Enhancing Show-and-Tell with a polymorphic type system and higher-order functions

Marc A. Najork Eric Golin
Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

We describe enhancements to the visual dataflow language Show-and-Tell (STL). These enhancements enrich STL by a polymorphic type system similar to the one used in ML, and they introduce user-definable higher-order functions.

1 Introduction

Show-and-Tell (abbrev. STL) [1] is a visual programming language based on dataflow and completion. It was primarily targeted at elementary school children. We believe that a visual syntax is a powerful way of communicating the meaning of a program, and that the benefits of visual programming can be realized by a wide range of users. We have enhanced STL by introducing features oriented to the sophisticated programmer. Our new language is called ESTL, for Enhanced-Show-and-Tell.

The first objective of this paper is to introduce a strong, polymorphic type system for STL that allows static type checking. The type system includes related functions (e.g. constructors and selectors), and visualization of typed objects. Our second objective is provide a more convenient and intuitive mechanism for handling higher-order functions.

In the following sections, we first give a short overview of Show-and-Tell, and then describe our enhancements to the language. These enhancements fall into four categories:

(1) A polymorphic type system for STL that stems from ideas incorporated into a multitude of today's functional languages, like ML or Miranda.
(2) The application of our typing mechanism to the definition of (first-order) functions. We introduce a number of primitive functions and show how to define derived functions.
(3) A notation for functions with variable arity.
(4) An elegant visualization for higher-order functions that improves STL's Lisp-style capabilities of quoting functions.

The final section summarizes the findings of this paper and gives an outlook on future research.

The Show-and-Tell language

Show-and-Tell, developed by Kimura et al. [1] [2], is a visual language based on the dataflow paradigm and the completion principle [3].

Constants, variables and operations are shown as *boxes*. Data flows from boxes to other boxes through *pipes* depicted as arrows. In this paper, we will refer to the interface between a box and a pipe as a *port* (this is a slightly more general notion of a port than the one used by Kimura). A picture composed of boxes and pipes is called a *puzzle*. STL tries to *complete* this puzzle by performing every possible dataflow.

If data flows into a box already containing a different value, the box becomes *inconsistent*. Inconsistency can be limited to a simple box, or it can "flow out" of this box and turn its spatial environment inconsistent as well. Inconsistent areas are shaded grey and are considered to be removed from the diagram. If a pipe leads through an inconsistent area, no data can pass through it. This novel notion of inconsistency can be utilized in many ways, in particular, it fulfills the same purposes as a conditional or selection function in traditional languages.

Function diagrams in STL can be associated with user-defined icons, which provides *named functions*. These functions may be recursive. Conditionals, recursion, and variable-sized data structures guarantee that STL is as powerful as Turing machines, the lambda calculus or other models of computation.

In addition, STL allows iteration through the *Iterator* construct. STL distinguishes between *sequential iterators*, *parallel iterators*, and combinations thereof. An iterator is visualized by an icon with fat borderlines containing a function diagram.

Sequential iterators have a number of input ports, shown as inward pointing triangles located on one side of the iterator, and an equal number of output ports, depicted as outward pointing triangles located at the opposite side of the iterator. The iteration process, termed *unfolding of the iterator*, goes as follows: The function diagram within the iterator is evaluated, using the data provided by the input ports. If it becomes inconsistent, the data flows directly from the input ports to the output ports. Otherwise the output of this function diagram is connected to the input ports of an identical iterator.

Parallel iterators can import or export data through parallel ports. If a list data item with *n* elements provides the input for a parallel port, the corresponding iterator will be unfolded into *n* similar function diagrams. If an iterator outputs one item through a parallel port during each of *n* iterations, the overall result will be a list with n elements.

Fig. 1 shows an example STL program containing a sequential and parallel iterator. After its execution, the two upper empty variable boxes will contain a 6, the lower one will contain the list ⟨1,2,3,4,5⟩, and the area of the iterator will be shaded in order to indicate that it turned inconsistent.

