Transformational Derivation of Programs Using the Focus system*

Uday S. Reddy
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Net address: reddy@a.cs.uiuc.edu

September 2, 1988

1 Introduction

Transformational derivation, formal derivation of programs using the paradigm of program transformation, has been known for several years [3, 20, 34]. But, its practical use in program development is still elusive. Hand derivation of any significant program is fraught with long sequences of low-level formal manipulations which are tedious and error-prone. The practicality of the program derivation method thus depends on automated support for such manipulations. We are constructing a program derivation support system called Focus, designed along the following guiding principles:

- The system must be *interactive*. It should let the user make the program *design decisions* and handle the *formal* details.
- The interaction with the system must be at a sufficiently *high-level.*
- The system must be *flexible.* It should not impose an overly restrictive discipline on the user regarding how programs should be derived.
- The automation of the formal details must be in such a way that the behavior of the system can be easily *predicted* and understood.
- The *search spaces* involved in the automated transformation operations must be reasonably small, so that the system does not produce a large number of choices as a result of any operation.

*This paper is to appear in the Proceedings of Third Symposium on Software Development Environments, ACM, 1988.

This research was supported in part by NASA grant NAG-1-613.

- The system must be able to construct *derivation histories*, summary descriptions of program derivations, which can be used by the programmer for understanding program designs.
- The system must also be able to construct *derivation scripts*, detailed records of derivation activity, which the system can use to automatically rederive a program for an altered specification.

In this paper, we discuss

1. the *integration* of validation and program derivation activities in the Focus system,
2. its *tree-based* user interface,
3. the control of *search spaces* in program derivation, and
4. the *structure* and organization of program derivation records.

Each of these issues is introduced below.

Validation and derivation

Validation is the activity of verifying if a specification (or an implementation) meets the intended behavior of the system. It requires, first of all, that the specification language be "executable" so that we can run test cases. A more effective method of validation is *symbolic execution* of specifications which allows the observation of all possible situations, rather than a few test cases [4, 5].

The symbolic execution method in Focus is based on *logic programming* principles [6, 19, 26]. Logic programming, through the use of *unification* [33], can effectively solve for the values of variables which make specific execution paths possible. We show that this is useful not only in validation, but in program derivation as well. During derivation, we have to consider the behavior of specifications in *all* possible situations, and construct efficient procedures in each case.

Yet another method of validation is *verification* (proof) of properties of programs. This too is an activity closely related to derivation. In fact, we need to routinely prove properties of program components and use them in *optimizing* programs. Thus, validation, verification and program derivation activities can be – indeed, need to be – integrated into a single system.

Tree editor interface

Conventionally, tree-based editors have been associated with syntax trees (see, for example, [36]). But, trees arise in other contexts as well. Program executions are trees with the hierarchy determined by procedure calls. Symbolic executions have another kind of tree structure: each subtree denotes an assumption on the symbolic values. Program derivations have a tree structure too. In deriving a program component, we derive its internal components as subtrees.

A flexible and programmable tree editor called Treemacs has been developed at University of Illinois to be applicable in such different contexts [16, 17]. As the name suggests, the tree editor has been built as an extension of the Emacs editor and inherits its programmability and the ability to interact with the operating system environment. Focus system uses this editor as its user interface, so that the user can structure program derivations as trees and develop them as he sees fit. This interface also allows us to integrate all the tools required in program development, such as compilers and debugging tools.

Search spaces in program derivation

One of the big stumbling blocks for the use of formal derivation tools in large problems is the *search space*. An examination of the transformation methodology of Burstall and Darlington [3] shows that, at each stage of the transformation, there are choices involved in what transformation operation to use, what part of the program (expression) to operate on, and what program or property rule to use for the operation. Unlike in other areas of problem solving, program derivation is often open-ended, i.e., there is no clear *a priori* goal. Various possibilities have to be tried and examined. The open-endedness means that there are usually no criteria that the system can employ to eliminate alternatives. But, if the user is swamped with a large number of alternatives to choose from, the utility of the automated tools is greatly diminished. The conventional solutions to this problem have been to force the derivation activity to be goal-directed or to give the user more control [9, 13, 40]. Though effective in cutting down the search space, these approaches increases the burden on the user. We show here that a few simple notions of efficiency can be used to direct the activity of the system so that the search space can be greatly reduced; in fact, unique results can be produced in many cases.

Recording derivation activity

The possibility of recording derivation activity and using it for automatic rederivation from modified specifications is the most exciting benefit of formal program derivation methods [1, 40]. However, the entire derivation record of a prior derivation does not usually work for a modified specification. There are bound to be components that are affected by a modification. Thus, it is necessary to isolate the components for which the rederivation *breaks*. It may be seen that structuring derivations as long sequences of transformations is not going to fit the bill. The organization of derivations as trees (as discussed earlier) was motivated in part by the desire to achieve such isolation.

In systems based on explicit user control of derivations [9, 13, 40], the derivation activity can be recorded by merely storing the commands issued by the user. But, in a more automated system like Focus, the system has to keep track of its own activity and represent all the dependencies between the derivations of components. We discuss how the dependencies are represented in the Focus system and used for propagation of changes.

