Software Development in PVS using Generic Development Steps*

Axel Dold

Universität Ulm
Fakultät für Informatik
D-89069 Ulm, Germany
dold@ki.informatik.uni-ulm.de

Proc. of a Dagstuhl Seminar on *Generic Programming*, April 1998.
To be published in Springer LNCS

Abstract. This paper is concerned with a mechanized formal treatment of the transformational software development process in a unified framework. We utilize the PVS system to formally represent, verify and correctly apply generic software development steps and development methods from different existing transformational approaches. We illustrate our approach by representing the well-known divide-and-conquer paradigm, two optimization steps, and by formally deriving a mergesort program.

Keywords: formal verification, mechanized theorem proving, generic development steps, transformational software development

1 Introduction

The transformational approach to software development is widely accepted in modern software engineering. The idea is to start from an abstract formal requirement specification and to apply a series of correctness-preserving software development steps to finally obtain an executable and efficient program.

This paper is concerned with a fully formally mechanized treatment of the transformational development process in a unified framework. The specification system PVS is utilized as a vehicle to formalize, verify, and apply well-known generic software development steps and development methods of different kind and complexity. Applying formal methods to the transformational process is important since it greatly increases the confidence in transformations and their application. Experience has shown the benefits of formal approaches, since we have found errors in published (paper-and-pencil) proofs of transformations and their application.

* This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) project *Verifix*

Development steps from different transformational approaches (PROSPECTRA [4], KIDS [14], CIP [10], BM CALCULUS [2]) are integrated into our framework. They can be grouped into four categories: the first group of development steps consists of problem solving strategies transforming a (non-constructive) requirement specification into a functional specification. Among them, one can find "algorithm theories" [15] used in the KIDS system such as *generate-and-test*, *global search*, *divide-and-conquer*, a general scheme for backtracking algorithms [1] or simpler transformations such as *split-of-postcondition* [4]. In the second group there are transformations which modify and optimize functional specifications [2, 10]. The third group consists of a library of implementations of standard data structures and operations (for example, different implementations for finite sets) while the fourth group is concerned with the translation of functional specifications into imperative programs.

In this paper we illustrate our approach by a formal representation of the general programming paradigm divide-and-conquer using a hierarchy of generic PVS theories, and by two selected optimization steps. These steps are then applied to derive a mergesort program from a formal requirement specification.

All generic development steps are represented within parameterized PVS theories which specify the semantic requirements on the parameters by means of assumptions and define the result of the transformation. Based on the semantic requirements on the parameters, correctness of the generic transformation step can be proved. Application of such a generic step to a given problem is then carried out by providing a concrete instantiation for the parameters and verifying that it satisfies the requirements. This paper extends [3] in several respects: first, the divide-and-conquer scheme is generalized to include n-ary decomposition and composition operators for an arbitrary n. Second, a hierarchy of divide-and-conquer theories is presented which enables a much more convenient application to specific problems. Third, the development of a sorting algorithm by applying several development steps is outlined, and fourth, the formal representation of two selected optimization steps is described.

We proceed as follows: the next section gives a brief introduction to PVS and presents the general form of requirement specifications. Sect. 3 is concerned with the representation of the divide-and-conquer paradigm while in Sect. 4 a formal treatment of two optimization steps is discussed. These transformation patterns are then applied in Sect. 5 to derive a mergesort program. Sect. 6 concludes. All PVS theories and proof scripts are available from the author.

2 Preliminaries

First, a brief introduction to PVS, the formal framework we are working in, is provided, then the general form of requirement specification is presented.

2.1 A Brief Introduction to PVS

The PVS system [8, 9] combines an expressive specification language with an interactive proof checker that has a reasonable amount of theorem proving capabil-

ities. The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, among others, the product type constructor `[A,B]` and the function type constructor `[A→B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*. A distinctive feature of the PVS specification language are *predicate subtypes*: the subtype {`x:A | P(x)`} consists of exactly those elements of type `A` satisfying predicate `P`. Predicate subtypes are used, for instance, for explicitly constraining the domains and ranges of operations in a specification and to define partial functions. Predicates in PVS are elements of type `bool`, and `pred[A]` is a notational convenience for the function type `[A→bool]`. Sets are identified with their characteristic predicates, and thus the expressions `pred[A]` and `set[A]` are interchangeable. For a predicate `P` of type `pred[A]`, the notation `(P)` is just an abbreviation for the predicate subtype {`x:A | P(x)`}. In general, type-checking with predicate subtypes is undecidable; the type-checker generates proof obligations, so-called *type correctness conditions* (TCCs) in cases where type conflicts cannot immediately be resolved. A PVS expression is not considered to be fully type-checked unless all generated TCCs have been proved. PVS only allows total functions, hence it must be ensured that all (recursive) functions terminate. For this purpose, a well-founded ordering or a *measure function* is used. The definition of a recursive function `f` generates a TCC which states that the measure function applied to the recursive arguments decreases with respect to a well-founded ordering. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories. Specifications are realized as possibly parameterized PVS theories and theory parameters can be constrained by means of *assumptions*. When instantiating a parameterized theory, TCC's are automatically generated according to the assumptions.

