An Iterator Construct for the Refinement Calculus

Steve King

Department of Computer Science
University of York
Heslington, York YO10 5DD, UK
Email: king@cs.york.ac.uk

Carroll Morgan

School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia
Email: carrollm@cse.unsw.edu.au

June 4, 2000

Abstract

This paper concerns the introduction of an iterator into the refinement calculus. The construct is based on concepts from functional programming, and the work gives an interesting example of cross-fertilisation between the functional and imperative programming worlds. Specifically, the iterator construct **it..ti** uses the idea of a catamorphism — the unique homomorphism from an initial algebra. The datatype for which the iterator is to be defined is considered as an initial algebra of an appropriate functor. The **it..ti** construct is formally defined as a recursive procedure, and it is shown that, if the value to be obtained by an iteration can be expressed as a catamorphism, then the **it..ti** construct provides a very natural implementation. Examples are given to show typical uses of the new construct.

1 Introduction

Eckart [8] has described iteration as ‘the ability to consider every element of a data structure’. For instance, a company’s personnel database might contain a list of records, each of which contains details of an employee, such as name, address, salary and so on. Given this structure, we could use an iterator to define a procedure which would examine each record in turn, and reduce the salary of all those who earned more than \$50K by 20%. Alternatively, we could obtain a list of all those employees earning more than \$30K, or even the total wage bill for the company. These three examples illustrate three common forms of iteration: the first takes the form of an ‘update in place’; the second is a filter, producing a smaller collection of records; and the third produces a scalar value.

The key fact about an iterator construct is that its user should need no knowledge of how the structure is implemented: it should be possible to abstract from such details as whether the list is singly or doubly linked and whether it is stored in main or secondary storage. The interesting challenge is that we want users, while working within the refinement calculus, to be able to use iterator constructs on elements of types they have themselves defined, rather than simply on built-in types. We therefore eventually need to pass, as parameters to such constructs, information about the action to be applied to each element of the structure: this leads on to the study of procedural parameters. Although this is outside the scope of this paper, it is explored in [12].

The paper is structured as follows: after a brief introduction to the refinement calculus, we define the new iterator construct over sequences. Section 4 summarises the results about catamorphisms, and we then make explicit the relationship with the **it..ti** construct. We also look at iterators over more complex recursive types. Iterators have appeared in various programming languages, such as Alphard [25], CLU [15, 16] and object-oriented languages like Eiffel [20]. The specific iterator forms found in these languages are detailed in Section 8, together with a comparison with the new construct we have defined.

2 The refinement calculus

The refinement calculus arose out of a simple extension of Dijkstra's language of guarded commands [6]. A *specification*, here written

$$w : [\alpha, \beta]$$

comprises a frame w , and two predicates: the precondition α and the postcondition β . It is a command in the programming language which, like the others, describes the intended effect of a computation. Unlike conventional programming commands however, it does not necessarily suggest a mechanism for the computation: it gives the *what*, but not the *how*. In the refinement calculus world, we do not distinguish specifications from programs: every specification is also a program (but not *vice versa*).

In the specification $w : [\alpha, \beta]$, the frame w is a (possibly empty) list of variables that the specification (command) may alter. When the precondition α is true initially, the specification is guaranteed to terminate in a state satisfying the postcondition β . On the other hand, when α is not true initially, no guarantees can be made about the behaviour of the specification: it might terminate in an arbitrary state or it might not terminate at all.

For example:

- $y : [true, y^2 = x]$ is a specification which states that a value should be assigned to y to make the predicate $y^2 = x$ true (thus assigning to y a square root of x);
- $e : [s \neq \{\}, e \in s]$ is a command which chooses an element e from a set s , provided s is non-empty; if s is empty initially, then this command might not terminate, or it might assign an arbitrary value to e ; and
- $x : [b^2 \geq 4ac, ax^2 + bx + c = 0]$ is a command which solves the quadratic equation for x , provided the discriminant is non-negative; if the discriminant is less than 0, then its behaviour is arbitrary.

The meaning of a specification statement can be given in terms of Dijkstra's weakest precondition semantics [7]:

$$wp(w : [\alpha, \beta], \phi) \triangleq \alpha \wedge (\forall w \bullet \beta \Rightarrow \phi) \quad ,$$

which means that, for example,

$$wp(e : [s \neq \{\}, e \in s], \phi)$$

is equal to

$$s \neq \{\} \wedge (\forall e \bullet e \in s \Rightarrow \phi) \quad .$$

($\triangleq$ is used for definitions, and quantified predicates are enclosed in parentheses with the quantified variables separated from the predicate by a large dot $\bullet$.)

Apart from the specification statement, the second essential ingredient of the refinement calculus is a relation, called refinement, between programs. We write

$$aaa \sqsubseteq bbb \quad ,$$

for two programs aaa and bbb , to say that aaa is refined by bbb ; and that, in turn, means informally that any client who has asked for the program aaa will be happy if given bbb instead. Formally, the definition of the refinement relation between programs is given by weakest preconditions:

$$aaa \sqsubseteq bbb \triangleq wp(aaa, \phi) \Rightarrow wp(bbb, \phi) \text{ for all postconditions } \phi \quad .$$

For example, the first specification mentioned above, $y : [true, y^2 = x]$, could be refined by the program $y := \sqrt{x}$. On the other hand, it could also be refined by the program $y := -\sqrt{x}$: any client who had agreed that their needs were met by the original specification would have no grounds for complaint, whichever program they were given.