2 Specifications in Focus

The language used in the Focus system for writing specifications (as well as programs) is a combination of functional programming and logic programming languages [25, 27]. All program derivations are carried out within this language and the derived programs are then translated into conventional programming languages (such as Lisp, C, and Ada). So, in the spirit of Larch [15], we call it the *Focus shared language.* It provides higher-order functions, logical variables, quantifiers and abstract data types. The primitive data structures are recursive data structures, arrays and collections (often called "sets"). We are also investigating the use of imperative features like assignments, input/output and device control operations [30]. The focus of this paper being on the methodology rather than the language, we limit ourselves to a first-order functional subset.

```
use
    type char
    function alpha : char → bool
        /* check if a character is alphanumeric */
definitions
    type list(t) = Nil + t.list(t)
    type word = list(char)

    function scan : list(char) → list(word)
    scan(Nil) → Nil
    scan(char.s) → fword(char.s).scan(rword(char.s))

    function fword : list(char) → word
        /* the first word */
    fword(Nil) → Nil
    alpha(c) :: fword(c.s) → c.fword(s)
    fword(' '.s) → Nil

    function rword : list(char) → list(char)
        /* remainder of the first word */
    rword(Nil) → Nil
    alpha(c) :: rword(c.s) → rword(s)
    rword(' '.s) → skip(s)

    function skip : list(char) → list(char)
        /* skip spaces */
    skip(Nil) → Nil
    alpha(c) :: skip(c.s) → c.s
    skip(' '.s) → skip(s)

properties none
```

Figure 1: Specification of scan

Figure 1 shows an example specification of a simple *lexical scanner* in this subset (taken from Feather [8, 12] with some modifications). The input to the scanner is a list of characters. Each character is either an alphanumeric character, (for which alpha(c) is true), or a space. The output of the scanner should be the list of words in the input, where a *word* is a consecutive sequence of alphanumeric characters separated by spaces.

A few aspects of the specification are to be noted. All the rules in the specification are *rewrite rules*, i.e., oriented equations. There are two parts in the specification: *definitions* and *properties*. The definition part defines the main function being specified together with other auxiliary functions used. The properties part may specify any simple properties that these functions satisfy. Other than defined functions, the specification also involves *constructors* (such as Nil and '.') used to build data structures. The *formal parameters* of functions, i.e., terms t_i in the left hand sides of definition rules $f(t_1, \ldots, t_n) \rightarrow e$, can only be constructor terms. (They cannot have defined functions). Definition rules of this form are called *constructor-based* [23] and are similar to those used in functional languages like Standard ML [22] and Miranda [38]. Methods for translating them to conventional code may be found in [24].

Rewrite rules can be *conditional* rules, such as the second definition rule of fword. The precondition to the left of :: has to be satisfied before such a rule can be used for rewriting. Constructor-based formal parameters and preconditions are important for making the simplification process completely automatic [31].

The definition rules specified using the → connective are called *simplification rules*. That means that, whenever an instance of the left hand side of a rule is encountered it can be replaced by the corresponding instance of its right hand side, and we consider the expression to be "simplified" as a result. Another connective that can be used in definition rules is ⟶ (long arrow), and rules using it are called *expansion rules*. Expansion rules have the same *logical* meaning as simplification rules, but specify different kind of *control* information, viz., that applying an expansion rule makes the program longer or more complex. For example, a Lisp style definition of scan:

$$\begin{aligned}\mathsf{scan(s)} \longrightarrow\ & \mathsf{if\ s = Nil\ then\ Nil} \\ & \mathsf{else\ fword(s).scan(rword(s))}\end{aligned} \tag{1}$$

should be classified as an expansion rule. Applying it to an expression does not necessarily produce a "simplification".

We briefly summarize other aspects of the Focus specification language. More detailed discussion appears elsewhere [27, 30, 31]. *Logic programming* features are provided through a number of quantifiers. As the Focus shared language is an expression-oriented language, this is considerably different from Prolog. The simplest quantifier is of the form $THE(x : p)$ which denotes the unique x satisfying the condition p. An abstract logical specification of scan(s) may say something like "the list of words wl such that appending the words of wl, separated by arbitrary number of spaces, gives s". This can be expressed in Focus as:

$$\begin{aligned}\mathsf{scan(s)} \rightarrow \mathsf{THE(wl :\ } & \mathsf{word\_list(wl)\ and} \\ & \mathsf{append\_with\_spaces(wl,s))}\end{aligned} \tag{2}$$

Other quantifiers used in Focus are

$SOME(\overline{x}:p)e$	for some $\overline{x}$ such that p, the value of e
$ANY(\overline{x}:p)e$	for any $\overline{x}$ such that p, the value of e
$\{x:p\}$	the set of x such that p
$SET(\overline{x}:p)e$	the set of values of e for all $\overline{x}$ such that p
$EXIST(\overline{x}:p)$	there exist $\overline{x}$ such that p
$ALL(\overline{x}:p)q$	for all $\overline{x}$ such that p, q holds

The execution of all such expressions is performed using the symbolic evaluation techniques discussed in section 3.2.

The *properties* part of the specification can supply any properties of the functions to be used in program derivation. All such properties are possibly conditional rewrite rules of them form $c :: l \rightarrow r$, without any restriction on the form of left hand sides l. Such a rule means that whenever c holds, l can be replaced by r, and performance is *improved* as a result of the replacement. So, these rules may be thought of as "domain knowledge" or "transformation rules". In the example, we have not provided any properties as part of the specification. However, property rules also come from two other sources:

1. they may be proved on the fly, using the techniques of section 3.3, or

2. they may be derived from specifications of internal program components designed during program derivation (section 4.1).

3 Validation of Specifications

As mentioned at the outset, program validation, verification and derivation activities are closely integrated in the Focus system. This is so because they all share the same basic program manipulation operations. We first discuss these operations from program validation point of view, and reconsider them for program derivation in section 4.

3.1 Simplification

"Execution" in the context of functional languages merely means evaluating an expression using function definition rules. Simplification is a minor generalization of evaluation, which operates on expressions with not only *constant* inputs, but also with *symbolic* inputs (variables).

Simplification of an expression is the operation of repeatedly rewriting any instance of the left hand side of a simplification rule by the corresponding right hand side, until no more simplification is possible. For example, given the specification of Fig. 1, the expression

```
scan('a'.'b'.' '.'c'.Nil)
```

can be simplified to

```
('a'.'b'.Nil).('c'.Nil).Nil
```

Using a more readable square bracket notation to denote lists [...], the simplification is

```
scan(['a', 'b', ' ', 'c']) → [['a', 'b'], ['c']]
```

Simplification also allows us to use variables in the inputs to functions. In that case, it performs as much computation as possible without the knowledge of the values of the variables. For example

scan('a'.'b'.' '.x) → ['a', 'b'].scan(skip(x))

The expression scan(skip(x)) denotes the computation that remains to be performed when the value of x is determined.

