Steps Towards Mechanizing Program Transformations Using PVS [⋆]

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/shankar.html
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

Abstract

PVS is a highly automated framework for specification and verification. We show how the language and deduction features of PVS can be used to formalize, mechanize, and apply some useful program transformation techniques. We examine two such examples in detail. The first is a fusion theorem due to Bird where the composition of a catamorphism (a recursive operation on the structure of a datatype) and an anamorphism (an operation that constructs instances of the datatype) is fused to eliminate the intermediate data structure. The second example is Wand's continuation-based transformation technique for deriving tail-recursive functions from non-tail-recursive ones. These examples illustrate the utility of the language and inference features of PVS in capturing these transformations in a simple, general, and useful form.

1 Introduction

Correctness-preserving program transformations [15] often capture deep algorithmic insight and therefore pose interesting challenges for mechanization. The mechanization of program transformations has typically been carried out

[⋆] This paper is a sequel to a paper entitled *Computer-Aided Computing* [19] that appeared in the proceedings of the 3rd International Conference on the Mathematics of Program Construction held at Kloster Irsee, Germany, during July 1995. The author is grateful to the programme committee of that conference, especially Bernhard Möller, for the invitation to participate in a thoroughly stimulating meeting. This research has been supported by NSF Grant CCR-930044 and ARPA under contract PR8556. Some of the work reported here was inspired by Richard Bird's talk at the conference and the related paper [1]. The feedback from the 1995 Dagstuhl workshop on induction theorem proving was also valuable. Healfdene Goguen, Bernhard Möller, Sam Owre, Harald Ruess, John Rushby, and Mandayam Srivas supplied valuable comments on draft versions of this paper. Sam Owre has been, and continues to be, instrumental in the design and implementation of PVS. Many others at SRI and elsewhere have contributed to PVS in important ways.

using special-purpose tools such as the KIDS system [21]. This paper examines the utility of the general-purpose verification system PVS [14,19], for mechanizing program transformation. The main challenge is that program transformations are normally expressed and applied in metatheoretic, i.e., syntactic, form and are therefore not easily formalized in a formal specification logic. We observe that the specification language and inference mechanisms of PVS are quite effective for the task of formalizing and verifying program transformations, but are not without certain drawbacks.

Richard Bird [1] makes a persuasive argument that functional programming can be used to elegantly derive reasonably efficient analogues of imperative algorithms. In that paper he presents a fusion theorem showing that the composition of a catamorphism (a function that is defined by structural recursion on a recursive datatype) and an anamorphism (a function that recursively constructs an instance of the recursive datatype) can be simplified to a single function where the intermediate data structure has been eliminated. This transformation is closely related to deforestation [22]. We show how various features of PVS can be exploited in order to give an elegant formalization of an instance of the fusion theorem for the specific recursive datatype of binary trees. In particular, we show that the technical difficulty engendered in defining anamorphisms can be easily handled using subtyping and dependent typing as implemented in PVS. Note that the general fusion theorem for arbitrary positive recursive datatypes cannot be proved within PVS since it is a metatheorem. We also apply this transformation to derive an applicative quicksort algorithm from a treesort specification, and demonstrate that this algorithm returns an ordered permutation of its input.

Wand's continuation-based program transformation strategy is a powerful technique for transforming non-tail-recursive definitions into tail-recursive form [23]. In fact, a number of otherwise difficult induction arguments can be seen as simple instances of continuation-based transformations. We show how such transformations can be easily mechanized using parametric theories and the higher-order logic of PVS.

In general, the insights and techniques underlying such transformations are also useful in other domains such as hardware verification. [1] The results in this paper constitute preliminary steps towards mechanizing program transformation techniques using the general-purpose verification system PVS. Dold [8] has already verified a divide-and-conquer scheme using PVS and has instantiated it to synthesize a binary search algorithm for arrays. Ruess [18] has carried out a similar development using the type theory of LEGO [11]. Neither of these efforts achieves the level of mechanization claimed below. Most of the theorems in this paper are proved by a single PVS proof step that invokes a strategy for measure induction. This strategy was defined during the course

[1] Rajan [17] describes the use of PVS in verifying hardware-oriented transformations on control data flow graphs.

of this work and is a straightforward combination of existing strategies. It should be emphasized that the proofs presented in this paper are among the more elementary proofs that have been checked using PVS. The main point of this paper is not that these are hard proofs but that these highly interesting theorems can be formalized, proved, and used with negligible effort because of the combination of language constructs and deductive apparatus present in PVS. Even so, several challenges remain as fodder for future research.

2 A Brief Introduction to PVS

PVS (Prototype Verification System) is intended as an environment for constructing clear and precise specifications and for developing readable proofs that have been mechanically verified [14, 19]. While many of the individual ideas in the system pre-date PVS, what is new in PVS is the coherent realization of these ideas in a single system. The key elements of the PVS design are captured by the combination of features listed below.

An expressive language with powerful deductive capabilities. The PVS specification language is based on classical, simply typed, higher-order logic with base types such as the Booleans `bool` and the natural numbers `nat`, and type constructors for functions `[A -> B]`, records `[# a : A, b : B #]`, and tuples `[A, B, C]`. The PVS type system also admits predicate subtypes, e.g., {`i : nat | i > 0`} is the subtype of positive numbers. The PVS type system includes dependent function, record, and tuple types, e.g., `[# size : nat, elems : [below[size] -> nat] #]` is a dependent record where the type of the `elems` component depends on the value of the `size` component. It is also possible to define recursive abstract datatypes such as lists and trees as discussed in Section 3 below. The PVS typechecker checks for simple type correctness and generates proof obligations (called TCCs for *type correctness conditions*) corresponding to predicate subtypes. Typechecking is undecidable for PVS to the extent that it involves discharging such proof obligations. PVS also has parametric theories, so that it is possible to capture, say, the notion of sorting with respect to arbitrary array sizes, types, and ordering relations. Constraints on the theory parameters can be stated by means of assumptions within the theory. When an instance of a theory is imported with concrete parameters, there are proof obligations for the corresponding instances of the parameter assumptions. A theory is a list of declarations of constants (with or without definitions) and theorems. A constant or function definition has the form

