

# Laws of Data Refinement

**Joseph M. Morris**

Department of Computing Science

University of Glasgow

Glasgow G12 8QQ, UK



Laws of Data Refinement, Joseph M. Morris, Acta Informatica 26, 287-308 (1989). © Springer-Verlag 1989

**Summary.** A specification language typically contains sophisticated data types that are expensive or even impossible to implement. Their replacement with simpler or more efficiently implementable types during the programming process is called data refinement. We give a new formal definition of data refinement and use it to derive some basic laws. The derived laws are constructive in that used in conjunction with the known laws of procedural refinement they allow us to calculate a new specification from a given one in which variables are to be replaced by other variables of a different type.

## 1. Introduction

We take the view that a specification language is a programming language with added fancy constructs and notions that admit ease of expression but may be expensive or even impossible to implement. The programming task is to make a specification and then step by step eliminate the fancy constructs with correctness-preserving transformations, or refinements, until, all going well, we arrive at a program. Among the added notions of specification languages are a richer set of data types – sets, bags, mappings, unbounded sequences, etc. – than we are used to seeing in programming languages; their replacement with simpler types during program development is called "data refinement". To be useful, data refinement must proceed piecewise. That is to say, we want to refine the constituent pieces of a specification more or less independently of one another so that the refinement of the whole specification is not much more than the composition of its refined constituents. We give a new formal definition of data refinement and use it to derive some basic laws. The derived laws are constructive in that they allow us to calculate a new specification from a given one in which variables are to be replaced by other variables of a different type.

A great deal of work has been done in recent years on the making of specifications [4, 7], but the business of formally deriving programs from specifications using some "programming calculus" is less well developed. We are here contributing to this goal. Our underlying philosophy is that no semantic distinction is to be made between specifications and programs. Programs *are* specifications – albeit somewhat special in that we know to extract automatically a computation from them. The programming language is the implementable subset of the specification language. We view programming as constructing a sequence of specifications, the first one being supplied by the customer and thereafter each one being a "more algorithmic" derivative of its predecessor. The final members of the sequence are programs; we don't necessarily finish when a program first emerges for we may wish to continue development to satisfy efficiency obligations. A specification notation is richer than its constituent programming language in two regards: it has more expressive statements and richer data types. Procedural refinement is the business of

reducing the expressive statements to executable ones; data refinement is concerned with replacing data types with simpler or more efficiently implementable types. It is obviously highly desirable that both these refinement techniques should coexist harmoniously within the one developmental framework. The laws of procedural refinement have been described in outline in [1, 8, 10]; here we are incorporating data refinement into the theory. The relationship between the present and earlier work is considered later

## 2. Specifications and Refinement

We take as the programming language that of guarded commands [2] augmented by blocks of the form

$$[[ \text{var1: T1; var2: T2; ... statement} ]]$$

The specification language is the programming language plus four extensions. Firstly, of course, we admit a richer set of data types — sets, bags, etc. — and a richer set of operations on data; in particular we allow guards to employ quantifiers. Secondly, we admit the nondeterministic assignment statement. This has the form  $x:-P$  for  $x$  any variable(s) and  $P$  an assertion; it asks for the establishment of  $P$  without changing the values of variables other than  $x$ . Thirdly, we regard assertions in chain brackets not as comments but as formal statements; we call these "assert statements". Fourthly we allow single guarded commands as statements:  $(P \rightarrow s)$  behaves like  $s$  when  $P$  holds and otherwise it behaves "miraculously" by which we mean that it achieves whatever goal we have in mind, and hence we call it a "minor miracle". It plays a somewhat special role in that it is not so much used to make specifications as to play a mediating role in refining them; we will describe it in more detail later.

We let capital letters stand for assertions, and the small letters  $p, q, \dots$  stand for specifications. Type names are  $T_x, T_y$ , etc.  $T$  and  $F$  are the assertions satisfied by all states and by no state, respectively.  $[X]$  stands for assertion  $X$  universally quantified over its free specification variables.  $X(x \setminus e)$  stands for  $X$  with each free occurrence of variable  $x$  replaced by expression  $e$ ; the same notation is used for substitution in expressions. By such phrases as " $x$  occurs in  $P$ " and " $P$  contains  $x$ " we mean that  $x$  occurs free in  $P$ . An infix period is used for function application. We do not distinguish notationally between specifications as syntactic and semantic objects, preferring to write the more compact  $s.X$  instead of the familiar  $\text{wp}(s, X)$  to denote the weakest precondition for  $s$  to establish  $X$ . The predicate transformer semantics of specifications is given in Figure 1, and explanatory

$s.X =$

- $s$  is **skip** :  $X$
- $s$  is  $x := e$  :  $X(x \setminus e)$
- $s$  is  $x:-P$  :  $(\mathbf{A}x: P \Rightarrow X)$
- $s$  is  $\{P\}$  :  $P \wedge X$
- $s$  is  $p;q$  :  $p.(q.X)$
- $s$  is  $[[ x:T_x; y:T_y; \dots t ]]$  :  $(\mathbf{A}x, y(x \in T_x, y \in T_y, \dots): t.X)$
- $s$  is  $(P \rightarrow t)$  :  $P \Rightarrow t.X$
- $s$  is **if**  $([ P_i \rightarrow t_i ])$  **fi** :  $(\mathbf{E}i: P_i) \wedge (\mathbf{A}i: P_i \Rightarrow t_i.X)$
- $s$  is **do**  $P \rightarrow t$  **od** :  $(\mu Y: (P \wedge t.Y) \vee (\neg P \wedge X))$

**Figure 1:** semantics of specifications

comments follow. **A** and **E** denote universal and existential quantification, respectively. We have made some simplifying assumptions in the semantic rules, as follows. In applying the rule for blocks it is assumed that the names of variables declared in the block are systematically changed if otherwise they would clash with names occurring in the postcondition. Types are assumed to be non-empty. We also don't bother to specify the number of guarded commands in an if-statement, writing  $\mathbf{if} (\ [] P_i \rightarrow t_i ) \mathbf{fi}$  rather than  $\mathbf{if} P_0 \rightarrow t_0 \ [] P_1 \rightarrow t_1 \ [] \dots \ [] P_{n-1} \rightarrow t_{n-1} \mathbf{fi}$ , and  $(\mathbf{E}i:P_i)$  instead of  $(\mathbf{E}i(0 \leq i < n):P_i)$  where  $n$  denotes the number of guarded commands. Neither have we specified in the rules that expressions must be well-defined in the context of their use. We make these simplifications for the sake of shortening formulae a little; the theory we will develop does not depend on them. Following convention we may omit the connecting semicolon before or after  $\{P\}$ .  $(\mu Y:f.Y)$  denotes the least fixpoint of function  $f$ , and will be elaborated on later. Assert statements and minor miracles are further explained below. We add that our notation is not intended to be a general specification language, but rather one suited to the purpose at hand.

