Program families: Program construction by context independent refinements

Eric W. van Ammers*

Computer Science Department, Wageningen Agricultural University, Dreijenplein 2, 6703 HB Wageningen, The Netherlands

Communicated by K. Apt; received 1 August 1994; revised 15 May 1997

Abstract

The concept of program families is a generalisation of the conventional stepwise refinement paradigm. We formalise program families by allowing Hoare-triplets to be parameterized. Next we derive a simple calculus to develop programs which are known a priori to be correct with respect to explicitly formulated pre- and postconditions.

Program families deal with at least two important problems of conventional refinement steps, i.e. program families are not context dependent and they apply just as well to top-down decomposition as to the bottom-up or middle-out approach. It turns out that the meaning of a pseudostatement in the context of program families is quite different from its meaning in the conventional refinement process.

A couple of examples illustrate the technique: the 1000 primes problem, a palindrome filter and a sorting routine.

The discussion relates program families to Morgan's refinement calculus, Knuth's literate programming and Soloway's programming plans. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Stepwise refinement; Decomposition; Family decomposition; Context independent refinements; Generalised Hoare-triplet; Parameterized Hoare-triplet

1. Introduction

In one of the oldest and best-known explications of structured programming, Dijkstra addresses several principles he considers essential to the construction of quality programs [8]. In this paper we will engage ourselves with two of those, that is the principle of stepwise refinement (Dijkstra speaks of step-wise program composition) and the principle of program families.

* Fax: 131 (0)317 483158; e-mail: Eric.vanAmmers@users.info.wau.nl.

1 Part of this research was performed while the author stayed with the Department of Mathematics and Computer Science at the Eindhoven University of Technology.
Due to the limited amount of information the human brain is capable to process at once, some form of decomposition is a prerequisite to understand and control any sufficiently complex system. In this respect the technique of stepwise refinement has firmly established its position in the programming community since it was promoted by Dijkstra, Wirth and many others from the early 1970s [1,8–10,19–21]. Stepwise refinement aims to develop a program and its (informal) correctness proof simultaneously by discrete steps. Originally, the technique is intuitive and not rooted in a formal model. Consequently, no hard rules exist to check, for instance, whether a certain refinement is "valid" in the sense that it does not destructively interfere with previously implemented decomposition steps. More recently, efforts have been made to describe stepwise refinement by means of a theory [5,15,16].

The principle of program families proclaims that we should not see a program as an isolated object, but rather as member of a family of programs that are related in certain way. To grasp the idea consider the following trivial example. Given an integer array \( a[1 : N] \), determine if at least one of the elements of \( a \) is even. The following program does the job:

\[
\begin{align*}
declare & k, l \text{ integer;} \\
& k \leftarrow 1; l \leftarrow N+1; \\
& \text{while } (k \neq 1) \\
& \quad \text{do} \\
& \quad \quad \text{if } (a \cdot k \mod 2) = 0 \\
& \quad \quad \quad \text{then } l \leftarrow k \\
& \quad \quad \quad \text{else } k \leftarrow k+1 \\
& \quad \quad \fi \\
& \quad \od \\
\end{align*}
\]

(1.1)

Afterwards \( k \) reflects the position of the first even element encountered and \( k = N+1 \) means that no element has been found. A formal proof of this program is not too difficult [13].

From a program family point of view we would derive program (1.1) as a particular instantiation of a more general iteration scheme

\[
\begin{align*}
declare & k, l \text{ integer;} \\
& k \leftarrow 1; l \leftarrow N+1; \\
& \text{while } (k \neq 1) \\
& \quad \text{do} \\
& \quad \quad \text{if } \text{property } (a \cdot k) \\
& \quad \quad \quad \text{then } l \leftarrow k \\
& \quad \quad \quad \text{else } k \leftarrow k+1 \\
& \quad \quad \fi \\
& \quad \od \\
\end{align*}
\]

(1.2)

This scheme defines a family of programs with the characteristic that afterwards \( k \) specifies the position of the first element of \( a[1 : N] \) which obeys a certain unknown property. Since this family characteristic of (1.2) is inherited by any off-spring, the
proof of (1.1) reduces to the trivial substitution

\[ \text{property } (a \cdot k) \leftarrow (a \cdot k \mod 2) = 0 \]  

With Dijkstra we believe there are good arguments to accept program families as a guiding principle in program development, especially in relation to program maintenance. Due to inevitable modifications, reality forces us to deal with several versions of a program, where newer versions are obtained from the older ones by some sort of text manipulation. To quote Dijkstra’s words [8, p.401]:

If a program has to exist in two different versions, I would rather not regard (the text of) the one program as a modification of (the text of) the other. It would be much more attractive if the two different programs could, in some sense or another, be viewed as, say different children from a common ancestor, where the ancestor represents a more or less abstract program, embodying what the two versions have in common. Hopefully, this common ancestor can be readily recognised in the (prae-) documentation. The intentions are

(1) that the two versions share their respective correctness proofs as far as possible;
(2) that the two versions share (mechanically) as far as possible the common (or “equal”) coding;
(3) that the two regions affected by the modification are already well-isolated, a condition which is not met when the transition requires “brain-made” modifications scattered all over the text.

Although the principle of program families is believed to be applied (at least unconsciously) by any competent programmer, it is hardly dealt with by the literature on structured programming. Wirth mentions the idea in one of his earlier papers, but does not work it out in concrete details [19]. Neither Dijkstra nor Wirth returns to the subject in later work [9–11, 20, 21]. Other authors seem to ignore the idea altogether (e.g. [1, 6]).

In this paper we formalise the concept of program families into a simple theory which we apply to the (de)composition of programs. The approach deals with some important problems of conventional stepwise refinement, that is program family refinements are context independent and not restricted to a top-down ordering. It induces a programming technique that is easily adopted in a practical programming environment. Moreover, the author believes that the theory describes the mental steps of an expert programmer more aptly than existing models of stepwise refinement do.

To avoid notational confusion we start with some basic terminology and use this to describe conventional stepwise refinement in terms of Hoare-triplets. By a suitable “parameterization” of the conventional Hoare-triplets we formalise the concept of program families and explain its basic properties. We illustrate the applicability of the theory by Dijkstra’s historical first thousand primes program [8] and two other
examples. Finally, we discuss its relation with existing research on programming
techniques.

2. Terminology and notations

We will write upper case italics to denote sets (domains), for example $X, Y, \ldots$. The
 corresponding lower case italics like $x, y, \ldots$ represent an element of the corresponding
set. Constants are symbolised by short strings in the same font as the running text.
As an exception we write the Boolean domain as $\text{Bool}$.

$$\text{Bool} = \{\text{true}, \text{false}\} \quad (2.1)$$

A predicate $p$ is a function which maps a given domain $D$ into $\text{Bool}$. The proposition
that $p$ is valid in every point of its domain is denoted as

$$[p] \quad (2.2)$$

Let $p$ and $q$ be predicates over $D$. We say that predicate $p$ is stronger than predicate
$q$ (or equivalently that $q$ is weaker than $p$) if

$$[p \Rightarrow q] \quad (2.3)$$

The relation “stronger” is therefore represented by the implication arrow “$\Rightarrow$”.

