Programming Without Refinement

Marwa Benabdelali[1], Lamia Labed Jilani[1], Wided Ghardallou[2], and Ali Mili[3]

[1] ISG Management Institute, Bardo, Tunisia
[2] University of Kairouan, Kairouan, Tunisia
[3] New Jersey Institute of Technology, Newark, NJ, USA
marwa.benabdelali@yahoo.com,lamia.labed@isg.rnu.tn,
wided.ghardallou@gmail.com, mili@njit.edu

Abstract. To derive a program for a given specification R means to find an artifact P that satisfies two conditions: P is executable in some programming language; and P is correct with respect to R. Refinement-based program derivation achieves this goal in a stepwise manner by enhancing executability while preserving correctness until we achieve complete executability. In this paper, we argue that it is possible to invert these properties, and to derive a program by enhancing correctness while preserving executability (proceeding from one executable program to another) until we achieve absolute correctness. Of course, this latter process is possible only if we know how to enhance correctness.

Keywords

Absolute correctness, relative correctness, program refinement, correctness preservation, correctness enhancement, program derivation, program projection, deriving reliable programs.

1 Introduction: Correctness Preservation vs Correctness Enhancement

To derive a program from a specification R means to find an artifact P that satisfies two conditions: P is executable in some target programming language; and P is correct with respect to R. Refinement-based program derivation achieves this goal in a stepwise manner by enhancing executability (substituting the specification notation by programming notation) while maintaining correctness until we achieve complete executability. In this paper we consider an orthogonal approach, where these two properties are inverted: We enhance correctness with respect to R while maintaining executability (all intermediate artifacts are executable programs) until we achieve absolute correctness. Figure 1 illustrates how these two iterative processes differ.

Program derivation by correctness enhancement was introduced in [7]. In this paper we build on the discussions of [7] by: Considering more sample examples of program derivation by correctness enhancement (this is the subject of section 3); in light of our experience with these sample examples, sketching the first

Paradigm	Refinement	Correctness Enhancement
Initial Condition	$P = R$	$P =$ `abort`
Invariant Assertion	P is correct	P is executable
Variant Function	Enhance Executability	Enhance Correctness
Exit Condition	P is Executable	P is Correct

Fig. 1. Orthogonal Derivation Processes

outlines of a methodology of correctness enhancement; considering the concept of *projection*, and its impact on the discipline of program derivation by correctness enhancement; using empirical evidence to analyze the evolution of program reliability through the correctness enhancement process.

In section 2 we introduce some elements of relational mathematics, then we discuss the basic concepts that we need in this paper. In section 3 we present a number of sample program derivations by correctness enhancement, and in section 4 we analyze the reliability growth of the programs generated in each example, using a simple experimental set-up. In section 5 we take stock of the experience gained through the examples of section 3 to sketch the outlines of a programming methodology that is adapted to correctness enhancement, and discuss the contrast between correctness enhancement and related properties. Finally, in section 6 we summarize and assess our findings, and sketch directions of future research.

2 Mathematics for Correctness

2.1 Relational Mathematics

We assume the reader familiar with elementary relational mathematics [4], and will merely present some definitions and notations. We represent sets in a program-like notation by writing variable names and associated data types; if we write S as: {`x: X; y: Y;`}, then we mean to let S be the cartesian product $S = X \times Y$; elements of S are denoted by s and the X- (resp. Y-) component of s is denoted by $x(s)$ (resp. $y(s)$). When no ambiguity arises, we may write x for $x(s)$, and x' for $x(s')$, etc. A *relation* R on set S is a subset of $S \times S$. Special relations on S include the *universal relation* $L = S \times S$, the identity relation $I = \{(s, s) | s \in S\}$ and the empty relation $\phi = \{\}$. Operations on relations include the set theoretic operations of union, intersection, difference and complement; they also include the *converse* of a relation R defined by $\widehat{R} = \{(s, s') | (s', s) \in R\}$, the *domain* of a relation defined by $dom(R) = \{s | \exists s' : (s, s') \in R\}$, and the product of two relations R and R' defined by: $R \circ R' = \{(s, s') | \exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$; when no ambiguity arises, we may write RR' for $R \circ R'$.

A relation R is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R = \widehat{R}$, antisymmetric if and only if $R \cap \widehat{R} \subseteq I$, asymmetric if and only if $R \cap \widehat{R} = \phi$ and transitive if and only if $RR \subseteq R$. A relation R is said to

be *total* if and only if $I \subseteq R\widehat{R}$ and *deterministic* if and only if $\widehat{R}R \subseteq I$ (we then say that R is a *function*). A relation R is said to be a *vector* if and only if $RL = R$; vectors have the form $R = A \times S$ for some subset A of S; we use them as relational representations of sets. In particular, note that RL can be written as $dom(R) \times S$; we use it as a representation of the domain of R.

2.2 Program Semantics

Given a program p on space S, we define the *function* of p (denoted by P) as the set of pairs (s, s') such that if program p starts execution in state s it terminates in state s'; when no ambiguity arises, we may refer to a program and its function by the same name, P.

