Representing, Verifying and Applying Software Development Steps using the PVS System*

Axel Dold

Abt. Künstliche Intelligenz,
Universität Ulm,
D-89069 Ulm, Germany
dold@informatik.uni-ulm.de

Abstract. In this paper generic software development steps of different complexity are represented and verified using the (higher-order, strongly typed) specification and verification system PVS. The transformations considered in this paper include "large" powerful steps encoding general algorithmic paradigms as well as "smaller" transformations for the operationalization of a descriptive specification. The application of these transformation patterns is illustrated by means of simple examples. Furthermore, we show how to guide proofs of correctness assertions about development steps. Finally, this work serves as a case-study and test for the usefulness of the PVS system.

1 Introduction

The methodology of stepwise refinement is widely accepted in modern software engineering. The idea is to start from an abstract requirement specification of a given problem and successively apply correctness preserving transformation patterns to finally yield an executable program. These transformations can comprise development steps of different complexity. One large powerful step can be sufficient to synthesize a program while a series of smaller steps has to be applied to reach a similar result.

In this paper we focus on the representation of development steps of different complexity and kind in a rigorous formal manner. "Large" steps encode general programming knowledge which forms the basis of many algorithms. Such knowledge is frequently applied implicitly when constructing programs but even when it is explicitly described in the literature it often appears informal and lacks a rigorous formal (error-free) treatment. In a formal treatment, such development steps can be represented as schematic algorithms which, instantiated with a

* to appear in the Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95. Part of the research reported herein has been funded by the German Federal Ministry of Research and Technology (BMFT) under contract no. 01 IS 203 K5 (KORSO).

specific problem, synthesize a solution to this problem. These "algorithm theories" have intensively been investigated by Doug Smith [12, 13, 14] who defines, among other things, a hierarchy of algorithm theories encoding well-known programming paradigms such as *divide-and-conquer*, *global search*, *generate-and-test* and others. However, his approach is only semi-formal, some important aspects remain informal.

The transformations developed in the CIP-project and its descendants [2, 3, 10] can be considered as "smaller" development steps since they mainly operate on the level of functions. Among them one can find transformations for optimizing functions, recursion simplification, and as well, steps which operationalize a descriptive specification.

The goal of this paper is to completely formalize and verify two selected development steps, one of each kind, and to correctly apply them to examples. In order to represent software development steps higher-order logic greatly facilitates the formalization process. Therefore, and in order to have adequate system support we choose the specification and verification system PVS [7] in which the whole process of representation and verification can be carried out. The Prototype Verification System (PVS) consists of a higher-order specification language with a rich typing system, a set of supporting tools for creating, analyzing, modifying and documenting theories and proofs, and a powerful interactive Gentzen-style theorem prover. Furthermore, a library of standard theories such as natural numbers, polymorphic sets, and lists, booleans, relations is predefined. The type system provides type constructors to form dependent and non-dependent function, tuple, record, and semantic subtypes. Specifications are realized as PVS theories which can be parameterized where the parameters can be constrained by means of *assumptions*. Detailed information about the language, prover and the usage of the system can be found in [8, 9]. A distinctive feature of the typing system is the automatic generation of proof obligations, especially when instantiating the general scheme with a specific problem.

All considered steps are represented within a parameterized PVS theory which defines the required data structures and formalizes the application conditions by means of assumptions. Applying this step to a specific situation is carried out by importing the parameterized theory where the formal theory parameters are replaced by the specific problem parameters.

Another purpose of this paper is to investigate the verification process and to provide comprehensible, reusable proof methods. We show, for example, how to utilize the use of *measure-induction* in order to prove properties about recursive functions and to use the subtyping mechanism in order to encode correctness properties. Finally, we hope that this work serves as an interesting case-study and test for the usefulness of the PVS system.

The rest of this paper is organized as follows: the next section presents a formalization and verification of the schematic algorithm *divide-and-conquer* and an application of it to a binary-search problem. The operationalization of a descriptive specification is presented in Sect. 3 and is applied to the problem of finding a minimum element in a list.