Consider a construct (mechanism, machine, program, etc.) $s$. The set of variables
that $s$ uses to exchange information with the environment is called the interface.
The interface variables define the domain $D$ of $s$. In this way we can understand $s$ as an
element in the set $S$ of relations from $D$ to $D$. Invoking the powerset operator $\mathbb{P}$ we
have

$$s \in S, \quad S = \mathbb{P}(D \times D) \quad (2.4)$$

An expression of the form

$$\{p\} s \{q\} \quad (2.5)$$

is called a Hoare-triplet. The predicates $p$ and $q$ are called the precondition and
postcondition, respectively. A Hoare-triplet expresses that given an arbitrary initial
state $d_i$ satisfying $p$ then $s$ will halt in final state $d_f$ satisfying $q$.

$$\forall d_i, d_f:\ p.d_i \land d_f = s.d_i: q.d_f \quad (2.6)$$

In general, we can think of many distinct Hoare-triplets for a given $s$, each high-
lighting a different property. The following is an example of a Hoare-triplet that is
valid for any $s$ (it is therefore semantically void)

$$\{\text{false}\} s \{\text{true}\}$$

In the practice of decomposition we often deal with partially completed programs.
This means that the program contains certain positions (pseudostatements, procedure
calls, etc.) representing components (program segments, procedure bodies, etc.) that are not yet available. We will refer to a partially completed program $ps$ as a *program scheme* and the positions to be satisfied as *slots*. To fit a component into a slot, its interface has to match the interface as prescribed by the slot. Observe that the interface of a slot may contain variables from the interface of $ps$ as well as local variables of $ps$. Assume the scheme $ps$ has slots $t_1, \ldots, t_N$. The interface of $t_i$ spans a domain $D_i$ and candidates to satisfy $t_i$ must be relations from the set

$$\mathbb{P}(D_i \times D_i)$$

(2.7)

In this notation $ps$ is an element of the set $T$ of functionals

$$T = (\mathbb{P}(D_1 \times D_1) \times \cdots \times \mathbb{P}(D_N \times D_N)) \cdot \mathbb{P}(D \times D)$$

(2.8)

Ignoring the interface variables proper we will write $ps$ as

$$ps(t_1, \ldots, t_N)$$

(2.9)

### 3. Conventional stepwise refinement

Assume we want to construct a program $s$ specified as

$$\{p\} s \{q\}$$

(3.1)

Unfortunately, we are seldom able to find the desired $s$ at once. The gap between $p$ and $q$ is simply too wide to bridge in one mental leap. Therefore, we conceive the problem as a set of smaller subproblems $s_i$, “glued” together by a program scheme $ps$ and we then try to solve the subproblems independently.

The specification of each of the subproblems $s_1, s_2, \ldots, s_N$ can be represented by a Hoare-triplet

$$\{p_i\} s_i \{q_i\} \quad \text{where} \ 1 \leq i \leq N$$

(3.2)

Let us take again formula (2.9) to represent the glue

$$ps(t_1, \ldots, t_N)$$

(3.3)

The refinement is correct if and only if the Hoare-triplet

$$\{p\} ps(s_1, \ldots, s_N) \{q\}$$

(3.4)

holds and this has to be proven explicitly. However, with stepwise refinement it is a good strategy to choose “small” refinement steps which tend to have very simple proofs. Of course, a subproblem $\{p_i\} s_i \{q_i\}$ can still be too large to be handled in one step, in which case the decomposition process is repeated recursively.

Note that the refinements $s_i$ are context-dependent; the $p_i$ and $q_i$ corresponding to $s_i$ will in general reflect properties that are specific to the original $p$ and $q$. 
4. Context-independent refinements

With the notion of *program families* we attack the decomposition problem from a different angle [8]. Rather than aiming immediately for the specified program, we start from a program scheme sufficiently general to include the wanted program as a special case. A refinement tailors the system better to its intended task by locking a component into a slot. The descendant is more specific then its ancestors, but retains the family qualities at the same time. Since the component may have slots of its own, the process can be applied recursively. The trick is, of course, to choose the components in such a way that we end up with the desired program.

4.1. Program families

Let us focus again on program scheme (2.9). Henceforth, it is convenient to write it as

\[ t(t_1, \ldots, t_N) \]  

(4.1)

Recall it is parameterized in \( N \) components \( t_i \) and the interface domain of the \( i \)-th parameter is \( D_i \). With every parameter \( t_i \) we associate predicates \( u_i \) and \( v_i \) over \( D_i \). In addition we associate with \( t \) itself a precondition \( u \) and a postcondition \( v \) over \( D \) which are parameterized in terms of \( u_i \) and \( v_i \), that is \( u \) and \( v \) are predicate schemes. Now we can construct a tripartite expression of the form

\[ \{ u(u_1, \ldots, u_N, v_1, \ldots, v_N) \} t(t_1, \ldots, t_N) \{ v(u_1, \ldots, u_N, v_1, \ldots, v_N) \} \]  

(4.2)

which accommodates \( 3N \) independent parameters \( u_1, \ldots, u_N, t_1, \ldots, t_N, v_1, \ldots, v_N \). One can imagine instantiations of \( u_i, t_i \) and \( v_i \) for which (4.2) does hold and in which case it represents a valid Hoare-triplet.

Observe that the predicates are not parameterized in \( t_i \) while program scheme is not parameterized in \( u_i \) and \( v_i \). In this way the relation between scheme and components is defined in terms of predicates only. The internal structure of the component, \( t_i \), is completely eliminated from our correctness concerns. This is a prerequisite for a proper modularization.

For the purpose of program families we consider (4.2) as a scheme of \( N \) composite parameters \( h_1, \ldots, h_N \). Each parameter \( h_i \) has three components \( u_i, t_i \) and \( v_i \) mutually related by a Hoare-triplet \( \{ u_i \} t_i \{ v_i \} \). In this way we can understand (4.2) as a predicate \( h \) in \( N \) variables, conventionally written as

\[ h(h_1, \ldots, h_N) \]  

(4.3a)

where

\[ h_i = \{ u_i \} t_i \{ v_i \} \quad i = 1, \ldots, N \]  

(4.3b)

Let \( H_i \) denote the set of values for which \( \{ u_i \} t_i \{ v_i \} \) actually constitutes a valid Hoare-triplet. We call (4.3) a *program family* if and only if

\[ \forall h_1 \in H_1 \ldots \forall h_N \in H_N [ h(h_1, \ldots, h_N) ] \]  

(4.4)
A parameter $h_i$ which only takes values from $H_i$ is called a $H$-parameter. Consequently, formula (4.4) states that a program family is a tautology in its $H$-parameters.

In terms of (4.2) we see that the formula represents a program family if and only if it is true for all combinations of $\{u_i\} t_1 \{v_i\}$ that are valid Hoare-triplets in their own respect.

A program family $h$ represents all possible choices that can be made for its $H$-parameters. More formally, we associate with program family $h$ the set

$$P\{h(h_1, \ldots, h_N) | h_1 \notin H_1 \ldots h_N \notin H_N\}$$

(4.5)

Every element of this powerset is called an instantiation of the family. Thus instantiation comes down to function application.

### 4.2. Some definitions and properties

#### Root property.

$$\{u\} t \{v\}$$

constitutes a program family with a single $H$-parameter, $\{u\} t \{v\}$

**Proof.** Let $s$ be an arbitrary component specified by the Hoare-triplet

$$\{p\} s \{q\}$$

This Hoare-triplet evidently qualifies as an instantiation of the $H$-parameter of (4.6). Substitution gives

$$\{p\} s \{q\}$$

(b)

which is valid due to assumption (a). □

Note that every program family can be obtained as a specialisation from (4.6). It can be considered as the root of all program families. Hence it is semantically void.

A conventional Hoare-triplet can be viewed as a parameterized Hoare-triplet with zero parameters. Since it is valid for every instantiation of its (non-existent) $H$-parameters we have the

#### Leaf property. Any conventional Hoare-triplet constitutes a program family.

Conventional Hoare-triplets are trivial instantiations of program families. All properties that apply to program families in general also apply to the conventional Hoare-triplets in particular.

Let $w(w_1, \ldots, w_N)$ be a predicate scheme where the parameters $w_i$ have to be selected from the sets $W_i$, respectively. Obviously, $w$ represents a set of predicates $W$ defined as

$$W = \{w(w_1, \ldots, w_N): w_1 \in W_1, \ldots, w_N \in W_N\}$$
An instantiation \( w' \) of \( w \) is obtained by making particular choices for the parameters of \( w \), that is by defining new \( W'_i \) which are subsets of the original \( W_i \). Clearly, the set of predicates \( W' \) associated with scheme \( w' \) is a subset of \( W \).

Note that the relation between a predicate scheme and the corresponding set of predicates is unambiguous. Dropping the notational difference we conclude the

**Instantiation property.** Let \( w \) and \( w' \) be predicate schemes and assume that \( w' \) is an instantiation of \( w \). We then have

\[
w' \subseteq w
\]  

(4.7)

**Composition theorem.** Let \( \{u\} t \{v\} \) and \( \{u'\} t' \{v'\} \) be program families and assume that \( t' \) qualifies as an instantiation of the \( \text{th} \) \( H \)-parameter of \( t \). The actual substitution of \( t' \) brings forth a new scheme \( \{u''\} t'' \{v''\} \) which again is a program family. In addition, the precondition and postcondition schemes of \( t'' \) are subsets with respect to those of \( t \), that is

\[
u'' \subseteq u \quad \text{and} \quad v'' \subseteq v
\]  

(4.8)

**Proof.** See Appendix A. □

**Definition.** Referring to the notation above we call the family \( t'' \) a *descendant* of \( t \). Conversely, we say that \( t \) is the *ancestor* of \( t'' \).