Proofs in PVS are presented in a sequent calculus. There exists a large number of atomic commands (for quantifier instantiation, automatic conditional rewriting, induction etc.), and in addition PVS has an LCF-like strategy language for combining inference steps into more powerful proof strategies. The built-in strategy `GRIND`, for example, combines rewriting with propositional simplification using BDDs and decision procedures.

2.2 Requirement Specification

Initial requirement specifications are given as quadruples

$$P = (D : TYPE, R : TYPE, I : pred[D], O : [D, R \to bool])$$

where D and R denote the problem domain and range, respectively, I is a predicate on D describing admissible inputs, and O is the input/output relation of the problem. A solution to such a problem is either

1. a function $f : D \to R$ such that $\forall\, x : (I).O(x, f(x))$
2. or a function $F : D \to set[R]$ such that $\forall\, x : (I).F(x) = \{z : R \mid O(x, z)\}$.

In the first case, function f calculates *one* solution to the problem, while in the latter case, function F produces the set of all solutions to the problem.

3 The Divide and Conquer Paradigm

The well-known algorithmic paradigm *divide-and-conquer* [13, 15] is based on the principle of solving primitive problem instances directly, and large problem instances by decomposing them into "smaller" instances, solving them independently and composing the resulting solutions. Typically, the smaller problem instances are solved in the same way as the input problem which results in a recursive algorithm. However, some of the subproblem instances could be solved in a different way. Fig. 1 illustrates the paradigm.

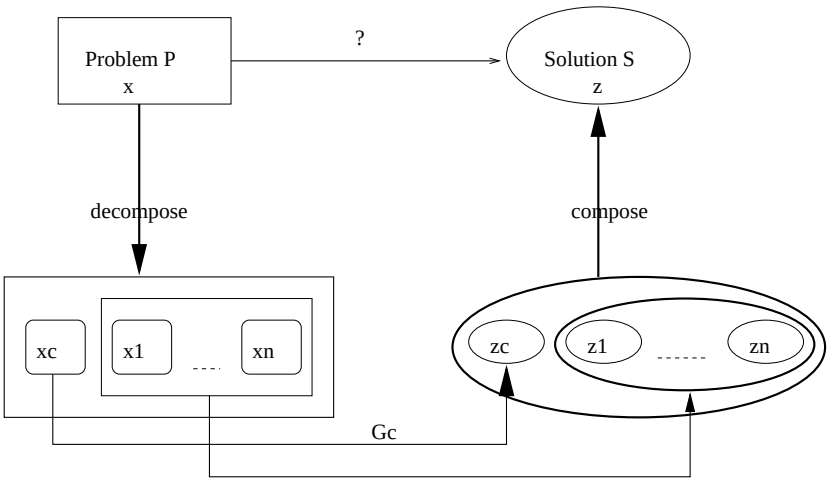

Fig. 1. Divide-and-Conquer Paradigm

3.1 The General Scheme

Our formalization follows [13]. However, our scheme is more general allowing the decomposition operator to produce an arbitrary but fixed number of subproblems.

Let (D, R, I, O) denote the problem specification, and (D_c, R_c, I_c, O_c) denote a "component problem" with a solution G_c, i.e. $\forall\, x_c : (I_c).O_c(x_c, G_c(x_c))$ Our most general theory uses a decomposition operator which decomposes a problem instance x into an arbitrary but fixed number N ($N \geq 1$) of subproblem instances $x_1, \ldots, x_n$ and a component problem instance x_c. The component problem instance x_c is solved directly by G_c producing an output z_c, and the problem instances $x_1, \ldots, x_n$ are solved recursively producing a vector of solutions $z_1, \ldots, z_n$ which are then composed by the composition operator to form a solution to the input problem x. To ensure termination of this process a measure

function on problem domain D is required. This completes the parameter list of the general scheme. [1]

```
% parameter list of general divide-and-conquer theory                    1
D:TYPE, R:TYPE, I:pred[D], O:[D,R -> bool],       % problem spec
Dc:TYPE, Rc:TYPE, Ic:pred[Dc], Oc:[Dc,Rc -> bool],% component problem
Gc          : [(Ic) -> Rc],                  % solution of (Dc,Rc,Ic,Oc)
N           : posnat,                        % number of subproblems
decompose   : [D -> [Dc, [below(N) -> D]]], % decomposition operator
dir_solve   : [D -> R],                      % primitive solution
compose     : [Rc, [below(N) -> R] -> R],    % composition operator
primitive? : pred[D],                        % set of primitive problems
mes         : [D -> nat]                     % termination measure on D
```