The operators used, grouped in order of decreasing precedence, are  $;$ ,  $\neg$ ;  $\wedge$ ,  $\vee$ ;  $\Rightarrow$ ;  $=$ .

Specifications are monotonic in the following sense:

$$[X \Rightarrow Y] \Rightarrow [s.X \Rightarrow s.Y]$$

for all  $s$ ,  $X$ ,  $Y$ ; the proof, which we omit, is by structural induction on  $s$ .

We briefly review procedural refinement. Programming consists in making a sequence of specifications starting from an initially given specification. At each step we make the current specification  $s$  "more algorithmic" – for example by replacing a nondeterministic assignment by a sequence of simpler assignments – producing a new specification  $t$  that will be the subject of the next step. We say that specification  $t$  is a "refinement" of specification  $s$  and denote the relationship by  $s \leq t$ , defined

$$s \leq t = (\mathbf{A}X: [s.X \Rightarrow t.X])$$

Procedural refinement is the process of eliminating, in such small steps, the unimplemented operators, minor miracles, and nondeterministic assignment statements. Stepwise procedural refinement is possible because of two properties of  $\leq$ . The first is that of monotonic replacement:

$$u \leq v \Rightarrow s(w \setminus u) \leq s(w \setminus v)$$

for all specifications  $u, v, s$  where  $s(w \setminus u)$  denotes  $s$  with each occurrence of component  $w$  in  $s$  replaced by  $u$ , and similarly for  $s(w \setminus v)$ . Monotonic replacement permits refinement of a specification by refinement of one of its components. Secondly,  $\leq$  is transitive. Together with monotonic replacement the transitivity of  $\leq$  allows us to refine a specification in small steps such that the derived specification is always a refinement of the original. See [1, 8, 10] for a discussion and proofs of these properties, and for the laws of stepwise procedural refinement in general.

The primary role of the assert statement is to supply the context for a refinement step:

$\{G\}s \leq t$  amounts to saying that  $t$  may replace  $s$  in a context in which  $G$  holds but not necessarily otherwise; such a  $G$  is called the context of  $s$  and by extension the context of the refinement. Refinements always take place in some context which may just be the context  $T$  but will usually be something stronger. The stronger the context the easier is the refinement — in the extreme  $\{F\}s \leq t$  for all  $t$ . Note that an implementation can ignore assert statements because, as is easily seen,  $\{X\} \leq \mathbf{skip}$  for any assertion  $X$ . The construct  $\{G\}x:-P$  — called a "prescription" — is a particularly important specification device being the mechanism through which many small components are specified. It says: given an initial state satisfying  $G$  establish the postcondition  $P$  while changing at most the value of  $x$ . Its semantics, derived from Figure 1, is:

$$(\{G\}x:-P).X = G \wedge (\mathbf{Ax}: P \Rightarrow X).$$

The statement  $x:-F$  is called **miracle** and satisfies  $[\mathbf{miracle}.X = T]$  for all  $X$ ; it is so called because  $s \leq \mathbf{miracle}$  for all  $s$ . Because  $\mathbf{miracle} \leq s$  iff  $s$  is **miracle** we can never derive a program that is a refinement of **miracle**. Nevertheless, miracles play a useful simplifying role in the theory of programming as we shall see. Of course a statement may be miraculous over just part of its domain; for example  $x:- (x \in \text{nat} \wedge x \leq n)$  (where the values of type  $\text{nat}$  are the natural numbers) is miraculous just for those values of  $n$  satisfying  $n < 0$ . The predicate characterising the miraculous domain of statement  $s$  is  $s.F$ .

The minor miracle  $(P \rightarrow s)$ , which is equivalent to  $\mathbf{if} P \rightarrow s \ [\ ] \neg P \rightarrow \mathbf{miracle} \mathbf{fi}$ , can play a useful intermediary role in programming. For example we may use it to strengthen a context, for it is easy to show that  $s \leq (P \rightarrow \{P\}s)$ , for all  $P$  and  $s$ , and we then proceed to refine  $\{P\}s$ . Ultimately we have to get rid of any minor miracles and there are various techniques for doing so. We shall only need the law, which is easily proved, that  $\{G\}(P \rightarrow s) \leq s$  if  $[G \Rightarrow P]$ . Obviously the weaker the guard  $P$  the more likely it is that we can replace  $(P \rightarrow s)$  with  $s$ . One may easily show that  $s \leq (P \rightarrow t)$  is equivalent to  $\{P\}s \leq t$ . It follows that minor miracles are a device for associating context with the refined specification rather than with the source specification. This will be useful in data refinement.

### 3. Data Refinement

In the context set out above data refinement is a special instance of refining a block. More specifically, given  $[[ b:Tb; s ]]$  we are to produce  $[[ v:Tv; t ]]$  such that  $[[ b:Tb; s ]]$   $\leq$   $[[ v:Tv; t ]]$ . In applying the theory of data refinement it is more convenient to make the initialization of  $b$  explicit; denoting the initialization by  $b:-Q$ , the problem is then that of constructing  $[[ v:Tv; t ]]$  satisfying  $[[ b:Tb; b:-Q; s ]]$   $\leq$   $[[ v:Tv; t ]]$ . In essence, we are to replace variable  $b$  and operations on it with variable  $v$  and corresponding operations on  $v$ . The given block is called the "abstract specification" and the derived block is called the "concrete specification". By extension we talk of "abstract statements" and "concrete statements", and so on.  $b$  is the "abstract variable" and  $v$  the "concrete variable". In general there may be more than one abstract and more than one concrete variable but we need not make this explicit just yet. Variables other than  $b$  and  $v$  are called the "common variables".

We intend to construct the concrete specification so that it has the same structure as the abstract specification and so that each primitive concrete statement is the translation of the corresponding abstract statement, according to a uniform rule. Central to this strategy is the introduction of a relationship between the concrete and abstract variables via a so-called "abstraction invariant" [5]. Throughout,  $I$  will stand for

the abstraction invariant,  $b$  (of type  $T_b$ ) for the abstract variable, and  $v$  (of type  $T_v$ ) for the concrete variable. We require that  $I$  does not contain common variables that appear on the left hand side of an assignment or nondeterministic assignment statement in the abstract specification; usually only  $b$  and  $v$  will occur in  $I$ , however. In choosing concrete variables and an abstraction invariant we must take into account how the abstract variable has been used; two abstract variables of the same type in different programs may require different concrete types and/or different abstraction invariants for a successful translation to concrete form in each case. There are no formal requirements imposed on the abstraction invariant except that it does not contain common variables that are assigned to, but one must be sensible in one's choice. It will turn out that for an inappropriate choice of  $I$  — say if we take  $F$  for  $I$  — the concrete specification can never be refined to a program because it is miraculous over some part of its domain.

