Acta Informatica 27, 481–503 (1990)

Acta Informatica

Data Refinement by Calculation

Carroll Morgan and P.H.B. Gardiner*
Oxford University, Programming Research Group, 8–11 Keble Road, Oxford, OX1 3QD, UK

Received July 18, 1988 / October 17, 1989

Summary. Data refinement is the systematic substitution of one data type for another in a program. Usually, the new data type is more efficient than the old, but possibly more complex; the purpose of the data refinement in that case is to make progress in program construction from more abstract to more concrete formulations. A recent trend in program construction is to *calculate* programs from their specifications; that contrasts with proving that a *given* program satisfies some specification. We investigate to what extent the trend can be applied to data refinement.

1. Introduction

In [1], Back proposed an extension of Dijkstra's calculus [7] where specifications and programs are given equal status during program construction. Later interest in specifications generally has led quite recently to further work on such constructions [3, 4, 18–20, 22, 24]. The style is now known as the *refinement calculus.*

Characteristic of any calculus is that it is used for *calculation*, not just description. The refinement calculus, therefore, should allow programs to be calculated from their specifications. It does indeed allow presentations in which each intermediate design follows from a previous design according to some *law of refinement.* That contrasts with the more well-known style in which intermediate designs are first proposed and then proved to follow from their antecedents. Our hope is that constructions in the refinement calculus will proceed more smoothly, and that proof obligations will be reduced. That is the point of a calculus, and it can be observed elsewhere: for example, in the differential calculus one uses laws of differentiation, not proofs from first principles. For differentiation, the process is now mechanical. In the integral calculus, we have laws too – but there, as in the refinement calculus, success is not guaranteed.

* Supported by British Petroleum Ltd.

Offprint requests to: C. Morgan

Data refinement is a special case of refinement : one replaces an abstract type by a more concrete type in a program while preserving its algorithmic structure. Abstract operations are similarly replaced by corresponding concrete operations. It is a well-established technique, with its own specialised proof rules [11, 13].

Our principal contribution is to draw data refinement into the calculational style: we show how to *calculate* data refinements rather than prove them. Hoare et al. [12] first introduced this concept, but their work, being in a relational setting, was not easily applied in practice. Here we build on our earlier work [8], which was in a predicate transformer setting, to produce practical methods of calculation. This paper gives applications of the theory in [8], though for convenience we have presented afresh some proofs which are corollaries by specialisation. Recent work by Morris [23] addresses the same concerns that we do.

In passing we fromalise *logical constants*, long used in program derivation, but not until now treated rigorously. Their use in programs loses the property of conjunctivity, another of Dijkstra's healthiness laws [7]. (The law of the excluded miracle, and continuity, have already been abandoned [6, 19, 24, 25].)

This work relies on the ideas of the refinement calculus, reviewed in Sects. 2 and 3 below. More detail can be found in [2, 19, 20, 24].

2. Refinement

We consider Dijkstra's programming langùage [7], whose meaning is given by *predicate transformers*. For any program P, we write $[\![P]\!]$ for its *meaning*; and that meaning is a function from (desired) final assertions to (necessary) initial ones:

> For any formula ψ over state variables, and program P, $[\![P]\!]\psi$ is the *weakest* formula whose truth in an initial state ensures that activation of P will lead to a final state in which ψ is true.

Thus we write $[\![P]\!]\psi$ for Dijkstra's $wp(P, \psi)$.

2.1. Algorithmic Refinement

A program P is *algorithmically* refined by another P' whenever every specification satisfied by P is satisfied by P' also. We restrict our specifications, however, to formulae $\phi \Rightarrow [\![P]\!]\psi$ which state "the program P must be such that its activation in a state in which ϕ is true will lead to a state in which ψ is true." We do not, for example, specify time or space constraints.

Note that, when we assert a formula in text (for example $\phi \Rightarrow [\![P]\!]\psi$ in the above paragraph), we mean that the formula is valid (i.e. true for all values of its free variables).

Definition 1. *Algorithmic refinement:* Program P is algorithmically refined by program P' precisely when, for all formulae ϕ and ψ over the program variables,

$$\phi \Rightarrow [\![P]\!]\psi \quad \text{implies} \quad \phi \Rightarrow [\![P']\!]\psi.$$

We write $P \sqsubseteq P'$ for that relationship. □

The following is an easy consequence of Definition 1, and is what we will use in practice:

Lemma 1. *Algorithmic refinement: For programs P and P', we have $P \sqsubseteq P'$ precisely when*

$$[\![P]\!]\psi \Rightarrow [\![P']\!]\psi \quad \textit{for all formulae } \psi \textit{ over the program variables.}$$

Proof. For *if*, note that $\phi \Rightarrow [\![P]\!]\psi$ and $[\![P]\!]\psi \Rightarrow [\![P']\!]\psi$ imply $\phi \Rightarrow [\![P']\!]\psi$ as required; for *only if*, take ϕ to be $[\![P]\!]\psi$ itself. □

We assume that in Lemma 1 we may limit our choice of formulae ψ to those containing only variables free either in P or P' or both.

2.2. Data Refinement

Data refinement arises as a special case of algorithmic refinement. A program P is data-refined to another program P' by a transformation in which some so-called *abstract* data-type in P is replaced by a *concrete* data-type in P'. The overall effect is an algorithmic refinement of the block in which the abstract data type is declared.

For that, we add *local variables* to Dijkstra's language in the following (standard) way:

Definition 2. *Local variables:* For (list of) variables l, formula I (the initialisation), and program P, the construction

$$|[\mathbf{var}\, l \,|\, I \cdot P]|$$

is a *local block* in which the local variables l are introduced for the use of program P; they are first assigned initial values such that I holds. We define, for ψ not containing l,

$$[\![\,|[\mathbf{var}\, l \,|\, I \cdot P]|\,]\!]\psi \triangleq (\forall l \cdot I \Rightarrow [\![P]\!]\psi). \quad \square$$

The symbol $\triangleq$ is read "is defined to be equal to".

Note that the scope of quantifiers is indicated explicitly by parentheses $(\forall \ldots)$.

Where a postcondition ψ does contain the local variable l, Definition 2 can be applied after systematic change of the local l to some fresh l'. We assume therefore that such clashes do not occur.

Where appropriate, we consider *types* to be simply sets of values, and will write $|[\mathbf{var}\, l{:}\ T \,|\, I \cdot P]|$ for $|[\mathbf{var}\, l \,|\, (l \in T \wedge I) \cdot P]|$; thus a variable is initialised to some value in its type. And if I is just *true* we may omit it, writing $|[\mathbf{var}\, l \cdot P]|$ or $|[\mathbf{var}\, l{:}\ T \cdot P]|$ as appropriate.

Now data-refinement transforms an abstract block $|[\mathbf{var}\, a \,|\, I \cdot P]|$ to a concrete block $|[\mathbf{var}\, c \,|\, I' \cdot P']|$. We assume that the concrete variables c do not occur

in the abstract program I and P, and *vice versa.* The transformation has these characteristics:

1. The concrete block algorithmically refines the abstract block:

$$|[\mathbf{var}\, a \mid I \cdot P]| \sqsubseteq |[\mathbf{var}\, c \mid I' \cdot P']|.$$

2. The abstract variable declarations **var** a are replaced by concrete variable declarations **var** c.

3. The abstract initialisation I is replaced by a concrete initialisation I'.

4. The abstract program P, referring to variables a but not c, is replaced by a concrete program P' referring to variables c but not a; moreover, the algorithmic structure of P is reproduced in P' (see below).

