Writing better code (in Dylan)

Fast development of object-oriented functional programs

Andreas Bogk and Hannes Mehnert

Chaos Communication Congress, 27.12.2005

Requirements for programming languages

- universal
- powerful
- easy to learn
- performant
- easy to read
- open source implementation
- tools (debugger, profiler, IDE)
- libraries, frameworks
- secure!

Dylan competes!

ICFP Programming Contest

- 2001: Second Prize
- 2003: Judges Prize
- 2005: Second Prize and Judges Prize

In 2005, we competed against 168 teams!

History of Dylan

- dialect of lisp
- Ralph, the programming language for Apple Newton
- Apple, CMU, Harlequin
- Dylan Interim Reference Manual
- Dylan Reference Manual (DRM)

since DRM no longer prefix (lisp) syntax

Dylan Features

- Object-Oriented (from the grounds up)
- Multiple Inheritance (with superclass linearization)
- Multiple Dispatch
- First class functions
- First class types
- Dynamic typing
- Metaprogramming
- Optimistic Type Inferencing

Apple Dylan

- technology release (based on MCL)
- Apple Cambridge labs
- implementation on 68k, later PowerPC
- 1996 abandoned for lack of money

CMU

- gwydion project
- goal: development environment
- DARPA funded between 1994 and 1998
- dylan interpreter in C
- dylan compiler to C written in dylan
- since 1994 open source license (mostly BSD)
- since 1998 open development process

Harlequin

- dylan compiler written in dylan
- developers had many experience (lisp machine, LispWorks)
- native compiler with IDE (only on win32 so far)
- debugger, profiler, interactor, hot code update
- command line compiler for linux/x86
- originally a commercial development, since 2004 open source license (LGPL)
- large scale professional development (30 person years work just for the garbage collector)

Existing libraries

- DUIM (Dylan User Interface Manager)
- corba (2.0, some 2.2 features)
- ODBC
- network
- regular expressions
- dood (persistent object store)
- file system
- XML parser
- C interfaces to png, pdf, postgresql, sdl, opengl
- stand alone web server and proof of concept wiki
- ...

Syntax

Algol like syntax:

```
begin
  for (i from 0 below 9)
    format-out("Hello world");
  end for;
end
```

Naming Conventions

- allowed in names: +=-*<>
- - instead of _
- classes begin and end with angle brackets: <number>
- global variables begin and end with asterisks: *machine-state*
- program constants begin with a dollar sign: $pi
- predicate functions end with a question mark: even?
- destructive functions end with exclamation mark: reverse!
- getters and setters: element element-setter

Dynamically and strongly typed

- strong vs weak typing
- static vs dynamic typing

Object oriented

- class based object system
- everything is inherited from class <object>
- multiple inheritance, but the right way: superclass linearization
- difference to widely deployed object oriented programming languages: functions are not part of classes

Class definition

```
define class <square> (<rectangle>)
  slot x :: <number> = 0, init-keyword: x:;
  slot y :: <number> = 0, init-keyword: y:;
  constant slot width :: <number>,
     required-init-keyword: width:;
end class;
```

Keyword arguments

```
define function describe-list
    (my-list :: <list>, #key verbose?) => ()
  format(*standard-output*,
         "{a <list>, size: %d",
         my-list.size);
  if (verbose?)
    format(*standard-output*, ", elements:");
    for (item in my-list)
      format(*standard-output*, " %=", item);
    end for;
  end if;
  format(*standard-output*, "}");
end function;
```

Higher order functions

- anonymous functions (lambda calculus)
- closures
- curry, reduce, map, do
- function composition

Anonymous functions and closures

```
define function make-linear-mapper
    (times :: <integer>, plus :: <integer>)
 => (mapper :: <function>)
  method (x)
    times * x + plus;
  end method;
end function;

define constant times-two-plus-one =
  make-linear-mapper(2, 1);

times-two-plus-one(5);
// Returns 11.
```

Curry, reduce, map

```
let printout = curry(print-object, *standard-output*);
do(printout, #(1, 2, 3));

reduce(\+, 0, #(1, 2, 3)) // returns 6

reduce1(\+, #(1, 2, 3)) //returns 6

map(\+, #(1, 2, 3), #(4, 5, 6))
 //returns #(5, 7, 9)
```

