Dylan

Hannes Mehnert
hannes@berlin.ccc.de

December 16, 2004

Overview

- DYnamic LANguage

Overview

- DYnamic LANguage
- Object Oriented: everything is an object

Overview

- DYnamic LANguage
- Object Oriented: everything is an object
- Safe: type-checking of arguments and values, no buffer overruns, no implicit casting, no raw pointers

Overview

- DYnamic LANguage
- Object Oriented: everything is an object
- Safe: type-checking of arguments and values, no buffer overruns, no implicit casting, no raw pointers
- Efficient: can compile to code nearly as efficient as C

Hello world

```
define method hello-world ()
  format-out(‘‘Hello world\n’’);
end method;
```

factorial

```
define method factorial ( i )
  if (i = 0)
    1
  else
    i * factorial( i - 1)
  end if
end method;
```

History

- Created by Apple Computer in the early 1990's

History

- Created by Apple Computer in the early 1990's
- sold as Technology Release by Apple in 1995

History

- Created by Apple Computer in the early 1990's
- sold as Technology Release by Apple in 1995
- CMU created Gwydion Dylan 1994 which is open source and maintained by Gwydion Dylan Maintainers (from USA, Germay, Japan, New Zealand, Hawaii,...)

History

- Created by Apple Computer in the early 1990's
- sold as Technology Release by Apple in 1995
- CMU created Gwydion Dylan 1994 which is open source and maintained by Gwydion Dylan Maintainers (from USA, Germay, Japan, New Zealand, Hawaii,...)
- Harlequin developed Harlequin Dylan, now called Functional Developer, owned by Functional Objects, since 9 months open source (also maintained by Geydion Dylan Maintainers)

Dylan roots

- Dylan was developed with object-orientated ideas from Lisp and smalltalk

Dylan roots

- Dylan was developed with object-orientated ideas from Lisp and smalltalk
- But Syntax looks more like Pascal

Dylan roots

- Dylan was developed with object-orientated ideas from Lisp and smalltalk
- But Syntax looks more like Pascal
- Unlike C and C++, Dylan uses no malloc() and free(), it has a garbage collector; no double-free bugs

Libraries

- Each Dylan program consist of one ore more Libraries

Libraries

- Each Dylan program consist of one ore more Libraries
- Libraries are the unit of compilation and boundaries of optimizations

Libraries

- Each Dylan program consist of one ore more Libraries
- Libraries are the unit of compilation and boundaries of optimizations
- Libraries contain modules

Library hello

```
define library hello
  use common-dylan;
  use io;
  export hello-world;
end library;
```

Modules

- Modules are namespaces

Modules

- Modules are namespaces
- Binding names are defined in a module

Modules

- Modules are namespaces
- Binding names are defined in a module
- Modules import names from other modules, and export names of bindings defined within

Modules

- Modules are namespaces
- Binding names are defined in a module
- Modules import names from other modules, and export names of bindings defined within
- Bindings that aren't exported are only visible in the module

Module hello-world

```
define module hello-world
  use common-dylan;
  use format-out;
  export hello-world;
end module;
```

Importing, Exporting and Renaming

```
use common-dylan, import: all;
use streams, import: { <string-stream> };
use io, exclude: { flush, seek };
use dylan, rename: { sort => dylan-sort };
use xml-parser, prefiz: ‘‘xml-’’;

use png-utils, export: { decode-png };
```

Interfaces

- Many Dylan libraries only contain one module

Interfaces

- Many Dylan libraries only contain one module
- Libraries may define several export modules that represent different "interfaces" on the same functionality, similar to c++'s public:, private:, and protected:

Library plug-in

```
define library plug-in
  use dylan;
  export plug-in,
         plug-in-implementor;
end library;
```

Module plug-in

```
define module plug-in
  use dylan;

  create <plug-in>,
         load-plug-in,
         plug-in-action,
         unload-plug-in;

  create plug-in-name;
end module;
```

Module plug-in-implementor

```
define module plug-in-implementor
  use dylan;

  create <simple-plug-in>,
         do-load-plug-in,
         do-plug-in-action,
         do-unload-plug-in;
end module;
```

Classes

- Classes define a type in the class hierarchy, and storage for object state, called "slots"

Classes

- Classes define a type in the class hierarchy, and storage for object state, called "slots"
- Every instance object has a class

Classes

- Classes define a type in the class hierarchy, and storage for object state, called "slots"
- Every instance object has a class
- Classes have no member functions

Classes

- Classes define a type in the class hierarchy, and storage for object state, called "slots"
- Every instance object has a class
- Classes have no member functions
- Classes are not namespaces (modules are)

Classes

- Classes are types. There are also types that are not classes.