$$\textit{constant} : \textit{type} = \textit{definition}$$

Powerful decision procedures with user interaction. PVS proofs are constructed interactively. The primitive inference steps for constructing proofs are quite powerful. They make extensive use of efficient decision procedures for equality and linear arithmetic [20]. They also exploit the tight integration between rewriting, the decision procedures, and the use of type information. PVS also uses BDD-based propositional simplification so that it can combine the capability of simplifying very large propositional expressions with equality, arithmetic, induction, and rewriting.

Higher-level inference steps can be defined by means of strategies (akin to LCF tactics [9]) written in a simple strategy language. Typical strategies include heuristic instantiation of quantifiers, propositional and arithmetic simplification, and induction and rewriting. The PVS proof checker tries to strike a careful balance between an automatic theorem prover and a low-level proof checker. Through the use of BDD-based simplification, simple PVS proof strategies can be defined for efficiently and automatically verifying simple processor designs and N-bit arithmetic circuits [6].

A useful strategy for well-founded induction (specifically, measure induction) was defined during the course of this work. This strategy is defined in terms of the existing `measure-induct` and `induct-and-simplify` strategies. It introduces the measure induction scheme instantiated with a suitable induction predicate, then simplifies the result to yield an induction goal. The strategy then expands the definitions of specified recursive functions and uses the case structure of these definitions to guide the remaining simplification steps.

Model checking with theorem proving. The details of this are not relevant to this paper. See [16, 19] for more details.

A variety of examples have been verified using PVS [7]. The most substantial use of PVS has been in the verification of the microcode for selected instructions of a commercial-scale microprocessor called AAMP5 designed by Rockwell-Collins [12].

3 Abstract Datatypes in PVS

Like many other specification and programming language, PVS has a construct for defining (possibly) recursive abstract datatypes corresponding to data structures that are freely generated by a collection of constructor operations. The `list` datatype is a simple example.[2] For example, the abstract datatype of lists is generated by the constructors `null` and `cons`. Similarly, the abstract datatype of stacks is generated by the constructors `empty` and

[2] The abstract datatype mechanism of PVS is partly inspired by the shell principle used in the Boyer-Moore theorem prover [3]. Similar mechanisms exist in a number of other specification and programming languages [5, 10, 13].

push. An unordered list or a bag is an example of a data structure that is not freely generated since two different sequences of insertions of elements into a bag can yield equivalent bags. The datatype of queues is freely generated by emptyqueue and enqueue, but it cannot be directly defined by the PVS abstract datatype mechanism since it is not recursive, i.e., the accessors top and dequeue are not inverses of the constructors.

The abstract datatype of lists of a given element type is declared in PVS as shown below.

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr:list):cons?

 END list
```

Here list is declared as a type that is parametric in the type T. The two constructors null and cons are introduced. The constructor null takes no arguments. The predicate null? holds for exactly those elements of the list datatype that are identical to null. The constructor cons takes two arguments where the first is of the type T and the second is a list. The *recognizer* predicate cons? holds for exactly those elements of the list type that are constructed using cons, namely, those that are not identical to null. There are two *accessors* corresponding to the two arguments of cons. The accessors car and cdr can be applied only to lists satisfying the cons? predicate so that car(cons(x, l)) is x and cdr(cons(x, l)) is l.

The PVS typechecker generates several theories corresponding to the declaration of the list abstract datatype. These generated theories are described in greater detail below for the case of the binary tree datatype. These theories can of course be generated by hand, but the datatype mechanism has the advantage that many of the datatype simplifications are built into the PVS inference mechanisms.

A binary tree is treated below as a recursive data structure that in the base case is an empty leaf node, and in the recursive case consists of a value component, and left and right subtrees that are themselves binary trees. The declaration for the binary trees datatype is similar to that for lists above. The two constructors leaf and node have corresponding recognizers leaf? and node?. The leaf constructor does not have any accessors. The node constructor has three arguments: the value at the node, the left subtree, and the right subtree. The accessor functions corresponding to these three arguments are val, left, and right, respectively.

```
binary_tree[T : TYPE] : DATATYPE
BEGIN
  leaf : leaf?
  node(val : T, left : binary_tree, right : binary_tree) : node?
END binary_tree
```

When the above datatype declaration is typechecked, the theories binary_tree_adt, binary_tree_map and binary_tree_reduce are generated. The initial portion of the binary_tree_adt theory is displayed below, and the remaining parts are discussed later.

```
binary_tree_adt[T: TYPE]: THEORY
  BEGIN

  binary_tree: TYPE

  leaf?, node?: [binary_tree -> boolean]

  leaf: (leaf?)

  node: [T, binary_tree, binary_tree -> (node?)]

  val: [(node?) -> T]

  left: [(node?) -> binary_tree]

  right: [(node?) -> binary_tree]

  ⋮

  END binary_tree_adt
```

Note that the binary_tree_adt theory is parametric in the value type T. The first declaration above declares binary_tree as a type. The two recognizer predicates on binary trees leaf? and node? are then declared. The subtypes corresponding to these predicates are written as (leaf?) and (node?), respectively. The three accessors on value (i.e., non-leaf) nodes are then declared. Each of these accessors takes as its domain the subset of binary trees that are constructed by means of the node constructor. This means that an expression of the form val(leaf) is not type correct, i.e., typechecking this expression yields an unprovable TCC proof obligation of the form node?(leaf).

Several axioms are generated in the binary_tree_adt theory. There is an *extensionality* axiom corresponding to each constructor that for the case of nodes asserts that any two value nodes that agree on all the accessors are equal. Each accessor–constructor pair generates an axiom indicating the effect of applying the accessor to an expression constructed using the constructor. The axiom asserting that the recognizers leaf? and node? hold of disjoint subsets of the type of binary trees, is not generated since its size is quadratic in the number of recognizers. However, this property is built into the proof checker simplifications and is also implicit in the semantics of the CASES construct used below.

The theory binary_tree_adt also contains an induction scheme and a few recursion schemes. The induction scheme for binary trees is shown below.