We denote "statement  $s$  on abstract variable  $b$  is data refined by statement  $t$  on concrete variable  $v$  under the abstraction invariant  $I$ " by  $s \ll_{I,b,v} t$ ; the subscript  $I,b,v$  is fixed for our presentation and so we omit it hereafter. We propose the definition

$$s \ll t = [(\mathbf{E}b: I \wedge s.X) \Rightarrow t.(\mathbf{E}b: I \wedge X)] \text{ for all } X \text{ not containing } v.$$

To be strict the preceding definition should include "and  $s$  does not refer to  $v$  nor  $t$  to  $b$ ", but we take this as understood. This definition is chosen for two reasons. Firstly, it is effective in that if we construct a  $t$  satisfying  $s \ll t$  then we have essentially constructed a refinement of

$\ll [b:T_b; b:-Q; s \ll] \ll$ ; see Theorem 1. Secondly we can use it to formulate a body of laws that allows us to construct formally a  $t$  satisfying  $s \ll t$  for any  $s$ . Note that if  $x$  does not occur in  $s$  or  $X$  then it does not occur in  $s.X$ , for all  $s, X$ ; we will be appealing to this in proofs. We prefer to apply the epithet "piecewise", rather than "stepwise", to data refinement because the steps in data refinement proceed more in parallel than in series; formally,  $\ll$  does not enjoy the transitivity and monotonic replacement properties of  $\leq$ .

If in refining  $\ll [b:T_b; b:-Q; s \ll] \ll$  with concrete variable  $v$  we construct  $t$  satisfying  $s \ll t$  then, apart from a requirement to initialise  $v$ , we will have solved the original problem:

**Theorem 1.** *If  $s \ll t$  then  $\ll [b:T_b; b:-Q; s \ll] \ll \leq \ll [v:T_v; v:-(\mathbf{E}b:I \wedge Q); t \ll]$*

*Proof.* (See the note on proofs below.)

$$\ll [b:T_b; b:-Q; s \ll] \ll \leq \ll [v:T_v; v:-(\mathbf{E}b:I \wedge Q); t \ll]$$

" $X$  arbitrary"

$$\ll [\ll [b:T_b; b:-Q; s \ll].X \Rightarrow \ll [v:T_v; v:-(\mathbf{E}b:I \wedge Q); t \ll].X] \quad (1)$$

We identify two cases:

(i)  $X$  does not contain  $b, v$ :

$$(1)$$

"semantics"

$$[(\mathbf{A}b(b \in T_b): (\mathbf{A}b: Q \Rightarrow s.X)) \Rightarrow (\mathbf{A}v(v \in T_v): (\mathbf{A}v: (\mathbf{E}b: I \wedge Q) \Rightarrow t.X))]$$

"calculus; types not empty"

$$[(\mathbf{A}b: Q \Rightarrow s.X) \Rightarrow (\mathbf{A}v: (\mathbf{E}b: I \wedge Q) \Rightarrow t.X)]$$

"absorption –  $v$  not in antecedent"

$$[(\mathbf{A}b: Q \Rightarrow s.X) \Rightarrow ((\mathbf{E}b: I \wedge Q) \Rightarrow t.X)]$$

"importation"

$$[(\mathbf{A}b: Q \Rightarrow s.X) \wedge (\mathbf{E}b: I \wedge Q) \Rightarrow t.X]$$

"calculus"

$[(\mathbf{E}b:I^Q \wedge s.X) \Rightarrow t.X]$   
 $\Leftarrow$  "calculus"  
 $[(\mathbf{E}b:I \wedge s.X) \Rightarrow t.X]$   
 $\Leftarrow$  "t monotonic"  
 $[(\mathbf{E}b:I \wedge s.X) \Rightarrow t.((\mathbf{E}b:I) \wedge X)]$   
 $\Leftarrow$  "X does not contain b"  
 $[(\mathbf{E}b:I \wedge s.X) \Rightarrow t.(\mathbf{E}b:I \wedge X)]$   
 $\Leftarrow$  "X does not contain v"  
 $s \ll t$   
 $=$  "given"  
 true

(ii) X contains b or v:

We use the following substitution rule:  $p.X = (p.(X(x \setminus y)))(y \setminus x)$  for  $p$  any statement,  $x$  any specification variable not occurring in  $p$  (except perhaps locally declared in  $p$ ), and  $y$  a fresh variable. More generally  $x$  and  $y$  may stand for lists of such variables. We take this rule as evident, just noting that it may be proved by structural induction on  $p$ . The rule allows us to rename  $b$  and  $v$  in  $X$  with fresh variables, and so the truth of (1) follows by almost the same argument as in (i); the details are routine and are left to the reader. (End)

*Note on proofs.* Hints for the justification of proof steps are written within double quotes. Proofs usually proceed by reducing the demonstrandum to 'true', some steps using  $\Leftarrow$ , pronounced 'follows from':  $A \Leftarrow B$  is the same as  $B \Rightarrow A$ . These steps often appeal to the rule

$$[A \Rightarrow D] \Leftarrow [A \Rightarrow B] \wedge [B \Rightarrow D]$$

For example,

$[A \Rightarrow D]$   
 $\Leftarrow$  "hint why  $[A \Rightarrow B]$ "  
 $[B \Rightarrow D]$   
 $\Leftarrow$  "hint why  $[C \Rightarrow D]$ "  
 $[B \Rightarrow C]$   
 $=$  "hint why  $[B \Rightarrow C]$ "  
 true.

Some rules are specifically named in proofs, viz.

importation/exportation:  $[(A \wedge B \Rightarrow C) = (A \Rightarrow (B \Rightarrow C))]$

absorption: for  $x$  a specification variable, and  $Q$  not containing  $x$

$$\begin{aligned}
 [P] &= [\mathbf{A}x:P], \\
 [P \Rightarrow Q] &= [(\mathbf{E}x:P) \Rightarrow Q] \\
 [Q \Rightarrow P] &= [Q \Rightarrow (\mathbf{A}x:P)]
 \end{aligned}$$

1-point

$$\begin{aligned}
 [(\mathbf{E}x: x = y \wedge P) = P(x \setminus y)] \\
 [(\mathbf{A}x: x = y \Rightarrow P) = P(x \setminus y)].
 \end{aligned}$$

We frequently use  $[(\mathbf{E}x:P) \wedge (\mathbf{A}x:Q) \Rightarrow (\mathbf{E}x:P \wedge Q)].$

The hint "calculus" records an appeal to rules of predicate calculus that are presumed to be familiar; "substitution" hints at the elementary laws of substitution of variables; and "semantics" hints at the predicate transformer semantics of specifications. (End)

Theorem 1 is the first formal indicator that the choice of  $I$  is important. If we choose  $F$  for  $I$ , for example, then in the subsequent procedural refinement of the concrete specification we will be faced with the miraculous and therefore unimplementable  $v:-F$ .

The relations  $\ll$  and  $\leq$  share in a kind of transitivity, viz.

$$\begin{aligned} s \ll t \wedge t \leq u &\Rightarrow s \ll u && \text{and} \\ s \leq t \wedge t \ll u &\Rightarrow s \ll u \end{aligned}$$

The proofs are easy and are left to the reader.

