Dylan (Dynamic Language)

A multi-paradigm language

Oliver Juwig

Aachen, 12. Februar 2003
Agenda

- History of Dylan
- Concepts of the language
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- Dylan in a real project
Dylan has been inspired by functional programming languages

- The language was inspired by Scheme and CommonLisp
  - Dylan is a superset of Scheme
  - Extensions:
    - CommonLisp Object System
    - CommonLisp Condition System
    - A production-rule based macro processor
- But: Language syntax is more like Pascal or Modula
- Dylan is a best of breed language; it combines object-oriented, functional and algorithmic programming paradigms
Dylan has been developed by several partners

- Three major partners has been involved in the development of the language
  - Carnegie-Mellon University (Project Gwydion)
  - Digital Corporation
  - Apple Computers, Inc.
  - Harlequin

- Dylan is a general purpose language, but it was targeted for small devices in the beginning (similar to Java)
  - Apple Newton

- The first language draft appeared 1993
Current resources for Dylan

- Functional Objects - a commercial IDE provider

- Gwydion project
  - http://gwydiondylan.org

- Goodies, libraries, ...
  - http://monday.sourceforge.net/wiki

- Forums / Newsgroups

History of Dylan
Agenda

- History of Dylan
- **Concepts of the language**
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- Dylan in a *real* project
Dylan combines extrem object-oriented concepts with static and dynamic programming style
Every little thing in Dylan is an object

- All objects are first class, even:
  - Numbers and characters
  - Classes
  - Functions
- Every object can be used as a function argument
- All objects are typed and type-safe
- Variables can be strongly typed
- All objects derive from class `<object>"
A strong subset of the CommonLisp Object System is used in Dylan

- Strong support of multiple inheritance
- Slots are functions (so called slot methods, they can be specialized)
- Classes define no methods in addition to the slot methods
- Scope of names is not defined by classes
  - Dylan has an explicit name space concept based upon libraries and modules

```dylan
define open abstract class <presentation> (<object>)
  keyword cached:, type: <boolean>, init-value: #f;

  sealed slot name :: <normalized-descriptor>, init-keyword: name:;
  sealed slot controller-class :: limited(<class>, subclass-of: <presentation-controller>);

  sealed slot state :: <object>, init-value: #f;

  sealed slot available-controllers :: <vector>, init-value: #[];
  sealed slot active-controllers :: <vector>, init-value: #[];

  sealed slot lru-count :: <integer>, init-value: 0;
  sealed slot cached? :: <boolean>, setter: #f, init-value: #f, init-keyword: cached:;
end class <presentation>;
```
The core concept of Dylan is the <function>.

- A <method> is a callable unit of code identified by a fixed parameter signature.

```dylan
define open generic (&sequence1 :: <sequence>, &sequence2 :: <sequence>, #key) => result :: <sequence>;

define method intersection (&sequence1 :: <lazy-sequence>, &sequence2 :: <sequence>, #key key: &key :: <function> = identity, test: &test :: <function> = \\==) => result :: <lazy-sequence>;

choose(method (&item)
    member?(&item, &sequence2, key: &key, test: &test)
end method,
&sequence1);
end method intersection;
```

- Functions support required parameters, named keyword parameters and an arbitrary number of additional (called rest) parameters.
A complete collection framework including functional iteration and mapping is supplied

- Supplied iteration and mapping functions:
  - do, map, map-as, map-into, member?, any?, every?, choose, ...
The condition (exception) handling of Dylan surpasses everything known in the C-world

- Both exceptions and handlers are objects
- Handlers can fix the problem and return to the signalling code block, or they can exit to any point in there lexical scope
- Exceptions can be tested for furthermore environmental conditions before handled
- Several restart protocols exist

Simple sample

```dylan
define method show-presentation (&presentation :: <descriptor>, &data, ...) => (#rest results :: <object>);
...
block ()
&environment := lookup-environment(&environment);
exception (<error>)
kill-thread(current-thread());
end block;
...
end method;
```

More complex sample

```dylan
local method run-block-lock ()
  let &run :: <integer> = 0;
  while (#t)
    block (return)
      let handler <kill> = method (&condition :: <kill>, &next :: <function>)
      if (&run > 1000)
        return();
      end if;
      &next();
    end method;
    &run := &run + 1;
  end block;
end while;
end method;
```