Definition 2.1. *Given two relations R and R', we say that R'* refines *R (abbrev: $R' \sqsupseteq R$ or $R \sqsubseteq R'$) if and only if $RL \subseteq R'L \wedge RL \cap R' \subseteq R$.*

This is the relational form of the usual interpretation of refinement as having a weaker precondition and a stronger postcondition.

Definition 2.2. *A program* p *on space S is said to be* correct *with respect to specification R on S if and only if its function P refines R.*

This definition is identical (modulo differences of notation) to traditional definitions of total correctness [2, 10, 12, 14].

2.3 Relative Correctness

Definition 2.3. *Due to [16]. Given a specification R and two deterministic programs P and P', we say that P' is* more-correct *(resp.* strictly more-correct*) than P with respect to R, denoted as $P' \sqsupseteq_R P$ (resp. $P' \sqsupset_R P$) if and only if $(R \cap P')L \supseteq (R \cap P)L$ (resp. $(R \cap P')L \supset (R \cap P)L$).*

To contrast relative correctness with correctness, we may refer to the latter as *absolute correctness*. We refer to $(R \cap P)L$ (or $dom(R)$) as the *competence domain* of P with respect to R. See Figure 2 for an illustration of relative correctness. We have: $(R \cap P) = \{(1,2),(2,3)\}$, hence $(R \cap P)L = \{1,2\} \times S$. On the other hand, $(R \cap P') = \{(1,0),(2,1),(3,2)\}$, hence $(R \cap P')L = \{1,2,3\} \times S$.

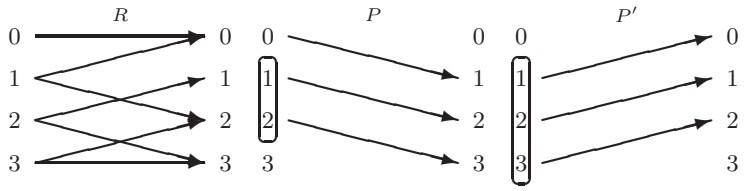

Fig. 2. $P' \sqsupseteq_R P$, Deterministic Programs

How do we know that our definition is any good? In [16], we find that relative correctness satisfies the following properties:

- *Ordering Properties*. Relative correctness is reflexive and transitive, but not antisymmetric. Two programs P and P' may be equally correct yet distinct.
- *Relative Correctness and Absolute Correctness*. A (absolutely) correct program is more-correct than (or as correct as) any candidate program. A deterministic program P is (absolutely) correct with respect to R if and only if its competence domain is $dom(R)$.
- *Relative Correctness and Reliability*. The reliability of a program P on space S is defined with respect to a specification R on S and a discrete probability distribution $\theta()$ on $dom(R)$. We measure it by the probability that the execution of P on a random state s of $dom(R)$ selected according to $\theta()$ terminates successfully in a state s' such that $(s, s') \in R$; in other words, it is the probability that a randomly selected element of $dom(R)$ following the probability distribution $\theta()$ falls within the competence domain of P with respect to R. From this definition, and from definition 2.3, we infer that if P' is more-correct than P, then P' is more reliable than P for any probability distribution $\theta()$.
- *Relative Correctness and Refinement*. Program P' refines program P if and only if P' is more-correct than P with respect to *any* specification R.

For illustration, we present below a specification and ten programs, ranked by relative correctness, as shown in Figure 3; correct programs are shown at the top of the graph. We let R be the specification defined on space $S = nat$ by: $R = \{(s, s') | s^2 \leq s' \leq s^3\}$, and we consider the following programs, where with each program we indicate its function, and its competence domain:

`p0`: {`abort`}. $P_0 = \phi$. $CD_0 = \emptyset$.
`p1`: {`s=0;`}. $P_1 = \{(s, s') | s' = 0\}$. $CD_1 = \{0\}$.
`p2`: {`s=1;`}. $P_2 = \{(s, s') | s' = 1\}$. $CD_2 = \{1\}$.
`p3`: {`s=2*s**3-8;`}. $P_3 = \{(s, s') | s' = 2s^3 - 8\}$. $CD_3 = \{2\}$.
`p4`: {`skip;`}. $P_4 = I$. $CD_4 = \{0, 1\}$.
`p5`: {`s=2*s**3-3*s**2+2;`}. $P_5 = \{(s, s') | s' = 2s^3 - 3s^2 + 2\}$. $CD_5 = \{1, 2\}$.
`p6`: {`s=s**4-5*s;`}. $P_6 = \{(s, s') | s' = s^4 - 5s\}$. $CD_6 = \{0, 2\}$.
`p7`: {`s=s**2;`}. $P_7 = \{(s, s') | s' = s^2\}$. $CD_7 = S$.
`p8`: {`s=s**3;`}. $P_8 = \{(s, s') | s' = s^3\}$. $CD_8 = S$.
`p9`: {`s=(s**2+s**3)/2;`}. $P_9 = \{(s, s') | s' = \frac{s^2+s^3}{2}\}$. $CD_9 = S$.