Classes

- Classes are types. There are also types that are not classes.
- There is a root class "¡object¿". It is the root of the class and type hierarchy.

Classes

- Classes are types. There are also types that are not classes.
- There is a root class "¡object¿". It is the root of the class and type hierarchy.
- Classes are objects; you can pass them around, test properties, etc.

Classes

- Classes are types. There are also types that are not classes.
- There is a root class "¡object¿". It is the root of the class and type hierarchy.
- Classes are objects; you can pass them around, test properties, etc.
- Dynamic: You can create classes and subclasses at runtime

Class <point>

```
define class <point> (<object>)
  slot x;
  slot y;
end class;

let point = make( <point> );
unless (slot-initialized?( point, x ))
  point.x := 0;
  point.y := 0;
end;
```

Init Expressions

```
define class <point> (<object>)
  slot x = 0;
  slot y = 0;
end class;

let point = make( <point> );
   {<point>: x = 0, y = 0}
```

Init Keywords

```
define class <point> (<object>)
  slot x = 0, init-keyword: x:;
  slot y = 0, init-keyword: y:;
end class;

let point = make( <point>, y: 42 );
   {<point>: x = 0, y = 42}
```

Slot Inheritance

```
define class <point> (<object>)
  slot x = 0;
  slot y = 0;
  slot size = 1;
end class;
define class <thick-point> (<point>)
  inherited slot size = 10;
end class;
```

Slot Types

```
define class <point> (<object>)
  slot x = 0;
  slot y :: <integer> = 0;
end class;
let point = make( <point> );
point.x = "some text";
   {<point>: x = "some text", y = 0}
point.y = "some text";
   {<type-error>}
```

Slot Allocation

- instance slot x;
- class slot instance-count = 0;
- each-subclass slot quux;
- virtual slot bar;

Class Adjectives

- define concrete class
- define abstract class

Class Adjectives

- define concrete class
- define abstract class

- define sealed class
- define open class

Class Adjectives

- define concrete class
- define abstract class

- define sealed class
- define open class

- define free class
- define primary class

Generic Functions

- Generic Functions are polymorphic functions

Generic Functions

- Generic Functions are polymorphic functions
- The basis of polymorphism and implementation inheritance in Dylan

Generic Functions

- Generic Functions are polymorphic functions
- The basis of polymorphism and implementation inheritance in Dylan
- Contain one or more methods that provide an implementation for specific classes and types

Generic Functions

- Generic Functions are polymorphic functions
- The basis of polymorphism and implementation inheritance in Dylan
- Contain one or more methods that provide an implementation for specific classes and types
- Dynamic: Methods can be added/removed at runtime

Generic Functions

- Generic Functions are polymorphic functions
- The basis of polymorphism and implementation inheritance in Dylan
- Contain one or more methods that provide an implementation for specific classes and types
- Dynamic: Methods can be added/removed at runtime
- Are not "owned" by classes

Generic Function Dispatch

- When you call a generic function it dispatches the call to a specific method

Generic Function Dispatch

- When you call a generic function it dispatches the call to a specific method
- The most-specific method for the argument types will be called

Generic Function Dispatch

- When you call a generic function it dispatches the call to a specific method
- The most-specific method for the argument types will be called
- Multiple Dispatch: The types of all the required arguments are used for method dispatching; there is no distinguished "self" or "this" argument

double()

```
define generic double (o);

define method double (o :: <object>)
  pair( o, o )
end method;

define method double (n :: <number>)
  2*n
end method;

define method double (s :: <string>)
  concatenate( s, s )
end method;
```

capture?()

```
define method capture?
      (a :: <boris>, b :: <moose>)
  #f
end method;

define method capture?
      (a :: <boris>, b :: <squirrel>)
  #f
end method;
```

capture?()

```
define method capture?
      (a :: <moose>, b :: <boris>)
  #t
end method;

define method capture?
      (a :: <moose>, b :: <natasha>)
  #t
end method;
```

Object-oriented Programming

- Modules: Interfaces, Access Control, Namespaces
- Generic Functions: Polymorphism, Behaviors, Algorithms
- Classes: Inheritance, Types, Attributes

Types

- A type is a set of one or more values

Types

- A type is a set of one or more values
- Classes are types

Types

- A type is a set of one or more values
- Classes are types
- There is a root class "<object>". It is the root of the class and type hierarchy.