All definition rules supplied in a specification are expected to satisfy the *confluence* property: no matter what order the subexpressions of an expression are simplified, the result is unique [18, 23]. This essentially means that all functions are well-defined. In addition, simplification rules should also satisfy the *strong termination* property: given any expression, symbolic or constant, simplification of the expression in any possible order should finitely terminate. Intuitively, this means that simplification rules should always map more complex expressions to less complex expressions. (Expansion rules, on the other hand, need not satisfy this property. This is the main technical distinction between simplification and expansion rules). These two properties allow the Focus system to simplify all expressions it encounters to their normal forms, with the guarantee that the normal forms exist and are finitely reachable.

3.2 Symbolic execution

The specification of scan in Fig. 1 contains a potential bug (inherited from Feather [12]). If the very first character in the input is a space, the scanner outputs an empty word as the first word in the output. For example,

scan([' ', 'a', 'b']) → [[], ['a', 'b']]

Whether this was the intent of the original specification is questionable. If it is indeed a bug, coming up with a test case that detects it during validation would be somewhat hard. Symbolic execution, which automatically enumerates all possible execution paths, would easily produce such a test case and uncover the problem.

To observe the symbolic execution of a function, we start with an expression applying the function to variables (symbolic values). For example, to observe scan, we may choose scan(x). Such an expression cannot be simplified unless the variables are given values. The *expansion* operation finds instantiations for variables which allow the expression to be simplified at least one step. For example, scan(x) is simplifiable if it is an instance of either scan(Nil) or scan(char.s)—the left hand sides of the two definition rules of scan. So, x must be either Nil or char.s. These conditions on the variable x and the results of execution in each case are displayed by the Focus system as a tree. Figure 2 shows the results of this expansion as well as two successive expansions on cases 2 and 2.2. The case 2.2.1 shows the test case where x is a list with a single space and the output is a singleton list with an empty word Nil.Nil, i.e., [[]], rather than the empty list Nil. This shows up the problem mentioned above.

In the normal mode of Focus usage, expansion is user-controlled. This is necessary because successive expansion, in general, does not terminate. Through the interactive tree editor interface, the user issues an *expand* command, and Focus immediately produces children nodes

```
observe scan(x)
cases from scan(x)
 1.  if x = Nil then Nil
 2.  if x = char.s then fword(char.s).scan(rword(char.s))
     cases from fword(char.s)
      2.1.  if x = c.s and alpha(c) then (c.fword(s)).scan(rword(s))
      2.2.  if x = ' '.s then Nil.scan(skip(s))
            cases from skip(s)
             2.2.1.  if x = ' '.Nil then Nil.Nil
             2.2.2.  if x = ' '.' '.s then Nil.scan(skip(s))
             2.2.3.  if x = ' '.c.s and alpha(c) then  Nil.(c.fword(s)).scan(rword(s))
```

Figure 2: Symbolic execution of scan(x)

with various cases. It is also possible to generate successive expansions up to a specific depth and examine all the generated test cases.

The reader familiar with logic programming would notice that expansion bears a close resemblance to the execution of logic programs. In fact, the formal technique used in this operation is called *narrowing* and has been used to develop logic programming languages based on functions [11, 14, 25, 27]. To expand an expression, first a subexpression is chosen. The entire expression must be *strict* in the chosen subexpression, i.e., the value of the latter must be necessary to compute the value of the whole expression. (Sometimes, it is necessary to consider several subexpressions, but we are ignoring the technicalities in favor of simplicity). Secondly, the chosen subexpression must be a function application whose arguments are sufficiently evaluated (but not necessarily instantiated). For example, in case 2 of Fig. 2, we find two subexpressions that meet these conditions: fword(char.s) and rword(char.s). The bigger subexpression scan(rword(char.s)) does not meet the second condition above, since scan cannot be applied without first evaluating rword(char.s). When there are several subexpressions which can be expanded, any one of them can be chosen. Focus always chooses the leftmost one by default, but the user can override it. The expanded subexpression in each expansion is displayed in Fig. 2 following the keyword **cases**.

Once a subexpression is chosen for expansion, the definition rules of the function involved in the subexpression determine the various evaluation paths for it. For each definition rule, we compute the *most general* instance of the subexpression rewritable using the rule, using the unification algorithm [33]. For the subexpression fword(char.s) of case 2, we find there is no instance rewritable using the first rule of fword. The remaining two rules give the cases 2.1 and 2.2, instantiating char to c and ' ' respectively (with the condition alpha(c) in the former case). When conditional rules are involved, the preconditions are added to the corresponding path condition. Note that, for each case, Focus simplifies the resulting expression as far as possible.

Rewriting using *expansion rules* is also a special case of the expansion operation. For example, if scan were defined by the expansion rule (1), the first expansion of Figure 2 would have involved such rewriting.

When logic programming features are used, in terms of the quantifiers mentioned in section 2, the *normal* execution of specifications incorporates symbolic execution. The instantiations of the quantified variables are obtained by the expansion operation. For instance, if scan were specified by the logical specification (2), then the value of wl would be determined by expanding word_list(wl), append_with_spaces(wl,s), or both.

