A View Extension to an Object-Oriented Type System

M. Naeem[1] and C. J. Harrison[2]

[1] *Department of Computer Science, Government College University, Lahore, Pakistan*
[2] *School of Informatics, University of Manchester, Manchester, UK*

Abstract

Many languages provide support for describing composite and other user-defined types which in turn depend upon built-in types. A built-in or primitive type is typically composed of a data structure, a set of operations and a view or concrete external representation for that data structure. User-defined types differ from primitive types in that they are typically composed of a data structure and a set of operations only. This paper describes a view model which is an integral part of an object-oriented development language named POOL (Persistent Object-Oriented Language). The language provides a facility for defining multiple and complex views of a user-defined type as an integral part of a type definition. These view definitions are used to enable values of user-defined types to be manipulated directly, for example, during marshalling and input/output operations. This paper also addresses the view inheritance problem associated with user-defined views, and also discusses a type inference strategy adopted for inferring types from values of user-defined types.

Keywords: View Model; Marshalling; Persistent Object-Oriented Language; View Inheritance; Type Inference Strategy

1. Introduction

In Programming Languages, a built-in or primitive type is typically composed of a data structure, a set of operations and a view or concrete external representation for its data structure. The view or concrete external representation is built into the language processor. When values of built-in types are used, their concrete representation guarantees that the resulting value is syntactically correct and type inference for such values is relatively straight forward.

Programming languages provide support for describing composite and other user-defined types which in turn depend upon built-in types. User-defined types differ from primitive types because they are typically composed of a data structure and a set of operations only, i.e., there is no view or concrete representation for a user-defined type and all input/output processes are performed in terms of built-in types. There is no general-purpose mechanism available in any language to define views or concrete representation of user-defined types. Such view mechanism can only be implemented via changes to the language processor which in turn often initiates other changes. An alternate solution is to use the external representation of a primitive type to generate a mapping to a user-defined type's own external representation. However, this solution makes type inference more complicated and compromises the data hiding principle.

There are a number of view mechanisms in common use but these are typically limited to the context of *O*bject-*O*riented *D*ata*B*ases (OODB) [1, 2, 3, 4]. However, little work has been done regarding the implementation or realization of object-oriented views [2, 5], i.e., there are no widely accepted concrete representation or view mechanisms in the context of object-oriented languages.

POOL [6] is an object-oriented class-based language whose simple Pascal-like syntax, combined with static type-checking, provides support for the rapid construction of reliable reusable software. POOL provides an extension to its object-oriented, class-based type system which supports a general purpose mechanism for describing concrete representations for values of user-defined types.

2. View Extension

In POOL's type system, a type specifies the structure of a set of values together with the valid operations, and one or more views that may be applied to those values. A view is an integral part of a type definition and can be used directly to describe concrete representations for values of a user defined type for marshalling and input / output operations and direct

Corresponding Author: M. Naeem (dr_majid_naeem@gcu.edu.pk)

manipulation of values of a user-defined type [7].

Further, every user defined type has at least one view of the type's name. This view is termed as the default view and is used to process those values of user-defined types where a view name is not explicitly specified, e.g., values of user-defined types in program text. A type inherits all the views of its super-type and these inherited views can be extended and modified in sub-types.

A view differs from an operation in that a view can be used within an expression for direct object-value manipulation, marshalling, assignment or processing, i.e., such processing cannot be done by a method (operation) within an expression. A view is constructed using the following three primitives:

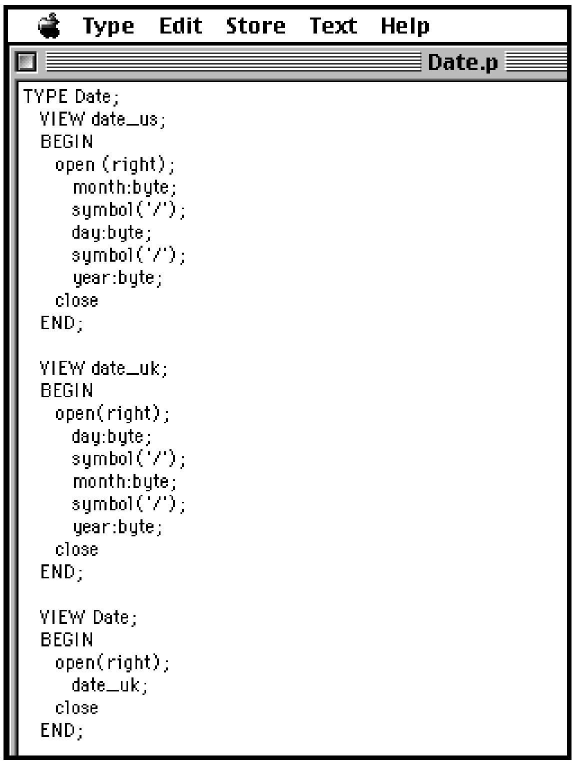
- **`open`** `(d : direction)` indicates the start of a new view whose nested sub-views are laid out along direction `d` which, in turn, can take one of two values, i.e. `right` and `down`. The value `right` indicates that all the components which appear in this view will be to the right of each other, and `down` indicates that all the components which appear in this view will be below each other.