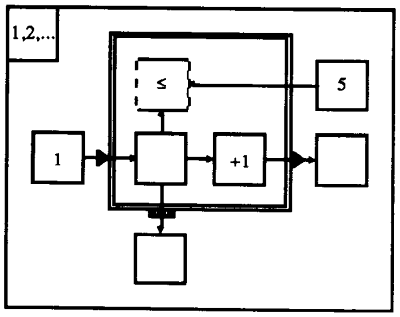

Fig. 1 : Example STL program

Iterators are an example of higher-order functions. Sequential iterators act as while-loops (or as fixed-point operators for tail-recursive functions), parallel iterators share many features with various map functions. In section 4, we will introduce a syntactic enhancement that allows the user to define iterators as higher-order functions.

2 A static polymorphic type system for STL

Type Definitions

ESTL is based on the type system proposed by Milner [4] for functional languages. This type system has been incorporated into a multitude of modern typed functional languages, like ML, Miranda, or Haskell.

Types are visualized by grey *type icons*. ESTL has four predefined base types: Integers, Reals, Characters, and Booleans (ESTL differs in this respect from STL, which does not have a boolean type, but handles all uses of booleans through the concept of consistency). Fig. 2 shows the representations of these predefined types.

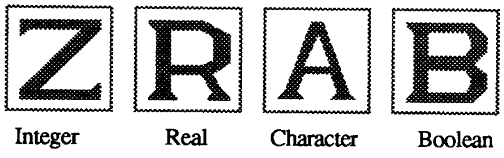

Fig. 2 : Predefined simple types

ESTL offers two families of *typemakers* that compose simpler types into higher ones:

- a typemaker family for *tuple (record) types*
- a typemaker family for *union types*

Typemakers are spatial arrangements of types. The top-level typemaker is enclosed by a grey box. A *tuple typemaker* is an area enclosing zero or more other type icons or type diagrams. A *union typemaker* is an area, divided into several non-overlaying subareas by broken lines. Each area represents one alternative and can contain zero or more other type icons or type diagrams. Fig. 3 shows examples of these typemakers.

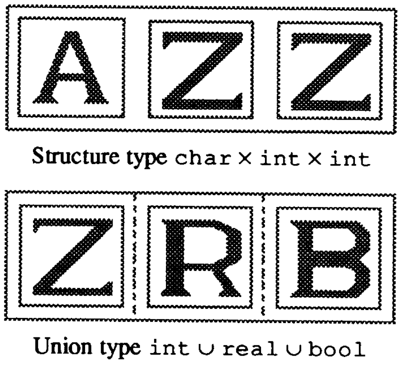

Fig. 3 : Typemakers

A *type diagram* or *type formula* is a set of types composed into one type by using typemakers. Type diagrams can be associated with a new type icon, i.e. they can be named. The type icon and its corresponding diagram constitute a type definition. As was the case with function definitions, type definitions can be recursive, so the name of a type can appear in its type diagram.

In addition to the predefined types discussed before, ESTL provides the *void* type. The void type is defined as a zero-tuple (which means that there is only one void value). The void type icon is an all-grey box, it names the empty diagram. Fig. 4 shows the definition of the void type.

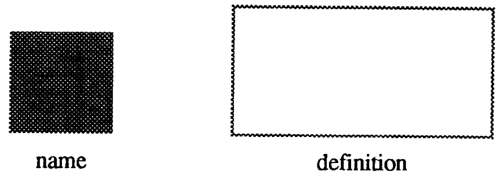

Fig. 4 : The void type

Using the union typemaker, the type naming mechanism, and 0-tuples, we can easily define *enumeration types*. For example, Fig. 5 shows an enumeration type t={day,night}. Each of the enumerants is first defined as a constant type (i.e. having an empty definition), and the enumeration is simply their union.

Union types together with a type naming mechanism also allow us to construct recursive types, like lists or trees. Trees, lists, stacks, queues, etc. are examples of polymorphic data types. We would like to be able to define a tree of T, where T is an arbitrary type, and

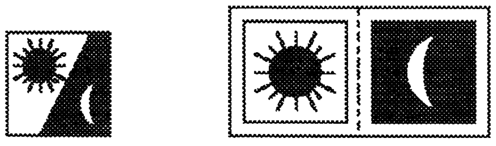

Fig. 5 : Enumeration type definition

then use this polymorphic tree type by simply "plugging in" an appropriate type for T, instead of defining instances of the tree type (or, more correctly, tree type scheme) from scratch.

A *type variable* is a placeholder for a type that can be used in a type diagram wherever a type icon could be used. Fig. 6 shows the predefined icon for a type variable. If a type definition involves more than one type variable, the T will have appropriate subscripts.

