SPUDS: Simple Pictures for Updating Data Structures

Todd Cochrane
School of Information Technology
Wellington Institute of Technology
Petone, NZ
Todd.Cochrane@weltec.ac.nz
Paul Lyons
Giovanni Moretti
Institute of Information Sciences and Technology
Massey University
Palmerston North, NZ

ABSTRACT

We introduce SPUDS (Simple Pictures for Updating Data Structure) – a construct used in the visual programming language HyperPascal to simplify programming of dynamic data structure manipulations. Each SPUD comprises a picture of a data structure - a linked list, for example - before and after an operation. The After-Picture in a SPUD defines an updated version of the data structure which should be generated if the Before-Picture matches the data structure's current configuration. An algorithm such as a linked list insertion (or an AVL tree insertion) can be constructed from a set of SPUDS.

Keywords

Interface Design, HCI, VPL, visual programming languages, HyperPascal, SPUDS.

1. INTRODUCTION

This paper describes SPUDS (Simple Pictures for Updating Data Structures), a construct used in the visual programming language HyperPascal [1] [4] to simplify programming of dynamic data structure manipulations.

Before describing SPUDS themselves, we will give the "flavour" of HyperPascal using a very simple example. Note that the simplicity of the example does not imply any restriction on the language's capabilities. HyperPascal is a superset of Pascal, which is a suitable language for experimenting with visual language constructs and demonstrating their wide applicability, because it has a simple syntax, but is also a "real" language that has been widely used for building applications. For example, the modern, and widely used, commercial IDE, Delphi, incorporates the programming language Object Pascal.

In HyperPascal, declarations reside in a "scope tree". This is a tree of subroutines, with the main program at the root (Figure 1). Each subroutine contains the declarations of its local identifiers; one of the identifiers is the subroutine name; its declaration (i.e., the code for the subroutine) is important enough to warrant its own representation, the "action tree" (Figure 2 The Action Tree defines the data manipulation

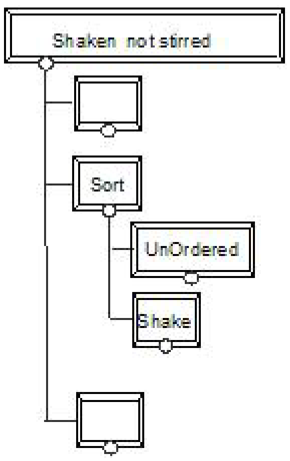


Figure 1. A scope tree shows the whole program as a tree of subroutines.

Proceedings of the 16th Annual NACCQ, Palmerston North New Zealand July, 2003 (eds) Mann, S. and Williamson, A. www.naccq.ac.nz

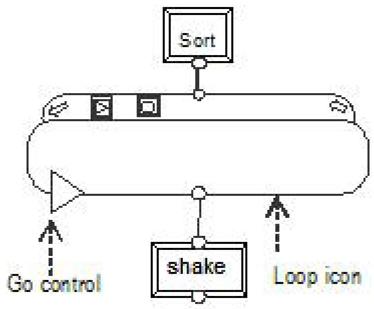


Figure 2. An Action Tree representing the Sort subroutine, showing, from top to bottom, a subroutine header, a loop icon, and subroutine ("Shake") that is called repreatedly till the expression in the GO control evaluates to FALSE.

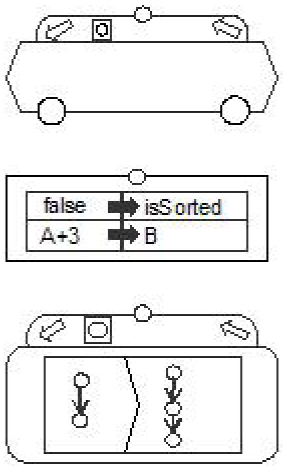


Figure 3. Other control icons; a Choice icon, an Action Sequence icon and a data structure traversal icon.

operations that occur in the subroutine and their order of execution).

Figure 2 shows a sort subroutine (the top box is the subroutine header) that uses a loop icon (the round-ended rectangle with a superimposed triangle) to control repeated calls to the shake subroutine (the bottom box is the subroutine invocation; the shake subroutine that sorts a list by the simplistic technique of randomly re-ordering it, then checking to see whether it is in order).