The composition theorem states that program families are *closed* under composition. In addition, it shows that the pre- and postcondition schemes of the successive families are monotonically decreasing sets of predicates. In other words, every instantiation step tailors the family more accurately to a specific task.

**Definition.** Let \( u \) and \( v \) be predicate schemes. We say that \( u \) is *stronger* than \( v \) (or equivalently that \( v \) is *weaker* than \( u \)) if every predicate in \( u \) is stronger than every predicate in \( v \), i.e.

\[
\forall u' \forall v' : \ u' \in u \land v' \in v : [u' \Rightarrow v']
\]  

(4.9)

Observe that the relation “stronger” as defined for predicates by (2.3) is simply a special case of (4.9). This justifies the symbol “\( \Rightarrow \)” to denote the stronger relation for predicate schemes as well.

Instantiation of a predicate means selection from a set of alternatives. This is not to be confused with strengthening. Thus, we have the following lemma to relate program families to program specifications.

**Lemma.** Assume we are looking for a program \( s \) specified by \( \{p\} s \{q\} \). If we can identify a program family \( \{u\} t \{v\} \) such that \( p \Rightarrow u \) and \( v \Rightarrow q \) then we know a priori that every instantiation of this family corresponds to a correct program.
Proof. Let \{u'\} t' \{v'\} be an instantiation of \{u\} t \{v\}. Then surely
\{u'\} t' \{v'\}
is a valid Hoare-triplet. Since \( u' \in u \) and \( p \Rightarrow u \) it follows that \( p \Rightarrow u' \). Likewise we have \( t' \Rightarrow q \). Therefore,
\{p\} t' \{q\}
is again a valid Hoare-triplet, which completes the proof. \( \square \)

4.3. Family decomposition

The relevance of program families is in the properties they establish a priori for every descendant. Whatever choice is going to be made for a \( H \)-parameter of \( t \), the off-spring necessarily inherits the semantics already established by the pre- and postcondition schemes \( u \) and \( v \) of the parent. This phenomenon can be exploited for decomposition purposes in which case we have family decomposition.

With family decomposition we approximate the desired program in steps through a series of program families in ancestor-descendant relationship. Every descendant is obtained by the substitution of a suitable family into a \( H \)-parameter of the already existing family. The composition theorem states that precondition scheme as well as postcondition scheme become successively "smaller" by instantiation, which justifies to write such a sequence of families as
\[
\{u_0\} t_0 \{v_0\} \supseteq \{u_1\} t_1 \{v_1\} \supseteq \{u_2\} t_2 \{v_2\} \supseteq \{u_3\} t_3 \{v_3\} \supseteq \cdots (4.10)
\]

The problem we face is to direct the selection (instantiation) process in such a way that after a certain number of steps, say \( N \), we have a program family which matches the desired specification. In other words the precondition scheme \( u_N \) should be at most as strong as the precondition \( p \) of the desired program while at the same time the postcondition scheme \( v_N \) should be at least as strong as the desired postcondition \( q \) (Fig. 1):
\[
p \Rightarrow u_N \quad \text{and} \quad v_N \Rightarrow q \quad (4.11)
\]

Precondition and postcondition schemes represent sets of predicates. Every step narrows both schemes. Note that the following conditions have to be met if the successive instantiations are to converge towards the desired program:
- The scheme used for instantiation must possess the family property. This is trivial in all situations where the instantiation is in terms of a conventional Hoare-triplet (leaf property).
- The newly constructed precondition scheme \( u' \) contains at least one predicate \( p' \) that is weaker than the precondition \( p \) of the desired program.
- The newly constructed postcondition scheme \( v' \) contains at least one predicate \( q' \) that is stronger than the postcondition \( q \) of the desired program.
The proof that the above conditions are actually met is referred to as the justification of the refinement step in question. Justification is a necessary condition to have monotonic convergence of the refinement process towards the desired goal.

Remark that in the realm of family decomposition, the meaning of a pseudostatement is quite different from its meaning in conventional stepwise refinement. With stepwise refinement a pseudostatement represents a specific well-defined action which could be specified by a conventional Hoare-triplet. But because the formulation of an explicit pre- and postcondition for this action is considered too cumbersome, we write the conditions sloppy and implicit through a natural language narrative. On the other hand, a pseudostatement of a program family represents every possible mapping that can be chosen for that particular slot without violating the family semantics.

5. Examples

The examples below are exclusively meant to illustrate the technique. Virtually, no arguments will be given with respect to the design steps we take successively. But before we can proceed, we need to agree on some additional notation.

In the process of decomposition it is often convenient to denote certain well-defined items by mnemonic identifiers (or sentences) rather than by a formal expression. At the same time we may want to use mnemonic strings to indicate parameters, pseudostatements and the like. Since this may raise ambiguities we use the following convention: "italics" will be used whenever a particular item is open for interpretation and "normal" (roman) font denotes an item whose interpretation is considered completely fixed. In addition, we identify actual code by the font "courier". For example,

\[ \text{square-first-value-into-second.x.y} \]

denotes a slot with an interface \( \{x, y\} \). Its meaning is to represent an arbitrary relation of the form \( (X \times Y) \times (X \times Y) \) and it is semantically equivalent to

\[ \text{add-second-value-to-first.x.y} \]
However, if we write

\[
\text{square-first-value-into-second.x.y}
\]

we mean a program fragment to implement the function

\[
y = x^2
\]

With this convention a Hoare-triplet like

\[
\{\text{precondition}\} \text{program}\{\text{postcondition}\}
\]

means that we are looking for an acceptable implementation to satisfy a fixed specification. On the other hand, the formula

\[
\{\text{precondition}\} \text{program}\{\text{postcondition}\}
\]

makes little sense because the strongest postcondition is already defined by the propagation of the precondition through the program. The only flexibility we have left here is to weaken the postcondition.

5.1. The first thousand primes

As a first example we will derive Dijkstra’s famous program to print the first thousand prime numbers [8]. Our aim is to design a program that with precondition

\[
\text{true} \quad \text{(5.1p)}
\]
satisfies postcondition

\[
\text{The first thousand primes have been printed} \quad \text{(5.1q)}
\]

We will treat this example in extensive detail and show the entire program after almost every refinement step.

(a) The most general program

From the root property we have the family

\[
\{u\} \quad \text{(5.1.1u)}
\]

\[
t \quad \text{(5.1.1t)}
\]

\[
\{v\} \quad \text{(5.1.1v)}
\]

Remark. No semantics are involved here. We only create an anchor point to start the refinement process.
(b) Producing some output

We first introduce a program family that will produce some output while no input-conditions are enforced. We postulate a global variable output that we can use to write the results of our calculation. Consider the scheme

\[
\{\text{true}\} \\
\text{begin} \\
\quad \text{Produce-something-on.output;} \\
\text{end} \\
\{\text{Something-has-been-produced-on.output}\}
\]

where its parameter is the Hoare-triplet

\[
\{\text{true}\} \\
\quad \text{Produce-something-on.output} \\
\{\text{Something-has-been-produced-on.output}\}
\]

Instantiation of (5.1.1) by (5.1.2a) means

\[
\begin{align*}
& u \leftarrow \text{true} \\
& t \leftarrow \text{begin} \\
& \quad \text{Produce-something-on.output;} \\
& \text{end} \\
& v \leftarrow \text{Something-has-been-produced-on.output}
\end{align*}
\]

This induces a new program family

\[
\begin{align*}
& \{\text{true}\} \\
& \text{begin} \\
& \quad \text{Produce-something-on.output;} \\
& \text{end} \\
& \{\text{Something-has-been-produced-on.output}\}
\end{align*}
\]

which is a better approximation of the desired program. In order to justify the refinement we must prove that

- (5.1.2a) actually constitutes a program family under the assumption (5.1.2utv). The proof of this claim is trivial.
- The interpreted precondition (5.1.2p) is not stronger than the desired precondition (5.1q). Since both are identical this condition is certainly met.
- The postcondition (5.1.2q) is a generalisation of (5.1q). There can hardly be a discussion on this point either.

Remark. Hoare-triplet (5.1.2psq) is obtained from (5.1.1utv) by an instantiation through scheme (5.1.2a). Moreover, the fact that (5.1.2psq) and (5.1.2a) are identical illustrates
once more that the root of all families is semantically void.