Types

- A type is a set of one or more values
- Classes are types
- There is a root class "<object>". It is the root of the class and type hierarchy.
- Types are objects; you can pass them around, test properties, etc.

Types

- A type is a set of one or more values
- Classes are types
- There is a root class "<object>". It is the root of the class and type hierarchy.
- Types are objects; you can pass them around, test properties, etc.
- Dynamic: You can create types at runtime

Singletons

- A singleton is a type with only one value; it is used to indicate a single object

Singletons

- A singleton is a type with only one value; it is used to indicate a single object
- Singleton types can be created with the singleton() function

Singletons

- A singleton is a type with only one value; it is used to indicate a single object
- Singleton types can be created with the singleton() function
- It is not the singleton pattern, where instantiating a singleton class always returns the one object

singleton()

```
define constant <just-42> = singleton(42);

instance?( 42, <just-42> );
   #t
instance?( 0, <just-42> );
   #f
```

```
define method fact (n == 0)
  1
end method;

define method fact (n :: <integer>)
  n * fact( n - 1 )
end method;

fact(0); // calls the first method
   1
fact(3); // calls the second method
   6
```

Union Types

- A union type is a type whose values include all the values of two or more other types

Union Types

- A union type is a type whose values include all the values of two or more other types
- Union types can be created with the type-union() function

Union Types

- A union type is a type whose values include all the values of two or more other types
- Union types can be created with the type-union() function
- Particularly useful for allowing values of disparate types without subclassing, e.g., "an rgb color, or a color table index, or a crayon color name string"

type-union()

```
define constant <speed> =
  type-union(<integer>, <symbol>);

define variable *speed* :: <speed> = 0;

*speed* := 93;
*speed* := #"fast";
*speed* := #"medium";
*speed* := #t;
   {<type-error>}
```

false-or()

```
define method false-or (type :: <type>)
  type-union( singleton( #f ), type )
end method;

let x :: false-or(<string>) = #f;
if (x)
  x
else
  x := "some text"
end if;
```

Limited Types

- A limited type is a type whose values are restricted to some subset of another type

Limited Types

- A limited type is a type whose values are restricted to some subset of another type
- Limited types can be created with the limited() function

Limited Types

- A limited type is a type whose values are restricted to some subset of another type
- Limited types can be created with the limited() function
- Limited integers can be used to represent subranges of integers

Limited Types

- A limited type is a type whose values are restricted to some subset of another type
- Limited types can be created with the limited() function
- Limited integers can be used to represent subranges of integers
- Limited collections can be restricted in the types of objects they can contain, and they can be length limited

limited(<integer>)

```
define constant <movie-rating> =
  limited( <integer>, from: 1, to: 10 );

let rating :: <movie-rating> = 10;

if (has-car-chases?( movie ))
  rating := rating + 1;
end if;
   {<type-error>}
```

Sealing

- Places limits on dynamism, both at runtime and compile time

Sealing

- Places limits on dynamism, both at runtime and compile time
- Reduces or eliminates runtime dispatch, type-checking, and other overhead

Sealing

- Places limits on dynamism, both at runtime and compile time
- Reduces or eliminates runtime dispatch, type-checking, and other overhead
- You can seal domains, generic functions, methods, classes, and slots

Sealing

- Places limits on dynamism, both at runtime and compile time
- Reduces or eliminates runtime dispatch, type-checking, and other overhead
- You can seal domains, generic functions, methods, classes, and slots
- Libraries are the boundaries of sealing

Element Reference - Function call

- sequence[i] – element(sequence, i)

Element Reference - Function call

- sequence[i] – element(sequence, i)
- array[i, j, ...] – aref(array, i, j, ...)