## 4. Data Refinement of Compositions

We proceed to show that the structure of the abstract specification carries over to the concrete specification. To begin with, data refinement distributes over sequential composition.

**Theorem 2.**  $p; q \ll s; t$  follows from  $p \ll s$  and  $q \ll t$ .

*Proof.* We are given, for any  $X$  not containing  $v$

$$[(\mathbf{E}b: I \wedge p.X) \Rightarrow s.(\mathbf{E}b: I \wedge X)] \quad (\text{i})$$

$$[(\mathbf{E}b: I \wedge q.X) \Rightarrow t.(\mathbf{E}b: I \wedge X)] \quad (\text{ii})$$

The proof is

$$p; q \ll s; t$$

" $\ll$ ;  $X$  arbitrary not containing  $v$ ; semantics"

$$[(\mathbf{E}b: I \wedge p.(q.X)) \Rightarrow s.(t.(\mathbf{E}b: I \wedge X))]$$

$\Leftarrow$ "(ii); monotonicity of  $s$ "

$$[(\mathbf{E}b: I \wedge p.(q.X)) \Rightarrow s.(\mathbf{E}b: I \wedge q.X)]$$

"(i) with  $X := q.X$ ;  $q.X$  does not contain  $v$ "

true.

(End)

Local blocks may be carried over to the concrete specification:

**Theorem 3.**  $\llbracket k:Tk; s \rrbracket \ll \llbracket k:Tk; t \rrbracket$  follows from  $s \ll t$ .

*Proof.* Exercise.

(End)

In translating guarded commands it turns out that a good translation of abstract guard  $P$  to concrete guard  $Q$  is to take  $(\mathbf{A}b: I \Rightarrow P)$  for  $Q$ . The resulting translations gives rise to minor miracles (which ultimately must be removed, if possible, by procedural refinement).

**Theorem 4.**  $\text{if } (\llbracket P_i \rightarrow s_i \rrbracket \mathbf{fi}) \ll ((\mathbf{E}i: Q_i) \rightarrow \text{if } (\llbracket Q_i \rightarrow t_i \rrbracket \mathbf{fi}))$  follows from  $s_i \ll t_i$  for each  $i$ , where  $Q_i$  stands for  $(\mathbf{A}b: I \Rightarrow P_i)$  for each  $i$ .

*Proof.*

$$\text{if } (\llbracket P_i \rightarrow s_i \rrbracket \mathbf{fi}) \ll ((\mathbf{E}i: Q_i) \rightarrow \text{if } (\llbracket Q_i \rightarrow t_i \rrbracket \mathbf{fi}))$$

="«; X arbitrary not containing v; semantics"  

$$[(\mathbf{E}b:I \wedge (\mathbf{E}i:P_i) \wedge (\mathbf{A}i:P_i \Rightarrow s_i.X)) \Rightarrow ((\mathbf{E}i:Q_i) \Rightarrow (\mathbf{E}i:Q_i) \wedge (\mathbf{A}i:Q_i \Rightarrow t_i.(\mathbf{E}b:I \wedge X)))]$$
 <"calculus"  

$$[(\mathbf{E}b:I \wedge (\mathbf{A}i:P_i \Rightarrow s_i.X)) \Rightarrow (\mathbf{A}i:Q_i \Rightarrow t_i.(\mathbf{E}b:I \wedge X))]$$
 <"calculus"  

$$[(\mathbf{A}i:(\mathbf{E}b:I \wedge (P_i \Rightarrow s_i.X))) \Rightarrow (\mathbf{A}i:Q_i \Rightarrow t_i.(\mathbf{E}b:I \wedge X))]$$
 <"calculus; generalisation over i"  

$$[(\mathbf{E}b:I \wedge (P_i \Rightarrow s_i.X)) \Rightarrow (Q_i \Rightarrow t_i.(\mathbf{E}b:I \wedge X))]$$
 ="importation;  $Q_i$ "  

$$[(\mathbf{E}b:I \wedge (P_i \Rightarrow s_i.X)) \wedge (\mathbf{A}b:I \Rightarrow P_i) \Rightarrow t_i.(\mathbf{E}b:I \wedge X)]$$
 <"calculus"  

$$[(\mathbf{E}b:I \wedge (P_i \Rightarrow s_i.X) \wedge (I \Rightarrow P_i)) \Rightarrow t_i.(\mathbf{E}b:I \wedge X)]$$
 <"calculus"  

$$[(\mathbf{E}b:I \wedge s_i.X) \Rightarrow t_i.(\mathbf{E}b:I \wedge X)]$$
 <"«; X does not contain v"  

$$s_i \ll t_i$$
 ="given"  
 true (End)

We can exploit the context of the abstract **if...fi** to construct weaker guards for the concrete **if...fi**, and so reduce the domain of miraculous behaviour, as follows. By the laws of procedural refinement we know that  $\{G\}\text{if} (\prod P_i \rightarrow s_i) \text{fi}$  is equivalent to  $\{G\}\text{if} (\prod (G \Rightarrow P_i) \rightarrow s_i) \text{fi}$ , and then the corresponding concrete guards  $Q_i$  are by Theorem 4  $(\mathbf{A}b:I \Rightarrow (G \Rightarrow P_i))$ , or equivalently  $(\mathbf{A}b: I \wedge G \Rightarrow P_i)$ . Moreover, because **if**  $(\prod P_i \rightarrow s_i)$  **fi** is equivalent to  $\{(\mathbf{E}i:P_i)\}\text{if} (\prod P_i \rightarrow s_i) \text{fi}$  we can in fact take  $(\mathbf{A}b: I \wedge (\mathbf{E}i:P_i) \Rightarrow P_i)$  as the concrete guard corresponding to the abstract  $P_i$  without further knowledge of the context.

We give an example of applying Theorem 4 with an inappropriate abstraction invariant. Let the abstract variable  $b$  and the concrete variable  $v$  be of type  $\text{nat}$ , and let the abstraction invariant be  $b \text{ div } 2 = v$ . Suppose we wish to translate **if**  $b = 0 \rightarrow s \ [] \ b \neq 0 \rightarrow t$  **fi** to concrete form. The translation of the first guard is  $(\mathbf{A}b: (b \text{ div } 2 = v) \Rightarrow (b = 0))$ , which can easily be shown to be false. The translation of the second guard is  $(\mathbf{A}b: (b \text{ div } 2 = v) \Rightarrow (b \neq 0))$  which reduces to  $v \neq 0$ . Because we may omit from an **if...fi** a guarded command whose guard is universally false, Theorem 4 yields the concrete  $(v \neq 0 \rightarrow \text{if } v \neq 0 \rightarrow t')$  **fi** where  $t \ll t'$ , which is equivalent to  $(v \neq 0 \rightarrow t')$ . This is only of use in the context  $v \neq 0$  for which we would need the abstract context  $b > 1$ , and presumably we don't have this. Essentially, the chosen abstraction invariant carries too little information for the translation to yield a program.

