;;;
;;; Three user interaction dispatchers that are used in the
;;; list->stream transform.
;;;

(define ask-user-about-length
```

```
  (ask-user-about-proc
   "A call to length has been found on the list being converted to ~
    a stream type.  Taking the length of a stream may not be ~
    desirable."))


(define ask-user-about-list-ref
  (ask-user-about-proc
   "A call to list-ref has been found on the list being converted to ~
    a stream type.  Applying stream-ref to such a stream may not be ~
    desirable."))

(define ask-user-about-passing-out
  (make-dispatcher
   (lambda (expression)
     (discard-ignore-continue-options
      expression
      "The value being changed is passed out of the program scope. ~
       If you choose to continue, it is possible that external ~
       procedures may be passed an argument type they are not ~
       expecting."))))

;;;
;;; One last user interaction procedure...
;;;

(define (ask-user-partial-match procs-not-found expression matchees)
  (discard-ignore-continue-options
   expression
   "Non-matching values can reach expression ~S.~%  The ~
    non-matching values are: ~S.~%  The current transformation rule ~
    applies only to the following types of expressions: ~S~%"
   expression
   procs-not-found
   matchees))

;;;
;;; Misc. procedures
;;;

;;;
;;; ->comb-expression is used to obtain a combination expression if
;;; <exp> is a variable and is one of the combination's arguments.
;;;

(define (->comb-expression exp)
  (if (variable-ref? exp)
      (enclosing-sexp exp)
      exp))

