The Transformational Paradigm as a Means of Smoothing Abrupt Software Design Steps

by
J.N.Oliveira

30.Dec.85 — Int.Rep. CCES-JNO:R2/85

C.C.E.S.
Universidade do Minho
Rua D. Pedro V, 88-2º
4700 Braga
Portugal

Abstract

Current state-of-the-art methods and specification languages devised for software design have not yet accomplished a completely satisfactory calculus of formal rules enabling a smooth, deductive style of software development.

Experience has shown evidence that the so-called *refinement* steps are hard to achieve in practice because too much intuition and *craft* are expected from the software designer. This problem is particularly apparent in constructive methodologies such as the Vienna Development Method (VDM).

This report is concerned with the proposal of alternative, *transformational* approaches to doing *model refinement* in VDM. This is intended to "smooth" such abrupt realization strategies, which are regarded as potential "bottlenecks" of the methodology.

1 Introduction

During the past decade, work on *abstract data types* [1, 2, 3] has uncovered (and proposed solutions for) a great deal of defficiencies in the traditional way of developing software. This work has emphasized the need for formal (mathematical) support, and puts forward strategies whereby the *specification* of a piece of software (which should be as abstract as possible) and its *implementation* (where detail and clumsiness are likely to crop up) are effectively separated.

The abstract-data-type theory is the kernel of many current approaches to Software Engineering, which are targetted at devising methods for building programs in the same, systematic way designs arise in other areas of Engineering. Typically, models of the physical world are constructed and denoted by some mathematical notation. Associated with this notation, there is available a *calculus* (sets of laws, or rules) which is used in a *deductive* way to develop and reason about designs (typically, these arise as solutions of systems of equations).

Most methodologies and specification languages hitherto devised for software design (*e.g. constructive* ones such as VDM [3], Z [4] and *me too* [5], or *algebraic* ones such as *e.g.* OBJ [6], CIP-L [7]) have not yet accomplished a completely satisfactory calculus of deductive rules which would

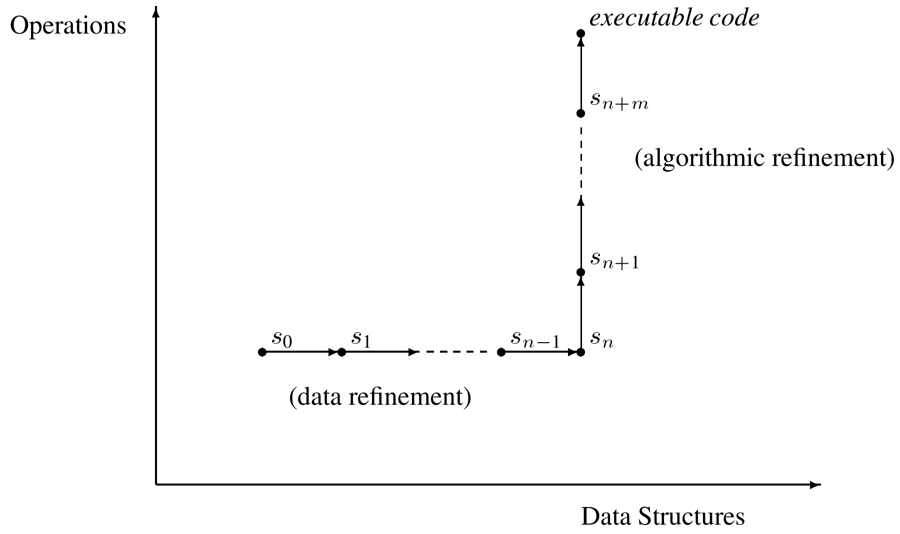

Figure 1: Data/operation-refinement

enable such a smooth, deductive style of software development. Experience has shown evidence that the hardest steps in development (from the above viewpoint) are the so-called *refinement* [3] (or *realization* [2]) steps, because too much intuition and *craft* are expected from the software designer. Realization steps are, in fact, *inductive* (as opposed to *deductive*) in nature. This problem is particularly apparent in constructive methodologies such as VDM (*cf. data refinement* and *operation refinement* [3] steps). This report is concerned with proposing an alternative, *transformational style* (in the sense of *e.g.* [8, 9]) of doing *model refinement* in VDM. This approach is intended to "smooth" such abrupt realization strategies, which are regarded as potential "bottlenecks" of the methodology.