**Theorem 5.**  $\{G\}\text{do } P \rightarrow s\{G\} \text{od} \ll \text{do } Q \rightarrow t \text{od}$  follows from  $s \ll t$  and  $[(\mathbf{E}b: I \wedge G) \Rightarrow Q \vee Q^~]$ , where  $Q = (\mathbf{A}b: I \wedge G \Rightarrow P)$  and  $Q^~ = (\mathbf{A}b: I \wedge G \Rightarrow \neg P)$ . (See comment below.)

*Proof.* Similar to that of Theorem 14 below. (End)

To see how the term  $Q \vee Q^~$  arises in the preceding theorem observe that

$$\{G\}\text{do } P \rightarrow s\{G\} \text{od}$$

is equivalent to DO where DO is the recursive routine

DO:  $\{G\}\text{if } P \rightarrow s; \text{DO } [] \neg P \rightarrow \text{skip fi.}$

With Theorems 2, 4 and 7 (below) in mind, Theorem 5 can be viewed as a special case of a more general theorem that says that data refinement distributes through recursion.

To summarise this section: the structure of the abstract specification carries over to the concrete specification, but for some choices of the abstraction invariant the concrete specification may be miraculous without the possibility of further refinement to a program. Data refinement is now reduced to refining the primitive abstract statements.

## 5. Data Refinement of Primitives

First we show that the data refinement of statements that do not refer to abstract variables is immediate.

**Theorem 6.** *If  $s$  refers to no abstract variables then  $s \ll s$ .*

*Proof.* By a routine structural induction left to the reader. (End)

**Theorem 7.**  $\{P\} \ll \{\text{Eb: } I \wedge P\}$

*Proof.* Exercise. (End)

We shall need some notation and results from the theory of procedural refinement. For any expression  $e$  and statement  $s$  we define  $s.\langle e \rangle = (\mathbf{Ai}(i=e): s.T \Rightarrow s.(i=e))$ ; roughly,  $s.\langle e \rangle$  is a predicate characterizing the initial states such that terminating executions of  $s$  leave the value of  $e$  unchanged from its initial value. Note that if  $s$  contains no assignments to free variables in  $e$  then  $[s.T \Rightarrow s.\langle e \rangle]$ ; this can be shown by an easy structural induction. We state the following results without proof.

**Lemma 1.** *Letting  $y$  stand for a list containing the program variables in assertion  $P$ ,*

*$[t.Q \wedge t.\langle y \rangle \wedge P \Rightarrow t.R]$  follows from  $[Q \wedge P \Rightarrow R]$  for all  $P, Q, R, t$ . (End)*

**Lemma 2.**  $\{G\}x:-Q \leq s = [G \Rightarrow s.Q \wedge s.\langle y \rangle]$  where  $y$  stands for a list of the program variables other than  $x$ . (End)

The main theorem for the translation of prescriptions follows:

**Theorem 8.** *Let  $c$  stand for some (or all) abstract variables and  $m$  stand for some (or all) common variables. Then for all  $t$  that do not refer to abstract variables*

*$\{G\}c,m:-Q \ll t$  follows from  $\{I \wedge G\}v,m:-(\mathbf{Ec: } I \wedge Q) \leq t$*

(See comment following proof.)

*Proof.* Let  $d$  be a list of the abstract variables other than  $c$  — in other words we can take it that  $b$  is  $c,d$ .

Let  $n$  be a list of the common variables other than  $m$ . By Lemma 2

$$\begin{aligned} & \{I \wedge G\}v,m:-(\mathbf{Ec: } I \wedge Q) \leq t \\ = & [I \wedge G \Rightarrow t.(\mathbf{Ec: } I \wedge Q) \wedge t.\langle b,n \rangle] \end{aligned} \quad (i)$$

The proof of the theorem is now:

$$\{G\}c,m:-Q \ll t$$

"«; X arbitrary not containing v; semantics"

$$[(\mathbf{E}c,d: I \wedge G \wedge (\mathbf{A}c,m: Q \Rightarrow X)) \Rightarrow t.(\mathbf{E}c,d: I \wedge X)]$$

"let Z:= (E c,d: I ^ X) "

$$[(\mathbf{E}c,d: I \wedge G \wedge (\mathbf{A}c,m: Q \Rightarrow X)) \Rightarrow t.Z]$$

"absorption – consequent does not contain c, d"

$$[I \wedge G \wedge (\mathbf{A}c,m: Q \Rightarrow X) \Rightarrow t.Z]$$

«(i)"

$$[t.(\mathbf{E}c: I \wedge Q) \wedge t.\langle b,n \rangle \wedge (\mathbf{A}c,m: Q \Rightarrow X) \Rightarrow t.Z]$$

«"Lemma 1 with Q:= (E c: I ^ Q); y:= b,n; P:= (A c,m: Q => X); R:= Z"

$$[(\mathbf{E}c: I \wedge Q) \wedge (\mathbf{A}c,m: Q \Rightarrow X) \Rightarrow Z]$$

«"calculus"

$$[(\mathbf{E}c: I \wedge Q \wedge (\mathbf{A}m: Q \Rightarrow X)) \Rightarrow Z]$$

«"calculus"

$$[(\mathbf{E}c: I \wedge Q \wedge (Q \Rightarrow X)) \Rightarrow Z]$$

"calculus"

$$[(\mathbf{E}c: I \wedge Q \wedge X) \Rightarrow Z]$$

"Z; calculus"

true

(End)

The above theorem essentially says that to data refine  $\{G\}c,m:-Q$  we need only procedurally refine  $\{I \wedge G\}v,m:-(\mathbf{E}c: I \wedge Q)$ . Although abstract variables appear in this concrete specification they act as "dummy" variables — their role is similar to that of  $x_0$  in  $\{x = x_0\}x:-(x > x_0)$  where  $x_0$  just denotes the initial value of  $x$ . In subsequent (procedural) refinements the dummy abstract variables must be refined out.

If  $m:-Q$ , where  $m$  is a common variable, is such that  $Q$  does not contain abstract variables then by Theorem 6 it carries over without change to the concrete specification; otherwise we can apply either of the following two theorems.

**Theorem 9.** *Let  $m$  stand for some (or all) common variables. Then for all  $t$  containing no abstract variables  $\{G\}m:-Q \ll t$  follows from  $\{I \wedge G\}m:-(I \wedge Q) \leq t$*

*Proof.* Let  $c$  be empty in Theorem 8.

(End)

**Theorem 10.** *Letting  $m$  stand for some (or all) common variables*

*$\{G\}m:-Q \ll \{\mathbf{E}b: I \wedge G\}m:-(\mathbf{A}b: I \Rightarrow Q)$ .*

*Proof.* Exercise.