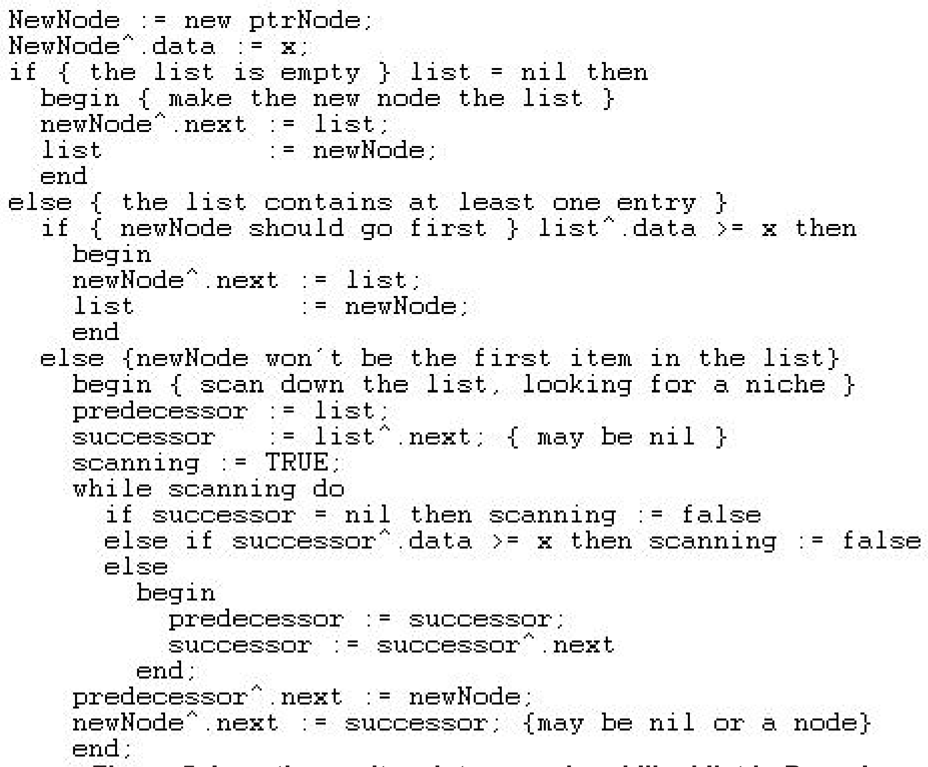
A subroutine is represented as a tree of icons. Some processing is associated with each icon, and the icons are executed in modified depth-first traversal, under the control of the loop, choice and structure-traversal icons. The loop icon repeatedly executes its children in left-to-right order while the value of the Boolean expression in its GO control (Figure 2 contains one GO control - the right-pointing arrow – green on the screen - in the loop icon) is TRUE and the value of the Boolean expressions in its STOP controls (red octagons that also can be placed along the bottom of the loop control – none are currently visible, though a tool for adding them can be seen at the top of the loop icon) evaluate to FALSE. An arbitrary number of STOP and GO controls can be added anywhere along the bottom of the loop control.

A Choice icon (exemplified by the hexagonal icon shown in Figure 3) depicts a number of Boolean expressions along its base. Two are shown in Figure 3, the second contains ELSE, which makes the icon equivalent to an IF-statement with an ELSE clause; adding more makes it a nested IF-statement). A subtree of the choice icon is attached to each Boolean expression. The Boolean expressions are evaluated until one returns a TRUE result, whereupon control is passed to its subtree. None of the other subtrees is executed.

An Action Sequence icon (the rectangular icon in Figure 3) contains an arbitrary number of assignments. They have a donor field on the left and an acceptor field on the right, which is inconsistent with many other programming languages, but internally consistent with other HyperPascal constructs.

A data structure traversal icon (Figure 3, and subsequent discussion) has a number of Before-After Picture-Pairs as its children. Each Before-After Picture-Pair depicts a section of a dynamic data structure such as a linked list, before and after an operation on the data structure. The Before-Picture in a Before-After Picture-Pair represent a Boolean test on part of the data structure. The Before-Pictures inside the Before-After Picture-Pairs attached to a data structure traversal icon are evaluated until one succeeds, whereupon the data structure is modified to make it identical to the image shown in the associated After-Picture – and no more of the Before-Pictures attached to the data structure traversal icon are evaluated.