3.3 Verification of Properties

Even though symbolic execution automates conventional exhaustive testing methods, it still involves examining a combinatorially explosive set of test cases. A more feasible method is to automatically verify the specification for certain properties. When an abstract logical axiomatization of the problem is available, such as (2) for scan, we can verify if the specification satisfies such axiomatization.

```
prove word_list(scan(x)) → True
simplifies to:
ALL(w : member(w, scan(x))) word(w) → True
cases from scan(x)
 1.  case x = Nil
     ALL(w : member(w, Nil)) word(w) → True
     proved
 2.  case x = char.s
     ALL(w : member(w, fword(char.s).scan(rword(char.s)))) word(w) → True
     simplifies to:
     word(fword(char.s)) and (ALL(w : member(w, scan(rword(char.s)))) word(w)) → True
     rewrites to:
     word(fword(char.s)) → True
     cases from fword(char.s)
      2.1.  case char = c and alpha(c)
            alpha(c) :: word(c.fword(s)) → True
      2.2.  case char = ' '
            word(Nil) → True
            cases from word(Nil)
             2.2.1.  Nil ≠ Nil and ALL(c : member(c, Nil)) alpha(c) → True
                     failed
```

Figure 3: Attempt at verification of scan

In Figure 3, we show an attempt at verification of the property:

word_list(scan(x)) = True

with the auxiliary definitions

word_list(wl) → ALL(w : member(w,wl)) word(w)
word(w) ⟶ w $\neq$ Nil and ALL(c : member(c,w)) alpha(c)
member(a, Nil) → Nil
member(a, b.x) → a = b or member(a,x)

Except for the additional documentation we have added to display the intermediate results, the case analysis in the verification is remarkably similar to that in symbolic execution (Figure 2). The verification *fails* in case 2.2.1, where the first character of the input is assumed to be a space.

Program verification differs from plain symbolic execution in one crucial respect: it involves *induction.* In Figure 3, induction is used in case 2. Note that, after simplification, the left hand side has two conjuncts and the second conjunct

ALL(w : member(w, scan(rword(char.s)))) word(w)

is an instance of the left hand side of the original property. The rewriting operation rewrites it to True by using the property as in inductive hypothesis. After performing an expansion using simplification rules, it is always valid to use the inductive hypothesis in the cases [28].

If this verification had succeeded, we would have proved the rewrite rule in the root node as an inductive theorem (together with a number of lemmas such as word(fword(s)) → True). Such properties are added to the rule base, and may be used for proving other properties or for optimization during program derivation. Important performance savings during derivation are almost always obtained by using properties of functions. For example, consider the property rule

last(reverse(a.x)) → a

in the domain of lists. Rewriting an expression using it eliminates an $O(n)$ computation. Given the definitions of last and reverse, we can prove the property using induction, just as in Figure 3, and then use it for optimization in derivations. Verification is thus an integral part of program derivation.

4 Program Derivation

Since the specifications we use are executable, program derivation can be viewed as *optimizing* the execution behavior of specifications. This requires that we first recognize opportunities for performance savings and then *realize* the savings in new procedures. This most often results in postulating a new program component by a specification of the form

$$f(\overline{x}) = e \qquad (3)$$

where f is the new function, $\overline{x}$ is a set of formal parameters and e is an expression that serves as the specification of f. The notion of "specification" here is that of a program component, and should be distinguished from the specification of the whole program. We call the design of such a new component a *focus*, $f(\overline{x})$ its *focus label* and e its *focus specification.* The derivation of the program for f can be viewed in two different, but equivalent, ways. One view is that we want to construct a definition for f that meets the specification $f(\overline{x}) = e$. This is often called the *constructive approach* and is exemplified by the work of [7, 21]. The second view is that we are merely transforming the expression e to a more efficient form and using $f(\overline{x})$ as the label for the efficient form. This view underlies the so-called *transformational approach* [2, 34, 40]. See [32] for a discussion of the two approaches. In our framework, the approach is one, but it can be viewed in two ways.

4.1 Rewrite rules from specifications

Suppose that we eventually produce an efficient procedure for f starting from a focus equation such as (3). The equation (3) then becomes a *property* of the newly defined function f. It contains important information that can be used for optimization of the rest of the program. The Focus system orients it as a property rule:

$$e \rightarrow f(\overline{x}) \tag{4}$$

The orientation of the rule backwards may seem counter-intuitive, but is easily justified. The fact that we have derived a procedure for f signifies that the focus expression e is deemed inefficient, and the savings available in the procedure can be obtained by replacing any encountered instance of e by a *call* to f. Thus, the rule is oriented along the direction of improved efficiency. We call property rules arising in this fashion *introduction rules*, as they serve to introduce applications of newly designed functions.

The use of an introduction rule need not be limited to the rest of the program. It can be used in the derivation of the procedure for f itself. It is more than likely that during the derivation of the procedure, we encounter another instance of e, which can then be replaced by a *recursive call* to f. By structuring the derivation process suitably, we can ensure that the recursion does not introduce nontermination. Using an introduction rule in the function's own derivation may be seen similar to using a property as an inductive hypothesis in its own proof.

Let us illustrate these concepts by deriving a minor optimization of the scanner specification in Fig. 1. An immediate opportunity for savings is the duplication of list traversal by fword and rword. We can postulate the following focus to eliminate the duplication:

$$\mathsf{scanword(s) = fword(s).scan(rword(s))}$$

(Notice that this is a minor generalization of the expression in the second definition rule of scan). In a large program, such opportunities for savings may not be immediately apparent. Observation of the symbolic execution of programs can be useful for suggesting the opportunities.

Once a focus is postulated, the process of program derivation is remarkably similar to that of symbolic execution. We perform one or more expansion operations on the focus, instantiating the variables for the various evaluation paths, and simplify the resultant expressions in each case. Figure 4 shows the result of one expansion on the scanword focus, together with the automatic simplification of the cases.

```
focus scanword(s) = fword(s).scan(rword(s))
introduction rule: fword(s).scan(rword(s)) → scanword(s)
cases from fword(s)
 1.   scanword(Nil) = Nil.scan(rword(Nil))
      simplifies to:
      scanword(Nil) = Nil.Nil
 2.   alpha(c) :: scanword(c.s) = (c.fword(s)).scan(rword(c.s))
      simplifies to:
      alpha(c) :: scanword(c.s) = (c.fword(s)).scan(rword(s))
 3.   scanword(' '.s) = Nil.scan(rword(' '.s))
      simplifies to:
      scanword(' '.s) = Nil.scan(skip(s))
```

Figure 4: Expansion of scanword focus

4.2 Rewriting

The main difference between observation of symbolic executions and program derivations is the use of property rules for *rewriting* or *transformation.* Recall that property rules are rewrite rules that yield guaranteed performance savings. They are either supplied as part of the initial specification, proved on the fly, or derived by back-orienting focus specifications as introduction rules. For example, when the scanword focus of Figure 4 is expanded, its specification is back-oriented to an introduction rule. Rewriting using such rules is a sophisticated operation and involves complex matching procedures.