- **`close`** indicates the end of the currently open view.

- **`symbol`** `(str : string)` constructs a primitive view containing a string `str`. For an output operation it will show the value of the string `str`, for an input operation it will accept a string `str`.

POOL's type system supports multiple views for values of user-defined types in a modular manner. A view may consist of other views, called sub-views. This nesting of views forms a hierarchical structure that may be organized to any number of levels, directly or indirectly, by referencing other views in a given view. This approach also imposes the data hiding principle on every level of a system design.

The example given in Figure 1, below, defines three different views for a type `date` (only view definitions are shown). In this example three views namely `date_us`, `date_uk` and a default view `date` are defined. The view `date_uk` defines a view for date type in `day/month/year` format, `date_us` defines a view for `date` type in `month/day/year` format, and `date` is a default view which is defined using the `date_uk` view, i.e., when no view is explicitly mentioned for an object of type `date` this view will be use to manipulate values during input/output.



```
TYPE Date;
  VIEW date_us;
  BEGIN
    open (right);
      month:byte;
      symbol('/');
      day:byte;
      symbol('/');
      year:byte;
    close
  END;

  VIEW date_uk;
  BEGIN
    open(right);
      day:byte;
      symbol('/');
      month:byte;
      symbol('/');
      year:byte;
    close
  END;

  VIEW Date;
  BEGIN
    open(right);
      date_uk;
    close
  END;
```

Figure 1: View Definition for A User-defined Type '`date`'.

One advantage of building and organising views in this manner is that the resulting implementations have the potential for reuse since a given view, or element of a view, may be replaced by another without arbitrary constraints being imposed on such substitutions. In addition, the resulting representations of view components do not constrain their elements to particular layouts, e.g., layouts whose components appear in pre-determined positions relative to one another. Equally importantly, low-level details relating to such components need not be of concern to a programmer.

3. Using Views

A view can be used in three different ways: (1) It can be used in an input (output) operation via read (write) statements; (2) it can be used when object values are directly manipulated, e.g., during assignment and expression evaluation; (3) it can be used for data marshalling. The next three subsections describe these three cases in detail.

3.1 Input / Output Operations

This view activation mechanism provides a distinction between type operations and type views. The input/output operations currently supported by the system include:

```
write ( x:  view;   dev )
read  ( x:  view;   dev )
```

where x is object name and view is the name of the view used to input or output x. If no view name is given then the default view is used to input or output a value. The input and output operations are performed via some device dev, as given in Table 1.

Table 2 shows the usage of the views defined in Figure 1 during the input and output of an object d of type date. In Table 2, row 1, a value is obtained from the user via the date_uk view for an object d of type date. When this statement is executed it will obtain a value from the user through a standard input device std (i.e. a keyboard) and validate that value against the view date_uk for the type date. In row 2, the same value of object d is displayed on a standard output device std (i.e. a monitor) using the date_uk view and the value is displayed in day/month/year format. In row 3, the same value of object d is displayed on a standard output device std (i.e. monitor), using the date_us view and the same value will appear on the output device but in month/day/year format. In row 4, no explicit view is indicated to output the value of object d on a standard output device std (i.e. monitor). In this case, the default view date will be used by the system and the value is displayed on the output device in day/month/year format.

Table 1: Input / Output Devices for read/write Operations

Sr. No	Device	Explanation
1	std	standard input/output device, e.g. keyboard, monitor
2	sto	persistent store
3	prn	printing device
4	com	Communication Link (Socket, port etc)

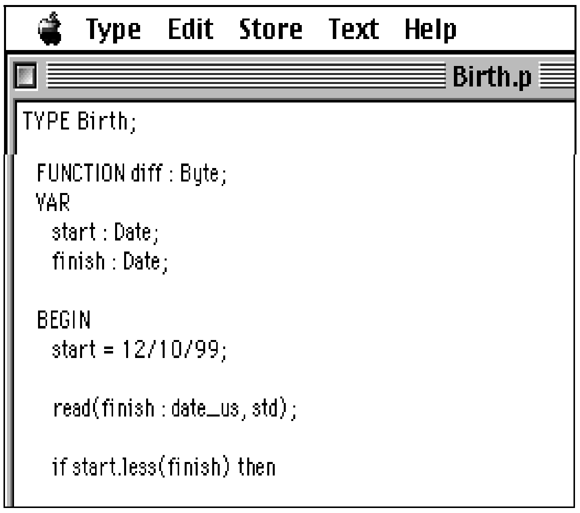
Table 2: System Interaction

Sr. No	Operation	Action	Output
1	`read (d:date_uk, std);`	user-input	25/10/98
2	`write(d:date_uk, std);`	output	25/10/98
3	`write(d:date_us, std);`	output	10/25/98
4	`write(d,         std);`	output	25/10/98