Related Work

The formalization of transformations using higher-order functions has been considered by several researchers. In [4], for example, program transformations for recursion removal are expressed as second-order patterns defined in the simply typed λ-calculus. Independently from the work described herein, my colleague Harald Rueß has formalized, among other things, the *divide-and-conquer* paradigm in his dissertation [11] using the calculus of constructions and has given a verification with the LEGO proof checker [6]. Similar work dealing with the representation of existing approaches to program synthesis, development steps, and programming paradigms as well as a library of standard theories in the context of the NUPRL system has been carried out by Christoph Kreitz [5].

2 Divide-and-Conquer

The well-known algorithmic paradigm *divide-and-conquer* 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. Here, we consider the decomposition of the problem into two subproblems. A general decomposition scheme would be treated analogously. Following Smith's notation of a problem specification, we parameterize theories with a descriptive problem specification described as a 4-tuple (D, R, I, O) where D denotes the problem domain, R denotes the problem range, I is a predicate constraining D to meaningful inputs, and O describes the problem as an input-/output predicate. A solution of such a problem is a function computing feasible solutions, i.e. for an input x satisfying condition I it computes a y of type R such that condition $O(x, y)$ holds. We represent this principle as a parameterized PVS theory which has a problem description (D, R, I, O) and functions *decompose*, *compose*, *dir_solve*, a predicate *primitive?*, a map *lt* from domain D to natural numbers as its parameters. PVS only allows total functions, it must be ensured that all (recursive) functions terminate. For this purpose, a *measure-function* is used. Its domain matches that of the recursive function, and its range is `Nat` or `Ordinal`. The definition of a recursive function f generates a type correctness condition (TCC) which must be discharged in order to guarantee well-definedness of f. Here, the function *lt* serves as a measure-function. Four assumptions describe the meaning of the parameters. They state that

1. the subproblems created by the decomposition operator are smaller than the original problem when applied to the measure-function *lt*.
2. if the problem is primitive enough *dir_solve* creates a solution.
3. solutions to the subproblems z_1, z_2 can be composed to build a solution to the original problem.
4. all subproblems generated by the decomposition operator satisfy the input condition I of the problem (D, R, I, O).

```
div_and_conq[D : TYPE, R : TYPE, I : [D → boolean],
             O : [D, R → boolean], decompose : [D → [D, D]],
             dir_solve : [D → R], compose : [D, R, R → R],
             primitive? : [D → boolean], lt : [D → nat]] : THEORY
 BEGIN
 ASSUMING
   x : VAR D  z1, z2 : VAR R
   ax1 :
     ASSUMPTION
       (I(x) ∧ ¬ (primitive?(x)))
          ⊃
        ((lt(proj_1(decompose(x))) < lt(x))
             ∧ (lt(proj_2(decompose(x))) < lt(x)))

   ax2 : ASSUMPTION (I(x) ∧ primitive?(x)) ⊃ O(x, dir_solve(x))

   ax3 :
     ASSUMPTION
       (I(x)
           ∧ ¬ (primitive?(x))
             ∧ O(proj_1(decompose(x)), z1) ∧ O(proj_2(decompose(x)), z2))
          ⊃ O(x, compose(x, z1, z2))

   ax4 :
     ASSUMPTION
       (I(x) ∧ ¬ (primitive?(x)))
          ⊃ (I(proj_1(decompose(x))) ∧ I(proj_2(decompose(x))))

   ENDASSUMING

 USING measure_induction[D, nat, lt, (λ (x, y : nat) : x ≤ y)]
 f_dc(x : {y : D | I(y)}) : RECURSIVE R =
     IF primitive?(x) THEN dir_solve(x)
      ELSE LET
              x1 = proj_1(decompose(x)), x2 = proj_2(decompose(x)),
              rec1 = f_dc(x1), rec2 = f_dc(x2)
         IN compose(x, rec1, rec2)
      ENDIF
   MEASURE (λ (x : {y : D | I(y)}) : lt(x))

 correct : THEOREM (∀ (x : D) : I(x) ⊃ O(x, f_dc(x)))

 END div_and_conq
```