The following definition applies to non-deterministic programs.

Definition 2.4. *Due to [5]. Given a specification R and two programs P and P'. We say that P' is* more-correct *than P with respect to R if and only if:*

$$(R \cap P')L \supseteq (R \cap P)L \wedge (R \cap P)L \cap \overline{R} \cap P' \subseteq P.$$

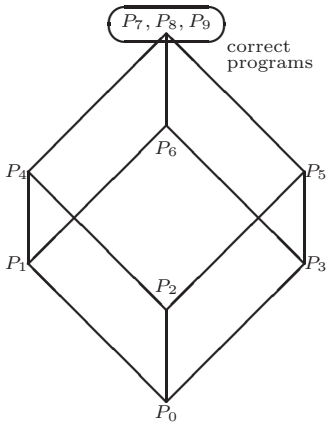

Fig. 3. Ordering Candidate Programs by Relative Correctness

2.4 Program Projection

We consider a space S defined by two integer variables x and y, and we let R be the following specification: $R = \{(s, s')|x' = x + y\}$. We let `p` be the following candidate program: {`while (y!=0) {x=x+1; y=y-1;}`}.
The function of `p` is: $P = \{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0\}$.
When we consider this function, we find that it has clauses (e.g. $x' = x + y$) that are mandated by the specification R and clauses (e.g. $y' = 0$) that are not mandated by R but stem instead from the design of P. In [6], we introduce an operator $\Pi_R(P)$, called the *projection of P over R*, which represents the functionality of P that is mandated by R. In the example above, we want the projection of P over R to be: $\Pi_R(P) = \{(s, s')|y \geq 0 \wedge x' = x + y\}$. Indeed, P delivers $(y' = 0)$ but R does not require it; and R mandates $(x' = x + y)$ for negative y but P does not deliver it.

Definition 2.5. *Due to [6]. Given a specification R on space S and a program P on S, the* projection of P over R *is the relation denoted by $\Pi_R(P)$ and defined as $(R \cap P)L \cap (R \cup P)$.*

The importance of projections is reflected in the following proposition ([6]).

Proposition 2.6. *Given a specification R and two programs P and P', P' is more-correct than P with respect to R if and only if $\Pi_R(P')$ refines $\Pi_R(P)$.*

Since the projection of a program on a specification reflects the functionality of the program that is relevant to the specification, it is only normal that it be the only part of P that determines relative correctness with respect to the specification.

3 Sample Examples

3.1 Fermat Decomposition

This example, due to [7], uses a specification due to [8]. We let space S be defined by natural variables n, x and y, and we let specification R be defined as:

$R = \{(s, s')|n = x'^2 - y'^2 \wedge 0 \leq y' \leq x'\}$. The domain of R is the set of states s such that $n(s)$ is either odd or a multiple of 4. Hence we write: $RL = \{(s, s')|n \bmod 2 = 1 \vee n \bmod 4 = 0\}$. Whereas Dromey [8] presents a sequence of designs that are increasingly more concrete, we present a sequence of increasingly correct programs. Starting from the initial program P_0 =`abort`, we resolve to let the next program P_1 find the required factorization for $y' = 0$:

```
void p1()
   {nat n, x, y; x=0; y=0;
    {nat r; r=0; while (r<n) {r=r+2*x+1; x=x+1;}}}
```

The function of this program is:

$P_1 = \{(s, s')|n' = n \wedge y' = 0 \wedge x' = \lceil \sqrt{n} \rceil\}$.

Whence we compute the competence domain of P_1 with respect to R:

$(R \cap P_1)L = \{(s, s')|\exists x'' : n = x''^2\}$.

In other words, P_1 satisfies specification R whenever n is a perfect square. We now consider the case where r exceeds n by a perfect square, making it possible to fill the difference with y^2:

```
void p2() {nat n, x, y;   //  input/output variables
    x=0; y=0; {nat r; r=0; while (r<n) {r=r+2*x+1; x=x+1;}
    if (r>n) {while (r>n) {r=r-2*y-1; y=y+1;}}}}
```

The function of this program is:

$P_2 = \{(s, s')|n' = n \wedge x' = \lceil \sqrt{n} \rceil \wedge y'^2 = x'^2 - n \wedge y' \geq 0\}$.

The competence domain of P_2 with respect to R is:

$(R \cap P_2) \circ L = \{(s, s')|\exists y'' : y''^2 = \lceil \sqrt{n} \rceil^2 - n\}$.