```
  binary_tree_induction: AXIOM
    (FORALL (p: [binary_tree -> boolean]):
       p(leaf)
         AND
         (FORALL (node1_var: T), (node2_var: binary_tree),
                 (node3_var: binary_tree):
                p(node2_var) AND p(node3_var)
              IMPLIES p(node(node1_var, node2_var, node3_var)))
         IMPLIES (FORALL (binary_tree_var: binary_tree):
                    p(binary_tree_var)))
```

In other words, to prove a property of all binary trees, it is sufficient to prove in the base case that the property holds of the binary tree `leaf`, and that in the induction case, the property holds of a binary tree `node(v, A, B)` assuming (the induction hypotheses) that it holds of the subtrees `A` and `B`. The PVS proof checker commands can automatically locate such induction schemes and hence they rarely need to be explicitly invoked.

As a consequence of induction, we can demonstrate the existence and uniqueness of functions defined by recursion over binary trees. It is, however, convenient to have an operation that can be used to explicitly define such recursive functions. Such a "recursion operator" can be parametric in the range type of the function being defined. A generic recursion operator `reduce` is defined in the theory `binary_tree_reduce`. The idea is that we want to define a function f by the following recursion over binary trees:

$$\begin{aligned} f(\texttt{leaf}) &= a \\ f(\texttt{node}(\texttt{v}, \texttt{A}, \texttt{B})) &= g(\texttt{v}, f(\texttt{A}), f(\texttt{B})) \end{aligned}$$

We define such an f by taking a and g as arguments to the function `reduce`. The definition of `reduce` uses the `CASES` construct to define a pattern-matching case split over a datatype value which in this case is a binary tree.

```
reduce(leaf?_fun: range, node?_fun: [[T, range, range] -> range]):
      [binary_tree[T] -> range] =
    LAMBDA (binary_tree_var: binary_tree[T]):
      CASES binary_tree_var OF
        leaf: leaf?_fun,
        node(node1_var, node2_var, node3_var):
            node?_fun(node1_var,
                      reduce(leaf?_fun, node?_fun)(node2_var),
                      reduce(leaf?_fun, node?_fun)(node3_var))
        ENDCASES
```

Following the terminology of Lambert Meertens, Bird refers to datatype recursion operators such as `reduce`, as *catamorphisms*. The typechecker also generates functions `every` and `map` corresponding for binary trees. The former checks that a given predicate on the parameter type T holds of each val component in a binary tree, and the latter maps a function on T over each node in a binary tree.

Ordered binary trees are defined in the theory obt which takes the value type T as a parameter, but also takes a second parameter <= which is constrained (by a subtype restriction) to be a total order (i.e., linear, reflexive, transitive). In theory obt the natural number instance reduce_nat of the reduce function can be used to define the size of a binary tree, i.e., the number of nodes in it, which is then used to provide the termination measure for the ordered? predicate. The every predicate is used to define the ordered? predicate which recursively checks that each subtree is ordered and that the values in the left subtree are no greater than the value at the node, which in turn must be no greater than the values in the right subtree.

```
obt [T : TYPE,  <= : (total_order?[T])] : THEORY
 BEGIN
  IMPORTING binary_tree[T]
  A, B, C: VAR binary_tree
  x, y, z: VAR T
  pp: VAR pred[T]
  i, j, k: VAR nat

  size(A) : nat =
    reduce_nat(0, (LAMBDA x, i, j: i + j + 1))(A)

  ordered?(A) : RECURSIVE bool =
    (IF node?(A)
      THEN (every((LAMBDA y: y<=val(A)), left(A)) AND
            every((LAMBDA y: val(A)<=y), right(A)) AND
            ordered?(left(A)) AND ordered?(right(A)))
      ELSE TRUE ENDIF)
  MEASURE size
END obt
```

4 Bird's Fusion Transformation

Bird [1] starts with the example of an applicative quicksort function which he shows can be obtained as a fusion of the composition of:

(i) a mktree function (an *anamorphism*) which constructs an ordered binary tree from a given list, and
(ii) a flatten function (a *catamorphism*) which flattens the ordered binary tree into an ordered list.

Catamorphisms over binary trees are already captured by the reduce operation shown earlier. Bird defines anamorphisms in terms of the unfold function presented below. This definition is not straightforward. In defining this function, Bird writes that “the recursion is not well-defined unless *f* is ‘well-founded’ in a suitable sense that we will not make precise.” The PVS definition below does make this notion of well-foundedness precise through the use of subtyping and dependent typing. The subtype `smaller(x)` of the type `S` contains all and only those `y` in `S` such that `size(y) < size(x)`, where `<` is the usual ordering on natural numbers.[3] Given a predicate `p` over the type `S`, we write `(p)` for the subtype containing all the elements `x` of `S` such that `p(x)` holds. The dependent function type `well_fnd(p)` contains functions whose domain is `(p)` and that map an `x` in `(p)` to an element of the 3-tuple `[T, smaller(x), smaller(x)]`.