Fig. 1. Theory of *Divide-and-conquer*

The built in selector `proj_i(x)` selects the i-th element of a tuple. The recursive function f_dc realizes the schematic algorithm. Its domain is specified using the subtype-mechanism of PVS. It is given by the type D such that input condition I holds. Termination is established by the given measure function lt for which we must show that it decreases in size for the recursive arguments. This is given immediately by the first assumption. Figure 1 shows the (LaTeX pretty-printed) PVS theory. Type-checking this theory generates the following type correctness conditions (TCC's):

```
% Subtype TCC generated (line 35) for x1
  % proved - complete
f_dc_TCC2: OBLIGATION (FORALL (x: {y: D | I(y)}):
  NOT primitive?(x) IMPLIES I(PROJ_1(decompose(x))))

% Termination TCC generated (line 35) for f_dc
  % proved - complete
f_dc_TCC3: OBLIGATION (FORALL (x: {y: D | I(y)}):
  NOT primitive?(x) IMPLIES lt(PROJ_1(decompose(x))) < lt(x))

% Subtype TCC generated (line 36) for x2
  % proved - complete
f_dc_TCC4: OBLIGATION (FORALL (x: {y: D | I(y)}):
  NOT primitive?(x) IMPLIES I(PROJ_2(decompose(x))))

% Termination TCC generated (line 36) for f_dc
  % proved - complete
f_dc_TCC5: OBLIGATION (FORALL (x: {y: D | I(y)}):
  NOT primitive?(x) IMPLIES lt(PROJ_2(decompose(x))) < lt(x))
```

`f_dc_TCC2` and `f_dc_TCC4` are generated in order to ensure that the subproblem instances satisfy the condition I while both `f_dc_TCC3` and `f_dc_TCC5` ensure termination of f_dc. All obligations follow immediately from the assumptions $ax1$ and $ax4$. The correctness of this schematic algorithm is stated by theorem *correct*: f_dc exactly calculates a solution of the given problem (D, R, I, O).

2.1 Proof of *correct*

In order to prove properties about recursive functions, a measure-induction principle is required which is predefined in PVS:

measure_induction$[T, M:$ TYPE$, m: [T \to M], \leq: (\text{well_founded?}[M])]:$ THEORY
  BEGIN
  measure_induction :
    LEMMA
      $(\forall\ (p: \text{pred}[T]):$
        $(\forall\ (x: T): (\forall\ (y: T): m(y) \leq m(x) \wedge m(y) \neq m(x) \supset p(y)) \supset p(x))$
          $\supset (\forall\ (x: T): p(x)))$
  END measure_induction

Measure_induction builds on well-founded induction. It allows induction over a type T for which a measure function m is defined. Here, the theory is instantiated with $D, Nat, lt, \leq_{Nat}$.

In the following we give the main ideas of the PVS proof.[2] We start with the sequence:

```
  |-------
{1}   (FORALL (x: D): I(x) IMPLIES O(x, f_dc(x)))
```

We have written a strategy which instantiates the measure-induction principle and discharges the obligation that $\leq_{Nat}$ is well-founded. In the first step we apply this strategy, expand the definition of f_dc and obtain:

```
{-1}   (FORALL (y: D): lt(y) <= lt(x!1) AND lt(y) /= lt(x!1)
          IMPLIES I(y) IMPLIES O(y, f_dc(y)))
{-2}   I(x!1)
  |-------
{1}   O(x!1,
            IF primitive?(x!1) THEN dir_solve(x!1) ELSE
            compose(x!1, f_dc(proj_1(decompose(x!1))),
                         f_dc(proj_2(decompose(x!1))))
            ENDIF)
```

Case analysis on **`primitive?(x!1)`** yields two subgoals:

```
correct.1 :

{-1}   primitive?(x!1)
[-2]   (FORALL (y: D): lt(y) <= lt(x!1) AND lt(y) /= lt(x!1)
          IMPLIES I(y) IMPLIES O(y, f_dc(y)))
[-3]   I(x!1)
  |-------
{1}   O(x!1, dir_solve(x!1))
```

Applying assumption $ax2$ completes this branch.

For the other branch where **`primitive?(x!1)`** is false we apply $ax3$. This yields four subgoals:[3]

---

[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.