The competence domain of P_2 is the set of states s such that the difference between $n(s)$ and the square of the ceiling of the square root of $n(s)$ is a perfect square. This is a superset of the competence domain of P_1, hence P_2 is more-correct than P_1. The next program is derived from P_2 by resolving that if the ceiling of the integer square root of n does not exceed n by a perfect square, then we try the next perfect square, etc. We know that this process converges for any state s for which $n(s)$ is odd or a multiple of 4. Hence,

```
void p3() {nat n, x, y;   //  input/output variables
    {nat r; x=0; r=0; while (r<n) {r=r+2*x+1; x=x+1;}
    while (r>n)  {int rsave; y=0; rsave=r;
        while (r>n) {r=r-2*y-1; y=y+1;}
        if (r<n) {r=rsave+2*x+1; x=x+1;}}}}
```

If we let $\mu(n)$ be the smallest number whose square exceeds n by a perfect square, we write the function of P_3 as follows:

$P_3 = \{(s, s')|n' = n \wedge x' = \mu(n) \wedge y' = \sqrt{\mu(n)^2 - n}\}$.
We compute the competence domain of P_3 with respect to R and we find: $(R \cap P_3) \circ L = RL$. Hence P_3 is correct with respect to R hence it is more-correct than P_2 with respect to R. Hence we do have:

$P_0 \sqsubseteq_R P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3$.
Furthermore, we find that P_3 is correct with respect to R.

3.2 The Ceiling of the Square Root

This example, due to Reinfelds [21], consists in computing the non-negative integer square root of a non-negative integer n; its space is defined by variables n and x of type integer, and its specification is written as:

$$R = \{(s, s')|x'^2 \leq n < (x' + 1)^2 \wedge x' \geq 0\}.$$

In [21] Reinfelds offers two solutions to this problem; both solutions focus on the derivation of a while loop, and both start by deriving a loop invariant from the post-condition. We let the first program be `p0: abort`, whose competence domain (CD_0) is the empty set, and for the next program, we choose:

`p1: {int x, n; x=0;}`.
The function of this program and its competence domain are given as:

$P_1 = \{(s, s')|x' = 0 \wedge n' = n\}$

$CD_1 = (R \cap P_1)L = \{(s, s')|0 \leq n < 1 \wedge x' = 0 \wedge n' = n\}L = \{(s, s')|n = 0\}$.
This program satisfies the specification only for $n = 0$. For the next program, we want to satisfy R whenever n is a perfect square.

`p2: {int x,n;x=0; {int x2=0; while(x2<n){x2=x2+2*x+1;x=x+1;}}}`.
We compute the function of the while loop using invariant relations, as we discuss in [19], and we find the following function for P_2:

$P_2 = \{(s, s')|(x' - 1)^2 < n \leq x'^2 \wedge x' \geq 0 \wedge n' = n\}$.
We find that the competence domain of P_2 is: $CD_2 == \{(s, s')|\exists x' : n = x'^2\}$, which means that n is a perfect square. For the fourth program, we want to satisfy specification R even when n is not a perfect square. We consider the following program:

`p3: {int x, n; x=0;`
`{int x2=0; while(x2<n){x2=x2+2*x+1;x=x+1;} if (x2>n) {x=x-1;}}}`
The function of program P_3 can be obtained from that of P_2 by multiplying it on the right by the function of the `if-then` statement, which is:

$F = \{(s, s')|x^2 > n \wedge x' = x - 1 \wedge n' = n\} \cup \{(s, s')|x^2 = n \wedge s' = s\}$.
By computing the product, then simplifying the terms, we find:

$P_3 = \{(s, s')|x'^2 \leq n < (x' + 1)^2 \wedge x' \geq 0 \wedge n'' = n\}$.
The competence domain of P_3 is: $CD_3 = \{(s, s')|n \geq 0\}$. This is equal to the domain of R, hence P_3 is correct with respect to R.

3.3 Analyzing a String

This example is due to [9], and aims to scan a sequence q and count the number of letters, digits and other symbols. We let S be the space defined by a variable

q of type *string* and integer variables *let*, *dig*, and *other*; and we let R be defined as:

$$R = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \vartheta \cup \sigma \rangle \wedge \\ let' = \#_a(q) + \#_A(q) \wedge dig' = \#_\vartheta(q) \wedge other' = \#_\sigma(q)\}$$

where $list\langle T \rangle$ denotes the set of lists of elements of type T, and $\#_A$, $\#_a$, $\#_\vartheta$ and $\#_\sigma$ denote the functions that to each list l assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits and symbols. We generate the following programs:

p0: {abort}. $CD_0 = \phi$.

```
p1: {i=0; let=0; dig=0; other=0; l=strlen(q);
    while (i<l) {c=q[i];i++; if ('A'<=c && 'Z'>=c) let+=1;}}
```

$CD_1 = \{(s, s') | q \in list\langle \alpha_A \rangle\}$.

```
p2: {i=0; let=0; dig=0; other=0; l=strlen(q);
    while (i<l) {c = q[i]; i++;
       if ('A'<=c && 'Z'>=c) let+=1;
       else if ('a'<=c && 'z'>=c) let+=1;}}
```

$CD_2 = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \rangle\}$.

```
p3: {i=0; let=0; dig=0; other=0; l=strlen(q);
    while (i<l) {c = q[i]; i++;
       if ('A'<=c && 'Z'>=c) let+=1;
       else  if ('a'<=c && 'z'>=c) let+=1;
       else  if ('0'<=c && '9'>=c) dig+=1;}}
```

$CD_3 = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \nu \rangle\}$.

```
p4: {i=0; let=0; dig=0; other=0; l=strlen(q);
    while (i<l) {c = q[i]; i++;
       if ('A'<=c && 'Z'>=c) let+=1;
       else if ('a'<=c && 'z'>=c) let+=1;
       else if ('0'<=c && '9'>=c) dig+=1;
       else other+=1;}}
```

$CD_4 = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \nu \cup \sigma \rangle\}$.