The Hoare-triplet (5.1.2utv) indicates that the choices for Produce-something-on.output and Something-has-been-produced-on.output have to be made in a mutually consistent manner. They are linked to one another and cannot be interpreted independently.

\[(c)\] Preparing the calculation

Let \( p \) be a table (array) of 1000 integers. We can first fill \( p \) with elements that satisfy a given \( \text{property} \) and afterwards write the table values to output. A convenient scheme to implement this approach is

\[
\{ \text{true} \}
\begin{align*}
\text{begin} & \\
& \text{declare } p \text{ array } [1 : 1000] \text{ of integer;} \\
& \text{Fill-table-with-1000-particular-values.} p.\text{property}; \quad (5.1.3a) \\
& \text{Print-1000-values-of-table.} \text{output}. p; \\
\text{end} & \\
& \{ \text{1000-Particular-values-printed.} \text{property.} \text{output} \}
\end{align*}
\]

The postcondition indicates that the printing of the 1000 particular values is considered sufficiently clear. There are two parameters involved in this scheme. It definitely constitutes a family if the parameters obey the Hoare-triplets

\[
\{ \text{true} \}
\begin{align*}
& \text{Fill-table-with-1000-particular-values.} p.\text{property}; \quad (5.1.3u1) \\
& \{1 \leq i \leq 1000 \Rightarrow \text{property.} p.\text{i} \} \quad (5.1.3v1)
\end{align*}
\]

and

\[
\{1 \leq i \leq 1000 \Rightarrow \text{property.} p.\text{i} \} \quad (5.1.3u2)
\begin{align*}
& \text{Print-1000-values-of-table.} \text{output}. p \quad (5.1.3t2) \\
& \{ \text{1000-Particular-values-printed.} \text{property.} \text{output} \} \quad (5.1.3v2)
\end{align*}
\]

We claim that (5.1.3a) qualifies as an instantiation of (5.1.2utv). Its precondition is identical to (5.1.2u) and its postcondition is a stronger version of (5.1.2v). By this instantiation we obtain a new family

\[
\{ \text{true} \}
\begin{align*}
\text{begin} & \\
& \text{begin} \\
& \text{declare } p \text{ array } [1 : 1000] \text{ of integer;} \\
& \text{Fill-table-with-1000-particular-values.} p.\text{property}; \quad (5.1.3s) \\
& \text{Print-1000-values-of-table.} \text{output}. p; \\
& \text{end}
\end{align*}
\]
and two Hoare-triplets, (5.1.3utv1) and (5.1.3utv2), that must be satisfied. This family is again a better approximation of the desired program since the postcondition is more specific and still includes (5.1q) as a special case.

**Remark.** Since an instantiation of *property* applies to all occurrences simultaneously, we must realise that the families (5.1.3utv1) and (5.1.3utv2) are linked by this parameter. If a certain form is chosen for *property* in (5.1.3utv1), then that same form must also be used in (5.1.3utv2).

**(d) Printing the output**

A straightforward printing of the successive elements p to output does the job. Consider the scheme

\[
\{1 \leq i \leq 1000 \Rightarrow property.p.i\}
\]

begin
  declare k integer;
  for k ← 1 to 1000
    do
      write (p[k], output)
    od
end
\{1000-Particular-values-printed.property.output\} \hspace{1cm} (5.1.4a)

Note that the code segment is independent of *property*. Without proof we claim the correctness of (5.1.4a) for all possible interpretations of *property*. Thus, it qualifies as a family, which in turns sanctions (5.1.4a) as a valid interpretation of (5.1.3utv2). This refines (5.1.3psq) to

\[
\{true\}
\]

begin
  begin
    declare p array [1:1000] of integer;
    Fill-table-with-1000-particular-values.p.property;
    begin
      declare k integer;
      for k ← 1 to 1000
        do
          write (p[k], output)
        od
    end
  end
end
\{1000-Particular-values-printed.property.output\} \hspace{1cm} (5.1.4p)

(5.1.4s)
Note that the scheme remains subjected to condition (5.1.3utv1).

(e) **Filling the table**

Consider the scheme

\[
\{\text{true}\}
\begin{align*}
&\text{begin} \\
&\quad \text{declare 1 integer;} \\
&\quad \text{for } 1 \leftarrow 1 \text{ to } 1000 \\
&\quad \text{do} \\
&\quad \quad \text{Calculate-new-value.property.p} \cdot 1 \\
&\quad \text{od} \\
&\text{end}
\end{align*}
\]

\[
\{1 \leq i \leq 1000 \Rightarrow \text{property.p} \cdot i\}
\]

We want to fill the table with successive elements satisfying \text{property.p} \cdot 1. In general, we can only know the \(i\)th element with a particular property if we have the previous \(i-1\) elements with that property at our disposal. For this reason the parameter of scheme (5.1.5a) is specified as

\[
\{1 \leq i \leq 1-1 \Rightarrow \text{property.p} \cdot i\} \quad \text{(5.1.5u)}
\]

\[
\text{Calculate-new-value.property.p} \cdot 1 \quad \text{(5.1.5t)}
\]

\[
\{1 \leq i \leq 1 \Rightarrow \text{property.p} \cdot i\} \quad \text{(5.1.5v)}
\]

Again we skip the proof that scheme (5.1.5a) obeys the family property under restriction (5.1.5utv). Evidently, (5.1.5a) qualifies as an interpretation of (5.1.3utv1) and we get

\[
\{\text{true}\} \quad \text{(5.1.5p)}
\]

\[
\begin{align*}
&\text{begin} \\
&\quad \text{begin} \\
&\quad \quad \text{declare p array [1:1000] of integer;} \\
&\quad \quad \text{begin} \\
&\quad \quad \quad \text{declare 1 integer;} \\
&\quad \quad \quad \text{for } 1 \leftarrow 1 \text{ to } 1000 \\
&\quad \quad \quad \text{do} \\
&\quad \quad \quad \quad \text{Calculate-new-value.property.p} \cdot 1 \\
&\quad \quad \quad \text{od} \\
&\quad \quad \text{end} \\
&\quad \text{end}
\end{align*}
\]

\[
\{1 \leq i \leq 1000 \Rightarrow \text{property.p} \cdot i\} \quad \text{(5.1.5s)}
\]
begin
    declare k integer;
    for k ← 1 to 1000
        do
            write (p[k], output)
        od
    end
end

{1000-Particular-values-printed.property.output} (5.1.5q)

(f) Towards primes

To make our program produce primes, we interpret property as the quality of being "prime" and "prime.p.1" indicates that the 1-th element of the table p is identical to the 1-th prime number. We now have

{true} (5.1.6p)

begin
    begin
        declare p array [1:1000] of integer;
        begin
            declare l integer;
            for l ← 1 to 1000
                do
                    Calculate-new-value.prime.p.1
                od
        end
    end
begin
    declare k integer;
    for k ← 1 to 1000
        do
            write (p[k], output)
        od
    end
end

{1000-Particular-values-printed.prime.output} (5.1.6q)

while the parameter specification of (5.1.5utv) reduces to

{1 ≤ i ≤ 1-1 ⇒ prime.p.i} (5.1.6u)

Calculate-new-value.prime.p.1 (5.1.6t)

{1 ≤ i ≤ 1 ⇒ prime.p.i} (5.1.6v)
Remark. We consider postcondition (5.1.6q) equivalent to the more informal (5.1q). It no longer contains items that qualify for interpretation. Every implementation of Calculate-new-value that obeys (5.1.6utv) will complete the family into an instantiation of the desired program.

