Towards a Better Understanding of Dylan

Tamme D. Bowen
Kelly M. Hall
Laboratory for Applied Logic
University of Idaho

Research Report LAL-93-01
May 1993

Abstract

The Dylan programming language combines the glitz of the object oriented paradigm with the glitter of dynamic functional languages. In the paper we explore how Dylan binds these two approaches together to make a coherent language suitable for applications and systems programming.

1 Introduction

Dylan is a new object-oriented dynamic language from Apple. Dylan is designed to replace existing static languages for the development of large software systems, yet remain small and efficient enough for the next generation of portable computers. Dylan was developed from the language Scheme augmented with the Common-Lisp Object System (CLOS).

In this paper we will model Dylan's type system in a more formal manner than in [1]. We define formally what the terms *class*, *method*, *generic function*, and *instance* mean. We also discuss some features Dylan provides for efficiency and security. Function descriptions have been written to resemble Haskell.

2 Major Concepts

Most of Dylan can be characterized by two main concepts:

- Objects
- Functions

The core of Dylan implements objects and functions, but omits many of the features programmers would need to write useful programs. Dylan extends the core with ten required libraries that provide control flow, numbers, and abstract data types.

2.1 Objects

All data values in Dylan are considered *objects*. Dylan organizes objects by grouping them in *classes*. All objects are *instances* of at least one class. Dylan organizes classes into a heterarchy (direct acyclic graph) and classes inherit the features of classes above themselves in the heterarchy. The top of the heterarchy is the class `<object>`, the most general class in Dylan.

Every object is the *direct instance* of exactly one class. Objects are *indirect instances* of each *superclass* of their *direct class*.

Classes determine the structural characteristics of their instances by specifying *slots*, which hold the object's local state. Each slot has a name which identifies it. Special functions (called *getters* and *setters*) are used to read and write the values stored in slots.

2.2 Functions

Functions in Dylan are objects that perform actions. Dylan functions correspond to functions, procedures, methods, and messages in other languages. Dylan has two types of functions: *methods* and *generic functions*.

Methods are functions that contain a typed argument list and a body of code. Methods are defined with typed formal parameters and can be applied to arguments with either the same class as the defined parameters or to subclasses of the defined parameters. If functions are applied to incorrect argument types, an error is signaled. Note: this type checking is performed at runtime even though static analysis is possible.

Generic Functions are ways to group together zero or more methods (each having different types) under the same function name. The generic function examines the types of the arguments and chooses the most appropriate method to invoke. If no appropriate method can be found an error is generated. Since generic functions can be created and modified at runtime, all type checking is dynamic. Generic functions are by definition overloaded and give Dylan ad-hoc polymorphic capabilities.

2.3 Denotations for Objects and Functions

The Dylan reference manual [1] is often vague in its description of how classes, methods, and generic functions work together. To help us understand Dylan better, we have formalized the relationship of these entities. This section contains an abstract syntax and description of the basic operations on classes and functions. No static validity functions are provided since Dylan does no static analysis.

2.4 Abstract Syntax

Here is an abstract syntax of classes, methods, and generic functions. Note that *Symbol* and *Expr* are not defined. *Symbol* is analogous to $Char^+$ and *Expr* corresponds with function bodies in Scheme[2].

$$
\begin{array}{rcl}
\mathit{ClassList} & \hat{=} & \mathit{Class}^{+} \\
\mathit{Class} & \hat{=} & \text{name:}\mathit{ClassName}\ ;\ \text{sl:}\mathit{SlotList}\ ;\ \text{subclasses:}\mathit{ClassList} \\
\mathit{SlotList} & \hat{=} & \mathit{EmptySlot}^{*} \\
\mathit{ClassList} & \hat{=} & \mathit{ClassName}^{+} \\
\mathit{ClassName} & \hat{=} & \mathit{Symbol} \\
\mathit{EmptySlot} & \hat{=} & \text{name:}\mathit{SlotName}\ ;\ \text{class:}\mathit{ClassName} \\
\mathit{SlotName} & \hat{=} & \mathit{Symbol} \\
\mathit{Method} & \hat{=} & \text{name:}\mathit{FunName}\ ;\ \text{pl:}\mathit{ParamList}\ ;\ \text{body:}\mathit{Expr} \\
\mathit{ParamList} & \hat{=} & \mathit{Param}^{*} \\
\mathit{Param} & \hat{=} & \text{name:}\mathit{Symbol}\ ;\ \text{class:}\mathit{ClassName} \\
\mathit{FunName} & \hat{=} & \mathit{Symbol} \\
\mathit{Generic} & \hat{=} & \text{name:}\mathit{GenName}\ ;\ \text{ml:}\mathit{MethodList} \\
\mathit{MethodList} & \hat{=} & \mathit{FunName}^{*} \\
\mathit{GFList} & \hat{=} & \mathit{Generic}^{*} \\
\mathit{Instance} & \hat{=} & \text{name:}\mathit{IName}\ ;\ \text{slots:FullSlots}\ ;\ \text{class:}\mathit{ClassName} \\
\mathit{FullSlots} & \hat{=} & \mathit{FullSlot}^{*} \\
\mathit{FullSlot} & \hat{=} & \text{name:}\mathit{SlotName}\ ;\ \text{class:}\mathit{ClassName} \\
\mathit{IName} & \hat{=} & \mathit{Symbol} \\
\mathit{Key} & \hat{=} & \mathit{Symbol}
\end{array}
$$