The four characteristics are realised as follows. An *abstraction invariant AI* is chosen which links the abstract variables a and the concrete variables c. It may be *any* formula, but usually will refer to a and c at least. (See Sect. 4.1 below for a discussion of the impracticality of choosing *false* as the abstraction invariant.) The concrete initialisation I' must be such that $I' \Rightarrow (\exists a \cdot AI \wedge I)$. For the concrete program we define a relation $\preceq$ of data-refinement:

Definition 3. *Data refinement:* A program P is said to be data-refined by another program P', using abstraction invariant AI, abstract variables a and concrete variables c, whenever for all formulae ψ not containing c free we have

$$(\exists a \cdot AI \wedge [\![P]\!]\psi) \Rightarrow [\![P']\!](\exists a \cdot AI \wedge \psi).$$

We write this relation $P \preceq_{AI,a,c} P'$, and omit the subscript $_{AI,a,c}$ when it is understood from context. □

Definition 3 is appropriate for two reasons. The first is that it guarantees characteristic 1, as we now show.

Theorem 1. *Soundness of data-refinement: If* $I' \Rightarrow (\exists a \cdot AI \wedge I)$ *and* $P \preceq P'$, *then*

$$|[\mathbf{var}\, a \mid I \cdot P]| \sqsubseteq |[\mathbf{var}\, c \mid I' \cdot P']|.$$

Proof. Consider any ψ not containing a or c free. We have

$$\begin{aligned}
&[\![|[\mathbf{var}\, a \mid I \cdot P]|]\!]\psi \\
&= (\forall a \cdot I \Rightarrow [\![P]\!]\psi) && \text{Definition 2} \\
&= (\forall c, a \cdot I \Rightarrow [\![P]\!]\psi) && c \text{ not free in above} \\
&\Rightarrow (\forall c, a \cdot AI \wedge I \Rightarrow AI \wedge [\![P]\!]\psi) \\
&\Rightarrow (\forall c \cdot (\exists a \cdot AI \wedge I) \Rightarrow (\exists a \cdot AI \wedge [\![P]\!]\psi)) \\
&\Rightarrow (\forall c \cdot I' \Rightarrow (\exists a \cdot AI \wedge [\![P]\!]\psi)) && \text{assumption} \\
&\Rightarrow (\forall c \cdot I' \Rightarrow [\![P']\!](\exists a \cdot AI \wedge \psi)) && \text{assumption; Definition 3} \\
&\Rightarrow (\forall c \cdot I' \Rightarrow [\![P']\!]\psi) && \text{monotonicity; } a \text{ not free in } \psi \\
&= [\![|[\mathbf{var}\, c \mid I' \cdot P']|]\!]\psi && \text{Definition 2.}\ \square
\end{aligned}$$

The second reason our Definition 3 is appropriate is that it distributes through program composition. This is shown in [23, 8], and we refer the reader there for details. Here, for illustration, we treat sequential composition; alternation and iteration are dealt with in Sect. 4 and 6 below.

Lemma 2. *Data-refinement distributes through sequential composition: If* $P \preceq P'$ *and* $Q \preceq Q'$ *then* $(P; Q) \preceq (P'; Q')$.

Proof. Let ψ be any formula not containing c. Then

$$
\begin{array}{ll}
& (\exists a \cdot AI \wedge [\![P; Q]\!] \psi) \\
= & (\exists a \cdot AI \wedge [\![P]\!]([\![Q]\!] \psi)) \quad \text{semantics of ";"} \\
\Rightarrow & [\![P']\!](\exists a \cdot AI \wedge [\![Q]\!] \psi) \quad P \preceq P' \\
\Rightarrow & [\![P']\!]([\![Q']\!](\exists a \cdot AI \wedge \psi)) \quad Q \preceq Q'\text{; monotonicity} \\
= & [\![P'; Q']\!](\exists a \cdot AI \wedge \psi) \quad \text{semantics of ";".} \quad \square
\end{array}
$$

It is the distributive property illustrated by Lemma 2 that accounts for characteristic 4 above: if for example P is $P_1; P_2; \ldots P_n$ then we can construct P' with $P \preceq P'$ simply by taking $P' = P_1'; P_2'; \ldots P_n'$ with $P_i \preceq P_i'$ for each i. It is in this sense that the stucture of P is preserved in P'. We will see in Sect. 4 below that this carries through for alternations and iterations also.

3. Language Extensions

We extend Dijkstra's language in two ways. With the *specification statement* we allow specifications and executable program fragments to be mixed, thus promoting a more uniform development style. With *program conjunction* we make more rigorous the use of so-called *logical constants*, which appear in specifications but not in executable programs.

3.1. Specification Statements

A specification statement is a list of changing variables called the *frame* (say w), a formula called the *precondition* (say *pre*), and a formula called the *postcondition* (say *post*). Together they are written

$$w: [pre, post].$$

Informally this construct denotes a program which,

> if *pre* is true in the initial state, will establish *post* in the final state by changing only variables mentioned in the list w.

For the precise meaning, we have

Definition 4. *Specification statement:* For formulae *pre*, *post* over the program variables, and list of program variables w,

$$[\![w: [pre, post]]\!] \psi \triangleq pre \wedge (\forall w \cdot post \Rightarrow \psi). \quad \square$$

Specification statements allow program development to proceed at the level of refinement steps $\sqsubseteq$ rather than directly in terms of weakest preconditions, and are discussed in detail in [19, 20]. They are similar to the *descriptions* of [2] and the *prescriptions* of [24]. For now we extract from the above works a collection of *refinement laws*, given in the appendix to this paper. We illustrate their use with the following small program development:

"assign to y the absolute value of x"
$= y: [\mathit{true}, y = |x|]$
$= y: [(x \leqq 0) \vee (x \geqq 0), y = |x|]$
$\sqsubseteq$ Law 13
if $x \leqq 0 \rightarrow y: [x \leqq 0, y = |x|]$
$\square$ $x \geqq 0 \rightarrow y: [x \geqq 0, y = |x|]$
fi.
$=$ **if** $x \leqq 0 \rightarrow y: [-x = |x|, y = |x|]$
$\square$ $x \geqq 0 \rightarrow y: [x = |x|, y = |x|]$
fi.
$\sqsubseteq$ Law 12 twice
if $x \leqq 0 \rightarrow y := -x$
$\square$ $x \geqq 0 \rightarrow y := x$
fi.

3.2. Program Conjunction

Given a program P we write the generalised *program conjunction* of P over some variable i as $|[\mathbf{con}\, i \cdot P]|$. We call it conjunction because that new program is a refinement $\sqsubseteq$ of the original program P for *all* values of the *logical constant* i. For example, consider the statement $x: [x = i, x = i + 1]$, and suppose our variables range over the natural numbers. Its generalised conjunction over i refines all of the following:

$$x: [x = 0, x = 1]$$
$$x: [x = 1, x = 2]$$
$$x: [x = 2, x = 3]$$
$$\vdots$$

Each of those programs deals with a specific value of x, and can abort for all others. Yet, as Definition 5 will show, that generalised conjunction equals the statement $x := x + 1$, which is guaranteed to terminate.

Definition 5. *Program conjunction:* For program P and variable i not free in ψ,

$$[\![|[\mathbf{con}\, i \cdot P]|]\!]\, \psi \triangleq (\exists i \cdot [\![P]\!]\, \psi). \quad \square$$

As in Definition 2, systematic renaming can deal with occurrences of i in ψ.