... // Test code comes here...
Agenda

- History of Dylan
- Concepts of the language
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- Dylan in a *real* project
Dylan combines multiple inheritance with the so called multimethod dispatch

- Classes define no direct methods beside the slot methods
- Dylan uses generic functions for the active part of a program
- Each generic function can be compromised of methods that adhere to the parameter contract of this function
- Parameters of methods are strongly typed and define the type of arguments the method can be applied to
- The number of the so called applicable methods are computed for every concrete argument situation prior to the function invocation
- The applicable methods are sorted according to their specificity using a class precedence list (short: CPL) algorithm
- The most specific method is invoked
- A method can invoke the next most specific method by calling `next-method();`
Even operators are generic functions in Dylan and can be specialized

Define generic method 
\+
(x :: <object>,
 y :: <object>) => z :: <object>;

Define method 
\+(r :: <ratio>, i :: <integer>) => (s :: <rat>)
 r + make(<ratio>, numerator: i)
end \+;

Define method 
\+(s :: <string>, t :: <string>) => (r :: <string>)
 concatenate(s, t);
end method \+;

After this definition a program can write:

let h :: <string> = "Hello";
let w :: <string> = "World";

let hw :: <string> = h + " " + w;
Multimethods allows much more cleaner code than using traditional object-oriented techniques

class Shape {
    bool intersect(Shape s) {
        /* generic case - slow */
    }
}
class Rect {
    bool intersect(Shape s) {
        if (s instanceof Rect) {
            /* simple and fast */
        }
        else {
            super.intersect(s);
        }
    }
}
class Circle {
    bool intersect(Shape s) {
        if (s instanceof Circle) {
            /* simple and fast */
        }
        else {
            super.intersect(s);
        }
    }
}

define generic intersect (s1 :: <shape>,
                         s2 :: <shape>)
    => <boolean>;

define method intersect (s1 :: <shape>,
                         s2 :: <shape>)
    => <boolean>;
end method;

define method intersect (s1 :: <rect>,
                         s2 :: <rect>)
    => <boolean>;
end method;

define method intersect (s1 :: <circle>,
                         s2 :: <circle>)
    => <boolean>;
end method;
Methods can be added to generic functions at any time

- When a generic function is not sealed, methods can be added by everyone at any point in the program.
- Methods can be added syntactically according to the name scope or by using a call to `add-method(...)`;
- Every method must comply with the parameter signature of the generic function.
- Methods can belong to several generic functions (esoteric feature).
- Methods can be even removed from a generic function by calling `remove-method(...)`;
- A complete set of introspection function are available.
A complex sample

```
define generic likes? (a :: <life-form>, b :: <life-form>) => likes? :: <boolean>
  #f;
end method likes?;

define method likes? (cat :: <cat>, whoCares :: <life-form>) => likes? :: <boolean>
  gives-food?(whoCares);
end method likes?;

define method likes? (dog :: <dog>, human :: <human>) => likes? :: <boolean>
  pet?(human, dog);
end method likes?;

define method likes? (man :: <man>, woman :: <woman>) => likes? :: <boolean>
  looks-good?(woman);
end method likes?;

define method likes? (woman :: <woman>, man :: <man>) => likes? :: <boolean>
  intelligent?(man) & rich?(man);
end method likes?;
```
Agenda

- History of Dylan
- Concepts of the language
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- Dylan in a real project
Dylan provides a complete set of functional programming constructs