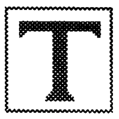

Fig. 6 : Type variable

To provide a mechanism for specifying the variables in a type definition, we have generalized the notion of *type icon*: an icon can be a *simple icon* (like those shown in Fig. 2) or a *structured type icon.* A structured type icon is a grey icon that consists of a pictorial part identifying the icon and one or more *slots* that can be filled with type variables (when naming a polymorphic type) or type icons (when creating an instance of a polymorphic type). All the type variables used in a type definition must also appear in the type's name. Fig. 7 shows the definition of a polymorphic tree type, and an instance of this polymorphic type, a tree of characters. Fig. 8 shows a polymorphic list type.

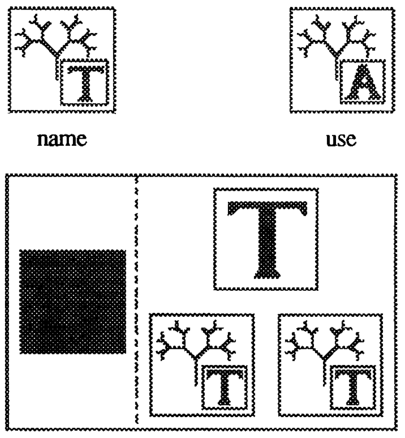

Fig. 7 : Polymorphic tree type

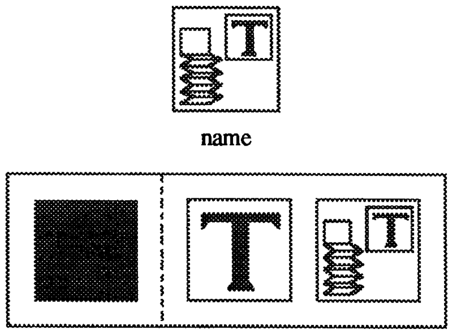

Fig. 8 : Polymorphic list type

These components — a few primitive types, tuple and union typemakers, type variables and a type naming mechanism — give us a type system equivalent in its expressive power to the one described by Milner [4], modulo mappings. In fact, our type system may be viewed as a visual syntax for Milner's. This type definition system can be used to type the arguments and the results of an ESTL function (represented by an *operation icon*). ESTL is a strongly typed language and does not provide automatic type coercion. Therefore the types of variables, arguments, and function results can be inferred automatically; type declarations are optional. Within a function diagram, the type of a variable box is shown by filling the black variable box with a grey type representation, which may be either a type icon or a type diagram. Fig. 9 shows the name and the definition of a recursively defined factorial function, with types specified.

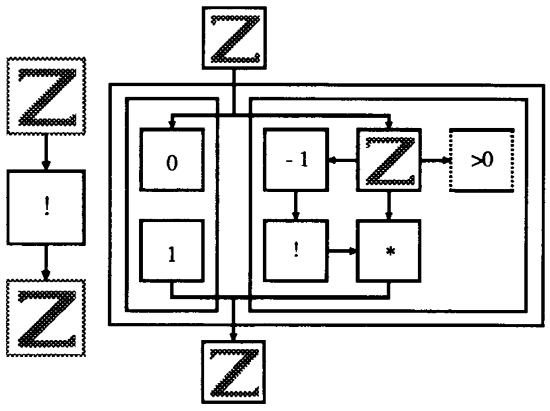

Fig. 9 : Typing of the factorial function

Constructors, Selectors, and Filters

There are two families of functions related to the type system. For every record type, there is an associated record constructor/selector function, and for every union type there is a type filter.

A record constructor/selector "function" (RCS) is a relation between a tuple type and its constituents types. An RCS for some n-tuple $T=T_1 \times ... \times T_n$ has ports on two sides: the ***structure side*** has one port with type T; the ***element side*** has n ports typed T_1 through T_n. An n–tuple input on the structure side will return its elements on the element side. Data items input into the element side will output an n-tuple on the structure side. Ports on the element side which have not received any data will lead to uninstantiated elements in the n-tuple. If both the structure and the element side receive data items, conflicts will lead to inconsistency.