Thus for the example above we can calculate

$$[\![|[\mathbf{con}\, i \cdot x: [x = i, x = i + 1]]|]\!]\, \psi$$

$= (\exists i \cdot [\![x: [x = i, x = i+1]]\!] \psi)$ Definition 5
$= (\exists i \cdot x = i \wedge (\forall x \cdot x = i+1 \Rightarrow \psi))$ Definition 4
$= (\exists i \cdot x = i \wedge \psi[x \backslash i+1])$ One point rule
$= \psi[x \backslash i+1][i \backslash x]$ One point rule
$= \psi[x \backslash x+1]$ i not free in ψ
$= [\![x := x+1]\!] \psi$

The notation $[x \backslash i+1]$ indicates syntactic replacement of x by $i+1$ with any changes of bound variable necessary to avoid capture.

Variables declared by **con** we call *logical constants.* They usually appear in program developments where some initial value must be fixed, in order to allow later reference to it. For example in the Hoare style [10], we might write "find a program P, changing only x, such that $\{x = X\} P \{x = X+1\}$". Here the upper case X makes use of a convention that such variables are not to appear in the final program: it is not $x := X+1$ that is sought, but $x := x+1$. We would just write

$$|[\mathbf{con}\ X \cdot x: [x = X, x = X+1]]|,$$

it being understood that we are looking for a refinement of that. Since our final programming language does not allow declarations **con**, we are forced to use refinements whose effect is to eliminate X. We do not need an upper-case convention.

It is interesting that program conjunction is the dual of local variable declaration (compare Definitions 2 and 5); thus logical constants are in that sense dual to local variables. It is shown in [8] that data refinement distributes through program conjunction.

4. Data Refinement Calculators

In Sect. 2 we defined the relation $\preceq$ of data-refinement between two statements S and S'. We gave there also a sufficient relation between the abstract initialisation I and the concrete initialisation I'.

In this section we show how the extensions of Sect. 3 allow us to *calculate* data-refinements S' and I' which satisfy the sufficient relations automatically. Following [14], we call these techniques *calculators.*

For the rest of this section, we will assume that the data-refinement is given by

abstract variables: a
concrete variables: c
abstraction invariant: AI

Moreover, we assume that the concrete variables c do not appear free in the abstract program.

4.1. *The Initialisation Calculator*

For concrete initialisation I' to data-refine the abstract I we know from Theorem 1 that $I' \Rightarrow (\exists a \cdot AI \wedge I)$ is sufficient; therefore we define I' to be $(\exists a \cdot AI \wedge I)$ itself. Law 5 (appendix) shows that we lose no generality, since any concrete initialisation I', where $I' \Rightarrow (\exists a \cdot AI \wedge I)$, can be reached in two stages: first replace I by the calculated $(\exists a \cdot AI \wedge I)$; then strengthen that, by Law 5, to I'.

If AI is *false*, then the calculated I' will be *false* also; indeed, Law 5 allows a refinement step to *false* initialisation directly. That is valid, though impractical, for the following reason: Definition 2 shows that the resulting program is *miraculous*:

$$[\![\,|\, [\mathbf{var}\ l \,|\, false \cdot P] \,|\,]\!]\ false = true.$$

It can never be implemented in a programming language. (And that is why programming languages do not have empty types.)

4.2. *The Specification Calculator*

Lemma 3 to follow gives us a calculator for the data-refinement of any abstract statement of the form $a, x: [pre, post]$, where a and x are disjoint (and either may be empty). Lemma 4 shows that taking that data-refinement loses no generality. The two results are combined in Theorem 2. Finally, we give as a corollary a calculator for statements $b, x: [pre, post]$ where b is a subset of a; that is an abstract statement which may require some abstract variables not to change.

Lemma 3. *Validity: The following data-refinement is always valid:*

$$\begin{array}{l} a, x: [pre, post] \\ \preceq c, x: [(\exists a \cdot AI \wedge pre), (\exists a \cdot AI \wedge post)] \end{array}$$

Proof. We take any formula ψ containing no free c, and proceed as follows:

$$\begin{array}{ll} & (\exists a \cdot AI \wedge [\![a, x: [pre, post]]\!]\ \psi) \\ = & (\exists a \cdot AI \wedge pre \wedge (\forall a, x \cdot post \Rightarrow \psi)) \quad \text{Definition 4} \\ = & (\exists a \cdot AI \wedge pre) \wedge (\forall c, a, x \cdot post \Rightarrow \psi) \quad c \text{ not free in } post, \psi \\ \Rightarrow & (\exists a \cdot AI \wedge pre) \wedge (\forall c, x, a \cdot AI \wedge post \Rightarrow AI \wedge \psi) \\ \Rightarrow & (\exists a \cdot AI \wedge pre) \\ & \quad \wedge (\forall c, x \cdot (\exists a \cdot AI \wedge post) \Rightarrow (\exists a \cdot AI \wedge \psi)) \\ = & [\![c, x: [(\exists a \cdot AI \wedge pre), (\exists a \cdot AI \wedge post)]]\!] (\exists a \cdot AI \wedge \psi). \quad \square \end{array}$$

Lemma 4. *Generality: For all programs CP, if* $a, x: [pre, post] \preceq CP$ *then*

$$c, x: [(\exists a \cdot AI \wedge pre), (\exists a \cdot AI \wedge post)] \sqsubseteq CP$$

Proof. We take any ψ containing no free a, and proceed as follows:

$$\begin{array}{ll} & [\![c, x: [(\exists a \cdot AI \wedge pre), (\exists a \cdot AI \wedge post)]]\!]\ \psi \\ = & (\exists a \cdot AI \wedge pre) \quad \text{Definition 4} \\ & \quad \wedge (\forall c, x \cdot (\exists a \cdot AI \wedge post) \Rightarrow \psi) \end{array}$$

$= (\exists a \cdot AI \wedge pre)$ c not free in $post$,
$\quad \wedge (\forall a, x \cdot post \Rightarrow (\forall c \cdot AI \Rightarrow \psi))$ a not free in ψ
$= (\exists a \cdot AI \wedge pre \wedge (\forall a, x \cdot post \Rightarrow (\forall c \cdot AI \Rightarrow \psi)))$
$= (\exists a \cdot AI \wedge [\![a, x: [pre, post]]\!](\forall c \cdot AI \Rightarrow \psi))$ Definition 4
$\Rightarrow [\![CP]\!](\exists a \cdot AI \wedge (\forall c \cdot AI \Rightarrow \psi))$ assumption, Definition 3
$\Rightarrow [\![CP]\!]\psi$ a not free in ψ, monotonicity. □

We now have the specification calculator we require: Lemma 3 states that it *is* a data refinement; Lemma 4 states that any other data-refinement of the abstract specification is an *algorithmic* refinement of the calculated one. We summarise that in Theorem 2:

Theorem 2. *The specification calculator: For all programs CP,*

$$a, x: [pre, post] \preceq CP$$

if and only if

$$c, x: [(\exists a \cdot AI \wedge pre), (\exists a \cdot AI \wedge post)] \sqsubseteq CP.$$

Proof. From Lemmas 3 and 4. □

Note that the quantifications $(\exists a \ldots)$ ensure that the abstract variables a do not appear in the concrete program.