(g) Calculating prime candidates

In order to interpret the concept Calculate-new-value, we propose

\[
\{1 \leq i \leq l-1 \Rightarrow \text{prime}.p.i\}
\]

begin

\begin{align*}
\text{declare candidate integer; } \\
\text{declare divisible Boolean; } \\
\text{if } l = 1 \text{ then } \\
\text{candidate } \leftarrow 2 \\
\text{else } \\
\text{divisible } \leftarrow \text{true; } \\
\text{candidate } \leftarrow p[l-1]; \\
\text{while divisible } \\
\text{do } \\
\text{candidate } \leftarrow \text{candidate } + 1; \\
\text{ls-divisible-by.candidate.p.(l-1).divisible; } \\
\text{od } \\
\text{fi } \\
p[l] \leftarrow \text{candidate; } \\
\text{end}
\end{align*}

\{1 \leq i \leq l \Rightarrow \text{prime}.p.i\}

which is a Hoare-triplet assuming the parameter obeys

\[
\{\text{true}\} \quad (5.1.7u)
\]

\[
\text{ls-divisible-by.candidate.p.(l-1).divisible; } \quad (5.1.7t)
\]

\[
\{\text{divisible } = (\exists i: 1 \leq i \leq l-1: (\text{candidate } \mod p[i]) = 0)\} \quad (5.1.7v)
\]

Although a bit tedious, it is not really difficult to prove the validity of (5.1.7a) from the assumption (5.1.7utv). Since (5.1.7a) is a conventional refinement of (5.1.6utv), this interpretation does change neither the precondition (5.1.6p) nor the postcondition (5.1.6q) of the program we have developed. Therefore we do not bother to spell out the result of this interpretation right now.

(h) Divisibility

The final step to complete the program is to implement the test for divisibility as specified by (5.1.7utv). This is again a simple conventional refinement. For
instance,
{true}
begin
declare m integer;
divisible ← false;
for m=1 to 1-1
do
if (candidate mod p[m])=0 then
divisible ← true
fi
od
end
{divisible = (∃i: 1 ≤ i ≤ 1−1: (candidate mod p[i]) = 0)}
is a Hoare-triplet which does the job.

(i) The completed program
By a straight forward substitution of the last two refinement steps in family (5.1.6psq) we get the complete program.

{true}
begin
begin
begin
declare p array [1:1000] of integer;
begin
declare l integer;
for l←1 to 1000
do
begin
declare candidate integer;
declare divisible Boolean;
if l=1 then
candidate ← 2
else
divisible ← true;
candidate ← p[l-1];
while divisible
do
candidate ← candidate+1;
begin
declare m integer;
divisible ← false;
for m=1 to 1-1
do
if (candidate mod p[m]) = 0 then divisible ← true fi od end od fi p[1] ← candidate; end od end begin declare k integer; for k ← 1 to 1000 do write (p[k], output) od end end

{1000-particular-values-printed.prime.output} (5.1.9q)

Remark. The example was meant to illustrate the technique and we did not bother too much about the efficiency of the program. Of course it is more efficient to restrict the testing process to values of \( p[\sqrt{\text{candidate}}] \) that are not greater than \( \sqrt{\text{candidate}} \) and to take only odd values into account as candidates.

### 5.2. A palindrome filter

A palindrome is a sentence with the property that reading the letters from left to right, gives the same result as reading them from right to left. In the comparison uppercase and lowercase letters are considered to be equivalent and all other characters are simply ignored. Hence an empty sentence is a palindrome. Other examples are:

Ada
1234567
Able was I, ere I saw Elba.
A man, a plan, a canal, Panama.
Norma is as selfless as I Am, Ron.

The program we are going for reads a given file named “input-txt” and writes the palindromic lines it encounters to a file named “output-txt”. A file consists of a
sequence of lines of text and the expression “textline.\textit{file}.k” indicates the \(k\)th line of \textit{file}. We can now define a palindrome filter as a program that given the precondition

\[
\text{true} \quad (5.2p)
\]
satisfies the postcondition

\[
\text{output-txt} = \{\text{line: } \exists k \ (\text{line} = \text{textline.input-txt.k}) \wedge \text{palindrome.line}\} \quad (5.2q)
\]

In the second example we will use a slightly different way to document refinement steps. The notation is more efficient, but less explicit. It is copied from Morgan with a minor modification [15]. The conventions are:

- Complete H-parameter descriptions are treated as code segments and placed at their appropriate position. We prefix a sequence number for easy reference.
- Parameter triplets are written in pre-order: body, precondition, postcondition.
- We will discuss the successive refinements without showing the intermediate programs every time. However, the evolution of precondition and postcondition of the over-all program will be shown after every refinement.

(a) Starting from the root

Since the root is a single H-parameter by itself we write it in the new convention as

\[
1. \ t\{u\}\{v\} \quad (5.2.1)
\]

(b) Reading input-txt and writing output-txt

The following program family is an adequate starting point for all programs that read a file “input-txt” and write a file “output-txt”.

\[
\{\text{true}\}
\]
begin
\begin{align*}
& \text{declare infile, outfile text;}
& \text{open (infile, input-txt); reset (infile);} \\
& \text{open (output, output-txt); rewrite (outfile);} \\
& 2. \ Process-file.infile.outfile; \\
& \quad \{\text{read-enabled.infile} \wedge \text{write-enabled.outfile}\} \\
& \quad \{\text{outfile=}File-processed.infile\} \\
& \text{close (infile); close (outfile);} \\
& \text{end}
\end{align*}
\[
\{\text{output-txt} = File-processed.input-txt\}
\]

The predicate “\text{read_enabled}” signifies that a file is readable and that its file pointer is right at the beginning. Likewise “\text{write_enabled}” means that an empty file is ready to be written.
Interpretation of (5.2.1) with (5.2.2) produces the following precondition and post-condition for the intermediate program:

\[ \{ \text{true} \} \]

\[ \{ \text{output-txt} = \text{File-processed}.\, \text{input-txt} \} \quad (5.2.2p) \]

\[ \{ \text{output-txt} = \text{File-processed}.\, \text{input-txt} \} \quad (5.2.2q) \]

Justification of the refinement:
- Within the semantics of a Pascal-like language it is fairly evident that scheme (5.2.2) is a program family.
- \((5.2p) \subseteq (5.2.2p)\)
- \((5.2q) \subseteq (5.2.2q)\)

(c) Line-based processing

The next step restricts our program to a line based processing of the files. A single input line is processed at a time and this may (but does not have to) result in a single output line. The internal representation of a line is a decision for later. We call upon the family

\[ \{ \text{read-enabled}.\, \text{infile} \land \text{write-enabled}.\, \text{outfile} \} \]

begin
\[ \text{declare inline, outline linetype}; \]
\[ \text{declare condition boolean}; \]
\[ \text{while not eof (infile)} \]
do
3. \text{Get-line}.\, \text{infile}.\, \text{inline}
\[ \{ \text{linenr}.\, \text{input}.\, (N) \}\]
\[ \{ \text{linenr}.\, \text{input}.\, (N+1) \land \text{inline} = \text{textline}.\, \text{input}.\, (N) \} \]
4. \text{Test} \text{inline}.\, \text{condition} ;
\[ \{ \text{true} \} \{ \text{condition} = \text{Tested}.\, \text{inline} \} \]
if condition then
5. \text{Process-line}.\, \text{inline}.\, \text{outline} ;
\[ \{ \text{true} \} \{ \text{inline} = \text{Line-processed}.\, \text{outline} \} \]
6. \text{Put-line}.\, \text{outline}.\, \text{outfile} ;
\[ \{ \text{linenr}.\, \text{output}.\, (N) \land \text{outline} = A \} \]
\[ \{ \text{linenr}.\, \text{output}.\, (N+1) \land \text{textline}.\, \text{outfile}.\, (N) = A \} \]
fi
od
end

\[ \{ \text{outfile} = \{ \text{line} : \exists k, ln (ln = \text{textline}.\, \text{infile}.\, k \}
\land \text{line} = \text{Line-processed}.\, \text{ln} \]
\land \text{Tested}.\, \text{ln} \} \} \]

With this scheme we have postulated a new primitive. Thus, the predicate "linenr. file.k" indicates that the pointer of file is positioned at the start of the kth line. Furthermore, we will accept additional overloading of the "="-operator in order to
compare the text-values within objects of different type (e.g. `textline.input.N` and `inline`).

Clearly, (5.2.3) is a legal interpretation of the parameter of (5.2.2). To apply it we must replace the term

\[
\text{File-processed . infile}
\]

by

\[
\{\text{line: } \exists k, ln \ (ln = \text{textline . infile . k} \\
\quad \wedge \ line = \text{Line-processed . ln} \wedge \text{Tested . ln})\}
\]

Consequently, the pre- and postcondition of the intermediate program transform into

\[
\{\text{true}\}
\]

\[
\{\text{output-txt} = \{\text{line: } \exists k, ln \ (ln = \text{textline . input-txt . k} \\
\quad \wedge \ line = \text{Line-processed . ln} \wedge \text{Tested . ln})\}\}
\]

Justification of the refinement:

- The hard part is to prove that (5.2.3) really is a program family with the four parameters 3–6. A crucial step is to establish a suitable loop invariant. In this case

\[
\text{linenr . infile . N} \wedge \\
\text{outfile} = \{\text{line: } \exists k, ln \ (ln = \text{textline . infile . k} \\
\quad \wedge \ line = \text{Line-processed . ln} \wedge \text{Tested . ln})\}
\]

will do. As before a proof in detail is omitted.