```
unfold   [ T, S:  TYPE, size : [S -> nat] ] : THEORY

  BEGIN
  IMPORTING binary_tree[T]

  p : VAR PRED[S]
  x, y, z : VAR S
  a : VAR T

  smaller(x) : TYPE = { y | size(y) < size(x)}

  well_fnd(p) : TYPE =
     [x : (p) -> [T, smaller(x), smaller(x)]]

  unfold(p, (f: well_fnd(p)))(x) :
     RECURSIVE  binary_tree  =
    (IF p(x)
      THEN (LET (a, y, z) = f(x)
             IN node(a,
                    unfold(p, f)(y),
                    unfold(p, f)(z)))
       ELSE leaf ENDIF)
   MEASURE size(x)

  END unfold
```

The curried recursive function `unfold` takes as arguments a predicate `p` and a function `f` in `well_fnd(p)`. It returns a function which when applied to an `x` satisfying `p`, computes the triple `(a, y, z)` using `f(x)`, and then returns the node constructed from the value `a`, the left subtree `unfold(p, f)(y)`, and the right subtree `unfold(p, f)(z)`. In the base case when `p(x)` is false, `unfold` returns the leaf node `leaf`. When typechecked, the theory generates two termination TCC proof obligations that are automatically proved by the default TCC proof strategy.

[3] The specification of `unfold` can easily be modified to use any well-founded ordering instead of the less-than relation on natural numbers.

The fusion theorem is stated and proved in the theory `fusion` below. The `fusion` theory is parametric in the binary tree value type `T`, the domain type `S` of the unfold operation, and the range type `Range` of the `reduce` operation. The parameter `size` serves the same role here as in the `unfold` theory. We have already seen that `reduce(c, g)(A)` is defined to return `c` when `A` is `leaf`, and `g(a, reduce(c, g)(B), reduce(c, g)(C))` when `A` is of the form `node(a, B, C)`. The fusion theorem asserts that the composition of `reduce` and `unfold`, namely, `reduce(c, g)(unfold(p, f)(x))` which involves two recursive passes can be reduced to a single recursion given by the definition of `fun` below. As with `unfold`, there are two termination TCC proof obligations associated with `fun` that are easily discharged by the default proof strategy.

```
fusion   [ T, S, Range:  TYPE, size : [S -> nat] ] : THEORY
  BEGIN
   IMPORTING unfold[T, S, size]
   p : VAR PRED[S]
   x, y, z : VAR S
   c : VAR Range
   g: VAR [T, Range, Range -> Range]
   a : VAR T

   fun(p, (f : well_fnd(p)), c, g)(x) :
     RECURSIVE  Range =
   (IF p(x)
    THEN (LET (a, y, z) = f(x)
          IN g(a, fun(p, f, c, g)(y),
                  fun(p, f, c, g)(z)))
    ELSE c ENDIF)
   MEASURE size(x)
   ⋮
  END fusion
```

The fusion theorem stated below states the equivalence between the composition of `reduce` with `unfold` and the fused version `fun`. The PVS proof of `fusion` proceeds by a straightforward measure induction on the measure `size(x)` and is in fact proved by a single command that invokes the measure induction strategy. Due to space restrictions, we do not outline the details of this and other proofs in the paper.

```
  fusion: THEOREM
   (FORALL (p, (f: well_fnd(p)), c, g, x):
      reduce(c, g)(unfold(p, f)(x))
    = fun(p, f, c, g)(x))
```

The next step in the development is that of applying the fusion theorem to derive `quicksort` as a fusion of `flatten` and `mktree`, where the latter constructs an ordered binary tree from a given list of elements, and the former constructs a list by an in-order traversal of the resulting tree. The rest of the development of this example is carried out in the theory treesort partially displayed below. This theory takes a parameter list that is identical to that of ordered binary tree theory obt. The theory imports the fusion theory with the PVS datatype list[T] as the actual parameter for both S and Range. The flatten operation is defined as a catamorphism.

```
treesort [T: TYPE, <= : (total_order?[T])]: THEORY
 BEGIN
  IMPORTING fusion[T, list[T], list[T], length[T]], obt[T, <=]
  A, B, C : VAR binary_tree[T]
  x, y, z : VAR list[T]
  a, b, c : VAR T
  p, q    : VAR PRED[T]

  flatten(A) : list[T] =
             reduce(null[T],
                    (LAMBDA a, x, y: append(x, cons(a, y))))(A)
  ⋮
 END treesort
```

A few more preliminary definitions and lemmas are needed. The curried predicate below(a)(b) asserts that b is below a in the ordering <=, and similarly, above(a)(b) asserts that a is above b. The PVS prelude which contains a number of basic theories already defines the filter operation to return a list of those elements in a given list that satisfy a given predicate. The lemmas length_append, length_filter, and filter_length are self-evident.

```
  below(a)(b): bool = (b <= a)

  above(a)(b): bool = NOT (b <= a)

  length_append: LEMMA length(append(x, y)) = length(x) + length(y)

  length_filter: LEMMA
     (FORALL (p: PRED[T]): length(filter(x, p)) <= length(x))

  filter_length: LEMMA
     length(filter(x, below(a)))
      = length(x) - length(filter(x, above(a)))
```

The definition of mktree is given as an anamorphism and is defined using unfold. The function unjoin constructs the triple consisting of the first element of the input list, the list of elements below it in the rest of the input list, and the list of elements above it.

```
unjoin: well_fnd(cons?[T]) =
 (LAMBDA (x: (cons?[T])):
    (LET a = car(x),
         y = cdr(x)
      IN
    (a, filter(y, below(a)), filter(y, above(a)))))

mktree(x) : binary_tree[T] =
   unfold(cons?, unjoin)(x)
```

The quicksort operation can also be directly defined by means of the recursion shown below. This is of course essentially the same definition one would obtain by applying the fusion theorem. This fact is proved by the theorem quicksort_by_fusion. The PVS proof of this theorem consists of a step in which the fusion theorem is used to rephrase the right-hand side in terms of fun, and a second in which the measure induction strategy is used to prove the resulting equality. This results in three trivial subgoals that are proved by applying the lemma length_filter.