3 Classes

The acyclic directed graph of classes can be represented in many ways. Our representation will be a list of *Class* where *classname* is the name of a class, *sl* is a list of that classes slots, and *subclasses* is a list of classes that directly inherit properties from *classname*. The basic operations on classes can be coded as follows:

IsClass returns `True` if `cl` is a valid class:

```
IsClass :: ClassName -> ClassList -> Boolean
IsClass (cl:ClassName) ([]:ClassList) = False
IsClass cl (c::cs) =
  if cl = c.name then True else IsClass cl cs
```

GetSlots returns the slot list for class `cl`:

```
GetSlots :: ClassName -> ClassList -> SlotList
GetSlots (cl:ClassName) ([]:ClassList) = error "class not found"
GetSlots cl (c::cs) =
  if cl = c.name then c.sl else GetSlots cl cs
```

GetKids returns a list of the direct descendants of class *cl*:

```
 GetKids :: ClassName -> ClassList -> ClassList
 GetKids (cl:ClassName) ([]:ClassList) = error "class not found"
 GetKids cl (c::cs) =
   if cl = c.name then c.subclasses else GetKids cl cs
```

GetSupers returns a list of all of the super classes for class *cl*:

```
GetSupers :: ClassName -> ClassList -> ClassList -> ClassList
GetSupers (cl:ClassName) ([]:ClassList) (CC:ClassList) = []
GetSupers cl (c::cs) CC =
 if element cl c.subclasses
  then unique((cl.name::GetSupers cl cs CC)@(GetSupers c.name CC CC))
  else GetSupers cl cs CC
```

GetSubs returns a list of all of the subclasses for class *cl*:

```
 GetSubs :: ClassName -> ClassList -> ClassList -> ClassList
 GetSubs (cl:ClassName) ([]:ClassList) (CC:ClassList) = []
 GetSubs cl (c::cs) CC =
   if cl = c.name
     then unique (direct @ indirect)
     else GetSubs cl cs CC
 where direct = c.subclasses
 and   indirect = fold '@' (map (\x. GetSubs x CC CC) c.subclasses)
```

Where `unique` removes duplicate items from a list and `element` checks for membership in a list.

New classes are created in Dylan with the function *make-class* which takes three parameters: the name of the new class, a list of the direct superclasses of the new class, and a list of the new class's slots. For new classes, Dylan defines functions to access and update the slots. These *getter* and *setter* functions are added to the generic function space later. The code to define new classes is as follows:

```
NewClass :: ClassName -> ClassList -> SlotList -> ClassList -> Classlist
NewClass (n:ClassName) (pl:ClassList) (sl:SlostList) (C:ClassList) =
 if IsClass n C
   then NewClass n pl sl (remove n C)
   else if fold and (map (\x. IsClass x C) pl)
          then Fixlinks n pl (n,pl,sl)::C
          else error "superclass does not exist"
```

FixLinks and *Update* add the new class to the heterarchy and make sure the `subclasses` lists get modified to reflect the new class.

```
 FixLinks :: ClassName -> ClassList -> ClassList -> ClassList
 FixLinks (n:ClassName) ([]:ClassList) (CC:ClassList) = CC
 FixLinks n (p::ps) CC = FixLinks n ps (Update n p CC)


 Update :: ClassName -> ClassName -> ClassList -> ClassList
 Update (n:ClassName) (p:ClassName) ([]:ClassList) = error
 Update n p (c:cs) =
   if p = c.name
     then (c.name, c.sl, n::c.subclasses) :: CS
     else c :: (Update n p cs)
```

NewClass adds new classes to the heterarchy by checking that the class doesn't already exist. If the class does exist, it is removed and then added to the heterarchy again. Otherwise, the new class is added to the old class list and the *subclass* pointers are updated accordingly.