Program development in the refinement calculus is usually carried out via a series of so-called refinement steps, starting from a specification *aaa*, say, and ending with an executable program *zzz*. In between might occur a number of ‘hybrid’ programs, containing both specifications and executable fragments:

$$aaa \sqsubseteq \dots \sqsubseteq lll \sqsubseteq mmm \sqsubseteq nnn \sqsubseteq ooo \sqsubseteq \dots \sqsubseteq zzz .$$

The overall desired result $aaa \sqsubseteq zzz$ follows from the transitivity of $\sqsubseteq$.

Historically, Back was the first to embed specifications in programs, using the weakest precondition calculus [1, 2], although his specifications contained only a single predicate. A recent book [3] gives a full account of Back’s work, using a higher-order logic formalisation. In the late 1980s, both Morris [24] and Morgan [21] published extensions of Back’s original work, where they used separate pre- and postconditions. All three authors have the same refinement relation. The refinement calculus continues the tradition of Hoare [9] and Dijkstra [7]; for example, the meanings of the specification statement and the refinement relation were deliberately chosen to make true the following theorem (Theorem 3 of [21]):

Taking w to be all program variables, and *aaa* to be an executable program,

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

has exactly the same meaning as

$$(\forall w \bullet pre \Rightarrow wp(aaa, post)) .$$

This theorem allows us to check the validity of the laws of the refinement calculus, such as this law, for decomposing a specification into the sequential composition of two specifications:

$$w : [pre, post] \sqsubseteq w : [pre, mid] ; w : [mid, post] .$$

There is an extensive collection of laws such as the above, some with side-conditions to be proved, which are used to justify the refinement steps in a program development. A tutorial introduction to these laws may be found in [22], while a collection of more theoretical papers may be found in [23].

The constructs of the language that we use here are summarised in Figure 1, on the next page, together with their usual meanings in terms of standard weakest preconditions.

3 The *it..ti* construct for sequences

We consider first an iteration over a sequence s of type $\text{seq } A$, defined by

$$\text{type } \text{seq } A \hat{=} \text{Empty} \mid \text{Cons } A (\text{seq } A) ,$$

where we make use of the usual refinement calculus notation for disjoint union types: each element of the type is either an empty sequence, or it is constructed from an element of A and another sequence. (Further details may be found in [22, Chapter 15].) For brevity, and the convenience of an infix operator, we will often use the abbreviations

$$\begin{aligned} \langle \rangle &\hat{=} \text{Empty} \\ a:as &\hat{=} \text{Cons } a \text{ } as . \end{aligned}$$

	Syntax	Semantics
assignment	$x := E$	$wp(x := E, \phi) = \phi[x \backslash E]$
sequential composition	$aaa; bbb$	$wp(aaa; bbb, \phi) = wp(aaa, wp(bbb, \phi))$
alternation	$\text{if}(\llbracket i \bullet \alpha_i \rightarrow aaa_i \rrbracket \text{fi})^1$	$wp(\text{if}(\llbracket i \bullet \alpha_i \rightarrow aaa_i \rrbracket \text{fi}), \phi) = (\bigvee i \bullet \alpha_i) \wedge (\bigwedge i \bullet \alpha_i \Rightarrow wp(aaa_i, \phi))^2$
recursion specification	$\text{mu } aaa \bullet \mathcal{P}(aaa) \text{ um}$ $w : [\alpha, \beta]$	given by least fixed point $wp(w : [\alpha, \beta], \phi) = \alpha \wedge (\forall w \bullet \beta \Rightarrow \phi)$
skip	skip	$wp(\text{skip}, \phi) = \phi$
abort	abort	$wp(\text{abort}, \phi) = \text{false}$
nondeterministic choice	$aaa \parallel bbb$	$wp(aaa \parallel bbb, \phi) = wp(aaa, \phi) \wedge wp(bbb, \phi)$
naked guarded command	$\alpha \rightarrow aaa$	$wp(\alpha \rightarrow aaa, \phi) = \alpha \Rightarrow wp(aaa, \phi)$

Figure 1: The constructs of the refinement calculus and their wp meanings

The purpose of the iteration is to perform some calculation which involves the consideration of each element of the sequence in turn. We suppose that the result of the iteration is to be stored in a variable r , which is of some type R . A first example of the construct that we propose to use for the iteration is the following:

```

it s into r with
  ⟨ ⟩ → r := x
  ∥ a:as → r := f(a, as)
ti

```

The **it..ti** construct begins with a statement of the variable over which the iteration is to be performed, s , and the variable where the result is to be stored, r . This is followed by a collection of branches, one for each part of the disjoint union definition of the type of s .³ In this case, the first branch covers the empty sequence, and the second branch covers non-empty sequences made up of an element a , together with a sequence as . The interpretation which we intend for this construct is as follows: if the sequence s is empty, then the result variable r is updated with a (constant) value x ; alternatively, if s is not empty, then the new value of r is found by applying a binary function f to the first element of s and to the result of the iteration over the remainder of s — it will be clear from the definition below that this result is determined by a recursion.

This brings us to one of the interesting points about this construct — the dual use of as in the second branch. On the left of the arrow, as is a pattern matcher, while on the right it is the result of a recursive call. Use of this abbreviation has the advantage that we do not have to give a name to the function we are applying to s .

We can now give the definition for the **it..ti** construct in terms of a recursive procedure.