Since $CD_0 \subseteq CD_1 \subseteq CD_2 \subseteq CD_3 \subseteq CD_4$, we do have $P_0 \sqsubseteq_R P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3 \sqsubseteq_R P_4$; also, we find $CD_4 = RL$, hence $P_4 \sqsupseteq R$, i.e. P_4 is correct with respect to R.

3.4 Word Wrap

The specification of this problem is borrowed from [15, 20]; for the sake of readability and brevity, we present the English text of the specification (due to [17]), but not the relational representation.

"The program accepts as input a finite sequence of characters and produces as output a sequence of characters satisfying the following conditions:

- If the input sequence contains MaxPos+1 consecutive non-break characters then a boolean flag (longWord) is set to true.
- Else,

- LongWord is set of false.
- All the words of the input appear in the output, in the same order, and all the words of the output appear in the input.
- Furthermore, the output must must satisfy the following conditions:
  - It contains no leading or trailing breaks, nor consecutive breaks, where a break is a blank, or a newline or the end-of-file.
  - Any sequence of MaxPos+1 consecutive characters includes a newline.
  - Any subsequence made up of no more than MaxPos characters and embedded between the head of the sequence or a new line on the left, and the tail of the sequence or a break on the right contains no newline."

Due to space limitations, we do not compute the function of each program in our sequence of solutions, but content ourselves with presenting their competence domains. We generate the following sequence of programs for this specification, starting from `p0: abort`, whose competence domain is empty. The first program merely echos the input to the output; we generate it to pin down the mechanics of file transfer in C++.

```
p1:  #include <fstream> using namespace std;
const int MaxPos = 80; const char blank = ' ';
ifstream inpstr;  ofstream outpstr;
int main ()
   {inpstr.open("inp1.dat"); outpstr.open("outp1.dat");
    char c;  c=inpstr.get();
    while (!inpstr.eof()) {outpstr << c; c=inpstr.get();}
    inpstr.close(); outpstr.close();}
```

The competence domain (CD_1) of this program is the set of input sequences that are not longer than MaxPos, have no newlines, no leading or trailing blanks, and single blanks between words. The second program assumes that the input contains no newlines, and merely removes leading and trailing blanks, as well as extra blanks between words.

```
p2:  #include <fstream>
#include <string> using namespace std;
const int MaxPos = 80; const char blank = ' ';
const string emptyword="";
ifstream inpstr; ofstream outpstr; char c;  string word;
void skipblanks();
void echoword();
int main ()
   {inpstr.open("inp2.dat"); outpstr.open("outp2.dat");
    c=inpstr.get(); skipblanks();
    while (!inpstr.eof()) {echoword();}
    inpstr.close(); outpstr.close();}
```

```
void skipblanks()
   {while ((!inpstr.eof()) && (c==blank)) {c=inpstr.get();};}
void echoword ()
   {bool leadingblanks; leadingblanks=(c==blank); skipblanks();
    string word; word=emptyword;
    while ((!inpstr.eof())&&(c!=blank)) {word+=c;c=inpstr.get();}
    if (word.length()>0) {if (leadingblanks)
    {outpstr << blank << word;} else {outpstr << word;}}}
```

The competence domain (CD_2) of this program is the set of sequences whose compacted version (when extra blanks are removed) is not longer than MaxPos, and have no newlines. The third program removes newlines in addition to extra spaces throughout the data stream.

```
p3:  //  same as p2, except:
const char lf = '\n'; const char cr = '\r';
void skipblanks()
   {while ((!inpstr.eof())&&((c==blank)||(c==lf)||(c==cr)))
      {c=inpstr.get();}}
void echoword ()
   {bool leadingblanks;
    leadingblanks=((c==blank)||(c==lf)||(c==cr)); skipblanks();
    string word; word=emptyword;
    while ((!inpstr.eof()) && (c!=blank) && (c!=lf) && (c!=cr))
       {word +=c; c=inpstr.get();}
    if (word.length()>0) {if (leadingblanks)
       {outpstr << blank << word;} else {outpstr << word;}}}
```