Element Reference - Function call

- sequence[i] – element(sequence, i)
- array[i, j, ...] – aref(array, i, j, ...)
- all-windows[0] – element(all-windows, 0)

Element Reference - Function call

- sequence[i] – element(sequence, i)
- array[i, j, ...] – aref(array, i, j, ...)
- all-windows[0] – element(all-windows, 0)
- tic-tac-toe[1, 2] – aref(tic-tac-toe, 1, 2)

Slot Reference – Function call

- argument.function – function(argument)

Slot Reference – Function call

- argument.function – function(argument)
- window.position – position(window)

Slot Reference – Function call

- argument.function – function(argument)
- window.position – position(window)
- window.view.origin – origin(view(window))

Slot Reference – Function call

- argument.function – function(argument)
- window.position – position(window)
- window.view.origin – origin(view(window))
- view(window).origin – origin(view(window))

Multiple Values

- Functions can return multiple values, just like they can accept multiple arguments

Multiple Values

- Functions can return multiple values, just like they can accept multiple arguments
- Eliminates the need for "output" parameters

Multiple Values

- Functions can return multiple values, just like they can accept multiple arguments
- Eliminates the need for "output" parameters
- There is no "wrapper" object for the values; for example, on PowerPC, function arguments are stored in r3, r4, r5, etc. Multiple values could be returned in r3, r4, r5, etc.

values()

```
values( 1, 2, 3 );
   1
   2
   3

define method square-and-sum (x, y)
  values( x ^ 2, x + y )
end method;

square-and-sum( 2, 3 )
   4
   5
```

if

```
if (camel.humps = 1)
  "dromedary"
elseif (camel.humps = 2)
  "bactrian"
else
  "not a camel"
end if;
```

unless

```
unless (danger?( will-robinson ))
  follow( dr-smith )
end unless;
```

case

```
case
  camel.humps = 1 => "dromedary";
  camel.humps = 2 => "bactrian";
  otherwise       => "not a camel";
end case;
```

select (by)

```
select (my-object by instance?)
  <window>, <view>   => "UI object";
  <number>, <string> => "computational";
  otherwise          => "unknown";
end select;
```

for

```
for (tree in forest)
  look-at( tree )
end for;

for (i from 1 to 10) ...

for (j from 0 below 10,
     k from 10 above 0 by -1) ...

for (thing = first-thing then next(thing),
     until: done?(thing)) ...
```

block

```
block (return)
  open-files();
  if (files-empty?())
    return( #f );
  end;
  process-files();
afterwards
  report-totals();
cleanup
  close-files();
end block;
```

let

```
let x = 0;

let sym :: <symbol> = #"green";

let (whole, rem) = truncate( amount );

let (whole :: <integer>, rem :: <real>) =
      truncate( amount );

let (x, #rest rest) = values(1, 2, 3);
x         1
rest      #(2, 3)
```

local

```
local method square (x)
        x*x
      end method;

let y = square( 12 );
   144

local method back ()
        forth()
      end,
      method forth ()
        back()
      end;
```

more thinks to talk about

- Collections

more thinks to talk about

- Collections
- Macros

more thinks to talk about

- Collections
- Macros
- C-Interface

Naming Conventions

- Names: multiple-words

Naming Conventions

- Names: multiple-words
- Types: <object>, <number>

Naming Conventions

- Names: multiple-words
- Types: <object>, <number>
- Globals: *foo*, *port*

Naming Conventions

- Names: multiple-words
- Types: <object>, <number>
- Globals: *foo*, *port*
- Constants: $months-per-year

Naming Conventions

- Names: multiple-words
- Types: <object>, <number>
- Globals: *foo*, *port*
- Constants: $months-per-year
- Predicates: odd?, subclass?

Naming Conventions

- Names: multiple-words
- Types: <object>, <number>
- Globals: *foo*, *port*
- Constants: $months-per-year
- Predicates: odd?, subclass?
- Mutative: sort!, reverse!

Implementations

- Gwydion Dylan: d2c compiles dylan to c, works on nearly every platfrom (Linux, FreeBSD, MacOSX,...)
- Lacks support for threads

Implementations

- Gwydion Dylan: d2c compiles dylan to c, works on nearly every platfrom (Linux, FreeBSD, MacOSX,...)
- Lacks support for threads
- Functional Developer: compiles to assembler, works only on Windows and x86 Linux
- Nice IDE for Windows

Implementations

- Gwydion Dylan: d2c compiles dylan to c, works on nearly every platfrom (Linux, FreeBSD, MacOSX,...)
- Lacks support for threads
- Functional Developer: compiles to assembler, works only on Windows and x86 Linux
- Nice IDE for Windows
- Apple Dylan: only released as Technology Preview

More information

- WWW: `http://www.gwydiondylan.org`
- IRC: freenode, #dylan
- Mail: gd-hackers@gwydiondylan.org