2 The Role of Refinement in Software Design

Wirth's formula "Algorithms + Data Structures = Programs" [10] has become famous as the title of a mandatory textbook on structured programming since the mid 1970s. The author prefers to read it as being probably the first, widespread message that *programs* "are" *algebras*, *cf.* "Operators + Objects = Algebras". This algebraic view of programming has the consequence that, whatever is said about (or happens to) programs, can always be decomposed across two complementary axes: the objects programs deal with (*e.g.* data structures *etc.*) and the operations which manipulate them (*e.g.* procedures, functions *etc.*).

Refinement is no exception in this respect. In VDM, program-design starts from abstract specifications (termed *abstract models*) whose algebraic structure is "coated" with imperative- programming-oriented features such as the global *state*, and pre- and post-condition pairs (which fix the input/output relation of each operation).

The way the VDM user proceeds from abstract specifications to more concrete models (closer and closer to the target machine, or available programming language) is by *model-refinement*. It is recommended that one the above axes should be considered first: in fact, VDM suggests that *data-refinement* be performed first (possibly encompassing several iterations) [3]. Once a satisfactory *implementation-model* is reached, refinement decisions are then taken on the orthogonal direction (*operation-refinement*) so as to, eventually, reach executable code *e.g.* in PL/1, Pascal *etc.*, *cf.* Figure 1.

This strategy of gradually reaching the complexity of an implementation is not totally systematic. Each refinement decision is taken inductively, based (maybe) on the designer's (arbitrary) experience and intuition. The method is, however, careful about systematic proof (using formal correctness-rules [3]) that such intuitions observe the original intentions captured by the specification, and everything ends up happily.

The following sections show (by giving two examples) that refinement steps can be (at least in part) replaced by deductive reasoning using transformational strategies.

3 Model Refinement Supported by Transformations

In VDM, the correctness-proof of model-refinements rely on the relationship between each model and its closest descendant, as captured (formally) by a *retrieve function* [3] mapping the object space of the latter onto the object space of the former (*cf.* Σ-morphisms in [2]). The next step is to provide (at refined-model level) specifications for the operations available at high-level. Finally, each of these new specifications has then to be proved correct wrt. the respective specification at high-level.

The example below shows a transformational way of by-passing this work once a retrieve function is fixed. Let us assume that we have made a design decision whereby arbitrary (finite) subsets of a given set X (X-set in VDM notation) are to be represented by linked-lists in Pascal. In order to achieve this, it is wise to decompose such a refinement step into the following sub-steps: firstly, we refine X-set into the VDM list-space X-list, and then we refine this further until we reach the Pascal level.

The first refinement decision is captured by the well-known operator

$$
\begin{array}{rcl}
elems & : & X\text{-list} \rightarrow X\text{-set} \\
elems(l) & \stackrel{\text{def}}{=} & if\ \ l = <> \\
 & & then\ \ \{\} \\
 & & else\ \ \{hd(l)\} \cup elems(tl(l))
\end{array}
\tag{1}
$$

which yields the set of all elements occurring in a given list. To complete this intermediate model, one has to specify list-level operations corresponding to each set-level operation, *e.g.* ∈ (membership test), ∪ (union) *etc.* Let us concentrate on an operation

$$member : X \times X\text{-list} \rightarrow Bool$$

which is to be our list-level realization of ∈. Guided by our intuition and experience on *head and tail* recursion, we might have decided to define *member* as follows:

$$
\begin{array}{rcl}
member(x,l) & \stackrel{\text{def}}{=} & if\ \ l = <> \\
 & & then\ \ false \\
 & & else\ \ if\ \ x = hd(l) \\
 & & \qquad then\ \ true \\
 & & \qquad else\ \ member(x, tl(l))
\end{array}
\tag{2}
$$

We would then have to prove that *member* is a correct realization of ∈. This amounts to showing that

$$member(x,l) = x \in elems(l) \tag{3}$$

holds (which partly proves that $elems$ behaves like a homomorphism from lists to sets), *i.e.* that the diagram of Figure 2 comutes. (This proof is presented in the Appendix, *cf.* Exercise 1).