[3] We omit the subgoals `correct.2.2` and `correct.2.4` since they correspond to `correct.2.1` and `correct.2.3` respectively, just substitute `proj_2` for `proj_1` in formula {1}.

```
correct.2.1 :

[-1]   (FORALL (y: D): lt(y) <= lt(x!1) AND lt(y) /= lt(x!1)
          IMPLIES I(y) IMPLIES O(y, f_dc(y)))
[-2]   I(x!1)
  |-------
{1}   O(proj_1(decompose(x!1)), f_dc(proj_1(decompose(x!1))))
[2]   primitive?(x!1)
[3]   O(x!1, compose(x!1, f_dc(proj_1(decompose(x!1))),
                         f_dc(proj_2(decompose(x!1)))))

correct.2.3 (TCC):

[-1]   (FORALL (y: D): lt(y) <= lt(x!1) AND lt(y) /= lt(x!1)
          IMPLIES I(y) IMPLIES O(y, f_dc(y)))
[-2]   I(x!1)
  |-------
{1}   I(proj_2(decompose(x!1)))
[2]   primitive?(x!1)
[3]   O(x!1, compose(x!1, f_dc(proj_1(decompose(x!1))),
                         f_dc(proj_2(decompose(x!1)))))
```

Consider the first subgoal `correct.2.1`. Automatically instantiation of the term `proj_1(decompose(x!1))` for `y` and applying assumptions $ax1$ and $ax4$ completes the proof. The second subgoal is immediately proved by applying $ax4$. □

2.2 Example: Binary Search

We apply the schematic algorithm *divide-and-conquer* to solve the following problem: Given a function f on natural numbers, a key element and two bounds i_1, i_2 denoting the interval $\{n : Nat \mid i_1 \leq n \leq i_2\}$ the problem is to check if f applied to one of the elements of the interval is equal to the key element. This problem is described by D_1, R_1, I_1, O_1 where

- D_1 is the problem domain consisting of a function f, a key element key and bounds i_1, i_2.
- R_1 is the type of booleans.
- I_1 constrains domain D_1 such that i_1 is positive and $i_2 \geq i_1$.
- O_1 describes the input-/output condition informally given above.

Furthermore, applying the scheme of divide-and-conquer we have to explicitly give functions and predicates *primitive1?*, *dir_solve1*, *decompose1*, and *compose1* plus a measure-function *lt1*. The idea is to use a binary-search mechanism to solve the given problem. Therefore, we choose the following *divide-and-conquer*-theory instance:

- The problem is *primitive* if the given interval is trivial, i.e. $i_1 = i_2$.
- If the problem is primitive we can *directly solve* it by checking if $f(i_1) = key$.

$$
\begin{array}{l}
\text{bs} : \textsc{theory} \\
\quad \textsc{begin} \\
\quad \textsc{importing} \text{ div} \\
\quad D_1 : \textsc{type} \; = [f : [\text{nat} \rightarrow \text{nat}], \text{key} : \text{nat}, \; i_1 : \text{nat}, \; i_2 : \text{nat}] \\
\quad R_1 : \textsc{type} \; = \text{boolean} \\
\\
\quad I_1(x : \; D_1) : \text{ boolean } \; = \\
\quad\; \textsc{let} \; i_1 \; = \; \text{proj\_3}(x), \; i_2 \; = \; \text{proj\_4}(x) \; \textsc{in} \; (i_1 \; > \; 0) \; \wedge \; (i_2 \; \geq \; i_1) \\
\\
\quad O_1(x : \; D_1 \; , y : \; R_1) : \text{ boolean } \; = \\
\quad\; \textsc{let} \; f \; = \; \text{proj\_1}(x), \; \text{key} \; = \; \text{proj\_2}(x), \; i_1 \; = \; \text{proj\_3}(x), \; i_2 \; = \; \text{proj\_4}(x) \\
\quad\; \textsc{in} \; y \; = \; (\exists \, (z : \; \{n_1 : \; \text{nat} \; \mid \; i_1 \; \leq \; n_1 \; \wedge \; n_1 \; \leq \; i_2\}) : \; (f(z) \; = \; \text{key})) \\
\\
\quad \text{primitive1?}(x : \; D_1) : \text{ boolean } \; = \\
\quad\; \textsc{let} \; i_1 \; = \; \text{proj\_3}(x), \; i_2 \; = \; \text{proj\_4}(x) \; \textsc{in} \; (i_1 \; = \; i_2) \\
\\
\quad \text{dir\_solve1}(x : \; D_1) : R_1 \; = \\
\quad\; \textsc{let} \; f \; = \; \text{proj\_1}(x), \; \text{key} \; = \; \text{proj\_2}(x), i_1 \; = \; \text{proj\_3}(x) \; \textsc{in} \; (f(i_1) \; = \; \text{key}) \\
\\
\quad \text{decompose1}(x : \; D_1) : [D_1, D_1] = \\
\quad\; \textsc{let} \; f \; = \; \text{proj\_1}(x), \; \text{key} \; = \; \text{proj\_2}(x), \; i_1 \; = \; \text{proj\_3}(x), \; i_2 \; = \; \text{proj\_4}(x) \\
\quad\; \textsc{in} \; ((f, \; \text{key}, i_1, \; \text{div}(i_1 \; + \; i_2, 2)), (f, \; \text{key}, 1 \; + \; \text{div}(i_1 \; + \; i_2, 2), i_2)) \\
\\
\quad \text{compose1}(x : \; D_1, \; y_1 : \; R_1, \; y_2 : \; R_1) : \; R_1 \; = \; (y_1 \; \vee \; y_2) \\
\quad \text{lt1}(x : \; D_1) : \text{ nat } \; = \textsc{let} \; i_1 \; = \; \text{proj\_3}(x), \; i_2 \; = \; \text{proj\_4}(x) \; \textsc{in} \; \text{abs}(i_2 \; - \; i_1) \\
\\
\quad \textsc{importing} \text{ div\_and\_conq}[D_1, R_1, I_1, O_1, \\
\qquad\qquad\qquad\qquad \text{decompose1, dir\_solve1, compose1, primitive1?, lt1}] \\
\textsc{end} \text{ bs}
\end{array}
$$