Definition 1 *sequence iterator*

¹This is a shorthand for

$$\text{if } \alpha_1 \rightarrow aaa_1 \parallel \dots \parallel \alpha_n \rightarrow aaa_n \text{ fi} .$$

The number of branches must be finite, but may be zero.

² $\bigvee$ and $\bigwedge$ denote distributed disjunction and conjunction, respectively.

³There are obvious similarities with the tagged alternation and iteration constructs of [22].

An iteration over a sequence s of the following form

```

it  $s$  into  $r$  with
     $\langle \rangle \rightarrow bbb$ 
     $\parallel a:as \rightarrow ccc$ 
ti

```

is defined as

$I(s, r)$,

where

```

procedure  $I(\text{value } s, \text{result } r) \hat{=}$ 
    if  $s$  is
         $\langle \rangle \rightarrow bbb$ 
         $\parallel a:as \rightarrow \parallel \text{var } l \bullet I(as, l); ccc[as \setminus l] \parallel$ 
    fi .

```

Notice that the local variable l is used to store the result of applying the procedure recursively to as , so that it can be used in the immediately following statement, by means of substitution. Since the recursive call is applied to the tail of s , we are guaranteed that the recursion will terminate.

For example, suppose we have the following declarations:

```

 $s : \text{seq } \mathbb{N}$ 
 $r : \mathbb{N}$ 

```

Then the following program fragment will obtain the sum of the sequence:

```

it  $s$  into  $r$  with
     $\langle \rangle \rightarrow r := 0$ 
     $\parallel n:ns \rightarrow r := n + ns$ 
ti

```

Similarly, we obtain the length:

```

it  $s$  into  $r$  with
     $\langle \rangle \rightarrow r := 0$ 
     $\parallel n:ns \rightarrow r := 1 + ns$ 
ti

```

4 Homomorphisms on initial algebras

To generalise iterators beyond sequences, and to expose the links between the two examples above, we use recent work in the functional programming community, which we summarise in this section. The work is based on homomorphisms — functions on recursively-defined data types whose inductive definition mimics the structure of the type. Further details may be found in [4, 17, 19].

First we give an informal indication of the direction of this work, before outlining its formal basis: since we are transferring work from functional programming into the refinement calculus, we give only a brief summary of the results required rather than the full details, which may be found in the papers cited.

4.1 An informal approach

Suppose we define a type T by

$$T \triangleq a A \mid b B T \mid c T C T .$$

This uses the previously defined types A , B and C and defines the constructor functions a , b and c , whose types are thus

$$\begin{aligned} a &: A \rightarrow T \\ b &: B \rightarrow T \rightarrow T \\ c &: T \rightarrow C \rightarrow T \rightarrow T . \end{aligned}$$

Each element of T can be thought of as ‘tagged’ with a constructor function.

Now, suppose that we wish to define a function on T , which will give as result an element of some type R , say $f : T \rightarrow R$. We can achieve this by defining three subsidiary functions, each designed to show the effect of f on the disjoint part of its domain corresponding to each of the constructor functions.

The simplest part of the domain is that formed by a : for this we define

$$f_a : A \rightarrow R$$

which maps every element of A to an element of R . Now to find the effect of f on an element of T of the form $a a$, we merely apply f_a to a .

The functions corresponding to b and c are slightly more complex, but the types of their domains are derived from the domains of the constructor functions, with each instance of T replaced by R (since, as the function is recursively applied, all of the elements of T in the lower-level structure have already been transformed to elements of R). So the subsidiary functions we need to define have the following types:

$$\begin{aligned} f_b &: B \rightarrow R \rightarrow R \\ f_c &: R \rightarrow C \rightarrow R \rightarrow R . \end{aligned}$$

Once we have given the functions f_a , f_b and f_c , we can combine them, using the so-called ‘banana brackets’ [17], to define a function from T to R :

$$f \triangleq ([f_a, f_b, f_c]) .$$

This function f can be applied to any element of T , however it has been constructed. Moreover, for any element of T , we can be sure that exactly one of the subsidiary functions is applicable.

We will use, as a concrete example running through this section, the type of non-empty lists of natural numbers. This is defined by

$$Natlist \triangleq Single \mathbb{N} \mid Cons \mathbb{N} Natlist ,$$

using the previously defined type $\mathbb{N}$ and the constructors **Single** and **Cons**.

For every function we want to define on *Natlist*, we need to give two subsidiary functions to show the effect on a singleton list, and on a longer list. Thus if we want to map an element of *Natlist* to its sum, we can define the function in two parts:

$$\begin{aligned} sum(Single n) &= n \\ sum(Cons n ns) &= n + sum(ns) . \end{aligned}$$

So the function *sum* can be defined:

$$sum = ([id, (+)]) .$$

Similarly, to obtain the product of the elements of a list, we define

$$product = ([id, (*)]) .$$

To add one to every element of a list, we define

$$inc\_list = ([inc, cons\_inc]) ,$$

where

$$\begin{aligned} inc\ n &= Single\ n + 1 \\ cons\_inc\ n\ ns &= Cons\ (n + 1)\ ns . \end{aligned}$$

Note that *inc_list* does not ‘update’ the list, but rather forms a new list of the desired values.