Alternatively, a way of *deducing* the definition of $member$ (instead of "inventing" it) is presented next. This is achieved by taking (3) as the definitional axiom of $member$, and performing *fold-unfold*

$$
\begin{tikzcd}
X \times & X\text{-set} \arrow[r, "\in"] & Bool \\
X \times \arrow[u, "id"] & X\text{-list} \arrow[u, "elems"'] \arrow[r, "member"'] & Bool \arrow[u, "id"']
\end{tikzcd}
$$

Figure 2: Refinement diagram for $\in$

transformations via the definition of $elems$ (1):

$$
\begin{array}{rcl}
member(x,l) & = & x \in elems(l) \\
 & = & x \in \ \begin{array}[t]{l} if \ \ l = <> \\ \quad then \ \ \{\} \\ \quad else \ \ \{hd(l)\} \cup elems(tl(l)) \end{array} \\
 & = & \begin{array}[t]{l} if \ \ l = <> \\ then \ \ x \in \{\} \\ else \ \ x \in \{hd(l)\} \vee x \in elems(tl(l)) \end{array} \\
 & = & \begin{array}[t]{l} if \ \ l = <> \\ then \ \ false \\ else \ \ x = hd(l) \vee member(x, tl(l)) \end{array} \\
 & = & \begin{array}[t]{l} if \ \ l = <> \\ then \ \ false \\ else \ \ \begin{array}[t]{l} if \ \ x = hd(l) \\ then \ \ true \\ else \ \ member(x, tl(l)) \end{array} \end{array}
\end{array}
$$

thus obtaining (2) [1].

Definitions for the remaining operations (intersection, union *etc.*) may be achieved in a similar way. For example, the empty list ($<>$) is the only object in X-list such that $elems(<>) = \{\}$. So, $<>$ represents the empty-set at list-level. Exercise 2 in the Appendix shows how the following definition of an operator int (modelling intersection):

$$
\begin{array}{rcl}
int & : & X\text{-list} \times X\text{-list} \rightarrow X\text{-list} \\
int(l,l') & \stackrel{\text{def}}{=} & \begin{array}[t]{l} if \ \ l = <> \\ then \ \ <> \\ else \ \ \begin{array}[t]{l} if \ \ member(hd(l), l') \\ then \ \ <hd(l)> \frown int(tl(l), l') \\ else \ \ int(tl(l), l') \end{array} \end{array}
\end{array} \tag{4}
$$

is derived from its definitional axiom:

$$
elems(int(l,l')) = elems(l) \cap elems(l') \tag{5}
$$

again via *fold-unfold* transformations through $elems$. Note that the algorithmic structure of $elems$ ("head and tail recursion") induces similar structures in $member$, int, *etc.* In other words, these operations are "contaminated" by the definition we have chosen for $elems$). This shows that the retrieve function is much more decisive as a design step than we might have thought. It may be regarded as a "generator" of the definitions of the low-level model operations.

[1] The last step above consists of replacing logical disjunction ($\vee$) by its *lazy version*. This means that, given two predicates a and b, $a \vee b$ is replaced by $if \ a \ then \ true \ else \ b$. Note that the reverse replacement is not valid.

4 Operation Refinement Supported by Transformations

In VDM, operations are (abstractly) specified via pre- and post-condition pairs. The target of operation-refinement is to develop executable code from such specifications, and its nature is also inductive. For example, the designer might "suspect" that a particular specification OP may be refined into a *while-loop*, *e.g.*

$$\begin{aligned} OP = \ & while\ \ Exp \\ & do\ \ BODY \end{aligned}$$

A specification ($BODY$) for the body of this loop would have to be provided first and, before further refinement decisions are taken on $BODY$, this step is proved correct by using rules devoted to *while-loop* decomposition [3].

The example below illustrates an alternative, transformational approach to operation-refinement in VDM. The specification used in the example, borrowed from [11] (see also [12, 13]), appears in an intermediate model between an abstract mapping

$$Key \xrightarrow{m} Data$$

and its realization (in Pascal) in terms of binary trees. An intermediate type $BinTree$ is defined as follows:

$$\begin{array}{lcl} BinTree & = & [BinNode] \\ BinNode & :: & \text{LT: BinTree} \\ & & \text{K: Key} \\ & & \text{D: Data} \\ & & \text{RT: BinTree} \end{array}$$

subject to the *data-type invariant* (*cf.* [3]):

$$\begin{array}{rcl} invnd & : & BinNode \to Bool \\ invnd(\text{mk-}BinNode(lt,k,d,rt)) & = & (\forall lk \in collkeys(lt) : lk < k) \wedge (\forall rk \in collkeys(rt) : k < rk) \end{array} \quad (6)$$

where

$$\begin{array}{rcl} collkeys & : & BinTree \to Key\text{-set} \\ collkeys(t) & \stackrel{\text{def}}{=} & if\ \ t = NULL \\ & & then\ \ \{\} \\ & & else\ \ collkeys(LT(t)) \cup \{K(t)\} \cup collkeys(RT(t)) \end{array} \quad (7)$$