The competence domain (CD_3) of this program is the set of sequences whose compacted version (when blanks and newlines are removed) is not longer than MaxPos. The fourth program places newlines at the appropriate places in the output stream.

```
p4:
//  same as p3 except:
int linelen;
void echoword ()
   {bool leadingblanks;
    leadingblanks=((c==blank)||(c==lf)||(c==cr));skipblanks();
    string word; word=emptyword;
    while ((!inpstr.eof()) && (c!=blank) && (c!=lf) && (c!=cr))
        {word +=c; c=inpstr.get();}
    if (word.length()>0)
       {if (leadingblanks)
           {if (linelen+word.length()+1>MaxPos)
              {linelen=word.length();
               outpstr << endl << word;}
```

```
            else  {linelen=linelen+word.length()+1;
               outpstr << blank << word;}}
        else {outpstr << word; linelen=word.length();}}}
```

The competence domain (CD_4) of this program is the set of sequences which have no words longer than MaxPos. The fifth program takes into account the possibility of encountering long words in the input stream, and proceeds to set the boolean flag `longword` to true, while skipping the long words.

```
p5:   //  same as p4, except:
int main ()
   {// ... ... ...
    c=inpstr.get();linelen=0;longword=false;skipblanks();...}
void echoword ()
   {bool leadingblanks;
    leadingblanks=((c==blank)||(c==lf)||(c==cr));skipblanks();
    string word; word=emptyword;
    while ((!inpstr.eof())&&(c!=blank)&&(c!=lf)&&(c!=cr))
        {word +=c; c=inpstr.get();}
    if (word.length()>0)
       {if (word.length()>MaxPos)
           {longword=true;}
        else {if (leadingblanks)
              {if (linelen+word.length()+1>MaxPos)
                  {linelen=word.length();
                   outpstr << endl << word;}
               else  {linelen=linelen+word.length()+1;
                   outpstr << blank << word;}}
           else {outpstr << word; linelen=word.length();}}}}
```

The competence domain (CD_5) of this program is the set of all input sequences.

As we can see, $CD_0 \subseteq CD_1 \subseteq CD_2 \subseteq CD_3 \subseteq CD_4 \subseteq CD_5$, hence $P_0 \sqsubseteq_R P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3 \sqsubseteq_R P_4 \sqsubseteq_R P_5$. Also, because $CD_5 = dom(R)$, where R is the (unwritten, but available in [17]) specification, P_5 is correct with respect to R.

We have developed our programs in a stepwise manner, by considering broader and broader subsets of the domain of the specification, until we reach the whole domain. To some extent, the transition from one program to the next preserves much of the code that has been written, and modifies/ adds relatively little code. Perhaps more interestingly, the stepwise correctness enhancements enable us to tackle the complexity of the specification one issue at a time, and to validate our solution for one step before we tackle the next step: we have coded the file processing aspects in `p1`, the extra (leading, trailing, middle) blanks in `p2`, the removal of incoming newlines in `p3`, the insertion of outgoing newlines in `p4`, and the detection of long words in `p5`. At each step, we ensure that the program works properly for the targeted competence domain before we consider the next (broader) competence domain.

4 Programming for Reliability

The reliability of a program P can be defined with respect to two parameters: a specification R in the form of a binary relation; and a discrete probability distribution θ over the domain of R, reflecting a given usage pattern. We have seen in section 2.3 that for deterministic programs, enhanced correctness logically implies (but is not equivalent to) enhanced reliability. This means that if the derivation of a correct program P from a specification R proceeds through a sequence of increasingly correct programs, say P_0, P_1, P_2, etc.. $P_n = P$, then the sequence of P_i's is ordered by increasing reliability. So that the only difference between deriving a correct program and deriving a sufficiently reliable program (for a required reliability threshold) is that in the latter case we can end the derivation earlier, namely as soon as the reliability of P_i matches or exceeds the selected threshold. Given that correctness is the culmination of reliability, it is only fitting that the derivation of correct programs be the culmination of the derivation of reliable programs.

To illustrate our claim, we consider the four sample program derivations presented in the previous section, and for each example we derive a test driver and a (random) test data generator. Then we apply each test driver to the sequence of programs $\{P_i\}$ generated in the corresponding derivation. This allows us to estimate the reliability of each program P_i in each example; The table below shows how the reliability evolves as we proceed from one program to the next; the first column shows the size of the (random) test data used for each example.

	Test Data Size	P_0	P_1	P_2	P_3	P_4	P_5
Fermat	4000	0.0000	0.2535	0.3445	1.0000		
Sqrt Ceiling	4000	0.0000	$\approx$ 0.0000	0.1020	1.0000		
String Analysis	100	0.0000	0.0057	0.2790	0.2917	1.0000	
Word Wrap	3000	0.0000	0.0363	0.0873	0.1023	0.8990	1.0000

5 Critique

5.1 Refinement vs Correctness Enhancement

To elucidate the contrast between refinement and correctness enhancement, we revisit the concept of *projection*, discussed in section 2.4. This operator has many projection-like properties, hence its name, including:

- Idempotence: $\Pi_R(\Pi_R(P)) = \Pi_R(P)$.
- The projection of P over R is refined by P and by R.
- Program P is correct with respect to R if and only if $\Pi_R(P) = R$.
- Program P' is more-correct than program P with respect to R if and only if the projection of P' over R refines the projection of P over R.