We conclude this section with a corollary of Lemma 3; it calculates the data-refinement of an abstract specification in which not all variables are changing. In its proof we are able to reason at the higher level of the relations $\sqsubseteq$ and $\preceq$; weakest preconditions are not required.

This corollary is the first occasion we have to use logical constants in data refinement. Like local variables, logical constants are *bound* in a program; and it is the **con** declaration which binds the abstract variables a in Corollary 1, since the quantification $(\exists b \ldots)$ alone may leave some abstract variables free.

Corollary 1. *For any subset (not necessarily proper) b of the abstract variables a, the abstract specification $b, x: [pre, post]$ is data-refined by*

$$|[\textbf{con}\ a \cdot \; c, x: [AI \wedge pre, (\exists b \cdot AI \wedge post)] \;]|.$$

Proof. Let y be the set of variables containing the members of a that are not members also of b, and let Y be a set of fresh variables with size equal to that of y. Then

$b, x: [pre, post]$
$=$ Law 9
$\quad |[\textbf{con}\ Y \cdot b, y, x: [pre \wedge y = Y, post \wedge y = Y]]|$
$\preceq$ Lemma 3, data refinement distributes through **con**
$\quad |[\textbf{con}\ Y \cdot$
$\qquad c, x: [(\exists b, y \cdot AI \wedge pre \wedge y = Y), (\exists b, y \cdot AI \wedge post \wedge y = Y)]$
$\quad]|$
$= |[\textbf{con}\ Y \cdot$
$\qquad c, x: [(\exists b \cdot AI \wedge pre)[y \backslash Y], (\exists b \cdot AI \wedge post)[y \backslash Y]]$

$$
\begin{array}{l}
\quad]| \\
= \text{Law 8} \\
\quad |[\textbf{con}\ y\cdot \\
\qquad c, x: [(\exists b\cdot AI \wedge pre), (\exists b\cdot AI \wedge post)] \\
\quad]| \\
= \text{Law 6} \\
\quad |[\textbf{con}\ a\cdot \\
\qquad c, x: [AI \wedge pre, (\exists b\cdot AI \wedge post)] \\
\quad]|. \quad \square
\end{array}
$$

That data refinement distributes through **con**, is proven in [8].

4.3. The guard calculator

We saw in Corollary 1 that the specification calculator introduces **con** a and existentially quantifies over changing abstract variables only. For guards, changing nothing, we would expect that quantification to be empty. We have

Theorem 3. *The guard calculator: If $S_i \preceq S_i'$ for each i, then the following refinement is valid:*

$$
\begin{array}{l}
\quad \textbf{if}(\Box i\cdot G_i \to S_i)\ \textbf{fi} \\
\preceq |[\textbf{con}\ a\cdot \\
\qquad \textbf{if}(\Box i\cdot AI \wedge G_i \to S_i')\ \textbf{fi} \\
\quad]|.
\end{array}
$$

Proof. For any ψ not containing c, we have

$$
\begin{array}{l}
\quad (\exists a\cdot AI \wedge [\![\textbf{if}(\Box i\cdot G_i \to S_i)\ \textbf{fi}]\!]\,\psi) \\
= (\exists a\cdot AI \wedge (\vee i\cdot G_i) \quad \text{definition } [\![\textbf{if}\ldots\textbf{fi}]\!] \\
\qquad\quad \wedge (\wedge i\cdot G_i \Rightarrow [\![S_i]\!]\,\psi)) \\
= (\exists a\cdot (\vee i\cdot AI \wedge G_i) \wedge (\wedge i\cdot AI \wedge G_i \Rightarrow AI \wedge [\![S_i]\!]\,\psi)) \\
\Rightarrow (\exists a\cdot (\vee i\cdot AI \wedge G_i) \\
\qquad \wedge (\wedge i\cdot AI \wedge G_i \Rightarrow (\exists a\cdot AI \wedge [\![S_i]\!]\,\psi))) \\
\Rightarrow (\exists a\cdot (\vee i\cdot AI \wedge G_i) \quad \text{since } S_i \preceq S_i' \\
\qquad \wedge (\wedge i\cdot AI \wedge G_i \Rightarrow [\![S_i']\!](\exists a\cdot AI \wedge \psi))) \\
= [\![|[\textbf{con}\ a\cdot \ldots]|]\!](\exists a\cdot AI \wedge \psi). \quad \square
\end{array}
$$

A similar construction is possible for **do**... **od**, but in this general setting it is better to use **if**... **fi** and recursion. There are special cases for **do**, however, and they are discussed in Sect. 6.

5. Example of Refinement: The "Mean" Module

We can present a data refinement independently of its surrounding program text by collecting together all the statements that refer to the abstract variables or to variables in the abstraction invariant. Such a collection is called a *module*, and we can confine our attention to it for this reason: statements which do *not* refer to abstract variables, or to the abstraction invariant, are refined by themselves and we need not change them.

Consider the module of Fig. 1 for calculating the mean of a sample of numbers. We write bag comprehensions between brackets $\prec\succ$, and use $\sum b$

module *Calculator* $\triangleq$
  var b: **bag of** *Real*;
  procedure *Clear* $\triangleq b := \prec\succ$;
  procedure *Enter* (**value** r) $\triangleq b := b + \prec r\succ$;
  procedure *Mean* (**result** m) $\triangleq$
    if $b \neq \prec\succ \rightarrow m := \sum b / \# b$
    $[]$ $b = \prec\succ \rightarrow$ **error**
    fi
end

Fig. 1. The "mean" module

and $\# b$ for the sum and size respectively of bag b. The operator $+$ is used for bag addition. The statement **error** is some definite error indication, and we assume that **error** $\preceq$ **error**. The initialisation is $b \in$ **bag of** *Real*.

The module is operated by: first *clearing*; then *entering* the sample values, one at a time; then finally taking the *mean* of all those values.

For the data refinement, we represent the bag by its sum s and size n at any time.

abstract variables: b
concrete variables: s, n
abstraction invariant: $s = \sum b \wedge n = \# b$.

We data-refine the module by replacing the abstract variables b by the concrete variables s, n and applying the calculations of Sect. 4 to the initialisation and the three procedures. Stacked formulae below denote their conjunction.

- For the initialisation, we have from Sect. 4.1

$$\left(\exists b \cdot \begin{pmatrix} b \in \textbf{bag of } Real \\ s = \sum b \\ n = \# b \end{pmatrix}\right) = \begin{pmatrix} s \in Real \\ n \in Natural \\ n = 0 \Rightarrow s = 0 \end{pmatrix}$$

- For the procedure *Clear*, we have from Sect. 4.2

$$\begin{aligned} & b := \prec\succ \\ = {} & b : [true, b = \prec\succ] \\ \preceq {} & \text{Lemma 3} \\ & s, n : \left[\left(\exists b \cdot \begin{matrix} s = \sum b \\ n = \# b \end{matrix}\right), \left(\exists b \cdot \begin{matrix} s = \sum b \\ n = \# b \\ b = \prec\succ \end{matrix}\right)\right] \\ \sqsubseteq {} & \text{Law 1} \\ & s, n : [true, s = 0 \wedge n = 0] \\ \sqsubseteq {} & \text{Law 12} \\ & s, n := 0, 0 \end{aligned}$$