Fig. 2. A binary-search problem

- *Decomposition* is done by splitting the range given by i_1, i_2 into two parts.
- The results of the search process in both subintervals are disjunctively *composed.*
- The required measure $lt1$ is defined by the size of the interval.

All the entities are combined in the PVS theory *bs*, see Fig. 2. The PVS theory *div* defining the div-function on natural numbers together with some properties is imported. We omit this theory since it is not of great significance. All proof obligations and lemmata of *div* have been successfully discharged.

Consider the *bs* theory, we have to show that it is indeed a correct instance of the general divide-and-conquer theory. Type-checking *bs*, PVS automatically generates the four required obligations where the first one is given as

```
IMPORTING1_TCC1: OBLIGATION
  (FORALL (x: D1): (I1(x) & NOT (primitive1?(x)))
                   IMPLIES ((lt1(PROJ_1(decompose1(x))) < lt1(x)) &
                            (lt1(PROJ_2(decompose1(x))) < lt1(x))))
```

The proofs of all TCC's are established by simply expanding the definitions and using some elementary properties of *div*. Finally, having discharged all TCC's we obtain a correct solution using the instantiated algorithm f_dc.

3 Operationalization of a Descriptive Specification

In this section we represent a transformation called *operationalization of a choice* given in [10]. We closely follow the method described in the previous section in representing this step. However, we give another possibility to establish the correctness of such a formalization. Here, instead of using an explicit correctness theorem we encode this information into the type of the recursive function utilizing the subtyping mechanism of PVS. The transformation works as follows: Starting from a given problem (D, R, I, O), the idea is to find a predicate B such that

1. $B(x) \Rightarrow O(x, H(x))$, and
2. $\neg B(x) \Rightarrow O(x, y) = O(K(x), y)$

In the first case, whenever $B(x)$ holds $H(x)$ is a feasible solution, in the second case y is a solution to input x if and only if y is a solution to $K(x)$, where K modifies x such that it decreases w.r.t the given measure-function. A solution of the problem, i.e. a function f for which $O(x, f(x))$ holds for all x satisfying I, is then immediately obtained by the recursive function fun given as:

$$fun(x) = \text{ IF } B(x) \text{ THEN } H(x) \text{ ELSE } fun(K(x))$$

Figure 3 shows the PVS theory. As noted above, the correctness is stated using the subtype mechanism of PVS. The range of function fun is of type R such that the input-/output condition O of the problem holds. Type-checking this theory automatically generates the required correctness conditions:

```
fun_TCC2: OBLIGATION
          (FORALL (x: {x1: D | I(x1)}): B(x) IMPLIES o(x, H(x)))

fun_TCC3: OBLIGATION
          (FORALL (x: {x1: D | I(x1)}): NOT B(x) IMPLIES I(K(x)))

fun_TCC4: OBLIGATION
          (FORALL (v: [x: {x1: D | I(x1)} -> {y: R | o(x, y)}]),
                  (x: {x1: D | I(x1)}):
                    NOT B(x) IMPLIES o(x, v(K(x))))
```