```
quicksort(x): RECURSIVE list[T] =
   (CASES x OF
     null : null,
     cons(a, y) : append(quicksort(filter(y, below(a))),
                         cons(a, quicksort(filter(y, above(a)))))
    ENDCASES)
  MEASURE length(x)

quicksort_by_fusion: THEOREM
   quicksort(x) = flatten(mktree(x))
```

As one can see, the progress up to this point has been pretty smooth in the sense that it has been easy to capture the letter and spirit of Bird's definitions and the proofs have been essentially trivial given the automation available in PVS. However, the story takes a somewhat disappointing turn when one tries to show that quicksort returns an ordered permutation of its input by using its "specification", namely, flatten(mktree(x)). Bird loosely sketches an algebraic argument along such lines. We did not try to flesh out Bird's argument but instead proceeded along conventional lines. These proofs were not as straightforward as one might hope. The lemmas filter_every and every_filter are proved in a single step.

```
filter_every: LEMMA every(p, filter(x, p))

every_filter: LEMMA every(p, x) IMPLIES every(p, filter(x, q))
```

The lemmas every_mktree and every_mktree_implies are also essentially trivial and proved in about five steps apiece. The assertion that mktree always constructs an ordered binary tree is stated as ordered?_mktree below. This proof takes up about a dozen steps: the bulk of the work is completed by the measure induction strategy with the assistance of `every_mktree`, but the part involving the right branch of the `mktree` requires the explicit use of the lemma `every_mktree_implies` and the linearity of the ordering relation `<=` given by the type constraint on it.

```
every_mktree: LEMMA
  every(p, x) IMPLIES
  every(p, unfold(cons?, unjoin)(x))

every_mktree_implies: LEMMA
    (FORALL (p, q : PRED[T]):
       (FORALL a: p(a) IMPLIES q(a)) AND
       every(p, x)
     IMPLIES every(q, unfold(cons?, unjoin)(x)))

ordered?_mktree: LEMMA ordered?(mktree(x))
```

It remains to show that the result of `quicksort` is ordered by showing that `flatten` maps an ordered binary tree to an ordered list. This theorem is stated as `ordered?_flatten` below. It is proved in a single step using the standard PVS induction strategy and the lemmas `ordered?_append` and `every_flatten`. The lemma `ordered?_append` took up the most effort since it makes a fairly strong assertion of equivalence, and requires a nested induction. The verification of this proof had to be carried out at a fairly manual level and required about fifty interactions.[4]

```
ordered?_append: LEMMA
   ordered?(append(x, cons(a, y))) =
     (ordered?(x) AND
      ordered?(y) AND
      every((LAMBDA b: b <= a), x) AND
      every((LAMBDA b: a <= b), y))
```

The lemmas `every_flatten` and `every_append` were easily proved in a single step each. Observe that it would have been slightly easier to directly prove the orderedness property of `quicksort` since the lemma `ordered?_append` is the key result needed for this proof.

[4] Healfdene Goguen has been able to simplify this argument by using a definition of `ordered?` that is closer to the corresponding definition over binary trees. This definition checks that the first element in a list lies below all the remaining elements, rather than just the second element as done above, and thus avoids the awkwardness of checking whether a second element exists.

```
every_append: LEMMA
   every(p, append(x, y)) = (every(p, x) AND every(p, y))

 every_flatten: LEMMA
   checkall(p, A) = every(p, flatten(A))

ordered?_flatten: LEMMA  ordered?(flatten(A)) = ordered?(A)
```

The property that quicksort returns a permutation of its input list is stated as count_quicksort below and proved directly of the quicksort function itself. It asserts that the number of occurrences of any element a in the input and output lists agree. This proof is straightforward and uses the measure induction strategy and the lemmas length_append, count_filter, count_append, filter_length, and length_filter. These lemmas are proved trivially.

```
count(a, x): RECURSIVE nat =
  (CASES x OF
    null: 0,
    cons(b, y):
      IF a = b THEN 1 + count(a, y) ELSE count(a, y) ENDIF
   ENDCASES)
  MEASURE length(x)

count_filter: LEMMA
   count(a, filter(x, p)) =
    (IF p(a) THEN count(a, x) ELSE 0 ENDIF)

count_append: LEMMA
   count(a, append(x, y)) = count(a, x) + count(a, y)

count_quicksort: THEOREM
   count(a, quicksort(x)) = count(a, x)
```

The main observation here is that the transformation steps were easily formalized and mechanically verified in PVS, but the correctness proof required a large number of lemmas. Though these lemmas were proved trivially, the overall effort involved was surprisingly large. This seems to suggest that the source of the transformation, flatten(mktree(x)), is not as close to the specification of sorting as one might hope. Even so, the fusion transformation is a significant one since it frequently is the case that a good specification can be obtained by composing two operations using an intermediate data structure. As a simple example, consider the case of checking if a given variable has a free occurrence in a term by constructing the intermediate data structure consisting of the list of free variables in the term and then applying a list membership test.

5 Continuation-Based Program Transformation

Transforming non-tail-recursive functions to tail-recursive (iterative) form is one of the basic forms of program transformation. Wand [23] describes a powerful technique for such transformations where the non-tail-recursive part of the program is captured as a continuation, and the pattern of these continuations is used to convert the continuation into a data structure. This is perhaps one of the most ubiquitously used optimizations in algorithm design. We show how Wand's technique can be formalized using PVS. Consider the example of the list reverse operation. This is defined in the PVS prelude library as shown below, where `l` is a variable ranging over `list[T]`.