(define-integrable combination/operator car)
(define-integrable combination/args     cdr)
```

Bibliography

[1] **Steven Adams** *Personal Communication* January 1995.

[2] **David R. Barstow** *An Experiment in Knowledge-Based Automatic Programming* In *Artificial Intelligence* Volume 12, Pages 73-119, 1979.

[3] **Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper** *Formal Program Construction by Transformations–Computer Aided, Intuition-Guided Programming* In *IEEE Transactions on Software Engineering* 15(2), February 1989.

[4] **Manfred Broy and Peter Pepper** *Program Development as a Formal Activity* In *IEEE Transactions on Software Engineering* 7(1), Pages 14-22, January 1981.

[5] **David R. Chase** *An Improvement to Bottom-Up Tree Pattern Matching* In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages* München, W. Germany, January 1987.

[6] **Marina Chen** *A Parallel Language and its Compilation to Multiprocessor Machines or VLSI,* In *Proceedings of the Thirteenth Symposium on Principles of Programming Languages* Pages 131-139, 1986.

[7] **Marina Chen, Young-il Choo and Jingke Li** *Compiling Parallel Programs by Optimizing Performance,* In *The Journal of Supercomputing* 2(2), Pages 171-207, October 1988.

[8] **Thomas E. Cheatham, Jr.** *Reusability Through Program Transformations* In *IEEE Transactions on Software Engineering* 10(5), Pages 589-594, September 1984.

[9] **Anir Chowdhury and Scott Meyers** *Facilitating Software Maintenance by Automated Detection of Constraint Violations* In *IEEE Conference on Software Maintenance* Montreal, Quebec, September 1993.

[10] **James R. Cordy and Ian. H. Carmichael** *The TXL Programming Language Syntax and Informal Semantics: Version 7* Department of Computing and Information Science, Queens University at Kingston, Canada, June 1993.

[11] **James R. Cordy and Eric Promislow** *Specification and Automatic Prototype Implementation of Polymorphic Objects in TURING Using the TXL Dialect Processor* In *Proceedings of the IEEE International Conference on Computer Languages* New Orleans, March 1990.

[12] **James R. Cordy and Medha Shukla** *Practical Metaprogramming* External Technical Report 92-342, Department of Computing and Information Science, Queens University at Kingston, Canada, October 1992.

[13] **John Darlington** *An Experimental Program Transformation and Synthesis System* In *Readings in Artificial Intelligence and Software Engineering* Charles Rich and Richard C. Waters, eds., Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[14] **E. W. Dijkstra** *A Discipline of Programming* Prentice-Hall, Englewood Cliffs, NJ, Page 64, 1976.

[15] **Carolyn K. Duby, Scott Meyers, and Steven P. Reiss** *CCEL: A Metalanguage for C++* Technical Report CS-92-51, Brown University, October 1992.

[16] **Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa** *Demand-Driven Computation of Interprocedural Data-Flow* In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* San Francisco, CA, Pages 37-48, January 1995.

[17] **Charles Donald Farnum** *Pattern-Based Languages for Prototyping of Compiler Optimizers* Technical Report Number UCB/CSD 90/608, University of California, Berkeley, December 1990.

[18] **Charles Farnum** *Dora - An Environment for Experimenting with Compiler Optimizers* In *Proceedings of the 1992 International Conference on Computer Languages* Oakland, CA, April 1992.

[19] **Charles Farnum** *Pattern-Based Tree Attribution* In *Proceedings of the Nineteenth Annual Symposium on Principles of Programming Languages* Albuquerque, New Mexico, January 1992.

[20] **Charles Farnum** *Personal Communication* April 1995.

[21] **Martin S. Feather** *Constructing Specifications by Combining Parallel Elaborations* Technical Report ISI/ES-88-216, Information Sciences Institute, University of Southern California, August 1988.

[22] **Robert W. Floyd** *Toward Interactive Design of Correct Programs* IFIP, Ljubljana, Yugoslavia, Pages 7-11, August, 1971.

[23] **David Garlan** *Views for Tools in Integrated Environments* Ph.D. Dissertation, Carnegie Mellow University, May 1987. Also appears as CMU Technical Report CMU-CS-87-147.

[24] **David Garlan, Linxi Cai, Robert L. Nord, and Robert Stockton** *ASCENT: Application-Specific Environment Transformer* Technical Report CMU-CS-92-180 Carnegie Mellon University, 1992.

[25] **F. Giannesini, H. Kanoui, R. Pasero, and M. van Caneghem** *Prolog* Addison-Wesley, 1986.

[26] **David Gries and Dennis Volpano** *The Transform–A New Language Construct* Technical Report TR 89-1045, Department of Computer Science, Cornell University, October 1989.

[27] **A.N. Habermann, Charles Krueger, Benjamin Pierce, Barbara Staudt, and John Wenn** *Programming with Views* Technical Report CMU-CS-87-177, Carnegie Mellon University, January 1988.

[28] **C.M. Hoffman and M.J. O'Donnell** *Pattern Matching in Trees* In *Journal of the ACM* 29(1), pages 68-95, January 1982.

[29] **IEEE Std 1178-1990** *IEEE Standard for the Scheme Programming Language* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[30] **Elaine Kent** *On the Efficient Synthesis of Efficient Programs* In *Artificial Intelligence* Volume 20, Pages 253-305, 1983.

[31] **Gregor Kiczales** *Towards a New Model of Abstraction in the Engineering of Software* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 1-11, Tokyo, Japan, November 1992. A revised version is also available at: http://www.xerox.com/PARC/spl/eca/oi/gregor-invite.html.