Applying these functions to the list $\langle 1, 2 \rangle$ gives

$$\begin{aligned} ([id, (+)])\ \langle 1, 2 \rangle &= 1 + 2 = 3 \\ ([id, (*)])\ \langle 1, 2 \rangle &= 1 * 2 = 2 \\ ([inc, cons\_inc])\ \langle 1, 2 \rangle &= \langle 1 + 1, 2 + 1 \rangle = \langle 2, 3 \rangle . \end{aligned}$$

4.2 Formal definitions

It is well-known that recursively-defined types, such as *Natlist* above, can be viewed as initial algebras of an appropriate functor [18, 5]. For instance, if we define the functor *F* by its action on objects (sets) and functions:

$$\begin{aligned} F(A) &= \mathbb{N} + (\mathbb{N} \times A) \\ F(f) &= id + (id \times f) , \end{aligned}$$

then

$$F(Natlist) = \mathbb{N} + (\mathbb{N} \times Natlist) .$$

Now the two constructor functions of *Natlist* can be combined into a single function with the join operator

$$[Single, Cons] : \mathbb{N} + (\mathbb{N} \times Natlist) \longrightarrow Natlist ,$$

and we can deduce that we therefore have an *F*-algebra, which consists of *FNatlist*, *Natlist* and the function between them $[Single, Cons]$.

For any two *F*-algebras $f : FA \longrightarrow A$ and $g : FB \longrightarrow B$, an *F*-homomorphism from *f* to *g* is an arrow $h : A \longrightarrow B$ such that

$$h \cdot f = g \cdot Fh ,$$

which expresses equationally that the following diagram commutes:

$$\begin{array}{ccc} FB & \xrightarrow{g} & B \\ \uparrow Fh & & \uparrow h \\ FA & \xrightarrow{f} & A \end{array}$$

The join $[\text{Single}, \text{Cons}]$ is actually defined to be the initial algebra of F . It is therefore possible, given any other F -algebra — say g — to find a unique F -homomorphism from the initial algebra to g . This concept of the ‘unique homomorphism from an initial algebra’ is the basis of our iterator construct, and is called a catamorphism [17]. It is usually written with the ‘banana brackets’ mentioned above: $\llbracket g \rrbracket$. A simple way of thinking about catamorphisms is that the functions given between the brackets $\llbracket \rrbracket$ are used as ‘replacements’ for the constructor functions of the catamorphism’s argument.

5 Catamorphisms and the **it..ti** construct

We now give the connection between catamorphisms and the **it..ti** construct which we introduced earlier. From the definition of **it..ti** above in terms of a recursive procedure, we can prove a law which will allow us to carry out developments where we implement an assignment with an iterator. Note that we are now returning to work with the sequence type defined at the start of Section 3.

Law 1 *assignment iterator*

If the value to be assigned to a variable is formed by the application of a catamorphism to a sequence, then the whole assignment can be implemented with an **it..ti** construct.

$$\begin{array}{l} r := \llbracket f, g \rrbracket s \\ \sqsubseteq \\ \text{it } s \text{ into } r \text{ with} \\ \quad \langle \rangle \rightarrow r := f \\ \quad \llbracket a:as \rrbracket \rightarrow r := g(a, as) \\ \text{ti} \end{array}$$

Proof: By the definition of **it..ti**, it is enough to prove that

$$\begin{array}{l} r := \llbracket f, g \rrbracket s \\ \sqsubseteq \\ I(s, r) \end{array}$$

where, as before,

$$\begin{array}{l} \text{procedure } I(\text{value } s, \text{result } r) \hat{=} \\ \quad \text{if } s \text{ is} \\ \quad \quad \langle \rangle \rightarrow r := f \\ \quad \quad \llbracket a:as \rrbracket \rightarrow \llbracket \text{var } l \bullet I(as, l); r := g(a, l) \rrbracket \\ \quad \text{fi} \end{array}$$

We therefore develop the assignment until it is transformed into a recursive procedure:⁴

$$\begin{array}{l} r := \llbracket f, g \rrbracket s \\ \sqsubseteq \text{re } I(\text{value } s, \text{result } r) \text{ variant } V \text{ is } \#s \bullet \\ \quad \{V = \#s\} \\ \quad r := \llbracket f, g \rrbracket s \end{array}$$

---

⁴Note that we are using the refinement rule for recursion from the second edition of Morgan’s text [22]. We also use a technique of layout often found in refinement calculus developments: certain lines of the development are labelled with numbers, and these labels are used to continue the derivation at a later stage. We also sometimes label lines with a $\triangleleft$ symbol: this signifies that this line will be worked on in the very next step of the development. The complete program can eventually be found by collecting the code fragments from the branches of the resulting development tree.

$$\begin{aligned}
& \sqsubseteq r : [V = \#s, r = F(s)] \\
& \sqsubseteq \text{“tagged alternation”} \\
& \quad \text{if } s \text{ is} \\
& \quad \quad \langle \rangle \longrightarrow r : [s = \langle \rangle \wedge V = \#s, r = F(s)] \quad \triangleleft \\
& \quad \quad \parallel a:as \longrightarrow r : [s = a:as \wedge V = \#s, r = F(s)] \quad (1) \\
& \quad \text{fi} \\
& \sqsubseteq \text{“by definition of } F, \text{ and conversion to recursive form”} \\
& \quad r := f \\
(1) & \sqsubseteq \text{“by definition of } F, \text{ and conversion to recursive form”} \\
& \quad r : [s = a:as \wedge V = \#s, r = g(a, F(as))] \\
& \sqsubseteq \text{var } l \bullet \\
& \quad l : [s = a:as \wedge V = \#s, l = F(as) \wedge s = a:as \wedge V = \#s]; \quad (2) \\
& \quad r : [s = a:as \wedge V = \#s \wedge l = F(as), r = g(a, F(as))] \quad \triangleleft \\
& \sqsubseteq r := g(a, l) \\
(2) & \sqsubseteq l : [V > \#as \geq 0, l = F(as)] \\
& \sqsubseteq I(as, l) \quad \text{End of Proof}
\end{aligned}$$

We can use this law to give some very simple examples of iterations over sequences. Suppose we have the following declarations:

$s : \text{seq } \mathbb{N}$
 $r : \mathbb{N}$.