Five assumptions specify the semantic requirements:

```
a1: ASSUMPTION I(x) ∧ ¬ primitive?(x)                                    2
                ⇒ ∀n. mes(decompose(x).2(n)) < mes(x)
a2: ASSUMPTION Ic(xc) ⇒ Oc(xc, Gc(xc))
a3: ASSUMPTION I(x) ∧ primitive?(x) ⇒ O(x, dir_solve(x))
a4: ASSUMPTION I(x) ∧ ¬ primitive?(x) ∧ Oc(decompose(x).1, zc)
  ⇒ ∀v.(∀n. O(decompose(x).2(n), v(n))) ⇒ O(x, compose(zc,v))
a5: ASSUMPTION I(x) ∧ ¬ primitive?(x)
              ⇒ Ic(decompose(x).1) ∧ ∀n. I(decompose(x).2(n))
```

They state that

1. all subproblems created by the decomposition operator are smaller than the original problem with respect to the measure `mes`.
2. function Gc solves the component problem (D_c, R_c, I_c, O_c).
3. a primitive problem can be solved by `dir_solve`.
4. solutions to the subproblems $z_1, \ldots, z_n$ and a solution z_c to the component problem can be composed to build a solution to the input problem.
5. all subproblems generated by the decomposition operator satisfy the input conditions I of (D, R, I, O) and I_c of (D_c, R_c, I_c, O_c).

Based on these parameters, the generic divide-and-conquer algorithm can be defined by function `f_dc` in 3:

```
f_dc(x:(I)) : RECURSIVE R =                                              3
 IF primitive?(x) THEN dir_solve(x)
 ELSE LET (xc,dcomp)        = decompose(x),
          subsolutions      = λn. f_dc(dcomp(n))
     IN compose(Gc(xc), subsolutions)
 ENDIF
  MEASURE mes(x)
```

One has to prove that `f_dc` is a correct solution of the problem (D, R, I, O):

[1] To increase the readability of PVS specifications the syntax has liberally been modified by replacing some ASCII codings with a more familiar mathematical notation.

Theorem 1 (Correctness of Divide-and-Conquer).

```
dc_correctness: THEOREM ∀x:(I). O(x, f_dc(x))
```

We prove the theorem by measure-induction using measure function `mes` with the usual $<$ ordering on `nat`:

```
% measure induction principle
% T: TYPE, M: TYPE, m: [T → M], <: (well_founded?[M])
measure_induction: LEMMA ∀p:pred[T]:
 (∀x. (∀y. m(y) < m(x) ⇒  p(y)) ⇒ p(x)) ⇒ (∀x. p(x))
```

After the built-in measure-induction strategy has been applied the PVS proof state looks as follows: [2]

```
dc_correctness:

{-1}   ∀y:(I). mes(y) < mes(x!1) ⇒ O(y, f_dc(y))
  |-------
{1}    O(x!1, f_dc(x!1))
```

The proof can be finished by expanding the definition of `f_dc`, case-analysis on `primitive?`, and application of the theory assumptions 2.

3.2 A Hierarchy of Divide-and-Conquer Theories

In order to have more adequate support for application of the divide-and-conquer (DC) paradigm to specific problems we have developed a hierarchy of generic DC theories where the top most theory models the most general DC scheme presented above and lower theories in this hierarchy are specializations of the general scheme, see Fig. 2. Each child in this hierarchy is a partial instantiation or specialization of its parent theory. This hierarchy permits choice of the most specific theory in order to solve a given problem.

In theory `DC_ID` the component problem is given by $(D_c, D_c, TRUE, =[D_c])$ where $TRUE$ denotes the constant true predicate, and $=[D_c]$ denotes the equality relation on D_c, i.e. G_c is the identity on D_c. Theory `DC_BIN`, a refinement of `DC_ID`, uses a binary decomposition and composition operator where the type D_c of the (trivial) component problem is identified with domain D of problem (D, R, I, O). Theory `DC_1` specifies a binary decomposition and composition operator where the type D_c of the (trivial) component problem is in general different from D.