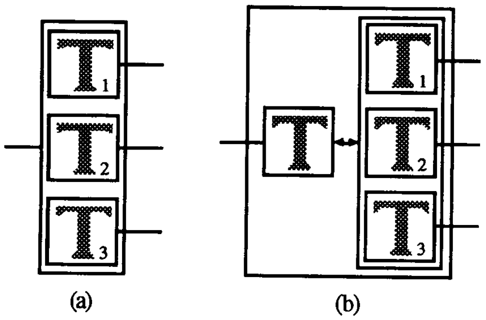

Fig. 10 : Record constructors/selectors

Fig. 10 shows two variations of RCS's. The RCS in Fig. 10.a constructs anonymous record types. Fig. 10.b visualizes the type of the elements of the tuple as well as the name of the resulting tuple type, T in this case.

Like an RCS, a type filter is a relation, in this case between a union type and its constituent types. A type filter for the union type $T=T_1 \cup ... \cup T_n$ also has ports on two sides: the ***union side*** with one port of type T, and the ***element side*** with n ports of type T_1 to T_n. A data item flowing in on the element side goes out on the union side tagged with its subtype T_n. If more than one port on the element side receives data, the type filter becomes inconsistent. Data flowing into the union side comes out on the port of the element side with the compatible type. Fig. 11 shows two variations of type filters, similar to the two variations of RCS.

Record constructors and type filters can be both combined in one icon, much as a single type diagram can both contain records and unions. Fig. 12 shows an example program using a combined record constructor/ selector and type filter. This program takes a data item of type `list of int` and sums the elements of this list. The incoming list is either empty or nonempty. In the first case, the void part of the type filter remains consistent, while the other part turns inconsistent. A `nil` value flows out of the type filter and into a variable box. This variable box is grouped together with a box containing the constant 0. As there are no more empty

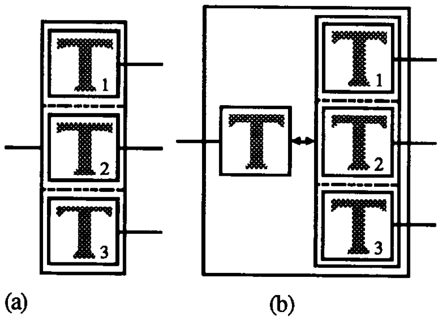

Fig. 11 : Type filters

variable boxes in this group, the 0 flows out of the group and is returned as a result of the function call. If the incoming list is nonempty, the void-part of the type filter becomes inconsistent and the upper part of the type filter remains consistent. The tail of the list is forwarded to a recursive occurrence of the sum function, the result of which is added to the head of the list, producing the final result.

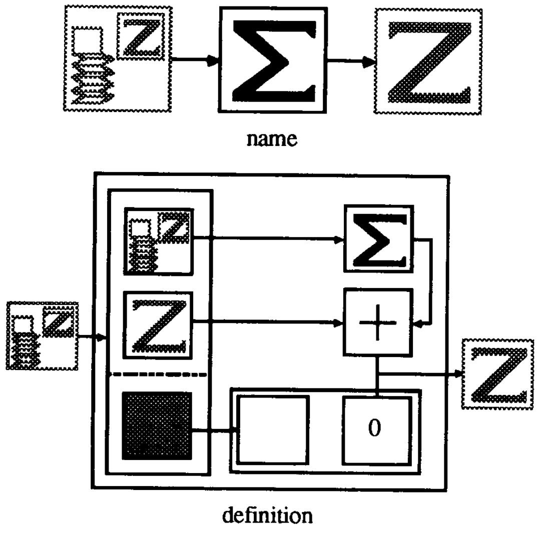

Fig. 12 : Summation of integer list

3 Functions with variable number of Ports

It is often convenient to allow functions with a variable number of arguments and results. Examples of these include:

- a preferrer function that takes *n* ordered inputs and returns the first input which receives consistent data (this construct resembles a case statement)
- a list composer that takes *n* objects of type T as arguments and returns a `list of T` containing those *n* objects, together with its corresponding decomposer
- an STL sequential iterator [1] with *n* sequential ports.

We have extended the syntax of STL to permit function icons with a variable number of ports. This extension generalizes a function to the notion of a *function schema*. A function schema is an abstraction of a set of similar functions which differ only by the number of ports they have. A concrete function belonging to this set is said to be an *instance of the function schema*.