```
reverse(l): RECURSIVE list[T] =
  CASES l OF
    null: l,
    cons(x, y): append(reverse(y), cons(x, null))
  ENDCASES
  MEASURE length
```

This definition is not tail-recursive because the recursive call to `reverse` is *surrounded* by the template `append(..., cons(x, null))`. By viewing this part as a *continuation* and adding it as an extra argument, we can convert `reverse` into a tail-recursive operation with an extra continuation argument.

```
revc(l, f): RECURSIVE list[T] =
  CASES l OF
    null: f(l),
    cons(x, y):
      revc(y, (LAMBDA u: f(append(u, cons(x, null)))))
  ENDCASES
  MEASURE length
```

It is easy to confirm that `revc(l, f) = f(reverse(l))`, and hence if `id` is the identity operation on lists, then `reverse(l)` can be computed by `revc(l, id)`. It is also easy to observe that the continuations have the pattern

```
(LAMBDA u: f(append(... append(u, cons(x_n, null)),
                              ..., cons(x_1, null)))).
```

By the associativity of `append` and by its definition, this is just `(LAMBDA u: f(append(u, cons(`x_n`, ..., cons(`x_1`, null)))))`. This continuation can be easily reconstructed from the list `cons(`x_n`, ..., cons(`x_1`, null))`. Hence `revc` can be transformed to the following definition of `reva` where the continuation has been replaced by an accumulator.

```
reva(l, w): RECURSIVE list[T] =
  CASES l OF
    null: w
    cons(x, y):
      reva(y, cons(x, w))
  ENDCASES
  MEASURE length
```

The relation between `revc` and `reva` is

```
reva(l, w) = revc(l, (LAMBDA u: append(u, w))),
```

so that `reverse(l) = reva(l, null)`. As shown by Wand, the sequence of steps shown above for the case of the `reverse` function can be generalized. We present the PVS mechanization of this generalization below. The theory `wand` shown below takes nine theory parameters. The parameters `dom` and `rng` are the domain and range types of the recursive function being transformed. This function is also supplied as the parameter `F`. The definition of `F` involves the use of the parameter `p` as the branching condition for the recursion, the parameter `a` in the base case, and the parameters `d`, `b`, and `c` in the recursion step. The parameter `b` is the continuation-builder, `c` is the recursion destructor, and `d` is the recursion parameter. The parameter `m` supplies the well-founded measure for the recursion. The measure yields a natural number but this can easily be generalized to an arbitrary type equipped with a well-founded relation.

```
wand [dom, rng: TYPE,        %function domain, range
      a: [dom -> rng],       %base case function
      d: [dom-> rng],        %recursion parameter
      b: [rng, rng -> rng],  %continuation builder
      c: [dom -> dom],       %recursion destructor
      p: PRED[dom],          %branch predicate
      m: [dom -> nat],       %termination measure
      F : [dom -> rng]]      %tail-recursive function
  : THEORY
  BEGIN
  ASSUMING
   ⋮
  ENDASSUMING
   ⋮
  END wand
```

There are three important assumptions on the theory parameters for `wand`. The first assumption `assoc` asserts the associativity of the continuation-builder `b`. The second assumption `wf` asserts that the destructor `c` must decrease the measure on any `x` where predicate `p` is false. The final assumption `F_def` asserts that `F` is given by the non-tail-recursive definition using the parameters `p`, `a`, `b`, `c`, and `d`.

```
  u, v, w: VAR rng
  assoc: ASSUMPTION b(b(u, v), w) = b(u, b(v, w))

  x, y, z: VAR dom

  wf : ASSUMPTION NOT p(x) IMPLIES m(c(x)) < m(x)

  F_def: ASSUMPTION
   F(x) =
   (IF p(x) THEN a(x) ELSE b(F(c(x)), d(x)) ENDIF)
```

The continuation-passing variant of F is defined as FC. The main invariant relating F and FC is proved as FFC. The theorem FFC can be proved in a single step using the measure induction strategy.

```
f: VAR [rng -> rng]

FC(x, f) : RECURSIVE rng =
  (IF p(x)
     THEN f(a(x))
    ELSE FC(c(x), (LAMBDA u: f(b(u, d(x)))))
   ENDIF)
MEASURE m(x)

FFC: LEMMA FC(x, f) = f(F(x))
```

The accumulator version of F is given by the function FA. The main invariant relating FC and FA is proved as FAFC. This theorem is also proved in a single measure induction step.

```
FA(x, u): RECURSIVE rng =
 (IF p(x)
    THEN b(a(x), u)
   ELSE FA(c(x), b(d(x), u)) ENDIF)
 MEASURE m(x)

FAFC: LEMMA FA(x, u) = FC(x, (LAMBDA w: b(w, u)))
```

Finally, we can apply this transformation to the non-tail-recursive `reverse` function to obtain the tail-recursive accumulator version. This step is carried out in the theory `reverse` shown below.[5] The first declaration in this theory introduces a *conversion* so that a list operation that is defined only on the domain `(cons?)` (namely on `conses`) is converted to an operation on all lists where the value `null` is returned on `null`. The next declaration imports and renames (as `reverse_wand`) the theory `wand` with `list[T]` for `dom`, `list[T]`

[5] PVS allows considerable leeway in the overloading of names so that we can have both a theory and a function named `reverse`.

for rng, the list identity operation id[list[T]] for a, the expression (LAMBDA (x: (cons?[T])): cons(car(x), null))[6], append for b, cdr for c, null for p, length for m, and reverse for F. This declaration generates three TCCs corresponding to the instances of each of the three assumptions in the theory wand. The associativity assumption on append is already proved in the prelude, and is, in any case, an easy induction. The remaining two TCCs are proved automatically by the default TCC strategy. It is easy then to prove that the function FA in the theory reverse_wand can be used to compute reverse as shown in tail_reverse.