3.2 Direct Value Manipulation

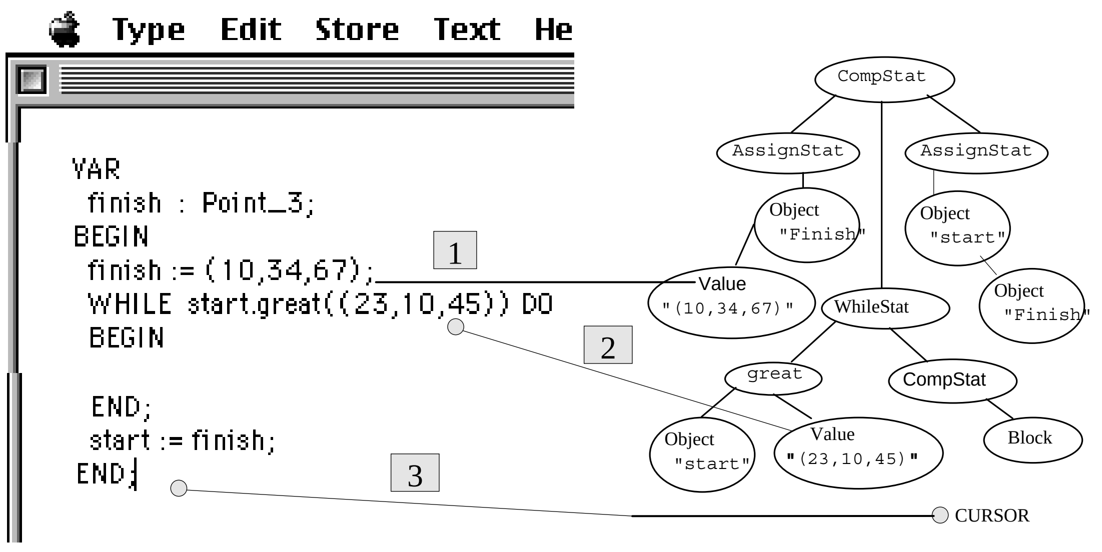
The code in Figure 2 demonstrates how value of an object of type date is used directly in a program text. In this example, two objects of type date are declared, and the object start is assigned a date value using the default view of the date type. The value for the object finish is obtained by user input via the date_us type. These two objects, i.e., start and finish, are then used in an expression which involves an operation less defined for the type date. In other programming languages, a user-defined type object can be initialized at the time of creation only and it become quite difficult in the body of the program where comma separated primitive values are used to initialize every object field.

In Figure 3, a fragment of code exploits the use of these views in a programming environment. This example shows how user-defined values can be used directly. The tree structure on the right-hand side of



```
TYPE Birth;

  FUNCTION diff : Byte;
  VAR
    start : Date;
    finish : Date;

  BEGIN
    start = 12/10/99;

    read(finish : date_us, std);

    if start.less(finish) then
```

Figure 2: Direct value manipulation



```
VAR
 finish : Point_3;
BEGIN
 finish := (10,34,67);
 WHILE start.great((23,10,45)) DO
 BEGIN

 END;
 start := finish;
END;
```

Figure 3: Code Fragment and underlying structure

Figure 3 shows the underlying structure of the code on the left-hand side. At Tag 1, an object named `finish` of a user-defined type `Point_3` is directly assigned a value, whereas at Tag 2, another value of the same type is used in a Boolean expression as a parameter to a function. In this example, `great` is an operation of a user-defined type `Point_3`, and `great: Point_3 → Boolean.`

It should be noted that only the default view is used to directly manipulate the value of a user-defined type in a program. This is a major limitation of direct value-manipulation in a program's source code. A solution to this problem is to specify the name of the view along with the direct value of a user-defined type. This solution solves the above mentioned problem but creates an ambiguity due to fact that the given view name can also be manipulated as a part of the value.