Like ordinary functions, function schemas have a name, visualized by a function schema name icon (FSNI), and a definition. An FSNI can have any number of ordinary ports, each one connected to an ordinary pipe and receiving one argument or returning one result. In addition, however, it can have any number of *multiports*, each one connecting to a *multipipe*. A multipipe is the abstraction of a bundle of n pipes, where n is variable. Fig. 13.a shows a multipipe abstracting zero or more pipes, Fig. 13.b a multipipe for one or more pipes. In an instance of a function schema, a multiport is connected to n ordinary pipes.

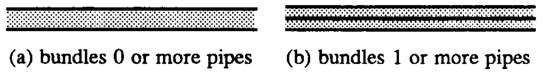

Fig. 13 : Multipipes

An ordinary pipe is associated with a type by a function definition. A multipipe can be associated with a type, but does not have to be. If it is not, type inference is done only when a concrete instance of the function schema is used (Note that this is still at program creation time, so that typing remains static). Fig. 14.a shows the name of a family of preferrer functions.

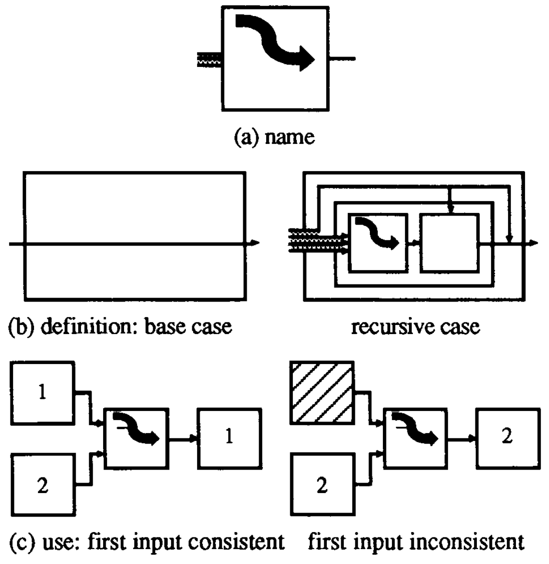

Fig. 14 : Preferrer function schema

A function schema is defined recursively. While the recursive part uses the multiport of the function schema, the base case replaces those multiports with ordinary ports. Fig. 14.b shows the definition of the preferrer function family. The base case arises when only one pipe (or a multipipe consisting of one pipe) is connected to the multiport. The recursive case arises when the multiport is connected to n ($n \geq 2$) pipes (or a multipipe composed of n pipes). In this case, the preferrer function schema instance taking n - 1 ports at this multiport is called.

Fig. 15 shows the naming and the definition of a family of list composers. Unlike the preferrer, the list composer uses typed inputs and outputs.

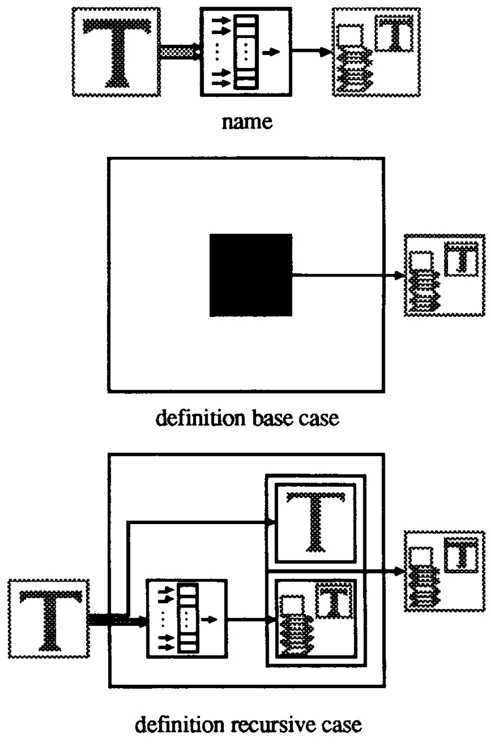

Fig. 15 : List composer function schema

4 Higher-Order Functions

A higher-order function is a function which takes other functions as arguments and/or returns functions as results. STL provides a quote/unquote mechanism similar to that in Lisp, for defining and using higher-order functions. Although this approach is powerful, syntactically it is not very elegant.