[2] The prover maintains a proof tree. The goal is to construct a proof tree which is complete, in the sense that all of the leaves are recognized as true. Each proof goal is a sequent consisting of a sequence of antecedent formulas (indicated by negative numbers) and consequent formulas (indicated by positive numbers). The intuitive meaning of such a goal is that the conjunction of the antecedents implies the disjunction of the consequents. Expressions of the form `x!i` denote constants introduced for universal-strength quantifiers.

The theories on the lowest level in the hierarchy define specific composition and decomposition operators for lists. Theory DEC_LST1 decomposes a (non-empty) list into its head and tail, theory DEC_LST2 splits a list into roughly two equal parts, while CMP_LST1 uses "cons" as its composition operator, and finally, theory CMP_LST2 composes lists by concatenation. In a similar way, the hierarchy can be extended for data types such as finite sets, balanced binary trees etc.

To illustrate a specific parent-child pair of the hierarchy theories DEC_LST2 and DC_BIN are explained in some more details. The parameters of DC_BIN are as follows:

```
% formal parameters of theory DC_BIN                                    4
D:TYPE, R:TYPE, I:pred[D], O:[D,R -> bool], % problem spec
decompose  : [D -> [D,D]],                  % decomposition operator
dir_solve  : [D -> R],                      % primitive solutions
compose    : [R,R -> R],                    % composition operator
primitive? : pred[D],                       % set of primitive problems
mes        : [D -> nat]                     % termination measure on D
```

There are four assumptions on the parameters. They are similar to assumptions a1,a3,a4,a5 of the general scheme 2 specialized to binary decomposition and composition operators:

```
% assumptions for DC_BIN                                                5
a1: ASSUMPTION I(x) ∧ ¬ primitive?(x) ⇒ LET (x1,x2) = decompose(x) IN
               ⇒ mes(x1) < mes(x) ∧ mes(x2) < mes(x)
a3: ASSUMPTION I(x) ∧ primitive?(x) ⇒ O(x, dir_solve(x))
a4: ASSUMPTION I(x) ∧ ¬ primitive?(x) ⇒ LET (x1,x2) = decompose(x) IN
               O(x1,z1) ∧ O(x2,z2) ⇒ O(x, compose(z1,z2))
a5: ASSUMPTION I(x) ∧ ¬ primitive?(x)
               ⇒ I(decompose(x).1) ∧ I(decompose(x).2)
```

Note that there is no assumption for the component problem since this has already been specialized in theory DC_ID.

Consider now theory DEC_LST2. Its formal parameter list is as follows:

```
% formal parameters of theory DEC_LST2                                  6
T          : TYPE,                          % list element type
R          : TYPE,                          % problem range
I          : pred[list[T]],                 % admissible inputs
O          : [list[T],R -> bool],           % input/output relation
dir_solve  : [list[T] -> R],                % primitive solutions
compose    : [R,R -> R]                     % binary composition operator
```

The problem domain is fixed by the type list[T], and the decomposition operator is defined by listsplit in 7. Here, first_n returns the first n elements of a list, and cut_n cuts n elements from a list. Note that the primitive? predicate is determined by the choice of the decomposition operator: primitive instances are lists containing at most one element; the measure is given by the length of the list.

```
% decomposition operator and primitive predicate                    7

decompose(x:list[T]): [list[T], list[T]] =
 IF length(x) < 2 THEN (x,x)    % dummy value
 ELSE listsplit(x) ENDIF

listsplit(l:{l1:list[T] | length(l1) >= 2}): [list[T], list[T]] =
 (first_n(l, div2(length(l))), cut_n(l, div2(length(l))))

primitive?(x:list[T]): bool = (length(x) < 2)
```

The assumptions on the parameters are given by `a3,a4,a5` in 5 using the specific operators above 7. A specialization of theory `DC_BIN` is then obtained by importing the instantiated theory:

```
IMPORTING
 DC_BIN[list[T],R,I,O,decompose,dir_solve,compose,primitive?,length]
```

TCCs are generated according to the assumptions 5. The only non-trivial TCC is to prove `a1`, stating that the length of the lists produced by `listsplit` is smaller than the length of the input list. This requires additional lemmas for list functions `first_n` and `cut_n`.