Function composition, interfacing to C

```
define interface
  #include "ctype.h",
    import: {"isalpha" => is-alphabetic?,
     "isdigit" => is-numeric?},
    map: {"int" => <boolean>};
end interface;

define constant is-alphanumeric? =
  disjoin(is-alphabetic?, is-numeric?);
```

Generic functions

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

define method double
    (x :: <number>) => result
  2 * x;
end method;
```

Multiple dispatch

```
define method inspect-vehicle
  (vehicle :: <vehicle>, i :: <inspector>) => ();
  look-for-rust(vehicle);
end;
define method inspect-vehicle
    (car :: <car>, i :: <inspector>) => ();
  next-method(); // perform vehicle inspection
  check-seat-belts(car);
end;
define method inspect-vehicle
    (truck :: <truck>, i :: <inspector>) => ();
  next-method(); // perform vehicle inspection
  check-cargo-attachments(truck);
end;
define method inspect-vehicle
    (car :: <car>, i :: <state-inspector>) => ();
  next-method(); // perform car inspection
  check-insurance(car);
end;
```

Optional type restrictions of bindings

```
define method foo
    (a :: <number>, b :: <number>)
  let c = a + b;
  let d :: <integer> = a * b;
  c := "foo";
  d := "bar"; // Type error!
end
```

Serves on the one hand as assert, on the other hand type inference.

Macros

```
define macro with-open-file
  { with-open-file (?stream:variable = ?locator:expression,
                    #rest ?keys:expression)
      ?body:body
    end }
  => { begin
         let ?stream = #f;
         block ()
           ?stream := open-file-stream(?locator, ?keys);
           ?body
         cleanup
           if (?stream & stream-open?(?stream))
             close(?stream)
           end;
         end
       end }
end macro with-open-file;
```

A simple for-loop...

```
let collection = #[1, 2, 3];
for (i in collection)
  format-out("%=\n", i);
end for;
```

… is in real a macro with iterator …

```
let (initial-state, limit, next-state, finished-state?,
     current-key, current-element) =
  forward-iteration-protocol(collection);
local method repeat (state)
   block (return)
     unless (finished-state?(collection, state, limit))
       let i = current-element(collection, state);
       format-out("%=\n", i);
       repeat(next-state(collection, state));
     end unless;
   end block;
 end method;
repeat(initial-state)
```

... which gets optimized to a simple loop.

```
while (1) {
  if ((L_state < 3)) {
    L_PCTelement = SLOT((heapptr_t)&literal_ROOT,
                        descriptor_t,
                        8 + L_state_2 * sizeof(descriptor_t));
    [...]
    L_state = L_state + 1;
  } else {
    goto block0;
  }
}
block0:;
```

Type unions

```
define constant <green-thing> =
  type-union(<frog>, <broccoli>);

define constant kermit = make(<frog>);

define method red?(x :: <green-thing>)
  #f
end;

red?(kermit) => #f
```

False-or, singleton

```
type-union(singleton(#f, type)) == false-or(type)

define method find-foo (x) => (index :: false-or(<integer>
 ... //returns index if found, false if not found
end method say;

type-union(symbol1, symbol2) == one-of(symbol1, symbol2)
define method say (x :: one-of(#"red", #"green", #"blue"))
   ...
end method say;
```

Nonlocal exits

```
block (return)
  open-files();
  if (something-wrong)
    return("didn’t work");
  end if;
  compute-with-files()
cleanup
  close-files();
end block
```

Exceptions

```
block ()
  open-files();
  compute-with-files()
exception (<error>)
  "didn't work";
cleanup
  close-files();
end block
```

Library and Module

```
define library hello-world
  use dylan, import: all;
  use io, import: { format-out };
  export hello-world;
end library;

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

Links

- WWW: `http://www.gwydiondylan.org/`
- Dylan Programming: `http://www.gwydiondylan.org/books/dpg/`
- Dylan Reference Manual: `http://www.gwydiondylan.org/books/drm/`
- IRC: irc.freenode.net, #dylan
- mailing list: gd-hackers@gwydiondylan.org