Control of the program loop is determined by the instance of a "Go control" placed prior to the branch representing the call to "Shake". The "Go control" evaluates a Boolean expression to determine continued execution of the loop. The loop icon also provides a "Stop control" that determines termination of the loop. Figure 4 depicts the programme with the "Go" control exploded; showing an expression that evaluates the function "UnOrdered". "Testlist" is the actual parameter that is passed in to the formal parameter "list". The value returned from "UnOrdered" is a Boolean that is



```
NewNode := new ptrNode;
NewNode^.data := x;
if { the list is empty } list = nil then
  begin { make the new node the list }
  newNode^.next := list;
  list          := newNode;
  end
else { the list contains at least one entry }
  if { newNode should go first } list^.data >= x then
    begin
    newNode^.next := list;
    list          := newNode;
    end
  else {newNode won´t be the first item in the list}
    begin { scan down the list, looking for a niche }
    predecessor := list;
    successor   := list^.next; { may be nil }
    scanning := TRUE;
    while scanning do
      if successor = nil then scanning := false
      else if successor^.data >= x then scanning := false
      else
        begin
          predecessor := successor;
          successor := successor^.next
        end;
    predecessor^.next := newNode;
    newNode^.next := successor; {may be nil or a node}
    end;
```

Figure 5. Inserting an item into an ordered liked list in Pascal.

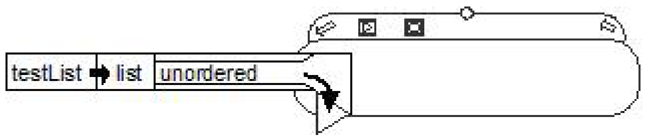


Figure 4. The loop icon with the GO control subicon expanded.

used to determine continued execution in the branches.

2. DATA STRUCTURE TRAVERSAL AND MANIPULATION USING SPUDS

The HyperPascal data structure traversal icon, and the related construct, the Before-After Picture-Pair, together comprise a formal visual representation for manipulations on dynamic data structures such as linked lists. They largely replace the "dot-and-caret" notation (e.g., thisNode^.data) used in Pascal and many other languages, which is regarded as opaque by beginning and experienced programmers alike. An earlier paper [2] has described how it is possible to examine the Boolean expressions in Before-Pictures and make inferences that can be used to automate a large part of the programming associated with dynamic data structure manipulations. Here we concentrate on how the constructs fit into the rest of the language.

Operations on dynamic data structures in conventional text programming languages use an impenetrable "dot-and-caret" notation to dereference pointer-based structures. For example, in Pascal inserting an item into an ordered linked list could be expressed as in Figure 5.

This notation very sensitive to minor alterations such as reordering of pointer assignments, and consequently most programmers steer clear of dynamic data structures if at all possible.

In teaching programming, it is conventional to clarify the operations on a data structure by showing them pictorially. For example, a linked list insertion is often illustrated by a picture such as the one in Figure 6, which shows the general case where the new node is preceded by a node with a value les than x, and followed by a node with a value greater than x. The programmer is left to work out the other cases (insertion into an empty list, as the first node in the list, or insertion at the end of the list, for her or himself). We have invented SPUDS, a formal notation for programming such operations visually. Figure 3 presents a combined the before-and-after picture of the data structure which has been separated into separate diagrams in the SPUD visualisation. Each diagram shows a "snapshot" of the data structure. The Before-Picture is effectively a complex Boolean expression that acts as a guard; if the data structure matches the image shown in the Before-Picture, an updated version of the data structure is generated, that matches the image shown in the After-Picture. In other words, the Before-After Picture-Pair can be interpreted as an if-statement that says "if the data structure looks like the Before-Picture, then make it look like the After-Picture."

From the description so far, it might seem as though a SPUD has to contain a representation of a complete data structure; if this were the case, it would be difficult to use SPUDS to program operations on arbitrarily complex data structures, because a separate SPUD would have to be drawn for every possible configuration of the data structure. Later in the paper, we introduce the notion of clouds, which allow the programmer to navigate to an interesting part of the data structure, and "hide" the uninteresting parts, so that the Before-Picture in a SPUD only needs to match a small part of the large data structure. Clouds are a more intuitive replacement for the temporary pointers (predecessor and successor in the example given earlier) that give programmers such trouble when they are manipulating data structures in conventional textual syntaxes.