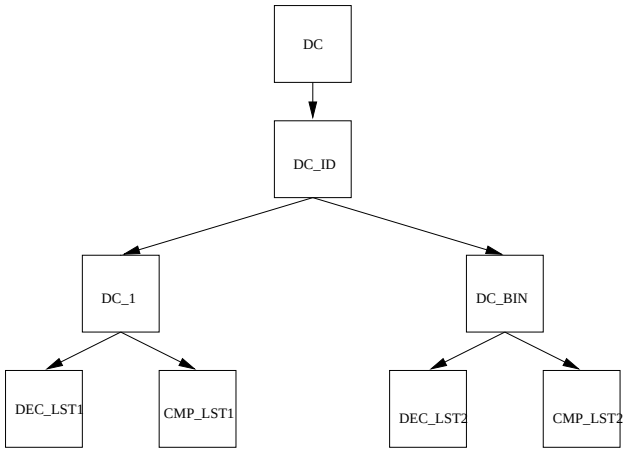

Fig. 2. A Hierarchy of Divide-and-Conquer Theories

In order to solve problems using the DC hierarchy the following general derivation strategy is utilized:

1. choose an appropriate DC theory from the hierarchy depending on the structure of the problem and/or the data types used in the specification

2. provide concrete instantiations for the formal theory parameters which are easy to find (i.e. determined by the problem)
3. set up new specifications for the parameters which are not obvious
4. solve the new problems in a similar way.

4 Optimization Steps

This section deals with the formal treatment of two examples of generic optimization steps: transformation of a linear recursive function into an equivalent tail-recursive one, and the introduction of a table to store and restore computed values.

4.1 Accumulation Strategy

Given a linear recursive function

$$f(x) = \text{IF } B(x) \text{ THEN } H(x) \text{ ELSE } p(K(x), f(K_1(x)))$$

the goal is to transform it into an equivalent tail-recursive function. In a tail-recursive function recursive calls appear on the top level of expressions, i.e. they have the form $f(K(x))$ and do not occur within an expression. Wand [16] presents a transformation where the non-tail-recursive part of f (expression p) is captured as an extra continuation argument. This results in a tail-recursive function which is further transformed by replacing the continuation by an accumulator. A PVS formalization of this transformation has been carried out by Shankar [12]. Partsch [10] presents a transformation rule which combines both steps and directly converts the linear recursive function into a tail-recursive one. In the following a formal treatment of this rule is given. The basic idea of this transformation is to add a parameter that accumulates the function result. It is required that operation p has a neutral element e and is associative. The respective PVS theory takes 10 parameters:

```
% parameters of accumulation strategy                          8
D  : TYPE,          % domain of f
R  : TYPE+,         % range of f
B  : pred[D],       % termination condition
H  : [D -> R],      % base case solution
p  : [R,R -> R],    % associative operator
K  : [D -> R],      % non-recursive argument modifier
K1 : [D -> D],      % recursive argument modifier
e  : R,             % neutral element w.r.t. p
m  : [D -> nat],    % measure
f  : [D -> R]       % linear function to be transformed
```

There are four assumptions on the theory parameters. The first one asserts the associativity of `p`, the second one states the neutrality of `e` with respect to `p`. The third one ensures termination of `f`, while the fourth one constrains `f` to have a linear form. A tail-recursive variant is then given by function `f1`:

```
f1(r:R,x:D) : RECURSIVE R =                                          9
 IF B(x) THEN p(r, H(x)) ELSE f1(p(r, K(x)), K1(x)) ENDIF
  MEASURE m(x)
```

Theorem 2 (Correctness of Accumulation Strategy).

```
f_trans(x:D): {r:R | r = f(x)} = f1(e,x)
```

Note that the correctness is expressed by type constraints in the definition of `f_trans`. The proof is by establishing an invariant `invar` (10) which itself is accomplished by a straightforward measure-induction proof.

```
invar: LEMMA f1(r,x) = p(r,f(x))                                     10
```

4.2 Memoization

In a recursive function's computation inefficiencies often arise due to multiple evaluations of identical calls. This is especially the case if the recursion is nested or cascaded. The idea of the transformation described in this subsection is to store computed values in a table and to remember that this computation has been carried out. If the result of this computation is needed it is looked up in the table. We present a formal representation of a generalization of the transformation called *memoization* [10]. The transformation is applicable to recursive functions which have the form

$$f(x) = \text{IF } B(x) \text{ THEN } H(x) \text{ ELSE } E(x, f(K_0(x)), f(K_1(x)), \ldots, f(K_n(x)))$$