This last property is interesting because it highlights the contrast between refinement and correctness enhancement. Whereas refinement mandates that we

refine all of P, relative correctness mandates that we only refine the projection of P over R, which is known to be less-refined than P.

The concept of projection enables us to distinguish between two sources of functional properties in a program P:

- *Functional attributes that are mandated by the specification.* This is the projection of P over R.
- *Functional attributes that are determined by design decisions.* This the functionality that is delivered by P but not mandated by R.

See Figure 4. The difference between refinement and relative correctness is that at each step, refinement refines all of P whereas relative correctness refines only those functional attributes of P that are mandated by the specification; we may, in the process of doing so, override design decisions made previously. Because it makes no distinction between specification-mandated attributes and design-dictated attributes the paradigm of refinement must refine all the functional attributes of P; consequently, every design decision taken during this process imposes constraints on subsequent steps.

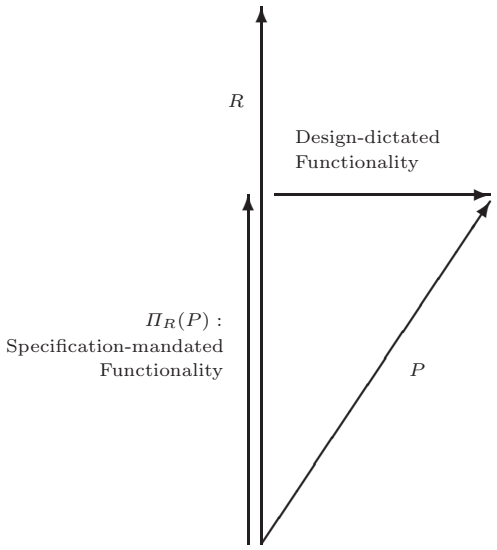

Fig. 4. Decomposing the Function of a Program

5.2 Critique: Is Correctness Enhancement a Viable Methodology?

Looking at the discussion of the previous section, one would be forgiven for thinking that correctness enhancement is a panacea for program derivation: it

involves refining a weaker specification than the traditional refinement based process ($\Pi_R(P)$ rather than P); we can take design decisions and not be constrained by them (override them subsequently, if needed); we can stop halfway through the derivation process and still have something to show for our effort (an executable program that runs correctly for part of $dom(R)$); the artifacts we generatee as we proceed are increasingly reliable; etc. But like everything else in life, this is too good to be true. Indeed, these advantages come at a cost, in terms of breadth of scope: In practice, correctness enhancement is viable only to the extent that each artifact P_i generated through this process can be derived easily from the previous artifact, P_{i-1}. But this is not always the case: just because P_i represents a small increment of relative correctness over P_{i-1} does not necessarily mean that P_i can be derived from P_{i-1} by a simple syntactic modidfication. In the examples of section 3 this was mostly true, particularly for the word wrap example, where the stepwise correctness enhancement process helped us deal with one aspect of the specification at a time.

Also, it is great that design decisions taken along the process do not constrain subsequent design decisions, but this comes at the cost of modularity: in a refinement process, each design decision is validated then fixed, and subsequent decisions are taken and validated accordingly. But with correctness enhancement, at each step much of the new code must be analyzed and verified anew; there is some potential for verification reuse, but it is not built into the process.

Still, we argue that correctness enhancement is worthy of study, not because it is better than refinement at the derivation of programs from scratch, but rather because unlike refinement, correctness enhancement models not only program derivation from scratch (a small and shrinking segment of software engineering practice), but also the vast majority of software engineering processes. Indeed, we find that corrective maintenance, adaptive maintenance, software merger, software upgrade, whitebox software reuse, extreme programming, and test driven design are all instances of correctness enhancement. Much of software engineering practice consists, not of developing a new product from scratch, but rather of taking a software product that does not quite meet our needs, and evolving it to meet new requirements; this is essentially a correctness enhancement (with respect to the new requirements) operation.

5.3 Related Work

To the extent that it can be seen as a weaker (less generic) form of refinement, relative correctness bears some similarity with *retrenchment* [3]. It is possible to think of relative correctness (more specifically: the property of being less-correct than) as being an instance of retrenchment, for a suitable choice of the *concedes* relation. Yet, whereas retrenchment appears to apply to data abstraction in the context of B specifications, relative correctness pertains primarily to programs modeled as mappings between initial states and final states.

Our notion of relative correctness is tightly coupled with our own version of refinement, both in terms of its definition and in terms of its notation. We use relational algebra and we define refinement as: $R' \sqsupseteq R \Leftrightarrow (RL \cap R'L \cap$

$(R \cup R') = R$). While this formula captures, in relational terms, a fairly generic understanding of refinement (larger domain, fewer images per argument), it is still fairly different from other definitions of refinement, such as those of [1, 2, 11–13, 18]. An interesting venue of research would be to explore how to derive a definition of relative correctness that corresponds to these refinement formulas.