(End)

For simple assignments to abstract variables we have

**Theorem 11.** *For all  $t$  containing no references to abstract variables, and any expression  $e$*

*$\{G\}c:=e \ll t$  follows from  $\{I \wedge G\}v:-I(c \setminus e) \leq t$*

*Proof.* Let  $Y'$  stand for  $Y(c \setminus e)$  for all  $Y$ . Let  $l$  stand for a list of all the common variables. Observe that by Lemma 2

$$\{I \wedge G\}v:-I(c \setminus e) \leq t$$

$$= [I \wedge G \Rightarrow t.I \wedge t.\langle b,l \rangle] \quad (i)$$

The proof of the theorem is then

$$\{G\}c:=e \ll t$$

="«; X arbitrary not containing v; semantics"  
 $[(\mathbf{E}b: I \wedge G \wedge X') \Rightarrow t.(\mathbf{E}b: I \wedge X)]$   
 ="Z:= (E**b**: I ^ X)"  
 $[(\mathbf{E}b: I \wedge G \wedge X') \Rightarrow t.Z]$   
 ="absorption - consequent does not contain b"  
 $[I \wedge G \wedge X' \Rightarrow t.Z]$   
 <="(i)"  
 $[t.I' \wedge t.\langle b,l \rangle \wedge X'] \Rightarrow t.Z]$   
 <="Lemma 1 with Q:= I'; y:= b,l; P:= X'; R:= Z"  
 $[I' \wedge X' \Rightarrow Z]$   
 ="Z"  
 true (End)

We give an example to show that it is not always possible to "refine out" the dummy abstract variables. With abstract variable  $b$  and concrete variable  $v$  both of type  $\text{nat}$ , and abstraction invariant  $b \text{ div } 2 = v$ , the data refinement of  $b := b+1$  is by Theorem 11 any (procedural) refinement of  $\{b \text{ div } 2 = v\}v:-((b+1) \text{ div } 2 = v)$  that has no references to  $b$ . By the laws of procedural refinement (see Lemma 2) we know that such a refinement must be an assignment to  $v$  only — say it is  $v := i$  for some  $i$ . By an application of Lemma 2 and some elementary calculus we find  $\{b \text{ div } 2 = v\}v:-((b+1) \text{ div } 2 = v) \leq v := i$  implies  $[i=v \wedge i=v+1]$ , which is false. Hence we find a second obstacle — miracles being the first — in the way of attempts to use the theory to data refine with inappropriate invariants. This observation is due to Carroll Morgan.

If  $m := e$ , where  $m$  is a common variable, is such that  $e$  does not contain abstract variables then by Theorem 6 it carries over without change to the concrete specification; otherwise we can apply the following theorem.

**Theorem 12.** *Let  $m$  stand for some (or all) common variables,  $m'$  be a fresh variable, and  $e$  denote an expression. Then for all specifications  $t$  that do not refer to abstract variables or  $m'$   $\{G\}m := e \ll t$  follows from  $\{I \wedge G \wedge m=m'\}m:-(I \wedge m=e) \leq t$  where  $e'$  stands for  $e(m \setminus m')$ .*

*Proof.* Exercise. (End)

## 6. Functional Invariants

When the abstract variable(s)  $b$  and concrete variable(s)  $v$  are related by  $b = e$  for  $e$  an expression not containing  $b$ , then life is simpler because data refinement becomes not much more than substitution. When this is the case we say that the abstract variables are functional in  $v$ . This is commonly the case. In this section  $e$  will always stand for an expression in which  $b$  does not occur. The requirement for functionality is  $[I \Rightarrow b = e]$ .

The first simplification attending functional invariants is in the handling of guarded commands.

**Theorem 13.** *if  $([ ] P_i \rightarrow s_i) \mathbf{fi} \ll \mathbf{if} ([ ] P_i(b=e) \rightarrow t_i) \mathbf{fi}$  follows from  $s_i \ll t_i$  for each  $i$  and  $[I \Rightarrow b = e]$ .*

*Proof.* Similar to that of Theorem 4. (End)

**Theorem 14.** *do P → s od « do P(b\|e) → t od follows from s « t and [I ⇒ b = e].*

*Proof.*

We introduce some abbreviations:

P' stands for P(b\|e)

f.X.Y = (P ∧ s.Y) ∨ (¬P ∧ X)

g.X.Y = (P' ∧ t.Y) ∨ (¬P' ∧ X)

It is shown in [10] that assertions partially ordered by [...] are embedded in a complete lattice, and as f.X.Y and g.X.Y are easily shown to be monotonic in Y it follows that the two least fixpoints (μY:f.X.Y) and (μY:g.X.Y) exist for all X [12]. We will be using the least fixpoint rule,

[h.X ⇒ X] ⇒ [(μY:h.Y) ⇒ X] for all monotonic h and all X.

Now **do P → s od « do P' → t od**

=< semantics; X arbitrary not containing v; definitions of f and g"

[(Eb: I ∧ (μY:f.X.Y)) ⇒ (μY:g.(Eb:I^X).Y)]

=< absorption - b not in consequent; Z:= (μY:g.(Eb:I^X).Y)"

[I ∧ (μY:f.X.Y) ⇒ Z]

=< exportation"

[(μY:f.X.Y) ⇒ (I ⇒ Z)]

=< absorption - v not in antecedent"

[(μY:f.X.Y) ⇒ (Av: I ⇒ Z)]

<="least fixpoint property"

[f.X.(Av: I ⇒ Z) ⇒ (Av: I ⇒ Z)]

=< absorption - antecedent does not contain v; importation"

[I ∧ f.X.(Av: I ⇒ Z) ⇒ Z]

=< Z a fixpoint"

[I ∧ f.X.(Av: I ⇒ Z) ⇒ g.(Eb:I^X).Z]

=< f, g"

[I ∧ ((P ∧ s.(Av: I ⇒ Z)) ∨ (¬P ∧ X)) ⇒ (P' ∧ t.Z) ∨ (¬P' ∧ (Eb:I^X))]

<="calculus"

[I ∧ P ∧ s.(Av: I ⇒ Z) ⇒ P' ∧ t.Z] ∧ [I ∧ ¬P ∧ X ⇒ ¬P' ∧ (Eb:I^X)]

=< [I ⇒ b=e]"

[I ∧ s.(Av: I ⇒ Z) ⇒ t.Z] ∧ [I ∧ X ⇒ (Eb:I^X)]

=< absorption - neither consequent contains v"

[(Eb:I ∧ s.(Av: I ⇒ Z)) ⇒ t.Z] ∧ [(Eb:I ∧ X) ⇒ (Eb:I^X)]

<="s « t; (Av: I ⇒ Z) does not contain v"

[t.(Eb:I ∧ (Av: I ⇒ Z)) ⇒ t.Z]

<="t monotonic"

[(Eb:I ∧ (Av: I ⇒ Z)) ⇒ Z]

=< absorption - Z does not contain b; exportation"