The respective PVS theory is parameterized as follows:

```
% parameters of memoization                                          11
D : TYPE,                          % domain of f
R : TYPE,                          % range of f
B : pred[D],                       % termination condition
H : [D -> R],                      % base case solution
N : nat,                           % N + 1 recursive calls
E : [D, [upto(N) -> R] -> R],      % expr. containing the recursive calls
K : [upto(N) -> [D -> D]],         % argument modifiers K0,K1, ..., KN
m : [D -> nat],                    % termination measure
f : [D -> R]                       % function to be transformed
```

There are two assumptions on the parameters: the first one constrains the input function f to have the specific form while the second one assumes that the arguments for the recursive calls decrease with respect to the measure `m`. The result of the transformation is given by function `f1` in 12.

```
f1(x:D)(a:(invar?)): RECURSIVE {a1:(invar?) | P_inv(x,a,a1)} =       12
 IF isdef(a,x) THEN a ELSIF B(x) THEN add(a, x, H(x))
 ELSE LET final_map = concat(λn. f1(K(n)(x)))(N)(a) IN
  add(final_map, x, E(x, λn. final_map^(K(n)(x))))
 ENDIF
  MEASURE m(x)
```

`f1` has an additional argument of type `map` (association list) (such that an invariant 13 is preserved) which denotes the table in which the computed values are stored. The type `map` associates elements of an index set with values: `empty` denotes the empty map and `add(m,i,e)` adds a new association to `m`: value `e` is associated with index `i`. There are two functions `isdef(m,i)` for testing whether the map has a defined value for a given index, and `^` for retrieving a value associated with a given index (if there is a defined value). Function `f1` is initially called with the empty map. If a result for some argument has already been computed then the value is looked up in the table otherwise it is calculated and then inserted into the table.

```
invar?(a:map): bool = ∀y:D. isdef(a,y) ⇒ a^y = f(y)                    13
P_inv(x:D,a1,a2:map): bool =
 isdef(a2,x) ∧ ∀y:D. isdef(a1,y) ⇒ (isdef(a2,y) ∧ a1^y = a2^y)
```

The invariant `invar?(a)` in 13 states that if there exists an entry in `a` for some `y` then the value associated with `y` equals to `f(y)`. `P_inv(x,a1,a2)` states that map `a2` has a defined value for `x` and that `a2` preserves all associations of `a1`. Function `concat` realizes the function composition $f_1(K_N(x)) \circ \cdots \circ f_1(K_0(x))$.

Theorem 3 (Correctness of Memoization).

```
correctness : THEOREM  ∀x:D. f(x) = f1(x)(empty)^x
```

Retrieving the value from the map `f1(x)(empty)` at index `x` equals to `f(x)`. This theorem is trivial to prove using the type constraints of the range of `f1` since the invariant directly states the correctness. However, some non-trivial TCC's are generated when type-checking this theory, since it has to be ensured that the invariant is preserved during computation. To prove this, additional lemmas have to be established which state that `concat` preserves the invariant and that the concatenation of maps can be split. Furthermore, it must be proved that there exists table entries for all $K_i(x)$, for $i \in \{0 \ldots N\}$ in the final map.

5 Derivation of a Sorting Algorithm

In the following the derivation strategy from Sect. 3.2 is applied to derive the well-known mergesort algorithm using different divide-and-conquer schemes. The problem specification of sorting a list of elements of some type T with respect to some total order $\leq$ can be stated as follows:

```
% specification of the sorting problem                                 14
% T:TYPE+; <=: (total_order?[T]); A,B: VAR list[T]
P_sort := (D:= list[T], R:= list[T], I:= TRUE, O:= λA,B. sorted?(A,B))

sorted?(A,B): bool = ordered?(B) ∧ perm?(A,B)
perm?(A,B): bool = ∀t: count(t,A) = count(t,B)
```

`ordered?` holds if all elements of a list are in nondecreasing order. Predicate `perm?(A,B)` holds if list `B` is a permutation of list `A`: the number of occurrences of any element in `A` and `B` agree.

In order to solve this problem an appropriate DC theory from the library has to be chosen. Since we decide to derive a mergesort algorithm we choose theory `DEC_LST2` where the decomposition operator splits a list into roughly two equal length parts (see 6 for the parameter list). Parameters `T,R,I` and `O` are determined by the sorting problem; `dir_solve` simply returns the input list since a list with at most one element is trivially sorted. The only creative work to do is to find an appropriate composition operator satisfying the assumption `a4`. Looking at this assumption one recognizes that the composition operator must "merge" two already sorted parts. Rather than inventing a merge function and proving its correctness, it is derived by first setting up a new specification and then selecting and instantiating another appropriate DC theory from the hierarchy:

```
% requirement specification of merge                                    15
P_merge :=
 (D1:= [list[T],list[T]],
  R1:= list[T],
  I1:= λ (d:D1). ordered?(d.1) ∧ ordered?(d.2),
  O1:= λ (d:D1,C:list[T]). perm?(append(d.1,d.2), C) ∧ ordered?(C))
```