The operation to be realized is specified by

$$\begin{array}{l} FINDBIN(K : Key)D : Data \\ globals\ T : rdBinTree \\ pre\ k \in collkeys(t) \\ post\ d = findb(t,k) \end{array}$$

where

$$\begin{array}{rcl} findb & : & BinTree \times Key \to Data \\ \text{pre-}findb(t,k) & \stackrel{\text{def}}{=} & k \in collkeys(t) \\ findb(\text{mk-}BinNode(lt,mk,md,rt),k) & \stackrel{\text{def}}{=} & if\ \ k = mk \\ & & then\ \ md \\ & & else\ \ if\ \ k < mk \\ & & \qquad then\ \ findb(lt,k) \\ & & \qquad else\ \ findb(rt,k) \end{array} \quad (8)$$

We start from $findb$, which may be regarded as a preliminary, functional realization of $FINDBIN$. $findb$ is a partial function and may be turned into a total function $findbt$ by embedding its pre-condition in the definition, *i.e.* we are led to

$$\begin{array}{rcl} findbt & : & BinTree \times Key \rightarrow Data \\ findbt(t,k) & \stackrel{\text{def}}{=} & if\ \ k \in collkeys(t) \\ & & then\ \ findb(t) \\ & & else\ \ error \end{array} \tag{9}$$

(Similarly, the pre-condition of $FINDBIN$ may be removed and $findbt$ be replaced for $findb$ in post-$FINDBIN$). Let us start from transforming pre-$findb$ via (7):

$$\begin{array}{rcl} k \in collkeys(t) & = & if\ \ t = NULL \\ & & then\ \ k \in \{\} \\ & & else\ \ k \in collkeys(LT(t)) \vee k = K(t) \vee k \in collkeys(RT(t)) \end{array}$$

Since $k \in \{\} \Leftrightarrow false$, we may rewrite (9) as follows:

$$\begin{array}{rcl} findbt(t,k) & = & if\ \ \ (if\ \ t = NULL \\ & & \qquad then\ \ false \\ & & \qquad else\ \ k \in collkeys(LT(t)) \vee k = K(t) \vee k \in collkeys(RT(t))) \\ & & then\ \ findb(t,k) \\ & & else\ \ error) \\ & = & if\ \ t = NULL \\ & & then\ \ error \\ & & else\ \ \ if\ \ (k \in collkeys(LT(t)) \vee k = K(t) \vee k \in collkeys(RT(t))) \\ & & \qquad then\ \ findb(t,k) \\ & & \qquad else\ \ error \end{array} \tag{10}$$

Using associativity and commutativity of disjunction, and making the disjunctions above *lazy*, the second condition in (10) becomes

$$\begin{array}{lcl} k = K(t) \vee & & \\ k \in collkeys(LT(t)) \vee & \Leftrightarrow & if\ \ k = K(t) \\ k \in collkeys(RT(t))) & & then\ \ true \\ & & else\ \ \ if\ \ k \in collkeys(LT(t)) \\ & & \qquad then\ \ true \\ & & \qquad else\ \ k \in collkeys(RT(t)) \end{array}$$

Substituting above, we obtain:

$$\begin{array}{rl} findbt(t,k) = & if\ \ t = NULL \\ & then\ \ error \\ & else\ \ \ if\ \ k = K(t) \\ & \qquad then\ \ findb(t,k) \\ & \qquad else\ \ \ if\ \ k \in collkeys(LT(t)) \\ & \qquad\qquad then\ \ findb(t,k) \\ & \qquad\qquad else\ \ \ if\ \ k \in collkeys(RT(t)) \\ & \qquad\qquad\qquad then\ \ findb(t,k) \\ & \qquad\qquad\qquad else\ \ error \end{array} \tag{11}$$