- \((5.2p) \subseteq (5.2.3p)\)

To see that \((5.2p) \subseteq (5.2.3q)\), one should realise that the test for palindromicity is a special case of \text{Tested} and that \text{Line-processed} can be taken to be the identity transformation.

(d) Choosing the structure of a line

Now it is time to become more specific about the internal representation of a line. That is we must choose an implementation of \text{linetype}.

Let us represent the characters of a line by an array \text{token} of 256 elements. An additional indicator \text{last} records which part of the array is being used. So we replace \text{linetype} by a \text{linetype} of the following structure

\[
\begin{align*}
\text{linetype} &= \text{record} \\
\text{token: array [1:256] of char;} \\
\text{last: integer;}
\end{align*}
\]

(e) The palindromicity test

To implement a test on palindromicity, we must interpret \text{H-parameter 4} of (5.2.3) in an appropriate way. The test itself can be performed as a two-step procedure:
1. Eliminate all non-letters characters and transform lower case to upper case.
2. Compare the remaining characters pair wise from outside inwards.
This is exactly the purpose of the code below:

{uuc}
begin
  declare i, j integer;
  declare letters linetype;

  {Fill letters with the upper case}
  {equivalent of the letters of line.}
  with inline
  do
    j ← 0;
    for i ← 1 to last
      do
        if is_uc_letter (token[i]) then
          j ← j + 1;
          letters.token[j] ← token[i]
        else if is_lc_letter (token[i]) then
          j ← j + 1;
          letters.token[j] ← to_uc(token[i])
        fi
      od
    letters.last + j
  od

  {Test subarray [1:last] of letters for}
  {palindromicity.}
  with letters
  do
    condition ← true;
    i ← 1;
    while (condition ∧ (i ≤ last div 2))
      do
        if chars[i] = chars[last - i + 1] then
          i ← i + 1
        else
          condition ← false
        fi
      od
  od
{condition = palindrome.inline}
Through this step the pre- and postcondition of the intermediate program transform to

\[ \{ \text{true} \} \]  
\[ \{ \text{output-txt} = \{ \text{line} : \exists k, ln \ (ln = \text{textline} \cdot \text{input-txt} \cdot k \} \} \]  
\[ \wedge \text{line} = \text{Line-processed} \cdot ln \]  
\[ \wedge \text{palindrome} \cdot ln) \} \]  

(5.2.5)

To justify (5.2.5) as a sensible interpretation note that

- (5.2.5) is a conventional Hoare-triplet. Consequently, its proof of the family property reduces to a correctness proof of its program segment with respect to its pre- and post condition. Once more we will skip this proof.

- (5.2p) \(\subseteq\) (5.2.5p)
- (5.2q) \(\subseteq\) (5.2.5q)

Remark. To prevent unnecessary programming detail it is convenient to postulate is_uc_letter, is_lc_letter and to uc as build-in functions of the programming language.

(f) Processing the line

In the case of a palindrome filter, we want to copy input lines that pass the palindromicity test. This reduces the transformation Process-line merely to a copy process. The following triplet is our candidate:

\[ \{ \text{true} \} \]  
begin 
 declare k integer 
 for k ← 1 to inline. last 
 do 
 outline.token[k] ← inline.token[k]; 
 do 
 outline.last ← inline.last; 
 end 
{inline = outline}

(5.2.6)

Replacing parameter 5 of (5.2.3) with (5.2.6) transforms pre- and postcondition of the intermediate program to

\[ \{ \text{true} \} \]  
\[ \{ \text{output-txt} = \{ \text{line} : \exists k, ln \ (ln = \text{textline} \cdot \text{input-txt} \cdot k \} \} \]  
\[ \wedge \text{line} = ln \wedge \text{palindrome} \cdot ln) \} \]  

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)

(5.2.6)
(g) A palindrome filter family

From the equivalence of (5.2.6q) and (5.2q) it follows that we have derived a program family of which every descendent will implement a valid palindrome filter. Putting all steps together we get

\[
\begin{align*}
\{ \text{true} \} \\
\text{begin } (1) \\
\text{declare infile, outfile text; } \\
\text{open (infile, input-txt); reset (infile); } \\
\text{open (output, output-txt); rewrite (output); } \\
\text{begin } (2) \\
\text{declare inline, outline linetype; } \\
\text{while not eof (infile) do } \\
\text{3. Get-line. infile . inline } \\
\{ \text{linenr.input.N} \} \\
\{ \text{linenr.input.(N+1) \land inline = textline.input.N} \} \\
\text{begin } (4) \\
\text{declare i, j integer; } \\
\text{declare letters linetype; } \\
\{ \text{Fill letters with the upper case } \} \\
\{ \text{equivalent of the letters of line.} \} \\
\text{with inline do } \\
\text{j }\leftarrow\text{ 0; } \\
\text{for i }\leftarrow\text{ 1 to last do } \\
\text{if is.uc.letter (token[i]) then } \\
\text{j }\leftarrow\text{ j+1; } \\
\text{letters.token[j] }\leftarrow\text{ token[i] } \\
\text{else if is.lc.letter (token[i]) then } \\
\text{j }\leftarrow\text{ j+1; } \\
\text{letters.token[j] }\leftarrow\text{ to.uc (token[i]) } \\
\text{fi } \\
\text{od } \\
\text{letters.last }\leftarrow\text{ j } \\
\text{od } \\
\{ \text{Test subarray [1: last] of letters for} \} \\
\{ \text{palindromicity.} \} \\
\text{with letters do } \\
\text{condition }\leftarrow\text{ true; }
\end{align*}
\]
i ← 1;
while (condition ∧ (i ≤ last div 2))
    do
        if chars[i] = chars[last - i + 1] then
            i ← i + 1
        else
            condition ← false
        fi
    od
od
end {4}
if condition then
    begin {5}
        declare k integer
        for k ← 1 to inline.last
            do
                outline.token[k] ← inline.token[k];
        od
        outline.last ← inline.last;
    end {5}
6. Put-line.outline.outfile;
   {linenr.output.N ∧ outline = A}
   {linenr.outfile.(N+1) ∧ textline.outfile.N = A}
fi
od
end {2}
close (infile); close (outfile);
end {1}
{output-txt = {line: \exists k(line = textline.input-txt.k) ∧ palindrome.line}]

The remaining H-parameters 3 and 6 represent conventional Hoare-triplets which can be elaborated in the conventional way. We will not bother about it this time.

Note that we have labelled begin–end pairs with the number of the parameter from which these code segments originated.

The justification of this refinement is as trivial as that of step (f).

5.3. Sorting

The concept of program families relates also to parameterization in terms of data types. Assume we want to sort a bag a of N elements of some type type. The elements of type are linearly ordered and the comparison operator is denoted by \(\subseteq\). The following program family does the job by a straight selection sort:
\{a = \{a_i \in \text{type}: 1 \leq i \leq N\} \land \text{multiset.}a = A\}

begin
  declare i, j, small integer;
  declare smaller Boolean;
  for i ← 1 to N-1
do
    small ← i;
    for j ← i+1 to N
do
      1. \text{Compare.type.}(a.\text{small}).(a.j).smaller;
         \{a.\text{small}, a.j \in \text{type}\}
         \{\text{smaller} = (a.j \subseteq a.\text{small})\}
      if smaller then
        small ← j
      fi
  od
  2. \text{Swap.type.}(a.i).(a.\text{small});
         \{X, Y, a.i, a.\text{small} \in \text{type} \land a.i = X \land a.\text{small} = Y\}
         \{a.i = Y \land a.\text{small} = X\}
  od
end
\{a = \{a_i \in \text{type}: 1 \leq i \leq N\} \land \text{multiset.}a = A\}
\land \forall i, j(1 \leq i < j \leq N \Rightarrow a_i \subseteq a_j)\}

Note that all code depending on the structure of the bag, \text{type}, is localised in the
two parameters. The family can be interpreted towards a file of random access records
as well as towards an array of integers. In fact the family property does not even
presuppose that the elements of a can be randomly accessed. But obviously this
is highly desirable if we are to implement the operations \text{Compare} and \text{Swap}
efficiently.