To solve this problem a theory with a specific composition operator is selected: theory `CMP_LST1` uses the list operator `cons` as its composition operator. The formal parameters of `CMP_LST1` then have to be instantiated properly.

```
% formal parameters of theory CMP_LST1                                  16
T           : TYPE,                        % type of list elements
D           : TYPE,                        % problem domain
I           : pred[D],                     % input condition
O           : [D, list[T] -> bool],        % input/output condition
decompose   : [D -> [T, D]],               % decomposition operator
dir_solve   : [D -> list[T]],              % solution of primitive instances
primitive?  : pred[D],                     % primitive instances
mes         : [D -> nat]                   % measure on domain D
```

Note that the problem range `R` is fixed here and defined by `list[T]`. The main effort to solve this problem is to find an appropriate decomposition function. In the sequel, let `A,B` denote the input tuple of type `D1`. Obviously, the problem is primitive if either `A` or `B` are empty. In this case `B` respectively `A` is returned (`dir_solve`). A non-primitive problem instance is split according to the result of comparing the two head elements: either the tuple `(car(A), (cdr(A), B))` is returned in case `car(A) <= car(B)` holds; otherwise the tuple `(car(B), (A, cdr(B)))` is returned. A decreasing measure on the domain `D1` is given by the sum of `A` and `B`'s length.

Most of the TCCs can simply be proved with the built-in `grind` strategy. The only non-trivial TCC (which requires additional proof effort) is the instantiated

composition assumption stating that solutions to the subproblems are correctly composed by the composition operator `cons`. The proof requires some additional properties about list permutations. Due to space limitations we omit the details here. After all TCCs have been proved an operational specification of mergesort has been derived which is correct by construction. It has the form:

```
mergesort(x:list[T]): RECURSIVE list[T] =                                  17
 IF length(x) < 2 THEN x
 ELSE merge(mergesort(listsplit(x).1), mergesort(listsplit(x).2)) ENDIF

merge(x:D1): RECURSIVE R1 =
 IF primitive?(x) THEN dir_solve(x)
 ELSE cons(decompose(x).1, merge(decompose(x).2)) ENDIF
```

This operational specification may be further improved by the optimization techniques presented in Sect. 4. For example, `merge` has a linear recursive form and the transformation from Sect. 4.1 can be applied (see 8 for the formal parameters): for the expression `p` the list concatenation operator `append` can be instantiated which is associative and has the empty list as neutral element. The `cons` operator can be realized by the concatenation operator. Functions `K`, and `K1` are realized by selecting the first and second part of the decomposition, respectively, and the measure is given by the length of the list. The generated TCCs can be proved automatically using lemmas from the PVS list library. On the other hand, `mergesort` has a tree-like recursive form and therefore the memoization technique presented in Sect. 4.2 can be applied. This might be useful if in the list to be sorted elements occur more than once or if even the same subsequences occur more than once.

Fig. 3 illustrates the derivation process: `mergesort` is derived using `DEC_LST2` where the composition operator, i.e. merging two ordered lists, is derived using `CMP_LST1`. Then both `mergesort` and `merge` are further optimized.

6 Conclusions

In this paper we have presented an approach to rigorous formal mechanized treatment of the transformational software development process in a unified framework. This process comprises the formalization, verification, and application of generic software development steps. We showed that transformations from different existing approaches can be integrated into this framework. The approach has been illustrated by a formal representation of the divide-and-conquer paradigm, two optimization transformations, and a derivation of *mergesort* using these steps.

In most cases the built-in strategies are powerful enough to prove the correctness of the instantiation of the generic software artifacts automatically. However, sometimes additional knowledge about the application domain is required. In general, it is advisable and useful to establish specialized theories for specific application domains. For example, we have constructed a "sorting theory"