Then we can develop simple iterators by referring directly to the catamorphism-style definitions of the functions concerned. For example, for the sum of a sequence:

$$\begin{aligned}
& r := \Sigma s \\
= & r := ([0, (+)])s \\
& \sqsubseteq \text{“assignment iterator 1”} \\
& \quad \text{it } s \text{ into } r \text{ with} \\
& \quad \quad \langle \rangle \longrightarrow r := 0 \\
& \quad \quad \parallel n:ns \longrightarrow r := n + ns \\
& \quad \text{ti}
\end{aligned}$$

What we are doing here is to import refinements⁵ like

$$\begin{aligned}
& r := \Sigma s \\
= & r := ([0, (+)])s
\end{aligned}$$

directly from other theories: we are able to use the literature on catamorphisms to simplify the development of our own iterators.

---

⁵actually an equality, in this case

The length of a sequence can also be obtained with an iterator:

```

    r := #s
=   “define  $a \oplus b = 1 + b$ ”
    r :=  $([0, \oplus])s$ 
 $\sqsubseteq$  “assignment iterator 1”
    it s into r with
         $\langle \rangle \rightarrow r := 0$ 
         $\parallel n:ns \rightarrow r := n \oplus ns$ 
    ti
 $\sqsubseteq$  r := 1 + ns

```

6 Iterators over more general data types

We now work in a more general framework, with an arbitrary recursive data type. The type we use is defined schematically by

type $T \triangleq a \mid b X \mid c Y T$.

Thus an element of T is either a constant, identified by a , or it is the image of an element of some set X , tagged by b , or it is formed from an element of Y and some other element of T and is tagged by c . It will become clear how the definitions and refinement laws can be extended from a type with these three ‘typical’ branches to any other recursive type.

First we extend Definition *sequence iterator 1*.

Definition 2 *general iterator*

If t is any element of the type T defined above, then

```

it t into r with
    a       $\rightarrow aaa$ 
     $\parallel b x \rightarrow bbb$ 
     $\parallel c y t' \rightarrow ccc$ 
ti

```

is defined to mean the same as

$I(t, r)$,

where

```

procedure  $I(\text{value } t, \text{result } r) \triangleq$ 
    if  $t$  is
        a       $\rightarrow aaa$ 
         $\parallel b x \rightarrow bbb$ 
         $\parallel c y t' \rightarrow \parallel \text{var } l \bullet I(t', l); ccc[t' \setminus l] \parallel$ 
    fi

```

Notice that the local variable is only needed on the third branch, where there is a recursive occurrence of T in the type definition. If there were several occurrences within one branch, then an equal number of local variables would be required.

Having extended the definition of an iterator itself, we can also extend the law which introduces an iterator as a refinement of an assignment:

Law 2 *assignment iterator*

$$\begin{array}{l}
 r := ([P, Q, R]) t \\
 \sqsubseteq \\
 \text{it } t \text{ into } r \text{ with} \\
 \quad a \quad \rightarrow r := P \\
 \quad \parallel b x \quad \rightarrow r := Q(x) \\
 \quad \parallel c y t' \quad \rightarrow r := R(y, t') \\
 \text{ti}
 \end{array}$$

7 Refinement of branches

Now that we have these laws over more complicated data types, we can use them to develop some more sophisticated examples. The observant reader may be wondering about the point of having “ $r :=$ ” in each branch of the **it..ti** construct. In fact, it is useful to have a program fragment (rather than an expression) in each branch, because it gives scope for further refinement: in cases where the expression being assigned to the result variable — for instance, $R(y, t')$ in the **it..ti** construct in Law *assignment iterator* 2 — cannot be easily evaluated in the target language, the assignment $r := R(y, t')$ can be refined until it is code. (In program developments where we are using libraries of abstract data types, it is also likely that we will want to refine branches until they can be replaced by calls of library procedures.)

We now give an example which shows the idea of this refinement of branches. A more substantial example can be found in [12]. The example here is based on an example in a paper about iterators in the CLU programming language [16]. The task is to count how many numeric characters are contained in a string which might also contain alphabetic characters.

We need to define two disjoint union types for characters, which are either alphabetic or numeric, and for strings, which are either empty, or contain a character and a string:

```

type
  Char  $\hat{=}$  alph Alpha | num Numeric
  String  $\hat{=}$  empty | ch Char String

```

(We assume that *Alpha* and *Numeric* have been suitably defined.)