In the first place, rewrite rules, unlike definition rules, do not satisfy the confluence property. So, different orders of rewriting and different choices of rewrite rules may produce different results. This is the main form of *nondeterminism* present in the Focus derivation procedures. To alleviate this nondeterminism, Focus first simplifies expressions to their unique normal forms (using only the simplification rules) before doing any rewriting by property rules. Secondly, during the default operation, Focus uses the *outermost* strategy for rewriting, i.e., it searches for instances of rewritable patterns outside in. The rationale is that, since rewriting is meant for optimizing, the larger the subexpression rewritten, the better is the optimization. This can be overridden by the user, but we found that the outermost strategy works admirably almost always.

A second complexity involved with rewriting is that the patterns on the left hand sides of rewrite rules are arbitrary expressions, not just constructor-based patterns. So, pure pattern matching is not adequate. The recursion rule of scanword above shows an important special case of this. Its left hand side is a data structure made of the "." constructor. So, the rule is applicable not only to an instance of the complete data structure, but also to instances of its individual *components.* Suppose we have two selector functions, head and tail, to access the components of a dotted pair. Then the introduction rule of scanword implicitly gives rise to the following derived rules:

```
fword(s) → head(scanword(s))                                          (5)
scan(rword(s)) → tail(scanword(s))
```

In the absence of such selector functions, we express the derived rules using the **where** construct:

```
fword(s) → x1 where x1.x2 = scanword(s)
scan(rword(s)) → x2 where x1.x2 = scanword(s)
```

where x1 and x2 are arbitrary new variables. A number of derived rules are used to handle conditionals and logical quantifiers. For example, a rule of the form $THE(x : p) \rightarrow r$ yields a derived rule $p \rightarrow (x = r)$. The details of the Focus rewriting operation in the context of conditionals and quantifiers will be given in a future paper.

Given the derived rules (5) for introduction of scanword, the focus in case 2 of Fig. 4 is automatically rewritten to

```
alpha(c) :: scanword(c.s) =
        (c.x1).x2 where x1.x2 = scanword(s)
```

This is the only rewriting possible in the entire expansion shown in Fig. 4. Note that the two calls to fword and rword have been eliminated in favor of a single recursive call to scanword.

Once the program for scanword is derived, we need to use it to optimize the original specification. This can be done by focusing on the function scan (when we focus on an already defined function, the focus expressions are obtained directly from the definition). Focus immediately rewrites the second definition equation of scan using the introduction rule of scanword to produce:

```
focus scan
cases
 1.    scan(Nil) = Nil
 2.    scan(char.s) = fword(char.s).scan(rword(char.s))
       rewrites to: scan(char.s) = scanword(char.s)
```

4.3 Search spaces

Let us now make explicit the division of work between the user and the Focus system. The user chooses focus expressions and creates focuses for them. Whenever a focus expression needs to be optimized, he issues an expand command. The system chooses the subexpression to be expanded, finds instantiations, creates cases, and for each case, simplifies and rewrites the focus expressions as far as possible. The nondeterminism involved in this process is, firstly, in choosing the subexpression to be expanded, and, secondly, in the multiple results which may be produced as a result of rewriting.

For expansion, the criteria that we have given for choosing expandable subexpressions eliminate a number of useless alternatives. Of the remaining alternatives, often the choice does not matter. As the reader may verify, the derivation of scanword would have produced identical results even if the other subexpression rword(char.s) was chosen for expansion. However, there are also several situations where the different choices for expansion lead to different programs.

Sometimes it is necessary to try the different alternatives and pick one. The user intervention is necessary in such situations.

Nondeterminism arises in rewriting much less often. It seems that the process of program derivation itself somehow prevents too much nondeterminism from creeping in. In most cases where we have encountered nondeterminism in rewriting, it was caused by the property rules supplied by the user rather than the introduction rules generated during derivation. Whenever rewriting yields multiple results, the system displays all of them and requires the user to pick one. We do not use backtracking. It may be noted that in the example given above, rewriting produced unique results.

Burstall and Darlington [3] use an operation called *folding*, which rewrites by using definition rules *backwards*. This operation causes much larger search spaces. For example, using the specification of Fig. 1, the constant Nil can be folded in five different ways. Our method of back-orienting focus specifications as introduction rules was a solution to the search space problem of folding. This allows us to restrict such backward rewriting to cases which improve efficiency. Similar approaches may be found in [10, 34, 35, 37]. Unfortunately, the program derivation procedure without folding is not logically complete [29]. So, occasionally, it is also necessary to use folding and attend to its large search space.

5 Structure of Program Derivations

5.1 Documenting Derivations

A program development system must organize the derivations of programs into a suitable structure so that the programmer can easily comprehend them. To this end, we find that there are two kinds of documentations required of program derivations. The first kind of documentation is to serve the same purpose as a program listing in the conventional software life cycle. It should be usable by program designers and maintainers to understand the program implementation and its structure. However, in contrast to conventional listings, this kind of documentation provides the *history* and the *rationale* of the implementation as well. Note, first of all, that the specification is the best readable account of the system. As the program derivation proceeds the components of the system are increasingly specialized in the context of their use [34]. Thus, the components of the final implementation cannot be understood in isolation. They have to be understood in relation to the more abstract components to be found in the specification. In practice, the system undergoes several changes in structure and abstraction in going from the most abstract specification to the most efficient implementation. The components to be found at each stage should then be understood in relation to the components of the earlier stages. A documentation of these relationships and the products of various stages of derivation is a *derivation history.*

The second kind of documentation includes, in addition to the products of the stages of the derivation and their relationships, all the details of the derivation activity itself. This additional level of detail is not necessary to *understand* the program structure and rationale, but necessary to make *changes.* Radical changes which necessitate discarding the whole derivation and implementation are rare. Instead, most changes require minor modifications to the specifications and can be propagated to the implementations by following the prior derivation activity. The documentation we call a *derivation script* serves this purpose.