```
reverse [T : TYPE]: THEORY
 BEGIN
  CONVERSION extend[list[T], (cons?[T]), list[T], null[T]]

  reverse_wand: THEORY =
      wand[list[T], list[T], id[list[T]],
             (LAMBDA (x: (cons?[T])): cons(car(x), null)),
              append[T], cdr[T],
              null?[T], length[T], reverse[T]]

  u, x, y, z: VAR list[T]

 tail_reverse: LEMMA FA(x, u) = append(reverse(x), u)

END reverse
```

5.1 Transforming Binary Recursive Schemes

Wand [23] presents several extensions of the above transformation of linear recursive definitions to other nonlinear forms of recursion. We round off our presentation of continuation-based transformation in PVS by illustrating how binary tree recursion schemes can be similarly transformed into iterative form. The theory binary below has some of the same parameters as wand. As with wand, dom and rng are the domain and range of the recursive function, F is the recursive function to be transformed, a is the function used in defining the base case, b is the continuation builder, p is the branching conditional for the recursion, and m is the termination measure. The main difference from wand is that the destructor c has been replaced by a pair of destructors l (for *left*) and r (for *right*), and the recursion parameter d has been eliminated.

[6] The domain subtyping (cons?[T]) of the lambda-abstraction is needed to make it type-correct to invoke car(x). The conversion extend is then automatically introduced to extend this operation to null lists as well.

```
binary [dom, rng: TYPE,       %function domain, range
        a: [dom -> rng],      %base case function
        b: [rng, rng -> rng], %continuation builder
        l: [dom -> dom],      %recursion destructor
        r: [dom -> dom],      %recursion destructor
        p: PRED[dom],         %branch predicate
        m: [dom -> nat],      %termination measure
        F : [dom -> rng]]     %non-tail-recursive function
  : THEORY
  BEGIN
 ASSUMING
  ⋮
  ENDASSUMING
  ⋮
  END binary
```

There are now six assumptions on the parameters. The first assumption asserts the associativity of the continuation builder b.

```
   u, v, w: VAR rng

   assoc: ASSUMPTION b(b(u, v), w) = b(u, b(v, w))
```

The assumptions wfl and wfr assert that the measure m decreases with the destructors l and r on any x where p(x) is false. The fourth assumption states that the measure always returns a positive natural number, and the fifth assumption states that when p(x) is false, the measure m(x) exceeds the sum of the measures m(l(x)) and m(r(x)).

```
   x, y, z: VAR dom

   wfl : ASSUMPTION NOT p(x) IMPLIES m(l(x)) < m(x)

   wfr : ASSUMPTION NOT p(x) IMPLIES m(r(x)) < m(x)

   mpos: ASSUMPTION m(x) > 0

   m_left_right: ASSUMPTION
     NOT p(x) IMPLIES m(x) > m(l(x)) + m(r(x))
```

The sixth assumption introduces the binary recursion scheme characterizing the parameter F.

```
   F_def: ASSUMPTION
    F(x) =
    (IF p(x) THEN a(x) ELSE b(F(l(x)), F(r(x))) ENDIF)
```

The transformation of F to the continuation-passing variant is captured by the function FC where there is now an additional contination argument f, and the result of the left recursive call is now part of the continuation argument given to the right recursive call. Lemma `FFC` captures the main invariant relating `F` and `FC`. It requires an additional constraint relating the contination argument `f` with the contination-builder `b`. The proof of `FFC` employs the measure induction proof strategy.

```
f: VAR [rng -> rng]

FC(x, f) : RECURSIVE rng =
  (IF p(x)
     THEN f(a(x))
    ELSE FC(r(x), (LAMBDA u: b(FC(l(x), f), u)))
   ENDIF)
MEASURE m(x)

FFC: LEMMA
 (FORALL u, v: f(b(u, v)) = b(f(u), v))
    IMPLIES FC(x, f) = f(F(x))
```

In the next transformation step, the continuation argument `f` in `FC` is replaced by an accumulator argument `v` in `FA`. The definition of `FA` is straightforward. The main invariant relating `FC` and `FA` is stated as `FAFC`. The proof of this invariant is also by a single measure induction step.

```
FA(x, v): RECURSIVE rng =
 (IF p(x)
    THEN b(v, a(x))
   ELSE FA(r(x), FA(l(x), v)) ENDIF)
 MEASURE m(x)

FAFC: LEMMA FA(x, v) = FC(x, (LAMBDA w: b(v, w)))
```

The accumulator passing version `FA` of `F` is still not tail-recursive. Wand [23] presents a further transformation to reduce `FA` to tail-recursive form. This "familiar" transformation is that of introducing a stack to save the right recursive calls. The resulting iterative definition is given by `FI` which takes an additional stack argument `Y`. Observe that in the case corresponding to the base case of `FA`, there is a further recursive call to `FI` where the stack argument `Y` is popped. Note that the termination argument for `FI` is nontrivial and the measure used is the sum of the `m(x)` and the `m(y)` for each element `y` in `Y`.