We define first an infix operator $\oplus$ which will form part of the catamorphism:

$$\begin{array}{l}
 (\text{alph } c) \oplus n = n \\
 (\text{num } c) \oplus n = n + 1 \quad .
 \end{array}$$

Now it is clear that, if *count_num* is the function which, when applied to a string, returns the desired number of numeric characters, then

$$\text{count\_num} = ([0, \oplus]) \quad .$$

We can therefore immediately introduce an iterator, as follows:

$$i := ([0, \oplus]) s$$

$\sqsubseteq$ “assignment iterator 2”
it s **into** i **with**
     empty $\rightarrow i := 0$
      $\parallel$ $ch\ c\ cs \rightarrow i := c \oplus cs$ $\triangleleft$
ti

Since the expression $c \oplus cs$ is not immediately implementable, we need to refine the second branch, which is not difficult using a tagged alternation:

$\sqsubseteq$ “tagged alternation”
if c **is** num $\rightarrow$
      $i : [c = \text{num } n, i = c \oplus cs]$ (1)
 $\parallel$ c **is** alph $\rightarrow$
      $i : [c = \text{alph } a, i = c \oplus cs]$ (2)
fi

The two branches of the tagged alternation are easily implemented, using the definition of $\oplus$:

(1) $\sqsubseteq i := cs + 1$

(2) $\sqsubseteq i := cs$.

This completes the development, giving overall:

$i := \text{count\_num}(s)$
 $\sqsubseteq$ **it** s **into** i **with**
     empty $\rightarrow i := 0$
      $\parallel$ $ch\ c\ cs \rightarrow$ **if** c **is** num $\rightarrow i := cs + 1$
                              $\parallel$ c **is** alph $\rightarrow i := cs$
                             fi
ti

8 Related work on iterators

Several programming languages include some sort of iterator mechanism, and we review a selection of them in this section. There has been much less work on formalising iterators — that is, providing a formal semantics and a mechanism by which an iterator construct can be proved correct. Where such work has been carried out for a particular language, it is mentioned below.

8.1 Alphard

The Alphard language [25], developed at CMU in the late 1970s, has two iterator constructs: a **for** construct which is used for iteration over a complete data structure; and a **first** construct which is used (primarily) for search loops. We will concentrate on the **for** construct, remarking only that execution of the **first** construct involves traversing a data structure until an element is found which meets some condition, and then performing some action. The **for** construct takes the following form:

for $x : \text{gen}(y)$ **while** $\beta(x)$ **do** $ST(x, y, z)$.

A local variable x is declared, which will take, in turn, the values specified by the generator $gen(y)$. For each value of x which satisfies the constraint β , the statement ST is executed, which may refer to other variables in scope (z) as well as x and y . Clearly, the heart of the **for** construct lies in the idea of the generator $gen(y)$, so we look a little more closely at this. A generator is a ‘form’ (the Alghard term for abstract data type) obeying certain conditions: it must provide two boolean-valued functions $\&init$ and $\&next$, which have the side-effect that their invocation will produce a sequence of values to be bound to the loop variable. For both functions, the boolean value returned indicates whether there are elements remaining which have yet to be iterated over.

Thus the meaning of the **for** construct can be given:

```

begin local  $x : gen(y)$ ;
     $\pi \leftarrow x.\&init$ ;
    while  $\pi$  cand  $\beta(x)$  do
        ( $ST(x, y, z)$ ;  $\pi \leftarrow x.\&next$ )
    end

```

where π is a compiler-generated boolean variable, $\leftarrow$ denotes assignment, and *cand* denotes the ‘conditional conjunction’ operator.

A simple example of an Alghard generator is the *upto* generator, which produces the sequence of numbers between a lower bound lb and an upper bound ub — that is, $\langle lb, lb + 1, \dots, ub \rangle$, or the empty sequence if $ub < lb$. This generator can be used in a **for** statement. For instance, summing the first n integers is achieved by:

```

 $sum \leftarrow 0$ ; for  $j : upto(1, n)$  do  $sum \leftarrow sum + j$  .

```

It is possible to add verification information to an Alghard form, using an invariant clause, an initially clause, and pre and post conditions for each function. It is also possible to give a concrete version of the state, and to give a representation function relating the concrete to the abstract view. Proof obligations can then be given to ensure the correctness of the form: this means ensuring that invariants are maintained by abstract and concrete operations, and that initial states correspond. Since a generator is just a special sort of form, it is possible to apply these proof rules to a generator. Using the expansion of the **for** construct given above, it is also possible to obtain a proof rule for the construct. However it is rather unwieldy. Because of this, there are various simpler proof rules for the **for** construct, which can be used when the generator satisfies certain conditions. Many generators do satisfy these conditions, so the full form of the proof rule is seldom needed. Thus the effort of the proof is transferred from the verification of the **for** construct, to verification that the generator obeys the necessary conditions.

When comparing the Alghard generator with our own **it..ti** construct, we can see two immediate differences from our own work. Firstly, we have hidden all the details of how to ‘move on’ to the next element of the collection, by using a recursive procedure. Thus users of the iterator have no need to know anything about the internal details of the object over which they are iterating, beyond its definition. Secondly, we have considerable flexibility with the **it..ti** construct: we have the freedom to choose whatever ‘view’ is most convenient of the type of the variable to be iterated over. However, in Alghard, once the $\&next$ function is defined, the iteration order is fixed. On the other hand, it should be noted that the Alghard mechanism has the advantage that it is easy to describe a generator that produces only part of some structure to be iterated over — perhaps every other element of a list. While possible, this would be more convoluted with **it..ti**.