[32] **Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez Jr., Amin Vahdat, and Daniel G. Bobrow** *Metaobject Protocols: Why We Want Them, and What Else They Can Do* In *Object-Oriented Programming: The CLOS Prospective* Andreas Paepcke, Ed., MIT Press, Cambridge, MA, Pages 101-118, 1993.

[33] **Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow** *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, 1991.

[34] **Gregor Kiczales, John Lamping, Luis H. Rodriguez Jr., and Erik Ruf** *Macros that Reach Out and Touch Somewhere* Internal Technical Report, Embedded Computation Area, Xerox PARC, August 1992.

[35] **Gregor J. Kiczales and Luis H. Rodriguez Jr.** *Efficient Method Dispatch in PCL*, In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* Nice, France, Pages 99-105, 1990. Also In *Object-Oriented Programming: The CLOS Prospective*, Andreas Paepcke, Ed., MIT Press, Cambridge, MA, Pages 335-348, 1993.

[36] **Thomas Kunz** *Distributed Debugging - A Case Study* Technical Report TI-3/92. Institut für Theoretische Informatik, Technische Hochschule Darmstadt, April 1992.

[37] **John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf** *An Architecture for an Open Compiler* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 95-106, Tokyo, Japan, November 1992.

[38] **Andrew Malton** *The Denotational Semantics of a Functional Tree-Manipulation Language* Computer Languages, 19(3), pages 157-168, 1993.

[39] **John McCarthy** *A Basis for a Mathematical Theory of Computation* In *Computer Programming and Formal Systems* P. Braffort and D. Hirschberg, Eds., North-Holland, 1967.

[40] **Scott Meyers and Steven P. Reiss** *A System for Multiparadigm Development of Software Systems* Technical Report CS-91-50, Department of Computer Science, Brown University, August 1991.

[41] **Scott Meyers and Steven P. Reiss** *An Empirical Study of Multiple-View Software Development* In *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, Tyson's Corner, VA, November 1992. Also in *ACM SIGSOFT Software Engineering Notes*, 17(5), November 1992.

[42] **Robin Milner, Mads Tofte, and Robert Harper** *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[43] **Mark S. Moriconi** *A Designer/Verifier's Assistant* In *IEEE Transactions on Software Engineering* 5(4), Pages 387-401, July 1979.

[44] **David S. Notkin** *Interactive Structure-Oriented Computing* Ph.D. Dissertation, Carnegie Mellow University, February, 1984. Also appears as CMU Technical Report CMU-CS-84-103.

[45] **Open Implementation** *OI Homepage* http://www.xerox.com/PARC/spl/eca/oi.html.

[46] **Open Implementation** *Record of the Workshop on Open Implementation*, Gleneden Beach, Oregon, October 1994. http://www.xerox.com/PARC/spl/eca/oi/workshop-94/default.html

[47] **Steven P. Reiss** *On the Use of Annotations for Integrating the Source in a Program Development Environment* Technical Report CS-91-32, Department of Computer Science, Brown University, April 1991.

[48] **Steven P. Reiss** *PECAN: Program Development Systems that Support Multiple Views* In *IEEE Transactions on Software Engineering* 11(3), March 1985.

[49] **Charles Rich and Richard C. Waters** *The Programmer's Apprentice* ACM Press, New York, NY, 1990.

[50] **Charles Rich and Richard C. Waters, Eds.** *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[51] **Luis H. Rodriguez Jr.** *Coarse-Grained Parallelism Using Meta-Object Protocols* S.M. Thesis, Massachusetts Institute of Technology, September 1991. Also Appears as Technical Report SSL-91-06, Xerox PARC, September 1991.

[52] **Luis H. Rodriguez Jr.** A *Study on the Viability of a Production-Quality Metaobject Protocol-Based Statically Parallelizing Compiler* In *Proceedings of the International Workshop on New Models for Software Architectures: Reflection and Meta-Level Architecture* Pages 107-112, Tokyo, Japan, November 1992.

[53] **Guillermo J. Rozas** *LIAR: an ALGOL-like Compiler for SCHEME* S.B. Thesis, Massachusetts Institute of Technology, June 1984.

[54] **Guillermo Rozas** *Translucent Procedures, Abstraction without Opacity* Ph.D. Dissertation, Massachusetts Institute of Technology, October 1993. Also appears as MIT Technical Report AI-TR-1427.

[55] **Olin Shivers** *Control-Flow Analysis of Higher-Order Languages or Taming Lambda* Ph.D. Dissertation, Carnegie Mellow University, May 1991. Also appears as Technical Report CMU-CS-91-145.

[56] **Andrew Stark** *UROP Project in Progress*, MIT Artificial Intelligence Laboratory, January 1995-present.

[57] **L. Sterling and E. Shapiro** *The Art of Prolog* MIT Press, 1981.

[58] **Elizabeth K. Turrisi** *Chapter and Verse Program Description* MIT AI Lab Working Paper 256, June 1984.

[59] **Jan L. A. Van de Snepscheut** *Proxac: An Editor for Program Transformation* Technical Report Caltech-CS-TR-93-33, California Institute of Technology 1993.

[60] **Richard C. Waters** *The Programmer's Apprentice: A Session with KBEmacs* In *IEEE Transactions on Software Engineering* 11(11), Pages 1296-1320, November 1981.

[61] **Deborah Whitfield and Mary Lou Soffa** *Automatic Generation of Global Optimizers* In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* Toronto, Ontario, CA, June 1991.

[62] **J. Allen Yang and Young-il Choo** *Meta-Crystal—A Metalanguage for Parallel-Program Optimization* Technical Report YALEU/DCS/TR-786, Yale University, April 1990.

[63] **J. Allen Yang and Young-il Choo** *Parallel-Program Transformation Using a Metalanguage*, In *Proceeding of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* Pages 11-20, 1991.

[64] **Ramin Zabih** *Dependency-Directed Backtracking in Non-Deterministic Scheme*, S.M. Thesis, Massachusetts Institute of Technology, August 1988. Also appears as Technical Report AI-TR-956.