The invariant on $BinNode$ (6) and the definition of $findb$ (8) guarantee the following facts, for all $t \in BinNode$ and $k \in Key$:

$$\begin{array}{rcl} k = K(t) & \Rightarrow & findb(t,k) = D(t) \\ k \in collkeys(LT(t)) & \Rightarrow & k < K(t) \Rightarrow findb(t,k) = findb(LT(t),k) \end{array} \tag{12}$$

and

$$k \in collkeys(RT(t)) \Rightarrow k > K(t) \Rightarrow findb(t,k) = findb(RT(t),k)$$

which, in turn, enable three substitutions in (11), leading to:

$$\begin{array}{rl} findbt(t,k) = & if\ \ t = NULL \\ & then\ \ error \\ & else\ \ \ if\ \ k = K(t) \\ & \qquad then\ \ D(t) \\ & \qquad else\ \ \ if\ \ k \in collkeys(LT(t)) \\ & \qquad\qquad then\ \ findb(LT(t),k) \\ & \qquad\qquad else\ \ \ if\ \ k \in collkeys(RT(t)) \\ & \qquad\qquad\qquad then\ \ findb(RT(t),k) \\ & \qquad\qquad\qquad else\ \ error \end{array} \tag{13}$$

By folding (13) according to (9),

$$\begin{array}{l} if\ \ k \in collkeys(RT(t))\ \ = findbt(RT(t),k) \\ then\ \ findb(RT(t),k) \\ else\ \ error \end{array}$$

we are led to

$$\begin{array}{rl} findbt(t,k) = & if\ \ t = NULL \\ & then\ \ error \\ & else\ \ \ if\ \ k = K(t) \\ & \qquad then\ \ D(t) \\ & \qquad else\ \ \ if\ \ k \in collkeys(LT(t)) \\ & \qquad\qquad then\ \ findb(LT(t),k) \\ & \qquad\qquad else\ \ findbt(RT(t),k) \end{array} \tag{14}$$

The final step consists of removing the remaining ocurrence of $collkeys$ in (14). This is achieved by weakening the predicate

$$k \in collkeys(LT(t))$$

in (14) to

$$k < K(t)$$

cf. (12) [2]. We thus broaden the set of values for which the test-condition of (14) evaluates to $true$, which thus includes the elements of

$$X = \{k \in Key |\ k < K(t)\} - collkeys(LT(t))$$

Since that, for any $k \in X$,

$$k \neq K(t) \wedge k \notin collkeys(LT(t)) \wedge k \notin collkeys(RT(t))$$

we have

$$findbt(RT(t),k) = error.$$

cf. (11). Thus, (14) may be replaced by

$$\begin{array}{l} if\ \ k < K(t) \\ then\ \ \ if\ \ k \in collkeys(LT(t)) \\ \qquad then\ \ findb(LT(t),k) \\ \qquad else\ \ error \\ else\ \ findbt(RT(t),k) \end{array} \tag{15}$$

[2] Given a predicate p, by "weakening p" we mean replacing p by some q such that $p \Rightarrow q$.

Folding (15) via (9), (14) is finally rewritten into

$$
\begin{aligned}
findbt(t,k) = \;& if \;\; t = NULL \\
& then \;\; error \\
& else \;\; if \;\; k = K(t) \\
& \qquad then \;\; D(t) \\
& \qquad else \;\; if \;\; k < K(t) \\
& \qquad\qquad then \;\; findbt(LT(t), k) \\
& \qquad\qquad else \;\; findbt(RT(t), k)
\end{aligned}
$$

or, making *tail-recursion* apparent,

$$
\begin{aligned}
findbt(t,k) \;\; = \;\;& if \;\; t = NULL \\
& then \;\; error \\
& else \;\; if \;\; k = K(t) \\
& \qquad then \;\; D(t) \\
& \qquad else \;\; findbt(\, if \; k < K(t) \; then \; LT(t) \; else \; RT(t), k)
\end{aligned} \tag{16}
$$

In summary, we have achieved (by transformations) a definition of $findbt$ which is directly convertible into a *while-loop*. A schematological result derived in the Appendix (Exercise 3) is directly applicable to (16) and converts this pattern of tail-recursion into a *while-loop* expressed by a *single-assignment* notation [14] *cf.* (21). This yields:

$$
\begin{aligned}
findbt(t,k) \;\; \stackrel{\text{def}}{=} \;\;& if \;\; t = NULL \\
& then \;\; error \\
& else \;\; while \;\; k \neq K(t) \\
& \qquad do \;\; \langle \;\; if \;\; t = NULL \qquad\qquad\qquad , k \rangle \\
& \qquad\qquad then \;\; error \\
& \qquad\qquad else \;\; if \;\; k < K(t) \\
& \qquad\qquad\qquad then \;\; new \;\; t = LT(t) \\
& \qquad\qquad\qquad else \;\; new \;\; t = RT(t)
\end{aligned} \tag{17}
$$