8.2 CLU

At around the same time that Alghard was being developed at CMU, Barbara Liskov’s team at MIT was developing a language called CLU [15]. One of the guiding principles behind CLU was that it should support abstraction in program construction [16]. The language contains mechanisms to support three forms of abstraction:

- procedural abstraction — supported by procedures;
- data abstraction — supported by the use of clusters, the CLU term for abstract data type; and
- control abstraction — as well as the usual **if** and **while** constructs, iterators can be defined.

An iterator is therefore a procedure-like construct, at the same level as procedures and clusters. Like Alphard, iterators are used in conjunction with **for** statements: as the iterator produces elements of a data structure one at a time, so the **for** statement consumes them.

The syntax of the definition of an iterator in CLU is:

```
id = iter [parms] args [yields] [signals] [where]
      routine_body
end id .
```

The *yields* clause specifies the number, order and types of the objects which will be delivered at each stage of the iteration. Within the *routine_body*, a *yield* statement is used to present the caller (a **for** statement) with the next element. (The *signals* clause specifies which exceptions may be raised, and the *where* clause specifies **own** variables.)

The CLU **for** statement takes the following form:

```
for [id1, ...] in invocation do body end ,
```

where *invocation* is the invocation of an iterator. Unlike Alphard, where the looping mechanism is found in the **for** statement, in CLU the looping must be explicitly programmed in the body of the iterator.⁶ Each time a **yield** statement is executed in the iterator's body, the objects yielded are assigned to the variables declared in the **for** statement, and the body of the **for** statement is executed. Then the iterator body is resumed at the point immediately following the **yield** statement. The **for** statement terminates on termination of the iterator.

In her thesis [26], Wing gave a method for specifying iterators, which has been extended to iterators for concurrent and distributed systems [27]. Wing's technique for specifying CLU iterators involves adding assertions to an iterator, similar to those used for a procedure, to give pre- and postconditions for each invocation. However, unlike a procedure, an iterator's specification is concerned with more than just two states — as well as the overall first and last states, there are the intermediate states for each invocation. There also needs to be a distinction between two kinds of termination for iterators — the 'real' termination when all the elements have been yielded, and the suspension that occurs after each **yield**. The assertions refer to state variables which can be decorated with subscripts *pre* and *post*, as well as a special state object (ie an auxiliary variable) *first*, which flags when we are in the very first state, and history variables which 'remember' values between invocations. Both of the remarks made above when comparing Alphard iterators to the **it..ti** construct still hold true for CLU iterators: the user has to program explicitly the method of progress through the collection, and the **it..ti** is more flexible. However there is more generality here, in that it is possible for the user of the CLU iterator to write a **yield** statement which returns a more complex expression than simply the current object in the collection — for instance, each **yield** statement might return a pair of consecutive characters in the sequence, allowing a **for** statement to calculate the frequency of pairs of characters. This would be considerably more complex using an **it..ti**.

On the verification side, while Wing's assertions and associated proof rules do allow the verification of a CLU iterator, the proof is at a very low level, dealing with the intermediate states during the iteration as well as the overall pre- and postconditions. In contrast, the effort of verification for the **it..ti** construct is at a much higher level, involving the reformulation of the postcondition as a catamorphism. Once this has been done, implementation as an **it..ti** is immediate, by *assignment iterator* 1, or a similar law for other types.

---

⁶In fact, it is usual to use another, more primitive, **for** statement in the body of an iterator.

8.3 Object-oriented languages

In recent years, many object-oriented environments have introduced libraries of abstract data types, often including container classes. These container classes, which describe such types as sets, bags, trees, etc., often contain some sort of iteration mechanism, whereby the user can traverse the data structure. However, these mechanisms are often based on the notion of a cursor, with the user having to supply cursor manipulation routines. As an example, we will describe the iteration mechanisms available in Eiffel [20].

The Eiffel library contains an Iteration Library which consists of classes which encapsulate various iteration mechanisms over arbitrary data structures — linear iteration, two-way iteration, tree iteration (preorder, postorder or inorder). These iterations are defined in terms of two sorts of deferred routines — these are routines which are called in the iterator, but not actually defined until the iterator is used: traversal routines and operation routines. The traversal routines are concerned with cursors, and need only be defined once for each data type. The operation routines are concerned with the particular actions to be taken as part of each iteration, and so can be given different values to achieve different iterations.

There are difficulties with the cursor approach to iterators, not least the problems of nested iterations, when it is not easy to keep track of several cursors, and the problem of robust iterations, when elements may be added or removed during an iteration. There has been work reported to solve these problems: [10] proposes a CLU-like mechanism for Eiffel and [11] does the same for C++, while [13] is concerned with robust iterators in a C++ class library.

8.4 Discussion

The chief disadvantage of the Alphard mechanism and the more recent proposals for object-oriented languages is that they require the user of the iterator to supply routines to control the iteration. This means that knowledge of the data structure's implementation is required, thus negating one of the primary advantages of using the iterator in the first place. The CLU mechanism is much cleaner in that respect, but the complications in the proof obligations caused by the suspend/resume semantics are non-trivial. Lamb has proposed [14] the use of trace specifications for the specification of Alphard-style iterators. He also mentions the use of procedure parameters, and shows how traces can be used to give 'partial specifications' of iterators which use procedure parameters.