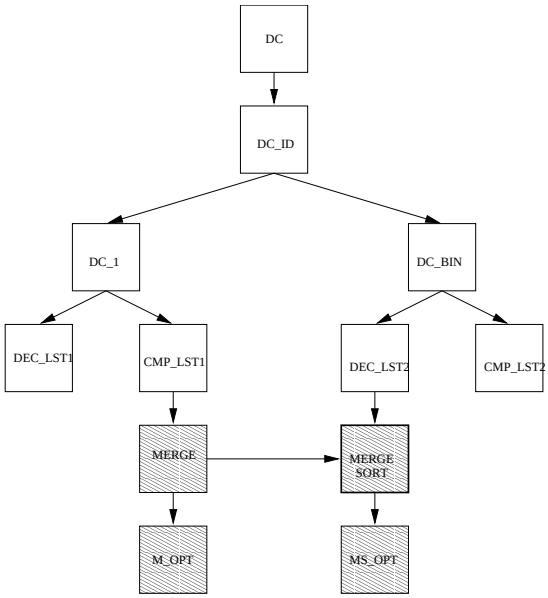

Fig. 3. Derivation of the Mergesort Algorithm

which includes the elementary definitions for sorting problems as well as a set of proved facts. In addition, specialized proof strategies can be defined which make use of the proved facts. Such a sorting theory then provides a basis for the derivation of sorting algorithms. In summary, I believe that the set of generic software development steps together with domain-specific knowledge provide a powerful tool for the formal transformational software development process: specific design/implementation decisions are made by the choice of an appropriate generic theory, and the correctness of an instantiation may be proved with the help of domain-specific knowledge.

The work presented in this paper is part of an ongoing effort to rigorous formal representation of software development steps. Besides extending the existing set of formalized development steps and development methods, we are in the process of applying the approach to the construction of a non-trivial compiler implementation. Starting from a given compiling specification which is proved to be correct with respect to the language semantics, a series of development steps is applied to derive an implementation in the source language itself which is then correct by construction. The development steps involved include functional as well as data structure refinement. This work is carried out within the context of the German compiler verification project *Verifix*.

Related Work

The mechanization of transformational approaches has been considered by several researchers.

In [5], for example, program transformations for recursion removal are expressed as second-order patterns defined in the simply typed λ-calculus. Kreitz [7] utilizes the NUPRL system which is based on a constructive type theory as a formal framework for representing existing approaches to program synthesis (such as KIDS [14], for example). Among other "algorithm theories" he presents a formal mechanization of a (binary) divide-and-conquer scheme. Rueß [11] formalizes the *divide-and-conquer* paradigm in his dissertation [11] using the calculus of constructions and has given a formal verification with the LEGO proof checker. More recently, Kolyang et.al [6] present a prototype development environment for implementing and applying transformations consisting of the tactical theorem prover Isabelle and a graphical user interface based on the toolkit TK. Their basic idea is to separate the soundness of transformations (stated by synthesis theorems) from the pragmatics of their application. They give a mechanization of the *global search* paradigm in Isabelle/HOL.

Acknowledgments

I would like to thank my colleague Martin Strecker for helpful comments on a draft version of this paper. The constructive criticisms and suggestions provided by the anonymous referees have greatly improved the paper.

References

1. Klaus Achatz and Helmuth Partsch. From descriptive specifications to operational ones: A powerful transformation rule, its applications and variants. Technical Report 96-13, Universität Ulm, December 1996.
2. Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM - Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.
3. Axel Dold. Representing, Verifying and Applying Software Development Steps using the PVS System. In V.S. Alagar and Maurice Nivat, editors, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95, Montreal*, volume 936 of *Lecture Notes in Computer Science*, pages 431–445. Springer-Verlag, 1995.
4. Berthold Hoffmann and Bernd Krieg-Brückner (Eds.). *Program Development by Specification and Transformation - The PROSPECTRA Methodology, Language Family, and System*, volume 680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
5. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order-patterns. *Acta Informatica*, 11:31–55, 1978.
6. Kolyang, B. Wolff, and T. Santen. Correct and User-Friendly Implementations of Transformation Systems. In J. Woodcock M.-C. Gaudel, editor, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 629–648, 1996.
7. C. Kreitz. Metasynthesis - deriving programs that develop programs. Technical Report AIDA-93-03, Fachgebiet Intellektik, Technische Hochschule Darmstadt, 1993.

8. 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.
9. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga NY, 1992. Springer-Verlag.
10. Helmuth A. Partsch. *Specification and Transformation of Programs - A Formal Approach to Software Development*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
11. H. Rueß. *Formal meta-programming in the calculus of constructions*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1995.
12. N. Shankar. Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1–3):33–57, 1996.
13. Douglas R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987.
14. Douglas R. Smith. KIDS - a semi-automatic program development system. *IEEE - Transactions on Software Engineering, Special Issue on Formal Methods in Software Engineering*, 16(9):1024–1043, September 1990.
15. Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2-3):305–321, 1990.
16. Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.