The early efforts towards documentation of derivations (e.g. [12, 40]) did not make this distinction between derivation histories and derivation scripts. The methods given in [9] seem to contain some elements at the level of what we call derivation histories, but they contain low level operations as well (such as unfold and fold). Another feature of these approaches is that they document derivations at a *meta-level*, in terms of operations on programs. We are of the opinion that the high level documentation of derivations should mainly contain *program level* information. In this context, we can draw an analogy with mathematical proofs. In writing proofs, we manage to communicate the reasoning and rationale for proof steps without mentioning the basic inference rules involved in them. We would like derivation histories to be descriptions at the level of such proofs. (Derivation scripts are akin to descriptions at the level of mechanically verifiable proofs). The notion of derivation trees is central to the high level descriptions that we seek.

5.2 Derivation Trees

Let us show how the tree structure arises through a sample scenario. Suppose we are trying to derive an efficient program for an expression e. We postulate a *focus* for it as

$$p(\overline{x}) = e$$

Sometimes, the expression e can be transformed directly into a more efficient form by a case analysis on its variables (like for scanword). But, more often, we have to focus on another expression s for which an efficient program can be derived. This new expression s may be merely a subexpression of e, or a collection of several subexpressions which have common subcomputations, or even a generalization of e. The choice of the expression s is a crucial *design decision* based on the programmer's insight into the nature of the computation. The Focus system owes its name to the centrality of the focusing operation in its usage.

Having chosen to focus on the expression s, we have a hierarchy like

$$\begin{array}{l} \textbf{focus}\ p(\overline{x}) = e \\ \qquad \textbf{focus}\ q(\overline{y}) = s \end{array}$$

The focus on s is *subsidiary* to that of e, because the development of s is an integral part of the development of e. When the development of s is complete, we have an efficient procedure for it in terms of q. We need then to realize these gains in the parent focus. This involves rewriting e to some e' using the back-oriented introduction rule $s \to q(\overline{y})$.

The new expression e' may still need further development. We may have to choose another internal focus expression s' and continue this process:

$$\begin{array}{l} \textbf{focus}\ p(\overline{x}) = e \\ \textit{program} : p(\overline{x}) = \cdots \\ \qquad \textbf{focus}\ q(\overline{y}) = s \\ \qquad \textit{program} : q(\overline{y}) = \cdots \\ \qquad \textbf{focus}\ q'(\overline{z}) = s' \\ \qquad \textit{program} : q'(\overline{z}) = \cdots \\ \qquad \vdots \end{array} \qquad (6)$$

Now, we only need to note that each subsidiary focus may in turn have other subsidiary foci, and it is clear that we have a tree structure. In fact, the tree structure of foci bears a close resemblance to the block structure of programs. There is, of course, no unique way to organize a derivation into a tree. The program designer is free to structure it as he/she sees fit. For example, instead of making the focus of q' subsidiary to the original focus node of p, the designer can open a new focus node for p with the specification e' and derive the procedure q' as a part of that.

We found that this flexibility in how the derivation is organized into trees and in how the trees themselves are developed, quite important. It is also necessary for the designer to examine the high-level structure of a tree and ignore the internal details of various of subtrees. The Treemacs editor, through which Focus is used, facilitates this. It provides functions to selectively view parts of the tree, to walk around the tree and to create and delete nodes. The functions of the Focus system such as opening and closing focus nodes and other program manipulation operations discussed in the last section are also provided through this interface.

The *derivation history*, characterized earlier as the high-level documentation describing the history and the rationale of the implementation, is now merely the focus tree with each focus node displaying two pieces of information: its specification and the derived program. Figure 5 shows a sample derivation history for the implementation of scan. This organizes the foci of scan and scanword (discussed in section 4) into a hierarchical structure, together with an additional focus for scanskip. More importantly, it suppresses the internal case analysis information that is generated during the derivation process.

5.3 Program Derivation State

Even though the *organization* of a program derivation is represented as a tree, the *process* of program derivation is a sequential process. Each derivation step takes an input program and produces an output program. For example, if we view the derivation tree of Figure 5 as a derivation step, its input program has the rule

scan(char.s) → fword(char.s).scan(rword(char.s))

whereas the output program has the rule

scan(char.s) → scanword(char.s)

in its place. The input program does not have the procedures scanword and scanskip, but the output program does. If we perform another program derivation step after that of Figure 5, e.g., to make scan tail recursive, then the input to this step would be the output program of Figure 5. Not only are programs transmitted between such derivation steps, but also all other kinds of rules like property rules, introduction rules, and specifications (old programs). We call the sum total of all the rules so transmitted, a *program derivation state.*

In systems which only present a *dynamic* view of this program derivation process (e.g. Popart [40] and KBEmacs [39]) there is only one program derivation state active at any time. It is not possible to go back to an earlier state and examine the differences from the current state, except by retracting derivation steps. This makes it hard for the user to keep track of what has been accomplished and what is left to be done. In contrast, the derivation trees of

```
Focus scan(char.s) = fword(char.s).scan(rword(char.s))
derived program:
scan(char.s) → scanword(char.s)
    Focus scanword(s) = fword(s).scan(rword(s))
    derived program:
    scanword(Nil) → Nil.Nil
    alpha(c) :: scanword(c.s) →
        (c.x1).x2 where x1.x2 = scanword(s)
    scanword(' '.s) → Nil.scanskip(s)
    introduction rules:
    rword(s).scan(rword(s)) → scanword(s)
        Focus scanskip(s) = scan(skip(s))
        derived program:
        scanskip(Nil) → Nil
        alpha(c) :: scanskip(c.s) →
            (c.x1).x2 where x1.x2 = scanword(s)
        scanskip(' '.s) → scanskip(s)
        introduction rules:
        scan(skip(s)) → scanskip(s)
```

Figure 5: A derivation history of scan

Focus present a *static* view of the entire program derivation process. Different parts of the tree refer to different derivation states. For this purpose, the nodes of a derivation tree may be viewed as being organized in a sequential fashion, from top to bottom. If a node is opened *above* the tree of Figure 5, its input state would be the state that existed *prior* to the derivation of Figure 5. It is possible, for example, to prove a property in such a node, and then rederive the tree of Figure 5 to take the new property into account. There is no need to retract or delete the tree in order to make such a modification.