6. Discussion

A program family isolates one single design decision from the program under con-
struction and is parameterized with respect to all other aspects of the problem to be
solved. We compare this approach to various existing programming techniques and we
discuss some potential applications.

6.1. Comparison to conventional stepwise refinement

Several drawbacks have been observed with respect to the conventional technique of
stepwise refinement. It is context sensitive, i.e. precondition and postcondition
of the parent problem propagate into the specifications of the off-springs. In other
words, the specifications of the components is strongly coupled to the over-all specification of the program in question. Consequently conventional stepwise refinement implies top-down decomposition. Since programming virtually never is an exclusively top-down matter, it is unlikely that the conventional approach faithfully reflects the mental steps of a programmer.

With conventional stepwise refinement the specifications of the subsystems are obtained as a result of the refinement and therefore the process is strictly top-down. But families are simply composed in order to derive new ones with more specific properties. This process is essentially symmetric and it can be applied just as well to top-down or bottom-up as middle-out decomposition.

Program families and conventional refinement steps compare to one another a bit like open and closed subroutines. The latter are context dependent while the former can be practised in various environments. However, the concept of a family is less restricted by syntactical requirements. Thus, the family of Section 5.3 represents a generic sorting routine. It is applicable to many different argument types and would sort an array of characters just as well as the rows of a table.

6.2. Comparison to refinement calculus

In earlier work the conventional technique of stepwise refinement has been formalised into a refinement calculus [5, 15, 16]. Within this formalism a program consists of abstract as well as executable constructs. The goal of the refinement process is to gradually replace abstract constructs (specifications) by executable constructs (actual code) until only executable code is left. A valid refinement step may strengthen the postcondition of the over-all program or weaken its precondition.

With program families the situation is quite different. Here a refinement step limits the range of $H$-parameters which may induce a stronger postcondition as well as a stronger precondition. But the precondition is certainly never weakened during the refinement process. Thus, we start with a sufficiently general program family that does not meet the required specification and squeeze it stepwise into one that does.

After a certain number of steps it may happen that the over-all pre- and postcondition of a family have no parameters left. All remaining parameters are local to the program family (schemes (5.1.6psq) and (5.2.7) are examples). With refinement calculus it can happen likewise that pre- and postcondition of the overall program are established and only local specifications remain to be transformed into executable code. In a situation like this both techniques are equivalent. The correspondence becomes even more evident with the notation we have employed for the last two examples.

6.3. Software engineering and software reuse

The program family model provides a clean formalism for the composition of programs from a store of components. Imagine a collection of individual families, each
representing a verified solution to some partial problem. Since the families are context independent, they are applicable to many different situations. The programmer selects the right components and locks them together. The resulting program is a solution to a composite problem with a specification that is automatically derived from the individual family specifications. To actually establish the program as a solution to the problem at hand, only requires validation of the specification.

Program families establish a 'blood relation' between programs assuming they are developed from the same store of components. Every program is derived by a sequence of parameter substitutions. This sequence can be recorded as a substitution tree. The more akin trees of different programs are, the closer the programs are related. Closely related programs are relatively easy to transform into one another; an important consideration when a program has to be modified.

6.4. Formality and informality

Due to the context independence of program families, it is no problem to mix formally specified families with families that are specified in some informal way. Every family implements one isolated property that can be specified and verified in a way that best suits the purpose.

Consider the example given in Section 5.1. To describe the parameters Fill-table-with-1000-particular-values and Is-divisible-by we have used formal specifications. In contrast the specification of Print-1000-values-of-table is very informal. But the concepts are treated entirely disjunct and the formality of the first two is in no way corrupted by the informality of the third parameter.

Formal specifications are often extremely inadequate to describe non-mathematical properties. Therefore the liberty to mix formal and informal descriptions without mutual interference comes as a big advantage. It allows the integration of “hard” and “soft” specifications in one formalism. Printing the table in Section 5.1 is an example. Assume we want a “nice” lay-out of the table. This would be hard to specify formally. But the informal specification of parameter (5.1.3utv2) makes it possible to develop the rest of the program independently of this problem.

6.5. Refinements versus subprograms

It is sometimes argued that a pseudostatement – as resulting from a refinement process – is but a clumsy way to write a subprogram invocation. From this point of view stepwise refinement is equivalent to writing a program in terms of a collection of sufficiently small subprograms.

However there is a controversy. Devoted stepwise refiners feel that their technique adds something to the comprehensibility of a program that a collection of subprograms lacks. But they have problems to refute the subprogram-rather-than-pseudostatement argument. Most often their objection will be that the syntactical overhead associated with definition and invocation of a subprogram is significant (e.g. all the interface
variables have to be represented as parameters). This makes programmers reluctant to implement every design step as a separate subprogram and the resulting collection of subprograms will in general not faithfully reflect the actual design steps. But somehow this argument alone doesn't seem to be satisfactory.

The family approach may shed some light on this problem. Recall that in the context of conventional stepwise refinement a pseudostatement is but a sloppy way to write a specific transformation of its interface. This case is conveniently represented as a subprogram invocation with the interface variables as arguments. The elaboration of the refinement is simply a properly parameterized subprogram definition elsewhere. Consider the slot of \((5.1.7a)\). As a code segment it would be written as

\[
isdivisibleby (\text{candidate}, \text{p}, (l-1), \text{divisible});
\]

and the following procedure definition would be a correct refinement with respect to requirements \((5.1.7 uv)\)

\[
\begin{verbatim}
procedure isdivisibleby (cand: integer;
  table: arraytype;
  table_index: integer;
  var testresult: boolean);
begin
  \{The body of this procedure is equivalent to\}
  \{the code segment of \((5.1.8a)\).\}
end;
\end{verbatim}
\]

But a slot inside a program family is a different beast. It manifests the set of all transformations of its interface that respect the family property. To represent this aspect in terms of formal code is more elaborate as is immediately evident from the notation we have applied in our examples. Take the first slot of \((5.1.3a)\)

\[
\text{Fill-table-with-1000-particular-values}.\ p.\ property;
\]

As subprogram invocation the slot would read

\[
\text{filltablewith1000particularvalues} \ (\text{p}, \text{property});
\]

where the second argument itself represents a subprogram. Technically this means that the programming language must support subprograms as parameters. But more important, the programmer must clearly separate the part of the transformation that is effected by the refinement \((\text{filltablewith1000particularvalues})\) from the part that is left invariant \((\text{property})\). Exactly this aspect is completely ignored by the conventional refinement approach. If our claim is true that expert programmers (unconsciously) use families rather than conventional refinements, then the above controversy is conveniently explained.
6.6. Supporting tools

Decomposition by program families is essentially a matter of substitution. In the examples we have performed this cumbersome process by hand. It would be desirable if we could restrict ourselves to a mere definition of the successive steps and leave the actual substitution to an automatic device. Thus, after any number of steps the device would be able to show the family obtained so far together with the corresponding precondition and postcondition.

This concept comes fairly close to what Knuth has dubbed “literate programming” [14]. With literate programming one builds a documentation in terms of individual design steps and their corresponding code lines (i.e. any sort of design step, not necessarily formulated in terms of a program family). A dedicated tool extracts the files that have to be fed to the compiler. Literate programming tools perform substitution on the level of pseudostatements (program schemes) but currently have no facilities to deal with predicate schemes as well. Fortunately it is quite possible to work around this limitation. We can in fact refrain from the actual substitution of an interpretation and simply keep track of the successive interpretations we have made. In this way we obtain an interpretation scheme which in case of the postcondition of our first example would look like

\[
\begin{align*}
\{ t \} & \leftarrow \{ \text{Something-has-been-produced-on.output} \} \\
\text{Something-has-been-produced-on} & \leftarrow 1000-\text{Particular-values-printed.property} \\
\text{property} & \leftarrow \text{prime}
\end{align*}
\]

In addition, we know of at least two literate programming tools, VAMP and CLiP, which have been designed to handle data type parameterization of program schemes (cf. example in Section 5.3) [2–4].

Our experience supports the conclusion that the technique of program family decomposition is supported by existing literate programming tools to a feasible degree. The substitution process itself can be effectively automated. Justification of a refinement step remains the responsibility of the programmer.