The other operation on classes is making new instances of them. This is done with the following code:

```
Make :: ClassName -> ClassList -> Instance
Make (n:ClassName) (CL:ClassList) =
 if IsClass n CL
  then BuildRecord unique (localslots @ superslots)
  else error "class not found"
where localslots = GetSLots n CL
and   superslots = fold '@' (map GetSlots (GetSupers n CL))
```

Where *BuildRecord* returns a record with field names identical to the *SlotNames* it is passed.

4 Generic Functions

Generic functions in Dylan are overloaded functions that examine the types of their parameters and invoke the most appropriate *Method* based on those types. We represent the space of all generic functions as a list. Each element in this list is itself a list of parameter lists and a pointer to the corresponding method that takes that parameter list. All top level function application in Dylan takes place through generic functions.

The operations on generic functions include defining new generic functions, removing generic functions, adding methods to existing generic functions, removing methods from generic functions, and applying generic functions to arguments.

IsGF returns `True` if `n` is a generic function:

```
 IsGF :: FunName -> GFList -> Boolean
```

```
IsGF (n:FunName) ([]:GFList) = False
IsGF n (g:gs) =
  if n = g.name then True else IsGF n gs
```

AddMethod adds a new method to generic function n:

```
AddMethod :: FunName -> ParamList -> Key -> GFList -> GFList
AddMethod (n:FunName) (pl:ParamList) (key:Key) ([]:GFList) = []
AddMethod n pl key (g:gs) =
 if n = g.name
   then ((g.name),(pl,key)::(g.methods)) :: gs
   else g :: AddMethod n pl key gs
```

RemoveMethod and *RMAux* remove a method with parameters pl from a generic function n:

```
RemoveMethod :: FunName -> ParamList -> GFList -> GFList
RemoveMethod (n:FunName) (pl:ParamList) ([]:GFList) = error
RemoveMethod n pl (g:gs) =
  if n = g.name
    then (g.name, (RMAux pl g.methods)) :: gs
    else g :: RemoveMethod n pl gs


RMAux :: ParamList -> MethodList -> MethodList
RMAux (pl:ParamList) ([]:MethodList) = error
RMAux pl (m:ms) =
  if foreach i in pl (pl.i.type = m.pl.i.type)
    then ms
    else m :: RMAux pl ms
```

NewGF creates a new generic function, overriding any old methods that might be there:

```
NewGF :: FunName -> GFList -> GFList
NewGF (n:FunName) (GF:GFList) =
  if IsGF n GF
    then NewGF n (RemoveGF n GF)
    else (n,[]) :: GF
```

RemoveGF removes a generic function n:

```
RemoveGF :: FunName -> GFList -> GFList
RemoveGF (n:FunName) ([]:GFList) = error
RemoveGF n (g:gs) =
```

```
  if n = g.name
    then gs
    else g :: RemoveGF n gs
```

ApplyGF applys a generic functions to an arguement list.

```
ApplyGF :: FunName -> ParamList -> GFList -> Object
ApplyGF (n:FunName) (pl:ParamList) ([]:GFList) = error
ApplyGF n pl (g:gs) =
  if n = g.name
    then SchemeApply (SpecificMethod pl g.methods) pl
    else AppleGF n pl gs
```

Where *SpecificMethod* chooses the most specific method to apply based on the types of the supplied parameters and *SchemeApply* is the *apply* function in Scheme modified so that if a function cannot be found in the Scheme top level namespace then the function name is treated as a generic function. This process resembles the *data-directed* programming of [2, pages 136–139]

5 Methods

Methods are Dylan's basic functional unit, taking a list of typed parameters and returning a typed value. Methods can be defined by the user, but are not applicable by the user: defining a method automatically creates a new generic function with the same name, and the new method is attached to the new generic function.

The only operation on methods is simply creating new methods. This is accomplished by creating a new, unique identifier (called a key) for this method, having Scheme bind the new key to the method's equivalent *lambda* expression in the Scheme namespace, and adding the parameter list and key to the appropriate generic function, creating a new generic function if required. Defining a new method must then alter the generic function space and the Scheme environment.