(17) is straightforwardly encoded in terms of imperative-language constructs. For example, the following excerpt of a Pascal program derived from (17):

```
TYPE Key = ....;
     Data = ...;
     BinTree = ^BinNode
     BinNode = RECORD
                 K: Key;
                 D: Data;
                 LT,RT: BinTree
               END;

VAR tree: BinTree;

PROCEDURE findbin (k: Key; VAR d: Data);
VAR p: BinTree; q: Key;
BEGIN p := tree;
      IF p = nil THEN terminate-error
                 ELSE q := p.K;
      WHILE q <> k DO
```

```
        BEGIN IF k < q THEN p := p^.LT
                       ELSE p := p^.RT;
              IF p = nil THEN terminate-error
                         ELSE q := p.K
        END;
      d := D(q)
  END (* findbin *);
```

is actually the realization proposed by Fielding [12] (with a slightly different choice of identifiers). According to the standard VDM procedure, this proposal would have to be proved correct by the rules of sequential, conditional and iterative composition [3].

5 Related Work and Concluding Remarks

The idea of generating a program by transformation of its functional specification is well-known from the literature on *program transformation* [8]. In the Project CIP [7] the emphasis has been put not only on the derivation of algorithms by means of transformations, but also on transforming data structures represented in terms of hierarchies of algebraic data types. Partsch [9] presents an example of how both data and control structures of a program can be jointly derived by concurrent transformations.

In this report we experiment with applying similar techniques to the development of programs using VDM. We show how the approach may contribute to alleviate VDM model-refinement and operation-refinement from too much "internal reasoning" preceeding "each change of the program" (quoting Partsch [16]). The algorithmic scheme underlying the definition chosen for the *retrieve-function* is shown to be an important design-decision wrt. the algorithmic struture of the low-level operations which are derived by transformations. It is also shown how algorithmically complex pre-conditions may be eliminated by (using transformations) intertwinning them with the definition of the relevant operations. Moreover, the exercises described in this report confirm the results in [16], in the sense that a small number of elementary transformations are sufficient to achieve a final implementation.

It should be noted that the results presented in this report have been transliterated from earlier exercises using the more terse notation of Backus' FP [15]. FP is perhaps less readable but much more concise wrt. "pen and paper" transformational exercises. This agrees with earlier experience on using FP for denoting and transforming dataflow programs [17].

References

[1] Guttag J.V. and Horning J.J. *The Algebraic Specification of Abstract Data Types*, in Acta Informatica, 10, 27-52, 1978.

[2] Goguen J.A., Thatcher J.W. and Wagner E.W. *Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*, in 'Current Trends on Programming Methodology', Vol. IV, (Yeh R.T. Ed.), Prentice-Hall Int., 1978.

[3] Jones C.B. *Software Development — A Rigorous Approach*, Prentice-Hall International, Series in Computer Science (Hoare C.A.R. -Ed.), New-Jersey, 1980.

[4] Abrial J.R. *Manuel du Language Z (Z/13)*, unpublished report, March 1977.

[5] Henderson P. *me too — a Language for Software Specification and Model Building — Preliminary Report*, Internal Report FPN-9, Department of Computer Science, University of Stirling (Scotland), December 1984.

[6] Goguen J.A. *Parameterized Programming*, Proc. of Workshop on *Reusability in Programming*, Perlis A. (Ed.), 1983.

[7] CIP-L Group *The Munich Project CIP*, Vol.1: *The Wide Spectrum Language 84*, T.U.Munchen, December 1983.

[8] Darlington J. *Program Transformation*, in *Functional Programming and Its Applications: An Advanced Course*, J. Darlington, P. Henderson, D.A. Turner (Eds.), 193-215, Cambridge, 1982.

[9] Partsch H. *An Exercise in the Transformational Derivation of an Efficient Program by Joint Development of Control and Data Structure*, Science of Computer Programming 3, 1-35, North-Holland 1983.

[10] Wirth N. *Algorithms + Data Structures = Programs*, Prentice-Hall Int., 1976.

[11] Bjorner D. and Jones C.B. *Formal Specification and Software Development*, Prentice-Hall International, Series in Computer Science, Hoare C.A.R. (Ed.), 1982.

[12] Fielding E. *The Specification of Abstract Mappings and Their Implementation as B+-trees*, Oxford PRG-18, September 1980.

[13] Welsh A. *The Specification, Design and Implementation of NDB*, M.Sc. Thesis, Dept. of Comp. Science, Univ. of Manchester, October 1982.