All but the third obligation are trivial and follow immediately from the assumptions. The only interesting obligation is the third one. The proof is established by adding type information of `v(K(x))` and applying assumptions $ax2$ and $ax3$.

```
op_of_choice_I[D : TYPE, R : TYPE, I : [D → boolean], O : [D, R → boolean],
               B : [D → boolean], H : [D → R], K : [D → D], lt : [D → nat]] :
 THEORY
  BEGIN
  ASSUMING
   x : VAR D  y : VAR R
   ax1 : ASSUMPTION (I(x) ∧ B(x)) ⊃ O(x, H(x))

   ax2 :
    ASSUMPTION (I(x) ∧ (¬ B(x))) ⊃ (O(x, y) = O(K(x), y))

   ax3 :
    ASSUMPTION
     (I(x) ∧ (¬ B(x))) ⊃ (I(K(x)) ∧ lt(K(x)) < lt(x))

   ENDASSUMING

  fun(x : {x1 : D | I(x1)}) : RECURSIVE {y : R | O(x, y)}  =
    IF B(x) THEN H(x) ELSE fun(K(x)) ENDIF
   MEASURE (λ (x : {x1 : D | I(x1)}) : lt(x))

  END op_of_choice_I
```

Fig. 3. Transformation op_of_choice_I

3.1 Example: Minimum Element

Suppose we are given the problem of finding a minimum element in a given (non-empty) list of natural numbers. We formalize this problem as the following 4-tuple (D_1, R_1, I_1, O_1):

- D_1, the problem domain, is defined as a tuple type $Nat \times List(Nat)$, where the first component denotes the temporary minimum.
- R_1 is the type of natural numbers.
- I_1 is the constant true function.
- $O_1(x, y)$ is true, if y is less than or equal to x's first parameter and every element of x's second parameter (denoting the list).

In order to correctly apply the above transformation we further have to supply specific values B_1, H_1, K_1, a measure function lt_1, and have to discharge all arising proof obligations.

- B_1 is true if and only if the list is empty.
- H_1 yields the first component of the tuple.
- K_1 yields the minimum of the head element and the temporary minimum element plus the tail of the list.
- lt_1 is defined as the length of the list.

minel : THEORY
  BEGIN
  IMPORTING list_prop

  D_1 : TYPE = [nat, list[nat]]
  R_1 : TYPE = nat
  $I_1 : [D_1 \rightarrow \text{boolean}] = (\lambda\ (x : D_1) : \text{TRUE})$

  $O_1(x : D_1, y : R_1) : \text{boolean} =$
   $((y = \text{proj\_1}(x) \lor \text{member?}(y, \text{proj\_2}(x)))$
   $\land\ (y \leq \text{proj\_1}(x)) \land (\forall\ (n : \text{nat}) : \text{member?}(n, \text{proj\_2}(x)) \supset y \leq n))$

  $B_1 : [D_1 \rightarrow \text{boolean}] = (\lambda\ (x : D_1) : \text{null?}(\text{proj\_2}(x)))$
  $H_1 : [D_1 \rightarrow R_1] = (\lambda\ (x : D_1) : \text{proj\_1}(x))$

  $K_1 : [D_1 \rightarrow D_1] =$
   $(\lambda\ (x : D_1) :$
    (IF null?(proj_2(x)) THEN x
    ELSE
     ((IF proj_1(x) $\leq$ car(proj_2(x)) THEN proj_1(x)
      ELSE car(proj_2(x))
      ENDIF),
     cdr(proj_2(x)))
    ENDIF))

  $lt_1 : [D_1 \rightarrow \text{nat}] = (\lambda\ (x : D_1) : \text{length}(\text{proj\_2}(x)))$