ESTL improves on this mechanism by introducing the concept of a *function slot*. A function slot is an empty (parameter) box in a function name icon. A function slot takes function icons or function boxgraphs as arguments ("the function is *slotted* in"). Each slot in a function name icon is associated with one or more variable boxes (referred to as *slot receivers*) in the boxgraph by means of slot variables (analogous to type variables). A lower-order function slotted into a slot of a higher-order function will appear in the variable box associated with the slot. Function slots are an alternative visual syntax for higher-order functions. They can be easily reduced to an equivalent quote and unquote.

Fig. 16 shows an example of higher-order functions using function slots. The functional defined in Fig. 16 performs an inorder traversal on a polymorphic binary tree. The function F slotted in will be called for each leaf node in the tree, it will use the leaf and the result of the traversal so far, and will return a new result. The functional takes both the tree and an initial value as arguments, and it returns the overall result of the traversal. Fig. 16.c shows a possible use of the functional. The addition function is slotted in. If a tree of integers and an initial value of 0 is provided, the functional will return the sum of the leaves of the tree.

5 Conclusion and Outlook

We described a number of extensions of the visual dataflow language Show-and-Tell. These extensions include

- a static polymorphic type system and its associated constructor and accessor functions,
- multiports and multipipes for describing functions with variable arities and
- function slots to ease the construction of higher-order functions.

We have implemented an editor and a type inference system for the type system described above.

6 References

[1] Takayuki Dan Kimura, Julie W. Choi, Jane M. Mack, A Visual Language for Keyboardless Programming, Tech. Rep. WUCS-86-6, Dept. of Computer Science, Washington Univ., St. Louis, MO, March 1986

[2] Takayuki Dan Kimura, Show and Tell Sample Programs, Dept. of Computer Science, Washington Univ., St. Louis, MO, January 1986

[3] Takayuki Dan Kimura, Determinancy of Hierarchical Dataflow Model -- a computation model for visual programming, Tech. Rep. WUCS-86-5, Dept. of Computer Science, Washington Univ., St. Louis, MO, March 1986

[4] Robin Milner, A Theory of Type Polymorphism in Programming, in: Journal of Computer and System Sciences, Vol. 17, 1978, pp. 348-375

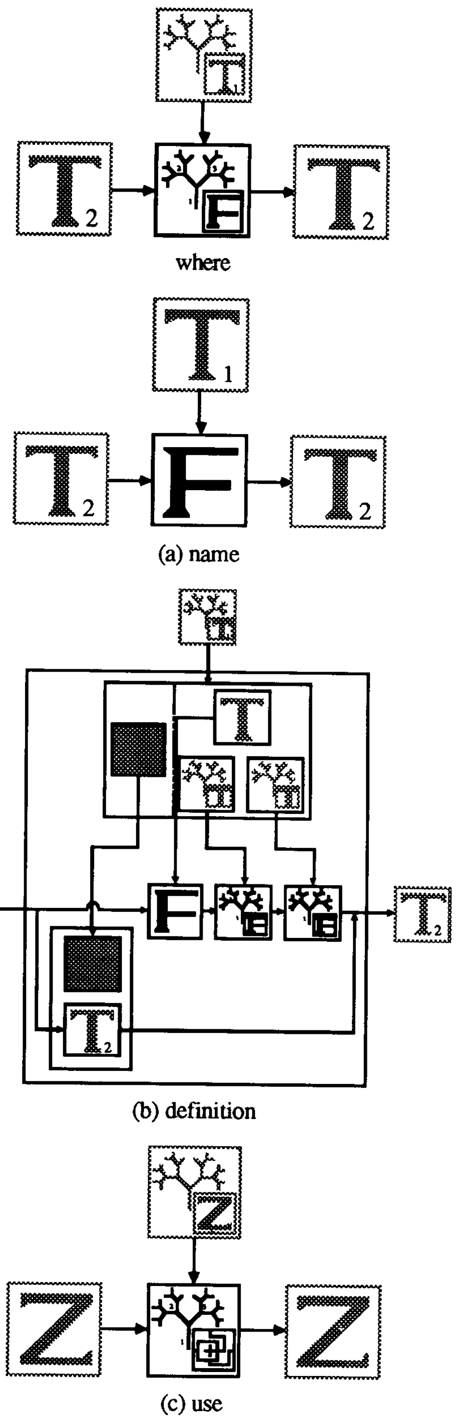

Fig. 16 : Preorder Tree Traversal Functional