3.3 Marshalling

In Remote Procedure Calls [8,9], the transfer of message data between two nodes requires encoding and decoding of the message data known as marshalling. In order to encode and decode such data the tagged or untagged representation methods are used to marshal arguments and results which must be known to both client and the server. The structure of this data reflects all primitive types, structured types and user defined types. The systems which support marshalling either require the user to explicitly define marshalling procedures for user-defined types [10] or provide some limited built-in procedures to marshal compound types built from the scalar ones. One solution of this problem is to define an initialising function or marshalling procedures for every user-defined type. This function/procedure takes the required value, in terms of values of built-in types, and initialises the variable (of some user–defined type) in order that it can be used in expressions or passed to a remote node for processing. Unfortunately, this approach is rather restrictive when there is a need to support the direct manipulation of values of user-defined types, and it also limits the expressive power of a language.

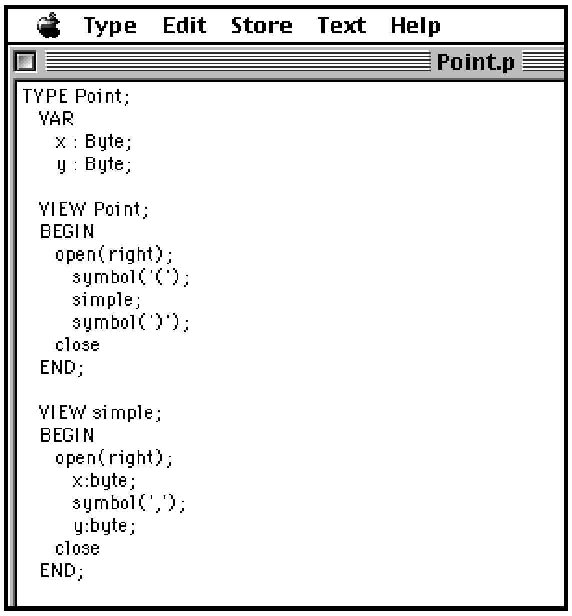
Whenever an object of a user-defined type is sent to another node in a distributed environment, its associated view will also be made available to the receiving node at compile time. When the object is received on that node the associated view will be use to manipulate the object according to its view on the sending node and no restriction will be imposed and complete transparency will be maintained.

4. View Inheritance

Sub-type views are defined using super-type views because the data hiding principle of object-oriented languages does not allow direct access to the attributes of a type by another type. If a type is inherited by another type, the views of both types should be defined with care because an inherited type includes all of the operations, attributes and views of its super-type. In order to define the views of a sub-type, the definitions of super-type views need special treatment, for example, if some start and end symbols are used in the definition of a super-type view, these symbols will also be inherited by the sub-types views which is not required there as they mark the beginning and end of a concrete representation of a value.

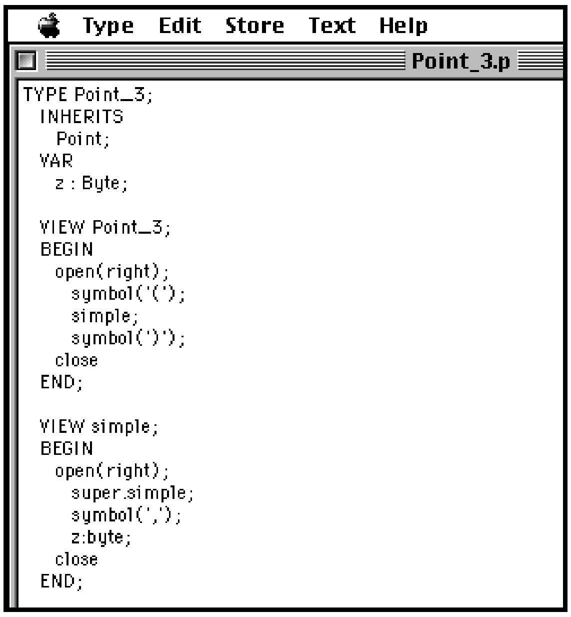
One solution is to define a view in two steps or parts. First a view without start and end symbols is defined.

Another view then uses this view and introduces the required start and end symbols. The first view will then also be used as the basis for defining all views of its sub-types. This technique is used to define two types `Point` and `Point_3`, shown in Figures 4 and 5 (no operations are shown). Type `Point` defines a `point (x,y)` in two-dimensions, whereas type `Point_3` defines a point in three dimensions having a format `(x, y, z)`. In this example, type `Point_3` inherits type `Point`. Type `Point` defines two views namely `simple` and `Point`, where the simple view defines a view without starting and ending symbols and the `Point` view uses the simple view to define the default view of the type `Point` by appending start and end symbols in the view simple.



```
TYPE Point;
  VAR
    x : Byte;
    y : Byte;

  VIEW Point;
  BEGIN
    open(right);
      symbol('(');
      simple;
      symbol(')');
    close
  END;

  VIEW simple;
  BEGIN
    open(right);
      x:byte;
      symbol(',');
      y:byte;
    close
  END;
```