- For the procedure *Enter*, we have from Sect. 4.2

$$
\begin{array}{l}
\quad b := b + \prec r \succ \\
= |[\mathbf{con}\ B \cdot b : [b = B,\ b = B + \prec r \succ]]| \\
\preceq \text{Lemma 3} \\
\quad \left\| \left[\mathbf{con}\ B \cdot s, n : \begin{bmatrix} s = \sum B & s = \sum (B + \prec r \succ) \\ n = \#B' & n = \#(B + \prec r \succ) \end{bmatrix} \right] \right\| \\
\sqsubseteq \text{Law 12} \\
\quad |[\mathbf{con}\ B \cdot s, n := s + r, n + 1]| \\
\sqsubseteq \text{Law 7} \\
\quad s, n := s + r, n + 1
\end{array}
$$

- For *Mean* we have first that from Sect. 4.2

$$
\begin{array}{l}
\quad m := \sum b / \# b \\
= m : [\# b \neq 0,\ m = \sum b / \# b] \\
\preceq \text{Corollary 1 (noting the quantification is empty)} \\
\quad |[\mathbf{con}\ b \cdot \\
\qquad m, s, n : \begin{bmatrix} \# b \neq 0, & m = \sum b / \# b \\ s = \sum b, & s = \sum b \\ n = \# b, & n = \# b \end{bmatrix} \\
\quad]| \\
\sqsubseteq \text{Laws 10, 2, 3, 1} \\
\quad |[\mathbf{con}\ b \cdot m : [n \neq 0,\ m = s/n]]| \\
\sqsubseteq \text{Laws 12, 7} \\
\quad m := s/n.
\end{array}
$$

Then we conclude from Theorem 3 that

$$
\begin{array}{l}
\mathbf{if}\ b \neq \prec \succ \rightarrow m := \sum b / \# b \\
\quad [\!]\ b = \prec \succ \rightarrow \mathbf{error} \\
\mathbf{fi} \\
\\
\preceq |[\mathbf{con}\ b \cdot \\
\qquad \mathbf{if} \begin{pmatrix} b \neq \prec \succ \\ s = \sum b \\ n = \# b \end{pmatrix} \rightarrow m := s/n \\
\qquad [\!] \begin{pmatrix} b = \prec \succ \\ s = \sum b \\ n = \# b \end{pmatrix} \rightarrow \mathbf{error} \\
\qquad \mathbf{fi} \\
\quad]|.
\end{array}
$$

To make further progress with *Mean*, we need to eliminate the abstract variable b from the guards; then Law 7 applies. That is assisted by the following lemma (which is generally applicable to the refinement of alternations, whether or not they occur within data refinements):

Lemma 5. *Refining guards: Given the conditions*

1. $(\vee i \cdot G_i) \Rightarrow (\vee i \cdot G'_i)$
2. $(\vee i \cdot G_i) \Rightarrow (G'_i \Rightarrow G_i)$ *for each* i

the following refinement is valid:

$$\textbf{if}(\Box i \cdot G_i \rightarrow S_i)\ \textbf{fi} \sqsubseteq \textbf{if}(\Box i \cdot G'_i \rightarrow S_i)\ \textbf{fi}.$$

Proof. By Lemma 1 and $[\![\textbf{if} \ldots \textbf{fi}]\!]$ we must show for all formulae ψ that

$$(\vee i \cdot G_i) \wedge (\wedge i \cdot G_i \Rightarrow [\![S_i]\!]\psi)$$
$$\Rightarrow (\vee i \cdot G'_i) \wedge (\wedge i \cdot G'_i \Rightarrow [\![S_i]\!]\psi).$$

That follows by propositional calculus from assumptions 1 and 2 above. □

We have immediately the following corollary:

Corollary 2. *Weakening guards: The following refinement is valid for any formula* X*:*

$$\textbf{if}(\Box i \cdot G_i \wedge X \rightarrow S_i)\ \textbf{fi} \sqsubseteq \textbf{if}(\Box i \cdot G_i \rightarrow S_i)\ \textbf{fi}.$$ □

Now we can continue the refinement of *Mean*:

$\sqsubseteq$ Lemma 5, Law 7
if $n \neq 0 \rightarrow m := s/n$
□ $n = 0 \rightarrow$ **error**
fi.

In Fig. 2 we give the resulting data refinement for the whole module.

module *Calculator* $\triangleq$
  var s: *Real*; n: *Natural*;
  procedure *Clear* $\triangleq s, n := 0, 0$;
  procedure *Enter*(**value** r) $\triangleq s, n := s + r, n + 1$;
  procedure *Mean*(**result** m) $\triangleq$
    if $n \neq 0 \rightarrow m := s/n$
    □ $n = 0 \rightarrow$ **error**
    fi
  initially $n = 0 \Rightarrow s = 0$
end.

Fig. 2. The "mean" module, after data refinement

To see the need for the initialisation, consider this alternative definition of *Clear*:

procedure *Clear* $\triangleq$
  if $b \neq \prec\succ \rightarrow b := \prec\succ$
  □ $b = \prec\succ \rightarrow$ **skip**
  fi

That is semantically identical to the original, in Fig. 1, but might be cheaper overall if the operation $b:=\prec\succ$ were expensive. Its calculated data refinement is

procedure *Clear* $\widehat{=}$
  if $n \neq 0 \rightarrow s, n:=0, 0$
  $[]$ $n=0 \rightarrow$ **skip**
  fi.

That would *not* work correctly if used immediately after an initialisation, say, of $s=1 \wedge n=0$! So our stated initialisation is necessary, after all; note however that since initialisations can always be strengthened (Law 5), we could use the simpler $s=0$ if desired.

6. Specialised Techniques

Now we specialise the techniques of Sect. 4: we consider *guards*, *functional* data-refinement, and the use of *auxiliary variables*.

6.1. Data-refining Guards

We have seen that data refinement takes an abstract program of the form

if $([]i \cdot G_i \rightarrow S_i)$ **fi**

to a concrete one of the form

$|[$**con** $a \cdot$
  if $([]i \cdot AI \wedge G_i \rightarrow S_i')$ **fi**
$]|$

where AI is the abstraction invariant, and S_i' is the result of applying data refinement to S_i.

One of the steps towards code is the removal of the **con**. Before this can be done the occurrences of abstract variables in the concrete guards must be eliminated. We use Lemma 5 for that: we replace each of the calculated guards $G_i \wedge AI$ by the guard $(\forall a \cdot AI \Rightarrow G_i)$, which does not contain a free. By that lemma, we must show

1. $(\vee i \cdot G_i \wedge AI) \Rightarrow (\vee i \cdot G_i')$,
2. $(\vee i \cdot G_i \wedge AI) \Rightarrow (G_i' \Rightarrow G_i \wedge AI)$ *for each* i

where G_i' is $(\forall a \cdot AI \Rightarrow G_i)$.

After expanding G_i' the validity of 2 is evident; and 1 can be rewritten to $(\exists a \cdot AI \wedge (\vee i \cdot G_i)) \Rightarrow (\vee i \cdot G_i')$. Thus we have the following

Lemma 6. *Data refinement of alternations: Given abstraction invariant* AI, *abstract guards* G_i, *and abstract statements* S_i, *let the concrete guards* G_i' *and concrete statements* S_i' *be such that*

1. $G_i' = (\forall a \cdot AI \Rightarrow G_i)$.
2. $S_i \preceq S_i'$.