This static picture of the program derivation process is achieved by *distributing* the program derivation state throughout the tree instead of maintaining a single global state. Each node contributes some rules to the state. The rules visible to a node are the rules contributed by the nodes which precede it in the textual top-to-bottom reading of the tree. These include the node's predecessor nodes (at the same level) and their descendants, its ancestor nodes and their respective predecessors and descendants. The rules contributed by the node's descendants are also visible. But, there is some dynamic character to this. A node can contribute *global* rules which are visible to all other nodes according to the visibility rules, and *local* rules which are only visible to its descendants.

The derivation process represented by a derivation tree is thus statically represented. However, the derivation process within a *single node* is dynamic. Notice (for example, in Figure 2) that each focus node has a state, viz., the focus equation. This is called a *focus state*. It changes dynamically as various derivation operations, like simplification and rewriting, are performed. It is not possible to go back to an earlier state except by retracting such operations. It would be possible to provide a static picture of the focus state as well, but we find this neither necessary nor convenient.

5.4 Dependencies

In order to make modifications to program derivations (and hence to programs), it is necessary to know how the modification of a node's derivation affects the other nodes. In principle, all the nodes that *follow* the modified node in top-to-bottom order are affected by the modification. However, maintaining finer-grained information about dependencies eliminates many candidates. First of all, if a modification only *adds* rules to a node's global contribution, then the derivations of the other nodes are still legal. However, performing replay on them may produce slightly different results because of the additional rules. It is usually not worth the trouble to keep an entire derivation tree "consistent" with respect to such weak modifications. It is better to keep the tree in a semiconsistent state and "clean it up" at the end by performing a replay. If, on the other hand, a modification deletes rules from a node's global contribution or replaces them by new ones, then the modification may make the derivations of other nodes *illegal.* To identify the nodes which are thus affected, Focus keeps track of dependencies and decertifies the nodes that become illegal due to a modification. Their derivations have to be reperformed either by an automatic replay or by explicit user intervention.

There are three kinds of dependencies that exist between focus nodes. The first kind, *specification dependency* arises as follows: Referring to the sample tree (6), suppose that, during the development of the program for q', we encounter an instance of the specification s of q. Such an instance may be replaced by an application of q using the back-oriented introduction rule. The procedure for q' may then have an application of q. The development of q' now *depends on the specification* of q. But, it does not depend on the procedure for q itself. The internal derivation of q can be modified without affecting q'.

The second kind of dependency is *program dependency* or *inheritance.* Suppose the derivation of q' encounters an application of q, and then unfolds the application to the body of the procedure, either through simplification or expansion. The derivation of q' now *depends on the program* (procedure body) for q. If the derivation of q is changed, the derivation of q' needs to be changed as well.

The third kind, called *subgoal dependency* can be present only between the children of a node and their parent. The specification expressions s and s' are meant to be subexpressions or generalizations of the parent's specification e (in general, subgoals of e). If the parent's specification is changed, s and s' may not be its subgoals any more and they have to be changed as well.

Subgoal dependencies are represented by the tree structure of the derivations. When the specification of a focus node is changed, the designer needs to examine all its children foci and rederive programs for them. The specification and program dependencies, on the other hand, are marked by the Focus system during program development. This information is used to propagate changes during program modification. The system attempts to automatically propagate the changes to the nodes that are specification or program dependent on the modified focus node. This is done by replaying the derivation scripts stored in the affected nodes, and comparing the results with those obtained in the earlier derivation. If the results are significantly different, the user is alerted to this fact and is given the option to modify the derivations.

5.5 Derivation Scripts

Derivation scripts are recorded by Focus internal to focus nodes. The scripts contain the details of all the derivation operations performed within each focus node in going from its specification to its program. In addition to the operations, the scripts also record all the intermediate results obtained during the original development. The intermediate results are matched against the results obtained during replay, so that errors can be automatically detected. The program designer never needs to directly construct the derivation scripts (unlike in [9, 12, 40]). He/She only manipulates the specifications/programs and the records of the activity are automatically constructed by the system, deleting any canceled steps and dead-ends.

~~6 Conclusion~~

~~We have outlined here the salient design principles of the Focus program derivation system.~~ The inference procedures of the system are based on the integration of *functional and logic programming* principles. This brings about a synthesis of paradigms that were heretofore considered far apart, such as logical and executable specifications and constructive and transformational approaches to program derivation.

A great emphasis has been placed, in the design of Focus, on achieving *small search spaces* during program derivation. The program manipulation operations such as expansion, simplification and rewriting were designed with this objective. The role of operations that are expensive in search spaces, such as folding, has been reduced.

Program derivations are documented in Focus in a way that the high level descriptions of derivations are expressed only using program level information. All the meta-level information, together with dependencies between derivations of program components, is automatically recorded by the system at a lower level of description for its own use in replay.

At the time of this writing, the implementation of the Focus system is at an advanced stage. The first-order functional subset of the specification language, used for examples in this paper, has been completely implemented. The logic programming part of the language has a prototype implementation which we are experimenting with. The tree structure of derivations has been formalized and is interfaced to the tree editor. A major unimplemented operation at this time is the replay of derivation scripts. We need this crucially before we can experiment with solving large programming problems using Focus.

An avenue for future research is to build heuristic and learning capabilities into the system. At the low level, most program derivations contain optimizations that are very similar. We are investigating the use of explanation-based learning and derivational analogy techniques, by which the system can observe the user's derivation activity and generalize it into heuristics.