Figure 4: View Definition of Point type



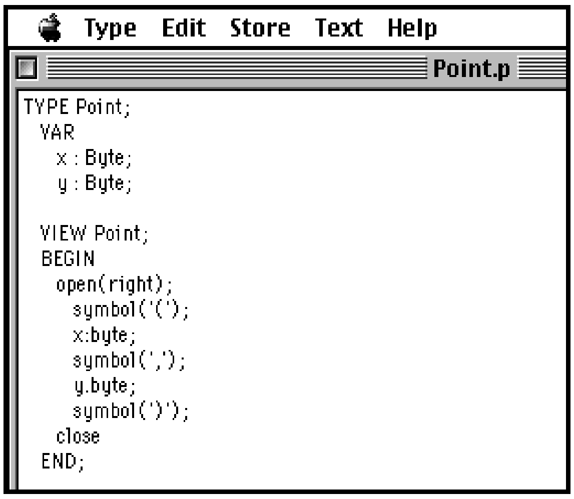
```
TYPE Point_3;
  INHERITS
    Point;
  VAR
    z : Byte;

  VIEW Point_3;
  BEGIN
    open(right);
      symbol('(');
      simple;
      symbol(')');
    close
  END;

  VIEW simple;
  BEGIN
    open(right);
      super.simple;
      symbol(',');
      z:byte;
    close
  END;
```

Figure 5: View Definition for Point_3 Type

When the type `Point_3` inherits from the `Point` type, it inherits both the view `simple` and the view `Point` together with other attributes and operations. When we define the `simple` view for `Point_3`, we will use the view which is defined without start and end symbols otherwise those symbols will appear in the `Point_3` view and the format for its view will become `((x,y),z)` instead of `(x,y,z)`. Therefore, we use the `simple` view of `Point` type and define a view define the default view `Point_3`. Note that in the simple view of the `Point_3` type we use its super-type view via `super`. This also helps to distinguish between super-type and sub-type views with the same name. These views can also be nested with any other super-type at other higher levels iteratively.

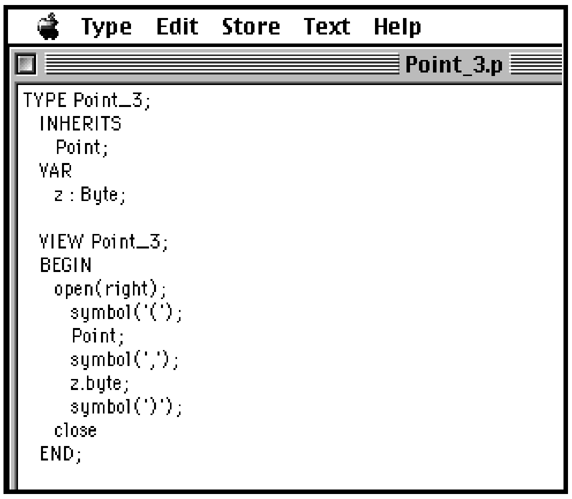
The disadvantage of this technique is that we have to define at least two views for every type, i.e. one without a start and end symbol, and the other with a start and end symbol which uses the first view. The logical improvement to the second technique is to ensure that the starting and ending symbols of every type are specific to that type and when any other type inherits from that type, the starting and ending symbols are truncated by the view mechanism. This enables new starting and ending symbols to be appended to an inherited view.

The improved version of the above examples is shown in Figures 6 and 7. In Figure 6, the type `Point` is shown with only one view `Point` which is a default view of this type. This view is defined with start and end symbols and will produce a value in `(x,y)` format. In Figure 7, type `Point_3` is shown with its default view `Point_3`. Since `Point_3` inherits from the type `Point`, it inherits the view `Point`. This view `Point` is directly incorporated into the definition of the view `Point_3` in the type `Point_3`. When the system encounters an inherited view in a subtype view it will truncate the starting and ending symbols, if any, of the inherited view and the resulting view will display the values with new start and end symbols. In this case it will display the values of type `Point_3` in `(x, y, z)` format by truncating the start and end symbols at the places used by the inherited view. The truncation operation is an integral part of language processor for view-mechanism.



```
TYPE Point;
  VAR
    x : Byte;
    y : Byte;

  VIEW Point;
  BEGIN
    open(right);
      symbol('(');
      x:byte;
      symbol(',');
      y.byte;
      symbol(')');
    close
  END;
```

Figure 6: Improved View for Point Type



```
TYPE Point_3;
  INHERITS
    Point;
  VAR
    z : Byte;

  VIEW Point_3;
  BEGIN
    open(right);
      symbol('(');
      Point;
      symbol(',');
      z.byte;
      symbol(')');
    close
  END;
```

Figure 7: Improved View for Point_3 Type

5. Type Inference

One major problem associated with direct manipulation of values of user-defined types is type inference for those values. This section describes the type inference strategy adopted to infer the type of values of user-defined types.