Then provided $(\exists a \cdot AI \wedge (\vee i \cdot G_i)) \Rightarrow (\vee i \cdot G_i')$, *the following data refinement is valid:*

$$\textbf{if } (\Box i \cdot G_i \to S_i) \textbf{ fi} \preceq \textbf{if } (\Box i \cdot G_i' \to S_i') \textbf{ fi}. \quad \square$$

For iterations the result is the same: we use the recursive formulation.

$$\begin{aligned}\textbf{do } (\Box i \cdot G_i \to S_i) \textbf{ od} \triangleq (\mu P \cdot \textbf{if } & (\Box i \cdot G_i \to S_i;\ P) \\ & \Box \neg(\vee i \cdot G_i) \to \textbf{skip} \\ & \textbf{fi}).\end{aligned}$$

Since data refinement distributes through recursion (see [8] for proof), we merely have to determine the conditions under which

$$\begin{aligned}&\textbf{if } (\Box i \cdot AI \wedge G_i \to S_i;\ P) \\ &\Box\ AI \wedge \neg(\vee i \cdot G_i) \to \textbf{skip} \\ &\textbf{fi} \\ \sqsubseteq\ &\textbf{if } (\Box i \cdot G_i' \to S_i';\ P) \\ &\Box \neg(\vee i \cdot G_i') \to \textbf{skip} \\ &\textbf{fi}.\end{aligned}$$

Here we apply Lemma 5 again, this time noticing that the disjunction of the guards of the initial program simplifies to AI, and that of the refined program simplifies to *true*. Thus Lemma 5 requires

1. $AI \Rightarrow true$

2a. $AI \Rightarrow (G_i' \Rightarrow G_i \wedge AI)$ *for each* i

2b. $AI \Rightarrow (\neg(\vee i \cdot G_i') \Rightarrow \neg(\vee i \cdot G_i) \wedge AI)$.

The validity of 1. and 2a. are evident; and 2b. can be rewritten to give $(\vee i \cdot G_i \wedge AI) \Rightarrow (\vee i \cdot G_i')$, which is the formula that was required in the treatment of alternation. Thus we have

Lemma 7. *Data refinement of iterations: Under the same conditions as Lemma* 6, *the following refinement is valid:*

$$\textbf{do } (\Box i \cdot G_i \to S_i) \textbf{ od} \preceq \textbf{do } (\Box i \cdot G_i' \to S_i') \textbf{ od}. \quad \square$$

Our choice of G_i' is used also in [23], where those two rules are proved from first principles (that is, from Definition 3). We have shown therefore how that technique is an instance of our Theorem 3.

6.2. Functional Refinement

In many cases, the abstraction invariant is *functional* in the sense that for any concrete value there is at most one corresponding abstract value. In [13], for example, this is the primary form of data-refinement considered.

Functional abstraction invariants can always be written as a conjunction

$$(a = AF(c)) \wedge CI(c)$$

where AF we call the *abstraction function* and CI the *concrete invariant*; the formula CI of course contains no occurrences of abstract variables a. We assume that $CI(c)$ implies well-definedness of AF at c.

Functional data-refinements usually lead to simpler calculations. First, the concrete formula $(\exists a \cdot AI \wedge \phi)$ – where ϕ is *pre* or *post* in the abstract specification – is simplified:

$$\begin{aligned}&(\exists a \cdot AI \wedge \phi)\\ &= (\exists a \cdot (a = AF(c)) \wedge CI(c) \wedge \phi)\\ &= CI(c) \wedge \phi[a \backslash AF(c)].\end{aligned}$$

Thus in this case data-refinement calculations are no more than simple substitutions. Note also that the resulting concrete formula contains no free abstract variables, and this allows any $|[\mathbf{con}\ a \cdot \ldots]|$ to be eliminated immediately. We have this corollary of Theorem 2:

Corollary 3. *Functional data-refinement: Given an abstraction invariant* $(a = AF(c)) \wedge CI(c)$, *the following data-refinement is always valid:*

$$a, x : [pre, post] \preceq c, x : \left[\begin{matrix} pre[a \backslash AF(c)] \\ CI(c) \end{matrix}, \begin{matrix} post[a \backslash AF(c)] \\ CI(c) \end{matrix}\right].$$

Moreover, it is the most general. □

A second advantage is in the treatment of guards, as is shown also in [23]. According to Theorem 3 we replace G_i by $G_i \wedge AI$, which becomes

$$\begin{aligned}&G_i \wedge (a = AF(c)) \wedge CI(c)\\ &= G_i[a \backslash AF(c)] \wedge (a = AF(c)) \wedge CI(c).\end{aligned}$$

Now by Corollary 2, we can eliminate the conjunct $a = AF(c)$ immediately, and hence the enclosing $|[\mathbf{con}\ a \cdot \ldots]|$ as well. (And we can eliminate the $CI(c)$, but that is optional: it contains no a.) So we have the following result for the functional data-refinement of alternations:

Lemma 8. *Functional data-refinement of alternations: Given abstraction invariant* $(a = AF(c)) \wedge CI(c)$, *abstract guards* G_i, *and abstract statements* S_i, *let concrete guards* G_i' *and concrete statements* S_i' *be such that*

1. $G_i' = G_i[a \backslash AF(c)] \wedge CI(c)$,
2. $S_i \preceq S_i'$.

Then the following data refinement is valid

$$\mathbf{if}\ (\Box i \cdot G_i \to S_i)\ \mathbf{fi} \preceq \mathbf{if}\ (\Box i \cdot G_i' \to S_i')\ \mathbf{fi}.$$ □

The same remarks apply to iteration (and again, the conjunct $CI(c)$ is optional in the concrete guards):

Lemma 9. *Functional data-refinement of iterations: Under the same conditions as Lemma* 8, *the following data refinement is valid*

$$\textbf{do}\ (\Box i \cdot G_i \to S_i)\ \textbf{od} \preceq \textbf{do}\ (\Box i \cdot G'_i \to S'_i)\ \textbf{od}. \quad \square$$

6.3. Auxiliary Variables

A set of local variables is *auxiliary* if its members occur only in statements which assign to members of that set. They can be used for data refinement as follows.

There are three stages. In the first, an abstraction invariant is chosen, relating abstract variables to concrete. Declarations of those concrete variables are added to the program, but the declarations of the abstract variables are *not* removed. The initialisation is strengthened so that it implies the abstraction invariant; every guard is strengthened by conjoining the abstraction invariant; and every assignment statement is extended, if necessary, by assignments to concrete variables which *maintain* the abstraction invariant.

In the second stage, the program is *algorithmically* refined so that the abstract variables become auxiliary. In the third stage, the (now) auxiliary abstract variables are removed (their declarations too), leaving only the concrete – and the data-refinement is complete.

That technique was proposed by [15], and a simple example is given in [7, p. 64]. It also appears in [11] and [9]. It is a special case of our present technique, as we now show. Suppose our overall aim is the following data-refinement:

abstract variables:	a
concrete variables:	c
abstraction invariant:	AI.

We decompose this into two data-refinements, applied in succession. In the first, there are *no* abstract variables:

abstract variables:	(none)
concrete variables:	c
abstraction invariant:	AI.

Clearly this refinement removes no declarations. And, for an abstract program S to be taken to a concrete program S' under this refinement, Definition 3 requires only that for all ψ not containing c free, we have

$$AI \wedge [\![S]\!]\, \psi \Rightarrow [\![S']\!](AI \wedge \psi).$$

That is precisely the first stage explained informally above.

The second stage remains: it is only algorithmic refinement. For the third stage, we use the following data refinement in which there are no *concrete* variables:

abstract variables:	a
concrete variables:	(none)
abstraction invariant:	*true*.