```
X, Y: VAR list[dom]

mlist(X): RECURSIVE nat =
  (IF cons?(X)
     THEN m(car(X)) + mlist(cdr(X))
    ELSE 0 ENDIF)
 MEASURE length(X)

FI(x, v, Y): RECURSIVE rng =
 (IF p(x)
   THEN (IF cons?(Y)
           THEN FI(car(Y), b(v, a(x)), cdr(Y))
          ELSE b(v, a(x))
         ENDIF)
  ELSE FI(l(x), v, cons(r(x), Y))
  ENDIF)
 MEASURE (m(x) + mlist(Y))
```

The invariant relating `FA` and `FI` essentially asserts that when the stack `Y` is empty, `FA` and `FI` coincide. The PVS proof of this lemma is the first nontrivial proof in the mechanization of these transformations. This proof could not be carried out automatically in a single command since the quantifier instantiation heuristics used by PVS were not powerful enough, and also several of the lemmas had to be invoked by hand. This proof required about forty interactive steps. The reader is invited to try out this proof as an exercise.

The main conclusion is that continuation-based transformations are extremely powerful and yet easily verified using PVS. Many examples that pose serious challenges for induction theorem provers [3,4] are often just straightforward instances of such continuation-based transformations. We have formalized these transformations in a schematic manner so that individual instances of these transformations can be easily obtained by means of suitable parameter instantiations rather than through the use of clever induction heuristics.

Wand makes heavy use of mutual recursion in writing his programs. PVS does not admit mutually recursive definitions. Mutual recursion is useful in an informal development, but is quite unwieldy for a formal approach since it can be hard to establish the termination of mutually recursive functions, and their correctness arguments typically involve simultaneous induction. [7]

Wand [23] shows how continuation-based transformations can be applied to nontrivial examples by deriving the alpha-beta form of minimax search from a naive minimax search algorithm. We did not retrace Wand's development steps but instead verified the correspondence between naive minimax search and alpha-beta search in PVS. This specification makes aggressive use of subtyping and dependent typing to constrain the β argument to be at least α,

[7] The SPIKE theorem prover is based on first-order term-rewriting and successfully mechanizes mutual recursion and simultaneous induction [2].

and the search result to lie in the subrange between α and β. The proof was only moderately difficult. It involved a fair amount of case analysis but the potentially laborious aspects of the proof were handled by the decision procedures.

6 Conclusions

We have studied the verification of two specific forms of program transformation using PVS. The first is a fusion theorem due to Bird [1] that can be used to eliminate the intermediate data structure in the composition of two recursive functions in order to obtain a more efficient algorithm. We showed how an applicative quicksort could be derived in this way from the composition of a tree flattening function with a function that constructs an ordered binary tree from a list. The mechanical proofs needed to justify the transformation were essentially trivial, but the functional correctness of the resulting quicksort remained a moderately serious challenge regardless of whether the source or the target of the transformation was used.

The second class of transformations (due to Wand [23]) involves the use of explicit continuation arguments to transform non-tail-recursive definitions into tail-recursive form. The mechanical proofs of these transformations were also mostly trivial. It should also be noted that the manual effort needed to construct and debug these specifications and proofs is quite small: all lemmas, theorems, and proof obligations used in this paper were established in about two or three days. The correspondence between minimax and alpha-beta search was established in about a day.

The main conclusion of this paper is that although general-purpose verification systems like PVS are not customized for program transformation, they are already quite effective at formalizing and verifying the mathematics underlying these transformations. Both the Bird and the Wand transformations could be captured at a useful level of abstraction through the use of the parametric theories. The continuation-based transformations also exploited the assumptions on parameters in order to state the associativity and well-foundedness assumptions. Through the use of predicate subtyping and dependent typing in PVS, we were able to overcome Bird's "problem" with well-foundedness in defining anamorphisms. A versatile proof strategy for measure induction was developed during the course of this work and it played a crucial role in automating all but a few of the proofs.

Conversely, program transformation should be based on general-purpose verification tools, since the mathematical tools needed are much the same as those used in other forms of verification. We have shown how some important transformations can be formalized using the specification tools available in PVS such as parameterized theories, higher-order logic, predicate subtyping, dependent typing, and that these transformations can be mechanized using equational, propositional, and quantificational reasoning combined with arithmetic decision procedures and induction strategies. It will be interesting to see whether other transformational strategies [15] can also be successfully formalized.

References

[1] Richard S. Bird. Functional algorithm design. In Bernhard Möller, editor, *Mathematics of Program Construction '95*, number 947 in Lecture Notes in Computer Science, pages 2–17. Springer Verlag, 1995.

[2] Adel Bouhoula. SPIKE: a system for sufficient completeness and parameterized inductive proofs (system description). In A. Bundy, editor, *Automated Deduction — CADE-12*, number 814 in Lecture Notes in Computer Science, pages 836–840. Springer Verlag, 1994.

[3] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[4] Alan Bundy, Frank van Harmalen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, September 1991.

[5] R. L. Constable, *et al. Implementing Mathematics with the Nuprl*. Prentice-Hall, New Jersey, 1986.

[6] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer Verlag.

[7] David Cyrluk, Patrick Lincoln, Steven Miller, Paliath Narendran, Sam Owre, Sreeranga Ragan, John Rushby, Natarajan Shankar, Jens Ulrik Skakkebæk, Mandayam Srivas, and Friedrich von Henke. Seven papers on mechanized formal verification. Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1995.

[8] Axel Dold. Representing, verifying and applying software development steps using the PVS system. In V. S. Alagar and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, AMAST'95*, number 936 in Lecture Notes in Computer Science, pages 431–445, Montreal, Canada, July 1995. Springer-Verlag.

[9] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[10] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[11] Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[12] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[13] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[14] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[15] Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer Verlag, 1990.

[16] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[17] Sreeranga P. Rajan. Correctness of transformations in high level synthesis. In Steven D. Johnson, editor, *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, pages 597–603, Chiba, Japan, August 1995. Proceedings published in a single volume jointly with ASP-DAC '95, CHDL '95, and VLSI '95, IEEE Catalog no. 95TH8102.

[18] Harald Rueß. Towards high-level deductive program synthesis based on type theory. In *The Tenth Knowledge-Based Software Engineering Conference*, pages 174–183. IEEE Computer Society Press, November 1995.

[19] N. Shankar. Computer-aided computing. In Bernhard Möller, editor, *Mathematics of Program Construction '95*, number 947 in Lecture Notes in Computer Science, pages 50–66. Springer-Verlag, 1995.

[20] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[21] Douglas R. Smith. KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[22] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[23] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.