  IMPORTING op_of_choice_I[$D_1, R_1, I_1, O_1, B_1, H_1, K_1, lt_1$]

  minelf(s : $\{s_1 : \text{list[nat]} \mid \neg\ (\text{null?}(s_1))\}$) : RECURSIVE nat =
   LET x = (car(s), cdr(s)) IN fun(x)
  MEASURE
   (λ (s : $\{s_1 : \text{list[nat]} \mid \neg\ (\text{null?}(s_1))\}$) : LET x = (car(s), cdr(s)) IN $lt_1(x)$)

  END minel

Fig. 4. The minimum element problem

The function *minelf* then computes the minimum element of a given non-empty list of natural numbers using function *fun* which is called with the tuple consisting of the list's head and tail, see Fig. 4. When type-checking this theory the three required TCC's (the instantiated assumptions of *op_of_choice_I*) are generated where the second one is given as

```
(*)  IMPORTING1_TCC2: OBLIGATION
      (FORALL (x: D1), (y: R1):
       (I1(x) & (NOT B1(x))) IMPLIES (O1(x, y) = O1(K1(x), y))
```

The first and last obligation are easy to prove (simply rewrite all definitions) whereas the proof of the second obligation is lengthy and requires analyses of many cases. The complete proof script is given in the appendix. All proof obligations, lemmata and theorems have been successfully discharged, and the proof scripts are available by the author.

4 Concluding Remarks

In this paper we have demonstrated that it is possible to elegantly formalize and verify software development steps in a rigorous mathematical manner using the PVS system. We have considered steps of different complexity, both steps coding well-known algorithmic paradigms and "smaller" steps defined in the context of the CIP project. The use of higher-order logic with a rich type system greatly supported the formalization of very general transformation schemes. The method described in this paper can readily be used to represent other development steps of both kinds as we have demonstrated within the BMFT project KORSO (correct software). In another paper, for example, we have formalized the theory of *global-search* algorithms using a type-theoretic framework [1]. This framework in which all entities of the software development process can be formally represented and reasoned about has also been developed within this project. We refer to [15] for more information about the framework and the project.

Furthermore, we have shown how to synthesize a specific algorithm for some given problem simply by "filling the holes" of a general scheme. The concept of semantic subtypes has turned out to be an adequate tool to establish the correctness of the formalized steps since the type-check mechanism of PVS produces the required proof obligations automatically.

There are of course some aspects which can be improved in future versions of PVS. For example, when representing hierarchies of software development steps following the ideas of Smith [14] it is desirable to define theories which have other theories as their parameters. This is not possible in the current version. Furthermore, it is not possible to express properties about theories such as refinements between theories (theory morphisms). PVS does not allow types as parameters or results of functions. Therefore, functions like the polymorphic identity cannot be expressed directly. Finally, pattern matching could improve the readability of PVS specifications avoiding the use of projections.

Acknowledgement

I wish to thank F. W. von Henke, Harald Rueß, Martin Strecker, Detlef Schwier, and Ercüment Canver for many discussions and comments on draft versions of this paper. The constructive criticisms and suggestions provided by the anonymous referees have greatly improved the paper.

References

1. A. Dold. Formalisierung schematischer Algorithmen. Technical Report UIB-94-10, Fakultät für Informatik, Universität Ulm, January 1994.
2. The CIP Language Group. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L.* LNCS 183. Springer-Verlag, 1985.
3. The CIP System Group. *The Munich Project CIP - Volume II: The Program Transformation System CIP-S.* LNCS 292. Springer-Verlag, 1987.
4. G. Huet and B. Lang. Proving and Applying Program Transformations Expressed with Second-Order-Patterns. *Acta Informatica*, 11:31–55, 1978.
5. C. Kreitz. Metasynthesis - Deriving Programs that Develop Programs. Technical Report AIDA-93-03, Fachgebiet Intellektik, Technische Hochschule Darmstadt, 1993.
6. Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual.* University of Edinburgh, May 1992.
7. 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.
8. S. Owre, N. Shankar, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
9. S. Owre, N. Shankar, and J.M. Rushby. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
10. H.A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.
11. H. Rueß. *Metaprogrammierung in einer typtheoretischen Umgebung.* PhD thesis, Universität Ulm, Abt. KI, to appear in 1995.
12. Douglas R. Smith. Applications of a Strategy for Designing Divide-and-Conquer-Algorithms. *Science of Computer Programming*, (8):213–229, 1987.
13. Douglas R. Smith. Structure and Design of Global Search Algorithms. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto, CA, 1987.
14. Douglas R. Smith and Michael R. Lowry. Algorithm Theories and Design Tactics. *Science of Computer Programming*, (14):305–321, 1990.
15. F.W. von Henke, A. Dold, H. Rueß, D. Schwier, and M. Strecker. Construction and Deduction Methods for the Formal Development of Software. In M. Broy and S. Jähnichen, editors, *KORSO, Correct Software by Formal Methods.* Springer-Verlag, Lecture Notes in Computer Science, to appear in 1995, also available as Technical Report UIB-94-09, Fakultät für Informatik, Universität Ulm.