References

[1] Balzer, R., Cheatham, E. J., and Green, C. Software technology in the 1990's: Using a new paradigm. *IEEE Computer 16* (Nov. 1983), 39–45.

[2] Balzer, R., Goldman, N. M., and Wile, D. S. On the transformational approach to programming. In *Proc. 2nd Intern. Conf. on Softw. Eng.* (1976), IEEE, pp. 337–344.

[3] Burstall, R. M., and Darlington, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.

[4] Cheatham, T. E., Holloway, G. H., and Townley, J. A. Symbolic evaluation and the analysis of programs. *IEEE Trans. Softw. Eng. SE-5*, 4 (July 1979), 402–417.

[5] Clarke, L. A. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng. SE-2* (Sep 1976), 215–222.

[6] Colmerauer, A., Kanouri, H., Pasero, R., and Roussel, P. *Un Systeme de Communication Homme-machine en Francais.* Research Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1973.

[7] Constable, R. Constructive mathematics and automatic program writers. In *IFIP* (Ljublijana, Yugoslavia, Aug. 1971), pp. 229–233.

[8] Darlington, J. Transforming specifications into efficient programs. In *IFIP Working Group 2.1 Conf. on Softw. Specifications* (St. Pierre-de-Chatreuse, France., 1976).

[9] Darlington, J. The structured description of algorithm derivations. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds., North-Holland, 1981, pp. 221–250.

[10] Dershowitz, N. Synthesis by completion. In *IJCAI* (Los Angeles, 1985), pp. 208–214.

[11] Dershowitz, N., and Plaisted, D. A. Logic programming cum applicative programming. In *Symposium on Logic Programming* (1985), IEEE.

[12] Feather, M. S. *A System for Developing Programs by Transformation.* PhD thesis, Univ. of Edinburgh, 1979.

[13] Feather, M. S. A system for assisting program transformation. *ACM Trans. Program. Lang. Syst. 4*, 1 (1982), 1–20.

[14] Goguen, J. A., and Meseguer, J. Equality, types, modules and generics for logic programming. In *Proc. 2nd Intern. Logic Prog. Conf., Uppsala* (1984), pp. 115–125.

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

[16] Hammerslag, D. *Treemacs Manual.* Tech. Rep. UIUCDCS-R-88-1427, Univ. Illinois at Urbana-Champaign, May 1988.

[17] Hammerslag, D. H., Kamin, S. N., and Campbell, R. H. Tree-oriented interactive processing with an application to theorem-proving. In *Second Conf. on Softw. Dev. Tools, Techniques, and Alternatives* (1985), IEEE Computer Society Press, pp. 199–206.

[18] Huet, G., and Oppen, D. C. Equations and rewrite rules: a survey. In *Formal Language Theory: Perspectives and Open Problems*, R. Book, Ed., Academic Press, New York, 1980, pp. 349–405.

[19] KOWALSKI, R. A. *Logic for Problem Solving.* North-Holland, 1979.

[20] MANNA, Z., AND WALDINGER, R. Synthesis: Dreams ⇒ programs. *IEEE Trans. Software Engineering SE-5*, 4 (1979), 294–328.

[21] MANNA, Z., AND WALDINGER, R. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan. 1980), 90–121.

[22] MILNER, R. A proposal for Standard ML. In *ACM Symp. on LISP and Functional Program.* (1984), pp. 184–197.

[23] O'DONNELL, M. J. *Equational Logic as a Programming Language.* MIT Press, 1985.

[24] PEYTON-JONES, S. L. *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1987.

[25] REDDY, U. S. Narrowing as the operational semantics of functional languages. In *Symp. on Logic Program.* (Boston, 1985), IEEE, pp. 138–151.

[26] REDDY, U. S. On the relationship between logic and functional languages. In *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom, Eds., Prentice-Hall, 1986, pp. 3–36.

[27] REDDY, U. S. Functional logic languages, Part I. In *Graph Reduction*, Springer-Verlag, 1987, pp. 401–425. (Lecture Notes in Computer Science, Vol 279).

[28] REDDY, U. S. *Rewriting techniques for program synthesis and inductive theorem proving.* Preprint, Univ. Illinois at Urbana-Champaign, Nov. 1987.

[29] REDDY, U. S. *Program transformation without folding.* Preprint, Univ. Illinois at Urbana-Champaign, Jan. 1988.

[30] REDDY, U. S. *Applicative Techniques for Imperative Programs.* Draft, Univ. Illinois at Urbana-Champaign, July 1988.

[31] REDDY, U. S. Design principles for an interactive program derivation system. In *Proc. AAAI-88 Workshop on Automating Software Design* (1988), AAAI. (to appear).

[32] RICH, C., AND WATERS, R. C. *Artificial Intelligence and Software Engineering.* Morgan Kaufmann, 1986.

[33] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *J. ACM 12* (1965), 23–41.

[34] SCHERLIS, W. L. Program improvement by internal specialization. In *ACM Symp. on Princ. of Program. Lang.* (1981), ACM, pp. 41–49.

[35] TAMAKI, H., AND SATO, T. Unfold/fold transformation of logic programs. In *Intern. Conf. on Logic Program.* (Uppsala, 1984), pp. 127–138.

[36] TEITELBAUM, T., AND REPS, T. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM 24*, 9 (Sept 1981), 563–573.

[37] TURCHIN, V. F. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst. 8*, 3 (1986), 292–325.

[38] TURNER, D. Miranda: a non-strict functional language with polymorphic types. In *Conf. on Functional Program. Lang. and Comput. Architecture* (1985), J. Jouannaud, Ed., Springer-Verlag, pp. 1–16.

[39] WATERS, R. The programmer's apprentice: A session with KBEmacs. *IEEE Trans. Softw. Eng. SE-11*, 11 (Nov. 1985), 1296–1320.

[40] WILE, D. S. Program developments: Formal explanations of implementations. *Comm. ACM 26*, 11 (1983), 902–911.