- **Functional composition**
  - **Closures**
  - **Function invocation**
  - **Iteration constructs**
    - do, map, reduce, etc.

```dylan
define method addX (n :: <number>, x :: <number>)
  n + x;
end;

define method add2 (x :: <integer>, y :: <integer>)
  # do(method (a, b) print(a + b); end, end)
    #((100, 200, 300, 400), #(1, 2, 3));
  101 202 303
  => #f
  # map(\+, #(1, 2, 3), #(4, 5, 6));
  # => #(5, 7, 9)
  => # reduce(max, 10, #(2, 4, 6, 11));
  => 11
  # score-david(0);
  => 100
  # score-david(10);
  => 110
  # score-david(20);
  => 130
  # score-diane(70);
  => 470

define method make-score (points :: <number>)
  method (increase :: <number>)
    points := points + increase;
  end;
end;

define constant score-david = make-score(100);

define constant score-diane = make-score(400);

# score-david(0);
=> 100
# score-david(10);
=> 110
# score-david(20);
=> 130
# score-diane(70);
=> 470

define method addX (n :: <number>, x :: <number>)
  n + x;
end;

define constant add10 = rcurry(addX, 10);

define constant less100? = rcurry(\<, 100);

define constant equal100? = curry(\=, 100);

define constant greater100? = conjoin(complement(less100?), equal100?);

define constant greater90? = compose(greater100?, add10);

# greater90?(99);
=> #t
# greater90?(87);
=> #f

define method add2 (x :: <integer>, y :: <integer>)
  x + y;
end;

# apply(add2, list(1, 2));
=> 3
# apply(max, list(4, 6, 7));
=> 7
# do(method (a, b) print(a + b); end, end)
  #((100, 200, 300, 400), #(1, 2, 3));
  101 202 303
  => #f
  # map(\+, #(1, 2, 3), #(4, 5, 6));
  # => #(5, 7, 9)
  # => #(5, 7, 9)
  # => # reduce(max, 10, #(2, 4, 6, 11));
  => 11
  # score-david(0);
  => 100
  # score-david(10);
  => 110
  # score-david(20);
  => 130
  # score-diane(70);
  => 470

# reduce1(\+, #(1, 2, 3, 4, 5));
=> 15

# choose(even?, #(3, 1, 4, 5, 6, 2));
=> #(4, 6, 2)

# choose-by(even?, range(from: 1),
  #("a", "b", "c", "d", "e", "f", "g", "h");
  => #("b", "d", "f", "h")
```
Sample from a real application

define function build-from-part (&environment :: <sequence>,
    &expression :: <select-query-expression>,
    #rest &args) => (from-part :: <string>,
    hints :: false-or(<string>));

let &variables :: <sequence> = remove-duplicates!(as(<vector>, first(&environment).variables));
let &classes :: <sequence> = choose(rcurry(instance?, <transformed-class>), &variables);
let &structs :: <sequence> = choose(rcurry(instance?, <transformed-struct>), &variables);
let &sequences :: <sequence> = choose(rcurry(instance?, <transformed-sequence>), &variables);
let &first? :: <boolean> = #t;  let &from-part :: <string> = $empty-string;
let &hints = #f;

local method make-table-from-string (&alias :: <string>, &table :: <string>)
  ...
end method;

unless (empty?(&classes))
  &classes := choose(complement(compose(empty?, specifity)), &classes);

unless (empty?(&classes))
  let &class :: <transformed-class> =
    first(sort!(&classes, key: compose(curry(reduce, \+, 0), specifity)));
  let &specifity :: <integer> = reduce(max, 0, &class.specifity);

  ...
end unless;
end unless;
Agenda

- History of Dylan
- Concepts of the language
- Multidimensional polymorphism - a silver bullet
- Functional programming with Dylan
- Dylan in a *real project*
Dylan can be used successfully in very large commercial information systems

- Project FIM for GEMA
  - Central application of the company
  - About 50 person-years development effort
  - About 4 million lines of code written completely in Dylan
    - 1.5 million lines of code for technical framework
    - 2.5 million lines of code for the application
  - About 400 dialogs and 45 batches
  - About 800 persistent classes (tables) in the underlying database
  - Some tables have about 30 million entries

- The project has been started before Java matured and has gone into production before the first J2EE application server was commercially available
Some screenshots of FIM (1/4)
Some screenshots of FIM (2/4)
Some screenshots of FIM (3/4)
Some screenshots of FIM (4/4)
Discussion

Contact: Oliver Juwig
sd&m Research
software design & management
Thomas-Dehler-Str. 27, 81737 München, Germany
Tel +49 89 63812-653, Fax -911
Tel +49 2241 9737-413
Mobile +49 171 3105384

mailto:oliver.juwig@sdm.de
http://www.sdm-research.de