A data structure manipulation algorithm, such as a linked list insertion, comprises a group of SPUDS under the control of a data structure traversal icon. This specialised version of the loop control icon executes each of the SPUDS attached to it. When a SPUD whose Before-Picture matches the data structure is executed, its After-Picture is also executed, and the current iteration of the loop ceases. That is, no more of the children are evaluated on that iteration. In general, an algorithm will start with a navigation phase, in which Before-Pictures that show "uninteresting" parts of the data structure match, and After-Pictures that hide the uninteresting parts of the data structure in a cloud are executed. Eventually, a stage will be reached when the unhidden part of the data structure is interesting; a different Before-Picture will match, and its corresponding After-Picture will be executed, updating part of the data structure, or extracting a value from it. The algorithm will then cease. The programmer discriminates between such a terminating SPUD and the others that merely effect navigation by attaching a stop sign (a red octagon) at the top of the link between the SPUD and the parent data structure. Thus when a SPUD with a stop sign at the top of its link is executed, the data structure traversal icon ceases to operate

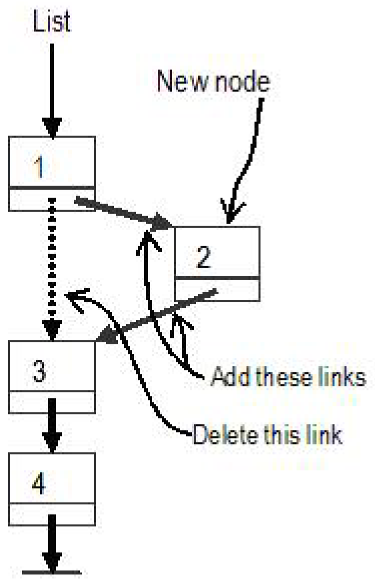


Figure 6. A diagram typical of those used in teaching texts to illustrate linked list insertion

The ease of use of SPUDS depends upon the ability of the HyperPascal editor to draw data structures with a minimum of programmer effort. The editor has a small drawing package (not unlike the table drawing tools in Microsoft Word). When declaring a pointer, the programmer uses this tool to draw an example of the type of node the pointer references. From this picture, the editor extracts enough information to draw a data structure based on these nodes, under the control of the programmer. Because of space limitations, graphical declarations are not described further in this paper.

3. EDITING BEFORE AND AFTER PICTURES – SPECIFIC NODES

A SPUD is presented as a box split into two regions by a broad arrowhead. The left region represents the Before-Picture and the right region represents the After-Picture. Clicking anywhere in the left region presents a list box into which the programmer inserts the name of a declared pointer variable. The HyperPascal editor places an arrow-tail next to the name (Figure 7 (A)). The shaft is like a stalk that, when dragged, extends down the diagram. It is dragged a short distance, at present between 5 pixels and 10 pixels, down the Before-Picture to indicate that the pointer is a nil pointer. This is depicted by placing an arrow head touching a horizontal line at the bottom of the shaft

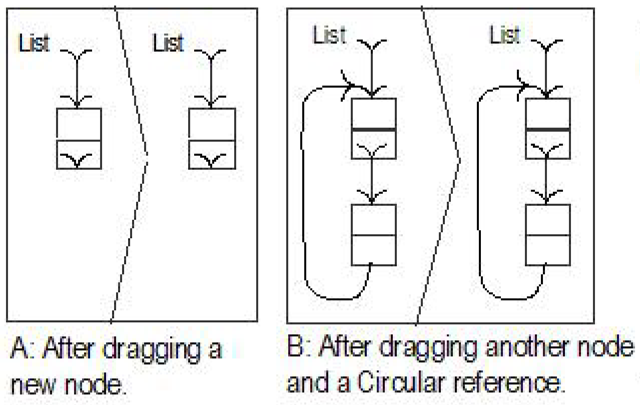

List
List
A: After dragging a new node.
B: After dragging another node and a Circular reference.

Figure 7. Creating an empty list in a SPUD.