In POOL's type system a view is an integral part of a type definition and a type can have more than one view. In other words, if Γ is a well formed (◊) static typing environment and α is a user-defined type then α must have one or more views υ.

$$\frac{\Gamma \succ \Diamond \quad \Gamma \succ \alpha}{\Gamma \succ (\upsilon_i \rightarrow \alpha)_{\; i \in 1..n}} \qquad (1)$$

In POOL, an object τ can have more than one valid type

$$\Gamma \succ \tau : \alpha_1 | \ldots | \alpha_n \qquad (2)$$

a valid type α can have more then one view υ

$$\frac{\Gamma \succ \Diamond \quad \Gamma \succ \alpha}{\Gamma \succ (\alpha \rightarrow \upsilon_i)_{\; i \in 1..n}} \qquad (3)$$

There are two different methods to infer the type of a given value in an expression. Either method can be used to infer the type of a value. In some cases both methods must be used to infer the type of a value. The following subsections describe these two methods.

5.1 Method 1

This method is used to infer the type of a value when it is used in an expression. The method is based upon determining the required type of the term in an expression as discussed in [11, 12, 13, 14]. Using this method, when a value is used directly in an expression the required type(s) of the expected value can be determined by inspection:

a) If it is used in as an operation argument, then the type(s) of the parameter can be determined from its definition, and hence the given value should be one of those type(s). In this case the type of the value will become the active type of that parameter. In Figure 3 an operation `great: Point_3 → Boolean` is shown. When a direct value `(23,10,45)` is directly passed to this operation, this value is inferred as of type `Point_3` by the definition of the operation.

b) If it is used as a term in an expression, then the allowed type(s) of that term can be used to infer the type of the value. That type will become the active type of that term. In Figure 2 an object `start` of type `Date` is declared. In the body of the program a value `12/10/99` is assigned to this object. In case of any ambiguity, the value assigned to this object will be inferred to the declared type of the object.

5.2 Method 2

This method is based upon using default or other views to infer the type of a value. In this method, concrete views are mapped onto a given value. To apply this method, a graph, as shown in Figure 8, is constructed. This graph is composed of a number of branches equal to the number of starting symbols of views in a type system. The depth of every tree is equal to the maximum number of symbols present in a view that starts from the symbol root of this tree. Each node stores information which corresponds to the view match for that sequence of symbols. When a user-defined value is encountered, the graph is traversed until the value is matched completely with some type's view.

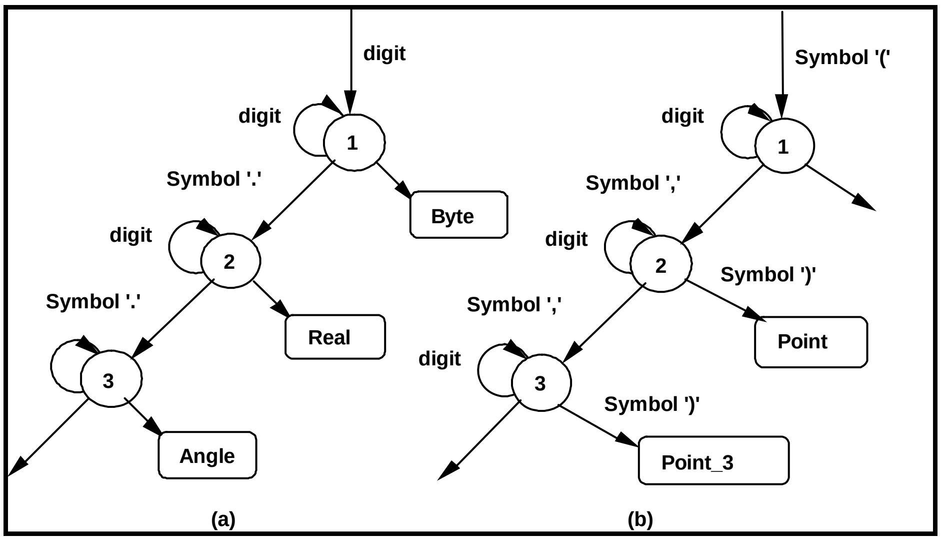


Figure 8: Type Inference Graph

In Figure 8(a), a path 1-2-3 is shown in which path 1 represents a `Byte` value, path 1-2 represents a `Real` value, and path 1-2-3 a value of type `Angle`. In Figure 8(b) the paths for values of type `Point` and `Point_3` are shown. The views for these types are defined in Figures 6 and 7. The path 1-2 represents a value of type Point whereas a value of type `Point_3` is represented by path 1-2-3.