A Proof Script

We give the proof script of formula (*). The obligation can be proved using the prover command (TERM-TCC) which expands all relevant definitions, skolemizes by automatically generating skolem constants, tries to automatically instantiate the quantifiers, and does propositional simplification. The application of this command results in two subgoals. The boolean equality is then converted to an equivalence by (IFF). Applying (PROP) for propositional simplification to each of the subgoals yields a lot of new subgoals each of which is proved either by (TCC) or by (THEN* (INST? :SUBST ("n" "car(proj_2(x!1))")) (ASSERT)) instantiating the term car(proj_2(x!1)) for quantifier n and invoking the decision procedures. The generated proof script looks as:

```
("" (TERM-TCC)
    (("1" (IFF)
          (SPLIT)
          (("1" (FLATTEN)
                (SPLIT)
                (("1" (SPLIT)
                      (("1" (FLATTEN) (PROPAX)) ("2" (PROPAX))
                                                ("3" (TCC))))
                  ("2" (SPLIT)
                       (("1" (FLATTEN)
                             (INST? :SUBST
                                    ("n" "car(proj_2(x!1))"))
                             (ASSERT))
                         ("2" (PROPAX))
                         ("3" (TCC))))
                  ("3" (SPLIT)
                       (("1" (FLATTEN) (PROPAX)) ("2" (PROPAX))
                                                 ("3" (TCC))))))
            ("2" (FLATTEN)
                 (SPLIT)
                 (("1" (SPLIT)
                       (("1" (FLATTEN) (PROPAX)) ("2" (PROPAX))
                                                 ("3" (TCC))))
                   ("2" (SPLIT)
                        (("1" (FLATTEN) (PROPAX)) ("2" (PROPAX))
                                                  ("3" (TCC)))))))
      ("2" (IFF)
           (SPLIT)
           (("1" (FLATTEN)
                 (SPLIT)
                 (("1" (SPLIT)
                       (("1" (FLATTEN)
                             (INST? :SUBST
                                    ("n" "car(proj_2(x!1))"))
                             (ASSERT)
                             (PROPAX))
```

```
            ("2" (INST? :SUBST
                        ("n" "car(proj_2(x!1))"))
                 (ASSERT)
                 (PROPAX))
            ("3" (TCC))))
      ("2" (SPLIT)
           (("1" (FLATTEN) (PROPAX))
             ("2" (INST? :SUBST
                        ("n" "car(proj_2(x!1))"))
                  (ASSERT))
             ("3" (TCC))))
      ("3" (SPLIT)
           (("1" (FLATTEN) (PROPAX))
             ("2" (INST? :SUBST
                        ("n" "car(proj_2(x!1))"))
                  (ASSERT)
                  (PROPAX))
             ("3" (TCC))))))
("2" (FLATTEN)
     (SPLIT)
     (("1" (SPLIT)
           (("1" (FLATTEN) (PROPAX))
             ("2" (INST? :SUBST
                        ("n" "car(proj_2(x!1))"))
                  (ASSERT))
             ("3" (TCC))))
       ("2" (SPLIT)
            (("1" (FLATTEN) (PROPAX))
              ("2" (INST? :SUBST
                         ("n" "car(proj_2(x!1))"))
                   (ASSERT))
              ("3" (TCC))))))))))
```

This article was processed using the $\LaTeX$ macro package with LLNCS style