6 Conclusion

Traditional refinement-based program derivation proceeds by successive correctness-preserving transformations starting from the specification and ending with an executable program, when all the specification notations have been replaced by program statements. In this paper we have explored an orthogonal approach, which starts from the trivially incorrect program `abort` and proceeds by successive correctness enhancing transformation until we reach a correct program, or a sufficiently reliable program. While it offers many advantages, the proposed method is only applicable to the extent that incremental correctness enhancements can be achieved by commensurably incremental amendments to the text of the program.

Regardless of its prowess in deriving programs from scratch, and of its relative merit by comparison with refinement based programming, correctness enhancement is worthy of study because it proves to be an adequate model for a wide range of software engineering activities. In order for these insights to be useful, we need to explore how mathematics of relative correctness can be turned into scalable methods and tools, and how the results that we have derived for our definition of refinement can be repurposed for other forms of refinement. This is currently under investigation.

References

[1] J.R. Abrial (1996): *The B Book: Assigning Programs to Meanings*. Cambridge University Press.

[2] R.J. Back & J. von Wright (1998): *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science, Springer Verlag.

[3] R. Banach & M. Poppleton (2000): *Retrenchment, Refinement and Simulation*. In: *ZB: Formal Specifications and Development in Z and B*, Lecture Notes in Computer Science, Springer, pp. 304–323, doi:10.1007/3-540-44525-0_18.

[4] Chris Brink, Wolfram Kahl & Gunther Schmidt (1997): *Relational Methods in Computer Science*. Advances in Computer Science, Springer Verlag, Berlin, Germany.

[5] J. Desharnais, N. Diallo, W. Ghardallou, M. F. Frias, A. Jaoua & A. Mili (2015): *Relational Mathematics for Relative Correctness*. In: *RAMICS, 2015*, *LNCS* 9348, Springer Verlag, Braga, Portugal, pp. 191–208.

[6] Jules Desharnais, Nafi Diallo, Wided Ghardallou & Ali Mili (2017): *Projecting Programs on Specifications: Definitions and Implications*. *Science of Computer Programming*.

[7] Nafi Diallo, Wided Ghardallou & Ali Mili (2015): *Program Derivation by Correctness Enhancements*. In: *Proceedings, Refinement 2015*, Oslo, Norway.

[8] Geoffrey Dromey (1983): *Program Development by Inductive Stepwise Refinement*. Technical Report Working Paper 83-11, University of Wollongong, Australia, doi:10.1002/spe.4380150102.

[9] Alberto Gonzalez-Sanchez, Rui Abreu, Hans Gerhart Gross & Arjan J.C. van Gemund (2011): *Prioritizing Tests for Fault Localization through Ambiguity Group Reduction*. In: *proceedings, Automated Software Engineering*, Lawrence, KS.

[10] David Gries (1981): *The Science of Programming*. Springer Verlag, doi:10.1007/978-1-4612-5983-1.

[11] Eric C. R. Hehner & Andrew M. Gravell (1999): *Refinement Semantics and Loop Rules*. In: *Formal Methods 1999*, Lecture Notes in Computer Science, Springer Verlag.

[12] Eric C.R. Hehner (1992): *A Practical Theory of Programming*. Prentice Hall, doi:10.1007/978-1-4419-8596-5.

[13] Tony Hoare (1997): *Unified Theories of Programming*. In Manfred Broy & Birgit Schieder, editors: *Mathematical Methods in Program Development*, Springer-Verlag, pp. 313–367.

[14] Zohar Manna (1974): *A Mathematical Theory of Computation*. McGraw-Hill.

[15] Bertrand Meyer (1985): *On Formalism in Specification*. *IEEE Software* 2(1), pp. 6–27.

[16] A. Mili, M. Frias & A. Jaoua (2014): *On Faults and Faulty Programs*. In P. Hoefner, P. Jipsen, W. Kahl & M. E. Mueller, editors: *Proceedings, RAMICS 2014*, *LNCS* 8428, pp. 191–207.

[17] A. Mili, X.Y. Wang & Y. Qing (1986): *A Relational Specification Methodology*. *Software- Practice and Experience*, pp. 1030–1030.

[18] Carroll C. Morgan (1998): *Programming from Specifications, Second Edition*. International Series in Computer Sciences, Prentice Hall, London, UK.

[19] Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Jules Desharnais & Ali Mili (2013): *Invariant Assertions, Invariant Relations, and Invariant Functions*. *Science of Computer Programming* 78(9), pp. 1212–1239. Available at http://dx.doi.org/10.1016/j.scico.2012.05.006.

[20] Daniel Perelman, Sumit Gulwani, Dan Grossman & Peter Provost (2014): *Test Driven Synthesis*. In: *Proceedings, 35th ACM SIGPLAN Conference, PLDI*, 49, Edinburgh, UK, pp. 408–418.

[21] Juris Reinfelds (1986): *A Brief Introduction to the Derivation of Programs*. Technical Report, University of Wollongong, Wollongong, NSW Australia.