[(Av: I ⇒ Z) ⇒ (I ⇒ Z)]

=< calculus"

true

(End)

Assignments to common variables (when the righthand side of the assignment contains abstract variables) are easily translated with functional invariants:

**Theorem 15.** *Let  $m$  stand for some (or all) common variables and  $g$  denote an expression. Then  $m := g \ll m := g'$  if  $[I \Rightarrow b = e]$  and  $g'$  denotes  $g(b \setminus e)$ .*

*Proof.*

$m := g \ll m := g'$

= "«; semantics;  $X$  arbitrary not containing  $v$ ; substitution"

$[(\mathbf{E}b: I \wedge X(m \setminus g)) \Rightarrow (\mathbf{E}b: I \wedge X(m \setminus g'))]$

= "let  $Z := (\mathbf{E}b: I \wedge X(m \setminus g'))$ "

$[(\mathbf{E}b: I \wedge X(m \setminus g)) \Rightarrow Z]$

← "[ $I \Rightarrow b = e$ ]"

$[(\mathbf{E}b: b = e \wedge I \wedge X(m \setminus g)) \Rightarrow Z]$

"1-point"

$[(I \wedge X(m \setminus g))(b \setminus e) \Rightarrow Z]$

= "substitution –  $e$  does not contain  $b$ "

$[(I \wedge X(m \setminus g'))(b \setminus e) \Rightarrow Z]$

= "Z"

true (End)

**Theorem 16.** *If  $[I \Rightarrow b = e]$  then for any common variable(s)  $m$ ,  $m: \underline{Q} \ll \{I(b \setminus e)\}m: \underline{Q}(b \setminus e)$ .*

*Proof.* By a routine application of Theorem 10 (End)

Finally we give a data refinement applicable to any abstract statement when the abstraction invariant is functional.

**Theorem 17.** *If  $[I \Rightarrow b = e]$  then  $s \ll \llbracket b: Tb; b: -I; s; v: -I \rrbracket$ .*

*Proof.* The proof is routine using the fact that when  $[I \Rightarrow b = e]$ ,  $[(\mathbf{E}b: I \wedge X) \Rightarrow (\mathbf{A}b: I \Rightarrow X)]$  for all  $X$ ; we leave the details to the reader. (End)

As it stands Theorem 17 is not very useful, but it shows that with functional invariants we are never faced with dummy abstract variables that cannot be refined out, because the (formerly) abstract variables now appear as variables local to the concrete specification. However the concrete specification may still be miraculous. Consider the following example. Suppose we elect to replace abstract variable  $b$  of type `int` with concrete variable  $v$  of type `small int` whose values are integers in the range 0 to 99, say. We choose the abstraction invariant  $I$  where  $I$  is  $0 \leq v < 100 \wedge b = v$ . Now by Theorem 17 the data refinement of  $b := b+1$  is

$\llbracket b: \text{int}; b: -I; b := b+1; v: -I \rrbracket$

whose non-miraculous domain is

$\neg \llbracket b: \text{int}; b: -I; b := b+1; v: -I \rrbracket.F,$

and by a routine calculation this turns out to be  $0 \leq v < 99$ . This, of course, is as we should expect.

We could use Theorem 17 to formulate rules of data refinement for particular statements when the abstraction invariant is functional. Further specialized rules could be derived for commonly occurring forms

of  $b = e$ , for example when  $b = e$  is  $(c,d) = (f,g)$ , i.e.  $c=f \wedge d=g$ , where expression  $f$ , say, has no variables occurring in expression  $g$ . We do not pursue this.

## 7. Example

We present a small example to illustrate the application of the foregoing theorems; it is based on an example in [9]. Using some notation explained below, and with global variables  $m$  of type sequence of int and  $n$  of type int, the specification of Figure 2 assigns to  $n$  the sum of the elements in  $m$ . (It may well be the case that in the program of Figure 2 procedural refinement has run too far ahead of data refinement, but that does not matter for the purposes of the example.) The type constructors 'bag of' and 'sequence of' have their obvious meaning. The exercise will be to replace variable  $b$  in Figure 2, introducing for the purpose the concrete variable  $v$  of type int.

```

[[      b: bag of int;
      b:= ∅; n:= 0;
      do b ≠ bag.m →
      [[      i: int;
              {b ≠ bag.m} i:-(i ∈ bag.m - b);
              {b ≠ bag.m} b:= b + ⟨i⟩;
              n:= n + i
            ]]
      od
    ]]

```

**Figure 2** : abstract specification

The following notation for bags and sequences is used:

#m : length of sequence  $m$   
 $m.v$  : element  $v$  of sequence  $m$ ,  $0 \leq v < \#m$   
 $\text{bag}.m$  : the bag containing precisely the elements of sequence  $m$   
 $m \downarrow v$  : the initial segment of length  $v$  of sequence  $m$ ,  $0 \leq v \leq \#m$   
 $\emptyset$  : the empty bag  
 $\langle i \rangle$  : the bag containing integer  $i$  only  
 $+, -$  : bag union and difference  
 $\in$  : bag membership

The precedence of operators is now  $., -, +; \downarrow; \in$  followed by that of the previously given operators as before. We will appeal to the following laws of "bag theory":

B1:  $(\text{bag}.(m \downarrow v) = \emptyset) = (v = 0)$   
 B2:  $(\text{bag}.(m \downarrow v) = \text{bag}.m) = (v = \#m)$   
 B3:  $0 \leq v < \#m \Rightarrow m.v \in \text{bag}.m - \text{bag}.(m \downarrow v)$   
 B4:  $0 \leq v < \#m \Rightarrow (\text{bag}.(m \downarrow v) + \langle m.v \rangle = \text{bag}.(m \downarrow v + 1))$

The abstraction invariant we propose to use is

$$I : 0 \leq v \leq \#m \wedge b = \text{bag.}(m \downarrow v)$$

Observe that, somewhat unusually, the abstraction invariant contains common variables. We summarize the variables

abstract variable	b: bag of int
concrete variable	v: int
common global variables:	m: sequence of int, n: int
common local variable	i: int

We refine the statements in turn, in each case the data refinement being followed by procedural refinement of the resulting specification. We will not describe the laws of procedural refinement, but will use them to the extent that the reader should find intuitively reasonable.