However, the main difference between the **it..ti** construct and all of the related work mentioned in this section is in the level of abstraction and the level of mathematical maturity required for their use: the iteration schemes for Alphard, CLU and Eiffel all require the user of the iterator to supply routines which need knowledge of the internal structure of the type of the variable being iterated over. Users of the **it..ti** can work at a more abstract level, but need to be more mathematically mature: while most average programmers can easily understand the ideas of initialising and advancing a cursor, they may have more difficulty with catamorphisms! However, they would probably benefit from thinking more deeply about the constructors of both the type of the variable to be iterated over and the target type of the iteration.

9 Conclusions

In this paper, we have introduced the iterator construct **it..ti**. The construct is based on the idea of a catamorphism, but is formally defined as a recursive procedure. Several examples were given, showing how particular functions can be seen as catamorphisms, and therefore implemented by an **it..ti** construct. Because the construct is closely based on catamorphisms, it is possible to take results from the functional programming community about catamorphisms and use them to simplify our developments. For instance, the so-called Fusion Law [5, Equation 2.12] can be used to

convert catamorphisms over one structure to those over another. [5, Chapter 7] contains several examples of the use of catamorphisms in optimisation problems.

In Section 6, we explained how the **it..ti** construct for sequences could be extended to act on a more general data type. The type T used there is intended to be a typical example of a type generated by a polynomial functor. This form of functor — formed from constants, products and coproducts — is general enough for our purposes, and we are guaranteed the existence of an initial object in the category of F -algebras (see [18]).

At the start of the paper, we gave illustrations of three common forms of iteration: the ‘update in place’, the filter and the scalar result. Most of the **it..ti** examples in the paper are of the scalar-result type — the sum or the length of a sequence, or the counting numeric characters example in Section 7. A fairly lengthy example of the filter type of iterator can be found in [12, Chapter 5], and it is not hard to see that the first form of iteration can be obtained by following an **it..ti** with an assignment of the result back to the original variable. For instance, in Section 4, we introduced the *inc_list* catamorphism:

$$inc\_list = \langle inc, cons\_inc \rangle ,$$

where *inc* simply forms a singleton list from the increment of its argument. Suppose that we had to implement the following ‘update in place’ form of iteration:

$$s := inc\_list(s) ,$$

adding one to each element of the non-empty list of numbers s . It is easy to see that this is achieved by the following program:

```

var  $r : Natlist$  •
it  $s$  into  $r$  with
     $Single\ n \rightarrow r := Single\ n + 1$ 
     $\parallel\ Cons\ n\ ns \rightarrow r := Cons\ (n + 1)\ ns$ 
ti;
 $s := r$ 

```

Thus all three common forms of iteration can be expressed in terms of our **it..ti** construct.

It can be argued that the most interesting feature of the construct is its close relationship with the type of the variable being iterated over: it is the constructor functions of this type and of the result type which define the underlying catamorphism, and the user of the iterator is not concerned with implementation details such as how to move on to the next element of the data structure.

References

- [1] R.-J.R. Back. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.
- [2] R.-J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
- [3] R.-J.R. Back and J. von Wright. *Refinement calculus: a systematic introduction*. Graduate texts in computer science. Springer, 1998. ISBN 0387984178.
- [4] R.S. Bird. Functional algorithm design. In B. Möller, editor, *Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

- [5] R.S. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1997. ISBN 013507245X.
- [6] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, August 1975.
- [7] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.
- [8] J.D. Eckart. Iteration and abstract data types. *SIGPLAN Notices*, 22(4), April 1987.
- [9] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.
- [10] M. Katrib and I. Martinez. Collections and iterators in Eiffel. *Journal of Object-Oriented Programming*, pages 45–51, November-December 1993.
- [11] M.H. Kim. A new iterator mechanism for the C++ programming language. *ACM SIGPLAN Notices*, 30(1), January 1995.
- [12] S. King. *Higher-level Algorithmic Structures in the Refinement Calculus*. DPhil thesis, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK, 1998. Also published as OUCL Technical Monograph PRG-124.
- [13] T. Kofler. Robust iterators in ET++. *Structured Programming*, 14:62–85, 1993.
- [14] D.A. Lamb. Specification of iterators. *IEEE Trans. Soft. Eng.*, 16(12):1352–1360, December 1990.
- [15] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Number 114 in Lecture Notes in Computer Science. Springer-Verlag, 1981.
- [16] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Comm. ACM*, 20(8):564–576, August 1977.
- [17] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
- [18] E.G. Manes and M.A. Arbib. *Algebraic approaches to program semantics*. Texts and monographs in computer science. Springer-Verlag, 1986.
- [19] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
- [20] B. Meyer. *Eiffel: the Language*. Prentice-Hall, 1992.
- [21] C.C. Morgan. The specification statement. *Trans. Prog. Lang. Sys.*, 10(3), July 1988. Reprinted in [23].
- [22] C.C. Morgan. *Programming from Specifications*. Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 2nd edition, 1994.
- [23] C.C. Morgan and T.N. Vickers, editors. *On the Refinement Calculus*. FACIT Series in Computer Science. Springer-Verlag, 1993.
- [24] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
- [25] M. Shaw, Wm.A. Wulf, and R.L. London. Abstraction and verification in Alphard: iteration and generators. In M. Shaw, editor, *Alphard: form and content*, pages 73–116. Springer-Verlag, 1981.
- [26] J.M. Wing. *A two-tiered approach to specifying programs*. PhD thesis, MIT, Lab. for Comp.Sci., 1983.
- [27] J.M. Wing and D.C. Steere. Specifying iterators for concurrent and distributed systems. Unpublished draft, 1994.