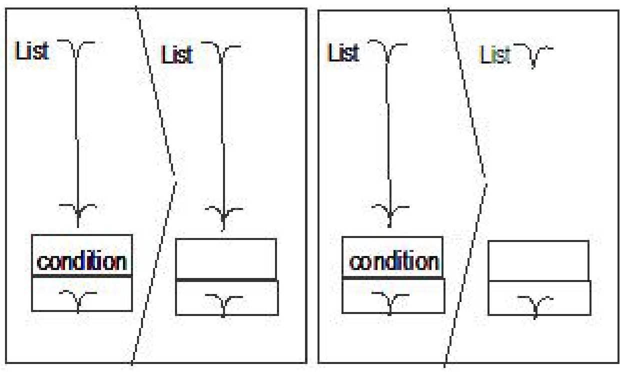


Figure 9. A pointer is detached from the node it originally referenced…

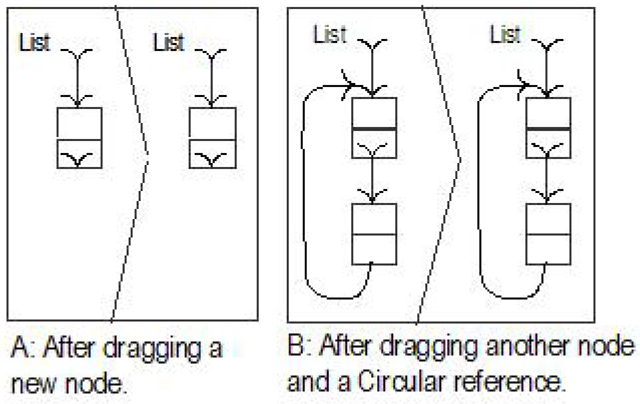

List
List
A: After dragging a new node.
B: After dragging another node and a Circular reference.

Figure 8. Adding nodes to the list, and making it circular.

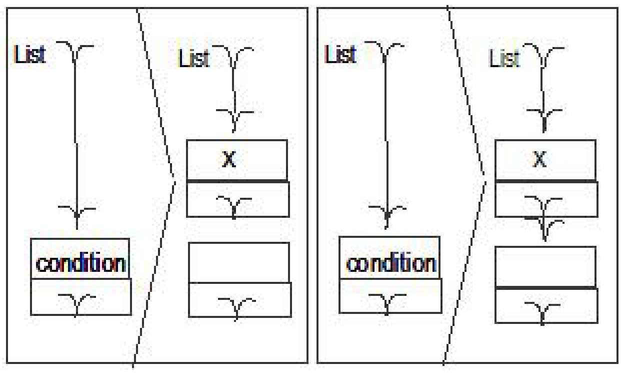


Figure 10. ...dragged out to form a new node, and the new node is linked back into the list.

(Figure 7(B)). If the programmer drags the shaft beyond the "nil pointer region," a new node is drawn attached to the shaft (Figure 7(A)). Pointers fields in a node are depicted with an arrow-tail ready for dragging out to new nodes or nil terminated pointers (Figure 8(A)). The shaft can also be dragged to a node of the correct type (Figure 8(B)). The type of the node pointed to can be as complex as required. For example a node of a binary tree has two pointer fields. As the diagram in the Before-Picture is extended to produce a more descriptive picture of a data structure, an identical picture is automatically created in the After-Picture; the programmer can later edit this to depict the new configuration of the data structure.

The After-Picture of a SPUD depicts the desired state of the data structure if the associated Before-Picture matches. Nodes, nil terminated pointers and circular references are indicated in the After-Picture as on a Before-Picture. A programmer may also drag an arrow off an existing node (Figure 9) and release it in its node creation region to create a new node. The new node can then be connected back into the list, depicting the insertion of the new node (Figure 10).

To delete a node from the list, the programmer drags another pointer to the tail of the pointer in the doomed node's next field (This has the effect of copying the value in the next field to the other pointer) The programmer can – and should – also destroy the "hanging node" by attaching a "destructor" (a set of teeth, in the current version of Hyperpascal) to the node (both of these operations have been shown in Figure 11).

In all operations depicted in the diagrams from Figure 6 to, the Before-Picture shows a node immediately following the start of the list. An operation on nodes further down the list can be depicted by inserting the specific number of nodes up to the point of the operation. As pointed out previously, this is not feasible for large data structures.