- By Theorem 1 the abstract initialization  $b := \emptyset$  is data refined by

$$v :-(\exists b: I \wedge b = \emptyset)$$

"1-point"

$$v :-(b \setminus \emptyset)$$

"substitution"

$$v :-(0 \leq v \leq \#m \wedge \emptyset = \text{bag.}(m \downarrow v))$$

"procedural refinement"

$$v :-(v = 0 \wedge \emptyset = \text{bag.}(m \downarrow 0))$$

"B1"

$$v :-(v = 0)$$

"procedural refinement"

$$v := 0$$
- $\{b \neq \text{bag.}m\} i :-(i \in \text{bag.}m - b)$

"Theorem 10"

$$\{(\exists b: I \wedge b \neq \text{bag.}m)\} i :-(\exists b: I \Rightarrow i \in \text{bag.}m - b)$$

"I; 1-point"

$$\{0 \leq v \leq \#m \wedge \text{bag.}(m \downarrow v) \neq \text{bag.}m\} i :-(0 \leq v \leq \#m \Rightarrow i \in \text{bag.}m - \text{bag.}(m \downarrow v))$$

"procedural refinement"

$$\{0 \leq v \leq \#m \wedge \text{bag.}(m \downarrow v) \neq \text{bag.}m\} i :-(i \in \text{bag.}m - \text{bag.}(m \downarrow v))$$

"B2"

$$\{0 \leq v < \#m\} i :-(i \in \text{bag.}m - \text{bag.}(m \downarrow v))$$

"procedural refinement"

$$\{0 \leq v < \#m\} i :-(0 \leq v < \#m \wedge i \in \text{bag.}m - \text{bag.}(m \downarrow v))$$

"B3; procedural refinement"

$$\{0 \leq v < \#m\} i :-(i = m.v)$$

"procedural refinement"

$$i := m.v$$
- $\{b \neq \text{bag.}m\} b := b + \langle i \rangle$

"Theorem 11 provided we refine out b"

$\{I \wedge b \neq \text{bag.m}\} v:-(b \setminus b + \langle i \rangle)$   
 ="I"  
 $\{0 \leq v \leq \#m \wedge b = \text{bag.(m}\downarrow v) \wedge b \neq \text{bag.m}\} v:-(0 \leq v \leq \#m \wedge b + \langle i \rangle = \text{bag.(m}\downarrow v))$   
 ≤"B2"  
 $\{0 \leq v < \#m \wedge b = \text{bag.(m}\downarrow v)\} v:-(0 \leq v \leq \#m \wedge b + \langle i \rangle = \text{bag.(m}\downarrow v))$   
 ≤"development steps omitted — verify using Lemma 2 and B4"  
 $(i = m.v \rightarrow v := v + 1)$

- For the translation of the loop we will use Theorem 14; the guard is  $(b \neq \text{bag.m})(b \setminus \text{bag.(m}\downarrow v))$

="substitution"  
 $\text{bag.(m}\downarrow v) \neq \text{bag.m}$   
 ="B2"  
 $v \neq \#m$

The assembled concrete specification is given in Figure 3, where we implicitly appeal to various theorems that allow us to carry over the abstract structure and statements not containing  $b$  to the concrete specification. The context of the minor miracle in Figure 3 clearly includes  $i = m.v$  and so the final program is that of Figure 3 with  $(i = m.v \rightarrow v := v + 1)$  replaced by  $v := v + 1$ .

```

[[
  v: int;
  v:= 0; n:= 0;
  do v ≠ #m →
    [[
      i: int;
      i:= m.v;
      (i = m.v → v:= v + 1);
      n:= n + i
    ]]
  od
]]

```

**Figure 3** : concrete specification

## 8. Conclusion

Although data refinement is a well-established technique – it was popularised in [5] and has been central to VDM [7] – there has been renewed interest in it recently [3, 6, 9] stemming from our better understanding of the mathematics of programming. The earliest treatment based on predicate transformers is that of Back [1] (which is a development of his earlier Ph. D. thesis). Back considers abstraction invariants of the shape  $Q \wedge b = e$  where  $Q$  and  $e$  contain concrete variables only. In this special case Back's definition of data refinement can be shown to be essentially the same as that of the present paper. Our first extension is to place no restrictions on the abstraction invariant other than that it should not refer to common variables that are assigned to in the abstract specification. That abstraction invariants containing common variables arise in practice has been shown in the example of Section 7; the need for non-functional invariants is shown in [7]. Back shows that for his choices of abstraction invariant data refinement distributes through the structure of the specification, but he does not supply rules for translating abstract

primitives such as prescriptions and assignments to concrete form — instead one guesses the translation and then verifies it using weakest preconditions. The present work goes further in that we develop a comprehensive set of laws for *calculating* concrete specifications. The laws require no knowledge of weakest preconditions which are only used to establish the validity of the laws. Neither does [1] use miracles which, we feel, leads to a simpler theory. Nevertheless [1] is a comprehensive treatment of procedural and data refinement and we commend it to the reader. A more recent treatment based on predicate transformers is [3]; this takes  $[I \wedge s.T \Rightarrow t.(¬s.(¬I))]$  for the definition of  $s \ll t$ , as does [9]. We have not proceeded from this definition for two reasons. The first is that it does not appear to accommodate comfortably variables common to the abstract and common domains. A naive application of the definition, for example, would suggest that  $m:= 1$  is data refined by  $m:= 2$  where  $m$  is a common variable; [3] uses a renaming device to overcome this difficulty. The second reason is that the definition of [3] does not seem to lend itself as readily to formal manipulation; [3] does not make a formal justification of the definition, nor supply laws of translation. None of this is to question the legitimacy of the definition in [3]. The present theory is a development of an earlier work [11], the main improvements being the admission of common variables to abstraction invariants and the wholly calculational methodology now employed.

In summary, we have proposed a new definition of data refinement and have used it to develop a set of laws allowing us to construct specifications from their abstract form. We have done so in a setting that places data refinement squarely beside procedural refinement in the one specification and developmental framework.

### **Acknowledgements**

I appreciate the efforts of two referees whose comments have led to an improvement in the presentation of the theory.

### **References**

1. Back, R. J. R.: Correctness preserving program refinements: Proof theory and applications. Mathematical Centre Tracts 131. Amsterdam: Mathematisch Centrum 1980
2. Dijkstra, E. W.: A discipline of programming. Englewood Cliffs, N. J.: Prentice–Hall 1976
3. Gries, D. and Prins, J. A.: A new notion of encapsulation. Proc. Symp. Language Issues in Programming Environments. Sigplan 20, 131–139 (1985)
4. Hayes, I (ed). Specification case studies. Englewood Cliffs, N. J.: Prentice–Hall 1987
5. Hoare, C. A. R.: Proofs of correctness of data representations. Acta Informat. 1, 271–281 (1972)
6. Hoare, C. A. R., He, J. F. and Sanders, J. W.: Prespecification in data refinement. Info. Proc. Letters 25, 71-76 (1987)
7. Jones, C. B.: Systematic software development using VDM. Englewood Cliffs, N. J.: Prentice–Hall 1986

8. Morgan, C.: The specification statement. *ACM Trans. Program. Lang. Syst.*, to appear.
9. Morgan, C.: Data refinement by miracles. *Info. Proc. Letters* 26, 243-246 (1988)
10. Morris, J. M.: A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Programming* 9, 287-306 (1987)
11. Morris, J. M.: Piecewise data refinement. Report CSC/87/R12, Dept. of Computing Sci., Univ. of Glasgow 1987
12. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. *Pac. J. Math.* 5, 285–309 (1955)