For an abstract program S to be taken to a concrete program S' under this refinement, Definition 3 requires that for all formulae ψ

$$(\exists a \cdot [\![S]\!]\, \psi) \Rightarrow [\![S']\!](\exists a \cdot \psi).$$

And this holds only when the abstract variables a are auxiliary.

We illustrate the auxiliary technique with two lemmas, derived from our general rules for data refinement:

Lemma 10. *Introducing concrete variables while maintaining the invariant: Let the abstract variables be none, the concrete variables be c, and the abstraction invariant AI. Then for abstract expression AE and concrete expression CE, we have*

$$a := AE \preceq a, c := AE, CE$$

provided $AI \Rightarrow [\![a, c := AE, CE]\!]\, AI$.

Proof

$$\begin{array}{lll}
AI \wedge [\![a := AE]\!]\, \psi & & \\
\quad = AI \wedge \psi[a \backslash AE] & & \text{by semantics of } := \\
\quad \Rightarrow [\![a, c := AE, CE]\!]\, AI \wedge \psi[a \backslash AE] & & \text{by assumption} \\
\quad \Rightarrow AI[a, c \backslash AE, CE] \wedge \psi[a \backslash AE] & & \text{by semantics of } := \\
\quad = AI[a, c \backslash AE, CE] \wedge \psi[a, c \backslash AE, CE] & & \text{since } \psi \text{ contains no } c \\
\quad = [\![a, c := AE, CE]\!](AI \wedge \psi) & & \text{by semantics of } := \quad \square
\end{array}$$

Lemma 11. *Eliminating auxiliary variables: Let the abstract variables be a, the concrete variables be none, and the abstraction invariant true. Then*

1. $a := AE \preceq \mathbf{skip}$,
2. $c := CE \preceq c := CE$

provided CE contains no occurrence of a.

Proof. For 1 we have

$$\begin{array}{lll}
(\exists a \cdot [\![a := AE]\!]\, \psi) & & \\
\quad = (\exists a \cdot \psi[a \backslash AE]) & & \text{by semantics of } := \\
\quad \Rightarrow (\exists a \cdot \psi) & & \text{predicate calculus} \\
\quad = [\![\mathbf{skip}]\!](\exists a \cdot \psi). & &
\end{array}$$

For 2 we have

$$
\begin{array}{ll}
(\exists a \cdot [\![c := CE]\!]\, \psi) & \\
\quad = (\exists a \cdot \psi[c \backslash CE]) & \text{by semantics of } := \\
\quad = (\exists a \cdot \psi)[c \backslash CE] & \text{since } CE \text{ contains no } a \\
\quad = [\![c := CE]\!](\exists a \cdot \psi) &
\end{array}
$$

(Note that in case 2 we did not assume that ψ contained no c.) □

If the abstract statement is a specification a: $[pre, post]$, then in the first stage we replace it by a, c: $[pre \wedge AI,\ post \wedge AI]$. If by the third stage (after algorithmic refinement) we still have a specification – say a, c: $[pre', post']$, then the removal of a as an auxiliary variable leaves us with c: $[(\exists a \cdot pre'), (\exists a \cdot post')]$.

Let us as a final illustration try to remove a variable which is *not* auxiliary: we take the data-refinement as for the third stage, and suppose that $c := a \preceq CP$ for some concrete program CP. We expect this to fail, since a is clearly not auxiliary in $c := a$. Now we have for all constants n that

$$
\begin{array}{ll}
true & \\
\quad = (\exists c \cdot c = n) & \text{predicate calculus} \\
\quad = (\exists a \cdot (c = n)[c \backslash a]) & \text{renaming bound variable } c \text{ to } a \\
\quad = (\exists a \cdot [\![c := a]\!](c = n)) & \text{by semantics of } := \\
\quad \Rightarrow [\![CP]\!](\exists a \cdot c = n) & \text{by assumption} \\
\quad = [\![CP]\!](c = n). &
\end{array}
$$

Since the above holds for any n, we have that CP always establishes both $c = 0$ and $c = 1$. Because no executable program can do this, we have shown that there is no such CP – as hoped, a cannot be eliminated from $c := a$. But what if we write $c := a$ as a specification? In that case, Corollary 1 would allow us to perfrom the data refinement as follows.

$$
\begin{array}{l}
\quad c := a \\
= c\text{: } [true, c = a] \\
\preceq \text{Corollary 1 (noting the quantification is empty)} \\
\quad |[\textbf{con } a \cdot c\text{: } [true, c = a]]|.
\end{array}
$$

So here we *have* a data-refinement, after all. But that is consistent with the above in the following way: there is no executable program CP (whether containing a or not) such that c: $[true, c = a] \sqsubseteq CP$. Thus the $|[\textbf{con } a \cdot \ldots]|$ still cannot be eliminated.

In [16] the auxiliary variable technique is presented independently of the refinement calculus.

7. Conclusions

Our calculators for data refinement make it possible in principle to see that activity as the routine application of laws. The example of Sect. 5 is a demonstration for a simple case. It is important in practice, however, to take advantage

of the specialised techniques of Sect. 6; otherwise, the subsequent algorithmic refinement will simply repeat the derivation of the techniques themselves, again and again.

That subsequent algorithmic refinement is in fact a lingering problem. In many cases, particularly with larger and more sophisticated refinements, the refined operations present fearsome collections of formulae concerning data structures for which we do not have an adequate body of theory. Their subsequent manipulations in the predicate caclulus resemble programming in machine code. Fortunately, there is work on such theories (and *their* calculi, for example [5]), and we see little difficulty in taking advantage of them.

Our work on data refinement has been aided and improved by collaboration with Morris and Back, who present their work in [23] and [2] respectively. We extend Morris's approach by our use of logical constants (which, however, he has later discovered in another context [22]). A second extension is our "if and only if" result in Theorem 2. That is necessary, we feel, for a data refinement to be called a calculator: $P \leqq Q$ is a *calculator* only if taking Q loses no generality. And Morris retains some restrictions on abstraction invariants which we believe are unnecessary. Conversely, Morris's specialised alternation calculator [23, Theorem 4] improves ours (Lemma 6) by introducing a miracle as the refined program [17]; his rule needs no proof obligation. Our work extends Back's by our emphasis on calculation, and our use of logical constants.

Acknowledgements. We are grateful to have had the opportunity to discuss our work with Ralph Back and Joe Morris, and for the comments made by members of IFIP WG 2.3. Much of our contact with other researchers has been made possible by the generosity of British Petroleum Ltd.