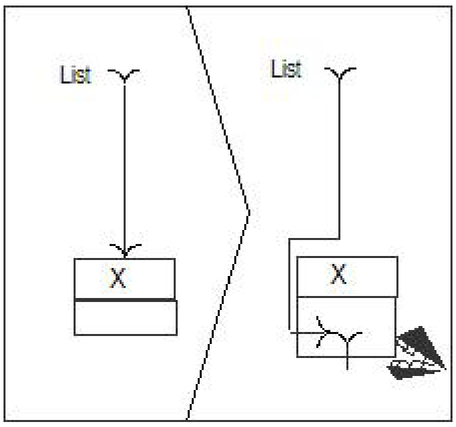


Figure 11. Deleting a node from a list. Copying the value from its next field pointer by dragging another pointer onto the tail of the pointer in the next field, then deleting the node by attaching a destructor to it.

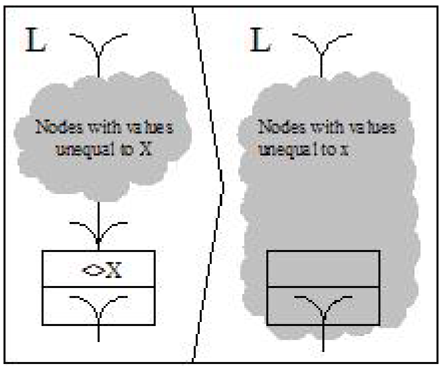

L
L
Nodes with values unequal to X
Nodes with values unequal to x
<>X

Figure 12. Traversing a list by hiding an "uninteresting" node inside a cloud. At the next iteration, the Before-Picture will examine the next node in the list.

4. GENERALISING TRAVERSAL, CLOUDS AND THE DATA STRUCTURE TRAVERSAL ICON

When the programmer wishes to specify that an operation shown in a SPUD occurs at an arbitrary point in a data structure, and not at the first node after the entry pointer, she or her draws a data structure traversal SPUD, in which the "uninteresting" parts of the data structure get hidden inside a cloud[4]. In this SPUD both the Before-Picture and the After-Picture show a cloud, but the cloud encloses a node in the After-Picture. On the next iteration of the data structure traversal loop, the node that was enclosed on the current iteration is assumed to be hidden inside the cloud, so the Before-Pictures start matching one node further into the data structure. This has the same effect as updating a temporary pointer in a textual language (currentNode := currentNode^.next), but is much more intuitive. And whereas in conventional representations, multiple temporary pointers are often required, a single cloud suffices with SPUDS.

To effect this, HyperPascal creates a temporary pointer when a data structure traversal icon is entered. This temporary pointer is initialised to the same value as the entry pointer for the data structure (usually the pointer that the programmer thinks of as the name of the data structure, L in Figure 12), and on each iteration of the loop (except the last), the temporary pointer is updated. This means that the cloud is empty at the start of the traversal and encloses more and more of the data structure. HyperPascal also keeps a reference to the last item in the cloud, so that its pointer can be updated if the first item in the After-Picture differs from the first item in the Before-Picture. That is, if the pointer "out of" the cloud gets redirected in the After-Picture, the last node "in" the cloud needs to have its next pointer updated. When the cloud is empty (as it would be the first time a list insertion routine is executed), the data structure entry pointer (L in Figure 11) is the pointer to be updated

5. SUMMARY

In HyperPascal, simple pictures called SPUDS represent the individual steps in a visual representation of a data structure manipulation algorithm. The representation is based on before and after picures of the data structure, and can be described by the statement "if the data structure looks like the Before-Picure, make it look like the After-Picture. The generality that comes from the use of temporary pointers in textual languages is achieved, more intuitively, by hiding parts of the data structures in clouds. The algorithm is under the control of a data structure traversal icon.

REFERENCES

[1] Lyons P J, Simmons C, Apperley M D, 1993 "HyperPascal: A Visual Language to Model Idea Space" In Proceedings of the 13th New Zealand Computer Society Conference, August 1993, pp 492-508.

[2] Lyons, P J, Apperley M D, Bishop A G, Moretti, G. S. 1994 “Active Templates Manipulating pointers with pictures” In Proceedings of the Computer Human Interaction Special Interest Group of the Ergonomics Society of Australia, OZCHI ’94, Melbourne, November 1994.

[3] Lyons P J, 1999 “Programming in several dimensions” In Proc. SoftVis ’99, (Ed A Quigley), December 3-4, 1999, 31-39.

[4] He Whakatauaki: En ngaro te kupua iti he tini whetu. Behind one small cloud many stars may be hidden.