NewMethod creates a new method by modifying the Scheme environment and the generic function space:

```
NewMethod (n:FunName) (pl:ParamList) (l:Expr) (GF:GFList) (SE:Env) =
let key = MkUniqueKey GF in
if IsGF n GF
  then (AddMethod n pl key GF) , (bind key l SE)
  else NewMethod n pl l (AddMethod n [] Nil GF)
```

Where *MkUniqueKey* generates a new, unique key in the generic function space; and *bind* binds the key and the *lambda* expression in the Scheme environment.

6 Example

To help illustrate how Dylan works, we will use an example most Scheme programmers are familiar with: calculating the square root of a number using Newton's method from [3, pages 394–395]. Here is the Dylan code from [1, page 81]:

```
(define-method newtons-sqrt (x)
  (bind-methods ((sqrt1 (guess)
                   (if (close? guess)
                       guess
                       (sqrt1 (improve guess))))
                 (close? (guess)
                   (< (abs (- (* guess guess) x)) 0.0001))
                 (improve (guess)
                   (/ (+ guess (/ x guess)) 2)))
  (sqrt1 1)))
```

The call to *define-method* will create a generic function called *newtons-sqrt* that has one method with a parameter of type `<object>`[1], and will store the *lambda* expression in Scheme's top-level environment. The local functions created with *bind-methods* do not generate generic functions themselves—these functions are stored like any other Scheme local functions (in nested environments).

As an example, we will trace through the call `(newtons-sqrt 4)` to see in more detail how Dylan and Scheme work together.

- The generic function `newtons-sqrt` is found and the argument class `<number>` matches the method `<object>`, so the Scheme function associated with that method is invoked with parameter `4`.
- In the Scheme environment, the call `(sqrt1 1)` is evaluated.
- This generates the call `(close?  1)`.
- This generates the call `(* 1 1)`.
- `*` is not found the Scheme environment so it is re-evaluated as a generic function in Dylan.
- Dylan matches the types and dispatches the proper method to multiply `guess`.
- (and so on ...)

7 Omissions

Our analysis of Dylan, though incomplete, shows the heart of what sets Dylan apart from other functional and object-oriented languages. Some of the features omitted from our analysis include:

[1] since `x` wasn't given a more specific class, it defaults to the most general class `<object>`

Slot options Slots may have initial values, initializing functions, and storage allocation modifiers.

Additional method functions Methods can defer execution to more general methods.

Security options Methods, generic functions, and classes may be *sealed*, which prevents any modifications to these items.

Efficiency options Classes may be declared as *abstract* or *instantiable*, preventing instantiation of classes too general to be implemented efficiently. Abstract classes could be used as parents to more concrete, instantiable classes. Generic functions can still be defined to operate on the abstract classes since they can be overloaded to deal with all of the instantiable subclasses.

Leaving out these features means that we are no longer talking about "Dylan". However, we feel these additional features are simple enough to implement on top of what is presented here.

8 Conclusions

The Dylan language is a small, efficient way to get the benefits of object-oriented programming without writing a new language from scratch. Using the language Scheme as a starting point, Dylan gives the programmer the ability to define new types (classes) and typed functions (methods). Polymorphism is fundamental to Dylan, and it is provided through inclusion (class inheritance) and ad-hoc polymorphic functions (generic functions).

Lacking from Dylan are type inference and static type checking. Type inference could help verify correctness in the above example by inferring the type for the parameter `x` as `<number>` instead of `<object>`. But inference is complicated by Dylan's dynamic nature which allows the functions `abs`, `+`, `*` *et cetera* to be overloaded at runtime to operate on non-numeric objects. This dynamic nature also limits the use of any static type checking.

Since Dylan can be readily built by adding two new namespaces on top of Scheme: a graph representing the class heterarchy and a list of generic functions, each with a list of specific methods. These new namespaces combined with new functions (to support the namespaces and replace the Scheme functions) will form the core of the Dylan language. At this point, the ten class libraries can be written to flesh out the core language thus creating a general purpose programming environment.

All told, Dylan represents a useful addition to Scheme and could be the prototype for adding ad-hoc polymorphism and class inheritance to other functional languages such as Haskell and Miranda. Whether Dylan will "transform the way we program [computers]"[2] remains to be seen.

References

[1] Apple Computer Eastern Research and Technology. *Dylan: An Object Oriented Dynamic Language*. Apple Computer Inc. 1992

[2] [1, page 9]

[2] Jonathan Rees and William Clinger (editors). *Revised^3 Report on the Algorithmic Language Scheme.*

[3] Harold Abelson and Gerald Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press. Cambridge. 1985