[14] Tesler L.G. and and Enea H.J. *A Language Design for Concurrent Processes*, AFIPS SJCC 32, 403-408, 1968.

[15] Backus J. *Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, CACM 21(8), 613-639, August 1978.

[16] Partsch H. *Structuring Transformational Developments: A Case Study Based on Earley's Recognizer*, Science of Computer Programming 4, 17-44, North-Holland 1984.

[17] Oliveira J.N. *The Formal Semantics of Deterministic Dataflow Programs*, Ph.D. thesis, Dept. of Comp. Science, Univ. Manchester, February 1984.

A Appendix

Exercise 1

Proof of (3) by structural induction on X-list:

- Case 1: $l = <>$

$$x \in elems(<>) = x \in \{\} = false = member(x, <>)$$

- Case 2: $l = <>$
  In this case

$$\begin{array}{rcl} member(x,l) & = & if\ \ x = hd(l) \\ & & then\ \ true \\ & & else\ \ member(x, tl(l)) \\ elems(l) & = & \{hd(l)\} \cup elems(tl(l)) \end{array}$$

Induction hypothesis: $x \in elems(tl(l)) = member(x, tl(l))$

– Case 2.1: $x = hd(l)$

$$\begin{aligned} member(x,l) &= true \\ x \in elems(l) &= hd(l) \in \{hd(l)\} \cup elems(tl(l)) \\ &= true \end{aligned}$$

– Case 2.2: $x \neq hd(l)$

$$\begin{aligned} member(x,l) &= member(x, tl(l)) \\ &= x \in elems(tl(l)) \quad \text{(Ind. hypothesis)} \\ &= x \in elems(l) \quad (\text{since } x \in \{hd(l)\}) \end{aligned}$$

□

Exercise 2

Derivation of (4) from the axiom (5), *cf.* the following diagram:

$$\begin{tikzcd}
X\text{-set} & \times & X\text{-set} \arrow[r, "\cap"] & X\text{-set} \\
X\text{-list} \arrow[u, "elems"] & \times & X\text{-list} \arrow[u, "elems"'] \arrow[r, "int"'] & X\text{-list} \arrow[u, "elems"']
\end{tikzcd}$$

$$\begin{aligned} elems(int(l_1, l_2)) &= elems(l_1) \cap elems(l_2) \\ &= (\ if\ l_1 = <>\ then\ \{\}\ else\ \{hd(l_1)\} \cup elems(tl(l_1))) \cap elems(l_2) \\ &= \ if\ \ l_1 = <> \\ &\quad\ then\ \ \{\} \\ &\quad\ else\ \ (hd(l_1) \cup elems(tl(l_1))) \cap elems(l_2) \\ &= \ if\ \ l_1 = <> \\ &\quad\ then\ \ \{\} \\ &\quad\ else\ \ \underbrace{\underbrace{\{hd(l_1)\} \cap elems(l_2)}_{A} \cup elems(int(tl(l_1), l_2))}_{B} \end{aligned}$$

where

$$\begin{aligned} A &= \ if\ \ hd(l_1) \in elems(l_2) \\ &\quad\ then\ \ \{hd(l_1)\} \\ &\quad\ else\ \ \{\} \\ &= \ if\ \ member(hd(l_1), l_2) \\ &\quad\ then\ \ \{hd(l_1)\} \\ &\quad\ else\ \ \{\} \end{aligned}$$

Substituting A in B:

$$\begin{aligned}
B &= (\,if\ member(hd(l_1), l_2)\ then\ \{hd(l_1)\}\ else\ \{\}) \cup elems(int(tl(l_1), l_2)) \\
&= \begin{array}[t]{l} if\ \ member(hd(l_1), l_2) \\ then\ \ \underbrace{\{hd(l_1)\} \cup elems(int(tl(l_1), l_2))}_{C} \\ else\ \ elems(int(tl(l_1), l_2)) \end{array}
\end{aligned}$$

where

$$C = elems(hd(l_1) \frown int(tl(l_1), l_2))$$

since

$$elems(l_1 \frown l_2) = elems(l_1) \cup elems(l_2)$$

Substituting C in B:

$$\begin{aligned}
B &= \begin{array}[t]{l} if\ \ member(hd(l_1), l_2) \\ then\ \ elems(<hd(l_1)> \frown int(tl(l_1), l_2)) \\ else\ \ elems(int(tl(l_1), l_2)) \end{array} \\
&= elems\ \begin{array}[t]{l} (if\ \ member(hd(l_1), l_2) \\ \quad then\ \ <hd(l_1)> \frown int(tl(l_1), l_2) \\ \quad else\ \ int(tl(l_1), l_2)) \end{array}
\end{aligned}$$