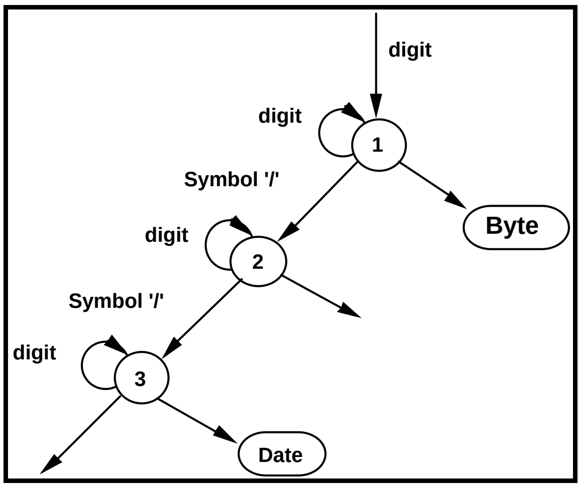


Figure 9: Type inference graph for type Date

In Figure 9, the type inference graph for type `Date` is shown. This graph shows the situation when two or more views have same inference graph. In this graph, it is difficult to determine whether a `date_us` view is used or a `date_uk` is used because both views have the same type inference graph. In this case, a direct value in text should be processed by the default view of that type, i.e., `Date`, the remaining occurrences of such values will be treated in the same manner in the rest of the program while in the case of an input operation it is decided by using view names with the object name of that type, i.e, `date_uk` or `date_us` is used to manipulate the direct value. If no view name is explicitly specified then the default view will be used. When these two methods are combined with rules number (2) and (3) by keeping rule (1) intact, there are four combinations for inferring the type of a value as discussed below:

Combination 1: This is the simplest case. In this case an object τ has only one type α and that type has only one view υ. In this case, the type of the object can be inferred directly by method 1.

$$\frac{\Gamma \succ \Diamond \quad \Gamma \succ \tau : \alpha}{\Gamma \succ (\alpha \rightarrow \upsilon)}$$

This case is illustrated by the examples shown in Figures 6 and 7. The type inference graph of these types is shown in Figure 8.

Combination 2: In this case an object τ has a type α, but that type can have more than one view υ. In this case, the type of the object can be inferred by method 1.

$$\frac{\Gamma \succ \Diamond \quad \Gamma \succ \tau : \alpha}{\Gamma \succ (\alpha \rightarrow \upsilon)_{i \in 1..n}}$$

This case is illustrated in Figures 4 and 5, where each type has two views. In Figure 10 the view graph or the view definitions of these types are shown. In Figure 10(a) the view graph for views `Point` and `Point_3` are shown, while Figure 10(b) shows a simple view graph for these types.

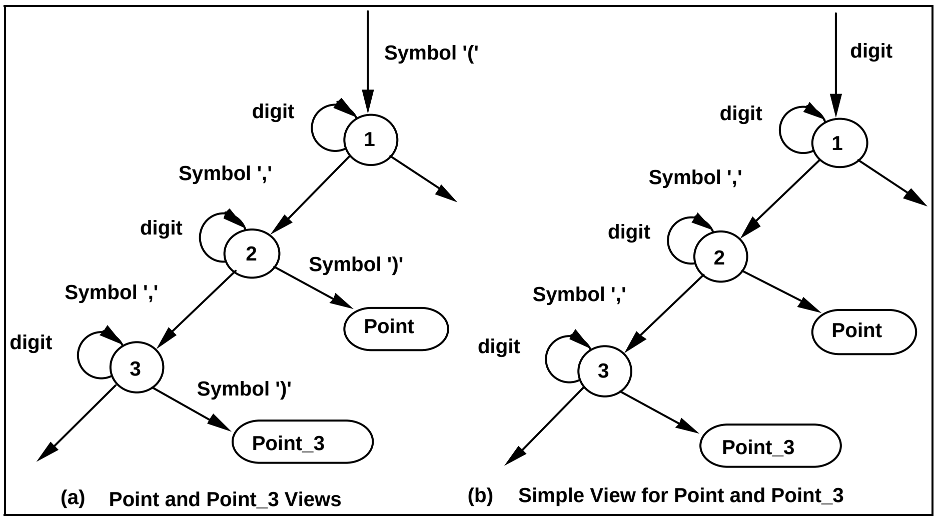


Figure 10: View Graph with two view definitions

Combination 3: An object τ has more than one type but each type has only one view. In this case the type of the value can be inferred by applying method 1 and then method 2.

$$\frac{\Gamma \succ \Diamond \quad \Gamma \succ \tau : \alpha_1 | \ldots | \alpha_n}{\Gamma \succ (\alpha_i \rightarrow \upsilon_i)} \; i \in 1..n$$

It will take a maximum of 'n' matches, where 'n' is the number of allowed types of the object τ.