6.7. Programming plans

There is accumulating psychological evidence that expert programmers compose their programs from a large set of “generic” solutions which they adapt to the particular problem statement at hand [17, 18]. These canned solutions, called plans, are not language dependent. They form the chunks (i.e. mentally meaningful units) in the programming domain.
We claim that the concept of program families is related to this kind of plans. That is, a program family formalises the generic properties of an intuitively invoked plan within the context of a programming language. Thus, we can understand a program family as the language specific instantiation of a plan.

To see the point let us take Soloway's discussion of the averaging problem: Write a program to read a list of integers, ended by the sentinel value 99999, and print its average [17]. Soloway observes that the expert programmer invokes two plans called SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN and SENTINEL-CONTROLLED COUNTER-LOOP PLAN – and merges them to a single loop calculating the number of integers together with their sum. In terms of program families we write the SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN as

\[
\{\text{last}.x=\text{sentinel} \land \text{sentinel} \notin \text{front}.x \land \text{front}.x=A\}
\]

\[
\begin{align*}
\text{begin} & \\
& \text{declare total, input integer;} \\
& \text{total} \leftarrow 0; \\
& \text{strip-head}.x.\text{input;} \\
& \text{while (input} \neq \text{sentinel}) \\
& \quad \text{do} \\
& \quad \quad \text{total} \leftarrow \text{total} + \text{input;} \\
& \quad \quad \text{strip-head}.x.\text{input;} \\
& \quad \text{od} \\
\text{end} \\
\{\text{total} = \sum A\}
\end{align*}
\]

where the reading process is abstracted to

\[
\{\text{head}.x=H \land \text{tail}.x=T\}
\]

\[
\text{strip-head}.x.\text{input}
\]

\[
\{\text{input}=H \land x=T\}
\]

THE SENTINEL-CONTROLLED COUNTER LOOP PLAN can be written as

\[
\{\text{last}.x=\text{sentinel} \land \text{sentinel} \notin \text{front}.x \land \text{front}.x=A\}
\]

\[
\begin{align*}
\text{begin} & \\
& \text{declare counter, input integer;} \\
& \text{counter} \leftarrow 0; \\
& \text{strip-head}.x.\text{input;} \\
& \text{while (input} \neq \text{sentinel}) \\
& \quad \text{do} \\
& \quad \quad \text{counter} \leftarrow \text{counter} + 1; \\
& \quad \quad \text{strip-head}.x.\text{input;} \\
& \quad \text{od} \\
\text{end} \\
\{\text{counter}=#A\}
\end{align*}
\]

with its parameter subjected to (6.1utv).
The idea to fuse both plans into a single master loop emerges because each of the plans can be accounted for as a particular interpretation of an even more abstract SENTINEL-CONTROLLED GENERIC LOOP PLAN which applies an arbitrary operator, \( \text{opr} \), to a list of elements:

\[
\{ \text{last}.x = \text{sentinel} \land \text{sentinel} \notin \text{front}.x \land \text{front}.x = A \}
\]

\[
\begin{align*}
\text{begin} \\
\text{declare generic, input integer;} \\
\text{generic } &\leftarrow \text{unity} \\
\text{strip-head}.x.\text{input} \\
\text{while (input } \neq \text{sentinel})
\text{do} \\
\text{opr}.\text{generic}.\text{input} \\
\text{strip-head}.x.\text{input} \\
\text{od}
\text{end}
\{\text{generic } = \text{opr}^*.A\}
\]

Again \( \text{strip-head} \) is subjected to (6.1utv). But in addition we have a parameterized \( \text{opr} \) which can be freely interpreted but for the restriction

\[
\text{opr}.z.\text{unity} = z, \text{ for all } z
\]

The \( ^* \)-notation denotes the repeated application of an operator over the successive elements of a list. More formally

\[
\begin{align*}
\text{opr}^* &[]= \text{unity} \\
\text{opr}^* &.(a; x) = \text{opr}.a.(\text{opr}^*.x)
\end{align*}
\]

Taking conventional addition for \( \text{opr} \) we get (6.1a). In order to derive (6.2a) we must interpret \( \text{opr} \) as a counting operation, count, defined as

\[
\text{count}.z.y = z+1, \text{ for all } y
\]

**Acknowledgements**

The origin of this work has been a three week long discussion of the author with Ralph-Johan Back during the Turku Summerschool in 1977. Our attempts to understand the very idea of stepwise refinement revealed a difference of approach we could not make explicit at the time. Since then several people have contributed to the ideas above: notably Maurice Elzas, Jan Smeenk, Mirjam Gerritsen, Jacek Leszczyłowski and Henk Barendregt. Hans van Vliet underlined the significance of context independency of refinement steps. The comments of the Eindhoven Algorithm Club have contributed significantly to the ideas in this paper. Mark Kramer spent such an
amount of time and effort to refine the document that he is virtually a co-author. The thought-out remarks of two anonymous referees have considerably shaped the final manuscript.

**Appendix A. Proof of the Composition theorem**

Let \( \{u'\} t' \{v'\} \) be used as a component of \( \{u\} t \{v\} \) producing \( \{u''\} t'' \{v''\} \). We have to show that

\[ \{u''\} t'' \{v''\} \]

again is a program family. In addition we must prove the formula

\[ u'' \subseteq u \text{ and } v'' \subseteq v \]

**Proof.** To keep the notation simple, we will prove the property for a program family \( \{u\} t \{v\} \) with two \( H \)-parameters and a program family \( \{u'\} t' \{v'\} \) with only one parameter. With the usual definition of \( H \)-parameters we have that

\[ \{u(u_1, u_2, v_1, v_2)\} t(t_1, t_2) \{v(u_1, u_2, v_1, v_2)\} \quad (A.1) \]

and

\[ \{u'(u'_1, v'_1)\} t'(t'_1) \{v'(u'_1, v'_1)\} \quad (A.2) \]

are both tautologies. Instantiation of the first \( H \)-parameter of \( t \) by \( t' \) gives

\[ \{u(u'(u'_1, v'_1), u_2, v(u'_1, v'_1), v_2)\} \]
\[ t'(t'_1, t_2) \]
\[ \{v(u'(u'_1, v'_1), u_2, v'(u'_1, v'_1), v_2)\} \quad (A.3) \]

By transitivity it follows that (A.3) is valid for all \( H \)-parameters which obey

\[ \{u'_1\} t'_1 \{v'_1\} \text{ and } \{u_2\} t_2 \{v_2\} \quad (A.4) \]

Note that the parameters of the precondition scheme as well as those of the postcondition scheme are \( u'_1, v'_1, u_2 \) and \( v_2 \). This justifies rewriting of these schemes, respectively, as

\[ \{u''(u'_1, v'_1, u_2, v_2)\} \quad (A.5) \]

and

\[ \{v''(u'_1, v'_1, u_2, v_2)\} \quad (A.6) \]

Likewise we can rewrite the program scheme as

\[ t''(t'_1, t_2) \quad (A.7) \]
which transforms (A.3) to
\[ \{u''(u_1', v_1', u_2, v_2)\\} \rightarrow (t_1', t_2) \{v''(u_1', v_1', u_2, v_2)\} \]  \hspace{1cm} (A.8)

The following lexical substitutions
\[ (u_1', t_1', v_1') \rightarrow (u_1'', t_1'', v_1'') \quad \text{and} \quad (u_2, t_2, v_2) \rightarrow (u_2'', t_2'', v_2'') \]  \hspace{1cm} (A.9)
transforms (A.8) to
\[ \{u''(u_1'', u_2'', v_1'', v_2'')\\} \rightarrow (t_1'', t_2') \{v''(u_1'', u_2'', v_1'', v_2'')\} \]  \hspace{1cm} (A.10)

From (A.4) and (A.9) it follows that (A.10) is valid for all parameters which satisfy
\[ \{u''\} \leftrightarrow (v') \quad \text{and} \quad \{u_2''\} \leftrightarrow (v'') \]  \hspace{1cm} (A.11)

Finally, we have to prove that \( u'' \) and \( v'' \) are stronger than \( u \) and \( v \), respectively. But this is fairly evident. For instance, \( u'' \) is derived from \( u \) by the instantiation
\[ u_1 \rightarrow u'(u_1', v_1') \quad \text{and} \quad v_1 \rightarrow v'(u_1', v_1') \]  \hspace{1cm} (A.12)
and the same instantiation derives \( v'' \) from \( v \). \( \Box \)

References

[8] E.W. Dijkstra, Notes on structured programming, in: [7].