Substituting B above:

$$\begin{aligned}
elems(int(l_1, l_2)) &= \begin{array}[t]{l} if\ \ l_1 = <> \\ then\ \ \{\} \\ else\ \ elems\ \ (if\ \ member(hd(l_1), l_2) \\ \qquad\qquad then\ \ (<hd(l_1)> \frown (int(tl(l_1), l_2))) \\ \qquad\qquad else\ \ int(tl(l_1), l_2)) \end{array} \\
&= elems\ \begin{array}[t]{l} (if\ \ l_1 = <> \\ \quad then\ \ <> \\ \quad else\ \ \ if\ \ member(hd(l_1), l_2) \\ \qquad\qquad then\ \ (<hd(l_1)> \frown (int(tl(l_1), l_2))) \\ \qquad\qquad else\ \ int(tl(l_1), l_2)) \end{array}
\end{aligned}$$

from which we finally derive:

$$int(l_1, l_2) = \begin{array}[t]{l} if\ \ l_1 = <> \\ then\ \ <> \\ else\ \ \ if\ \ member(hd(l_1), l_2) \\ \qquad then\ \ <hd(l_1)> \frown (int(tl(l_1), l_2)) \\ \qquad else\ \ int(tl(l_1), l_2))) \end{array} \quad .$$

i.e. (4). Since $elems$ is not an injective operator, the solution we have derived is not unique. For example, a more sophisticated solution might have been derived which might filter repeated occurrences of the same element in l_1 [3].

□

[3](4) is curiously similar to a function $intersect$ proved correct by set-induction on p.145 of [3].

Exercise 3

The abstract scheme underlying definition (16) is

$$f(x) = \begin{array}{l} if\ \ p(x) \\ then\ \ e(x) \\ else\ \ \begin{array}{l} if\ \ q(x) \\ then\ \ d(x) \\ else\ \ f(r(x)) \end{array} \end{array} \tag{18}$$

where r is of the form

$$r(x) = \ if\ c(x)\ then\ a(x)\ else\ b(x)$$

and e is a "zero function" wrt. composition, *i.e.*

$$w(e(x)) = e(x) \tag{19}$$

holds. In the case of (16), e is an "error function" in the same class of functions as *e.g.* the least defined function $\overline{\bot}$ [15], which is such that, for all x, $\overline{\bot}(x) = \bot$, where $\bot$ denotes "undefined". Clearly,

$$w(\overline{\bot}(x)) = w(\bot) = \bot = \overline{\bot}(x)$$

cf. (19). (18) can thus be rewritten into:

$$\begin{array}{rcl} f(x) & = & if\ p(x)\ then\ e(x)\ else\ w(x) \\ w(x) & = & if\ q(x)\ then\ d(x)\ else\ f(r(x))) \end{array} \tag{20}$$

Unfolding f in (20),

$$w(x) = \begin{array}{l} if\ \ q(x) \\ then\ \ d(x) \\ else\ \ (\lambda x.\ if\ p(x)\ then\ e(x)\ else\ w(x))(r(x)) \end{array}$$

Using (19), this may be rewritten into:

$$w(x) = \begin{array}{l} if\ \ q(x) \\ then\ \ d(x) \\ else\ \ w((\lambda x.\ if\ p(x)\ then\ e(x)\ else\ x)(r(x)) \end{array}$$

i.e. we have achieved a purely tail-recursive definition of w which is directly convertible into a *while-loop*. Using a *single-assignment* [14] description for *while-loops*, w is rewritten into:

$$\begin{array}{ll} w(x) = d & (while\ \ \neg q(x) \\ & \quad do\ \ new\ \ x = (\lambda x.\ if\ p(x)\ then\ e(x)\ else\ x)(r(x))) \end{array}$$

Therefore, (18) is rewritten non-recursively simply as:

$$f(x) = \begin{array}{l} if\ \ p(x) \\ then\ \ e(x) \\ else\ \ d\ \ (while\ \ \neg q(x) \\ \qquad\quad do\ \ new\ \ x = (\lambda x.\ if\ p(x)\ then\ e(x)\ else\ x)(r(x))) \end{array} \tag{21}$$

□