References

1. Back, R.-J.: On the correctness of refinement steps in program development. Report A-1978-4, Department of Computer Science, University of Helsinki, 1978
2. Back, R.-J.: Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980
3. Back, R.-J.: Procedural abstraction in the refinement calculus. Report Ser. A 55, Departments of Information Processing and Mathematics, Swedish University of Åbo, Åbo, Finland, 1987
4. Back, R.-J.: A calculus of refinements for program derivations. *Acta Inf.* **25**, 593–624 (1988)
5. Bird, R.S.: An introduction to the theory of lists. Technical monograph PRG-56, Programming Research Group, 8–11 Keble Road, Oxford OX1 3QD, UK, October 1986
6. Boom, H.: A weaker precondition for loops. *Trans. Prog. Lang. Syst.* **4**, 668–677 (1982)
7. Dijkstra, E.W.: *A Discipline of Programming*. Englewood Cliffs: Prentice-Hall 1976
8. Gardiner, P.H.B., Morgan, C.C.: Data refinement of predicate transformers. *Theor. Comput. Sci.* (to appear). Reprinted in [21]
9. Gries, D., Prins, J.: A new notion of encapsulation. In: *Symp. Language Issues in Programming Environments*, SIGPLAN, June 1985
10. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**, 576–580 (1969)
11. Hoare, C.A.R.: Proof of correctness of data representations. *Acta Inf.* **1**, 271–281 (1972)
12. Hoare, C.A.R., He, J.F., Sanders, J.W.: Prespecification in data refinement. *Inf. Proc. Lett.* **25**, 71–76 (1987)
13. Jones, C.B.: *Systematic Software Development using VDM*. Prentice Hall, 1986
14. Josephs, M.B.: The data refinement calculator for *Z* specifications. *Inf. Process. Lett.* **27**, 29–33 (1988)

15. Lucas, P.: Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, 1968
16. Morgan, C.C.: Auxiliary variables in data refinement. *Inf. Process. Lett.* **29**, 293–296 (1988) (Reprinted in [21]
17. Morgan, C.C.: Data refinement using miracles. *Inf. Process. Lett.* **26**, 243–246 (1988) (Reprinted in [21]
18. Morgan, C.C.: Procedures, parameters, and abstraction: Separate concerns. *Sci. Comput. Progr.* **11**, 17–28 (1988) (Reprinted in [21])
19. Morgan, C.C.: The specification statement. *Trans. Prog. Lang. Syst.* **10** (1988) (Reprinted in [21])
20. Morgan, C.C., Robinson, K.A.: Specification statements and refinement. *IBM J. Res. Dev.* **31** (1987) (Reprinted in [21])
21. Morgan, C.C., Robinson, K.A., Gardiner, P.H.B.: On the refinement calculus. Technical Report PRG-70, Programming Research Group, 1988
22. Morris, J.M.: Invariance theorems for recursive procedures. (In draft)
23. Morris, J.M.: Laws of data refinement. *Acta Inf.* (to appear)
24. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Progr.* **9**, 298–306 (1987)
25. Nelson, G.: A generalization of Dijkstra's calculus. Technical Report 16, Digital Systems Research Center, April 1987

8. Appendix: Refinement Laws

Below is a collection of laws which can in principle take most specification statements through a series of refinements into executable code. We have not tried to make them complete. "Executable code" means program text which does not include either specification statements or logical constants.

"In principle" means that these basic rules, used alone, will in many cases give refinement sequences which are very long indeed – rather like calculating derivatives from first principles. But with experience, one collects a repertoire of more powerful and specific laws which make those calculations routine.

Some of the laws below are equalities $=$; some are proper refinements $\sqsubseteq$. In all cases they have been proved using the *weakest precondition* semantics of the constructs concerned.

Section 8.2 contains notes relating to the laws of Sect. 8.1.

8.1. Laws of Program Refinement

Most of these laws are extracted from [20], retaining only those used in this paper. Logical constant laws have been added.

1. *Weakening the precondition:* If $pre \Rightarrow pre'$ then

$$w: [pre, post] \sqsubseteq w: [pre', post]$$

2. *Strengthening the postcondition:* If $post' \Rightarrow post$ then

$$w: [pre, post] \sqsubseteq w: [pre, post']$$

3. *Assuming the precondition in the postcondition:* See Note 1.

$$w: [pre, (\exists w \cdot pre) \wedge post] = w: [pre, post]$$

4. *Introducing local variables:* If x does not appear free in *pre* or *post*, then

$$w: [pre, post] \sqsubseteq |[\mathbf{var}\ x \mid I \cdot w, x: [pre, post]]|.$$

5. *Strengthening the initialisation:* If $I' \Rightarrow I$, then

$$|[\mathbf{var}\ x \mid I \cdot S]| \sqsubseteq |[\mathbf{var}\ x \mid I' \cdot S]|.$$

6. *Introducing logical constants:* If x does not appear free in *post* or w, then

See Note 2.

$$w: [(\exists x \cdot pre), post] = |[\mathbf{con}\ x \cdot w: [pre, post]]|.$$

7. *Eliminating logical constants:* If x does not appear free in P, then

See Note 3.

$$|[\mathbf{con}\ x \cdot P]| = P$$

8. *Renaming logical constants:* If y is disjoint from w, and does not occur free in *pre* or *post*, then

$$\begin{aligned} &|[\mathbf{con}\ x \cdot w: [pre, post]]| \\ =&|[\mathbf{con}\ y \cdot w[x \backslash y]: [pre[x \backslash y], post[x \backslash y]]]| \end{aligned}$$

9. *Expanding the frame:* If y is a fresh variable then

$$w: [pre, post] = |[\mathbf{con}\ y \cdot w, x: [pre \wedge x = y, post \wedge x = y]]|.$$

10. *Contracting the frame:* If w and x are disjoint, then

$$w, x: [pre, post] \sqsubseteq w: [pre, post].$$

11. *Introducing* **skip**:

$$w: [post, post] \sqsubseteq \mathbf{skip}$$

12. *Introducing assignment:* If E is an expression, then

See Note 4.

$$w: [post[w \backslash E], post] \sqsubseteq w := E.$$

13. *Introducing alternation:*

$$\begin{aligned} &w: [pre \wedge (\vee i \cdot G_i), post] \\ =&\mathbf{if}\ (\Box i \cdot G_i \rightarrow w: [pre \wedge G_i, post])\ \mathbf{fi}. \end{aligned}$$

8.2. Notes

1. Law 3 applies when information from the precondition is needed in the postcondition. We use it below to derive a stronger version of Law 2:

   If $((\exists w \cdot pre) \wedge post') \Rightarrow post$, then
   $w: [pre, post]$
   $\sqsubseteq$ by Law 2 and the assumption

w: $[pre, (\exists w \cdot pre) \wedge post']$
$\sqsubseteq$ by Law 3
w: $[pre, post']$.

2. Usually Law 6 is used to introduce an equality into the precondition which "saves an initial value for later." That is summarised in the following derived law:

If y is disjoint from w, and does not occur free in pre or $post$, then
w: $[pre, post]$
$\sqsubseteq$ by Law 1
w: $[(\exists y \cdot x = y \wedge pre), post]$
$\sqsubseteq$ by Law 6
$|[\mathbf{con}\ y \cdot w: [x = y \wedge pre, post]]|$.

3. *Logical constants*, introduced by **con**, are variables which we can use during program development but not in final programs. Usually they are used to fix initial values, as in

$|[\mathbf{con}\ X \cdot x: [x = X, x = X + 1]]|$
$\sqsubseteq$ by Law 12
$|[\mathbf{con}\ X \cdot x := x + 1]|$
$\sqsubseteq$ by Law 7
$x := x + 1$.

Since the keyword **con** does not occur in our executable programming language – just as specification statements do not – it must be eliminated (using Law 7 as above) during the development process. Thus logical constants never appear in the final program, since they cannot be declared there.

4. Law 12 is usually applied together with Laws 10 and 1, as in the following derived rule:

If w is a subset of the set of variables v, E is an expression, and $pre \Rightarrow post[w \backslash E]$, then
v: $[pre, post]$
$\sqsubseteq$ by Law 10
w: $[pre, post]$
$\sqsubseteq$ by Law 1 and the assumption
w: $[post[w \backslash E], post]$
$\sqsubseteq$ by Law 12
$w := E$.