Combination 4: An object τ can have more than one type and each type can have more than one view. In this case, the type of the value is first inferred by using method 1 and then method 2.

$$\frac{\Gamma \succ \Diamond \quad \Gamma \succ \tau : \alpha_1 | \ldots | \alpha_n}{\Gamma \succ (\alpha_1 \rightarrow \upsilon_i)_{i \in 1..k} \ldots \Gamma \succ (\alpha_n \rightarrow \upsilon_i)_{i \in 1..k}}$$

If `Vi` denotes the number of views in a user-defined type "`i`", and "`k`" denotes the total number of types which an object τ of be, then it will take a maximum of matches to infer the type of the given object.

$$\sum_{i=1}^{k} V_i$$

6. Summary and Conclusions

Some systems support sophisticated user-interfaces allowing the definition of regions and specific computations over these regions [15,13]. However, such approaches neither seem to allow for a wide variety of data structures, nor do they seem to provide strong typing mechanisms through which the system could check that valid data is used in the computation. These systems are tightly-coupled with a regular text-based programming language, and are used at execution time to allow the user to examine and explore current values found in the various data structures of the program. Such systems have a fixed representation for every data construct [12], although some systems allow the user to choose between varieties of alternative display methods [11], and only offer low-level support for dynamic data structures based on the concept of pointers.

The concrete representation extension presented here provides a systematic way of dealing with values of user-defined types. It fills a major gap between existing support for representing values of built-in types and a general lack of support for representing values of user-defined types. The view mechanism supports a hierarchically nested model of views in which type inference can be applied to values of user-defined types [16,17].

The resulting views are reusable, inherited by all the sub-types of a user-defined type, and the same view can be used to manage marshalling operations and input/output. This paper has also proposed rules for concrete representations and for type inference, and enumerated the possible combinations of types and views for an object in a polymorphic, static typing environment.

REFERENCES

[1] Heiler, S. B. Zdonik; Proc. IEEE International Conference on Data Engineering, (1990), 86-93.

[2] Kuno and E. A. Rundensteiner; *The MultiView OODB view system: Design and implementation*, Technical Report CSE-TR-246-95, University of Michigan, (Jul 1995).

[3] H. Scholl, C. Laasch, M. Tresch; Proc. The Second DOOD Conference, page 113-119 (Dec 1991).

[4] A. Rundensteiner; Proc. 18th VLDB Conference, (1992), 187-198.

[5] Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonic; *In Building an Object-Oriented Database System: The Story of O2*, Publisher Morgan Kaufmann Pub., (1992).

[6] C. J. Harrison and Majid Naeem; ACM Symposium on Applied Computing Page 1118-1121 Como, Italy (2000).

[7] C. J. Harrison and Majid Naeem *A Model-Oriented Programming Support Environment for Understanding Object-Oriented Concepts*, Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Vol. 1964/2000, ISSN: 0302-9743.

[8] Sun Microsystems; *Remote Procedure Call Protocol Specification*, Networking on the Sun Workstation, Sun Microsystems, Mountain View, CA. (1985).

[9] Tay B.H., Ananda A.L.; *Operating Systems Review*, 24(1990) 68-79.

[10] Bacon J.M., Hamilton K.G.; *Distributed Computing with RPC*, The Cambridge Approach. Technical Report No. 117. Computer Laboratory, University of Cambridge, England (1987).

[11] Tofte M.; *Information and Computation*, 89(1990) 1-34.

[12] Palsberg J.; Proc. 9Th Annual IEEE symposium on Logic in Computer Science, (1994), 186-195.

[13] Mitchell J.C.; Proc. 11Th Annual ACM Symposium on Principles of Programming Languages, (1984), 175-185.

[14] Aiken A., E. L. Wimmers; Proc. ACM Conference on Functional Programming and Computer Architecture, (1993), 31-41.

[15] A. Kuno, E. A. Rundensteiner; *Implementation experience with building an object-oriented view management system*, Technical Report CSE-TR-191-93, University of Michigan (1993).

[16] C. J. Harrison and Majid Naeem, Proc. IEEE INMIC 2004, 8th International (IEEE) Multi-topic Conference, Lahore, Pakistan, (2005), 133-139.

[17] C. J. Harrison and Majid Naeem, Proc. IEEE INMIC 2004, 8th International (IEEE) Multi-topic Conference, Lahore, Pakistan, (2004), 737-742.