Extensibility for DSL design and implementation

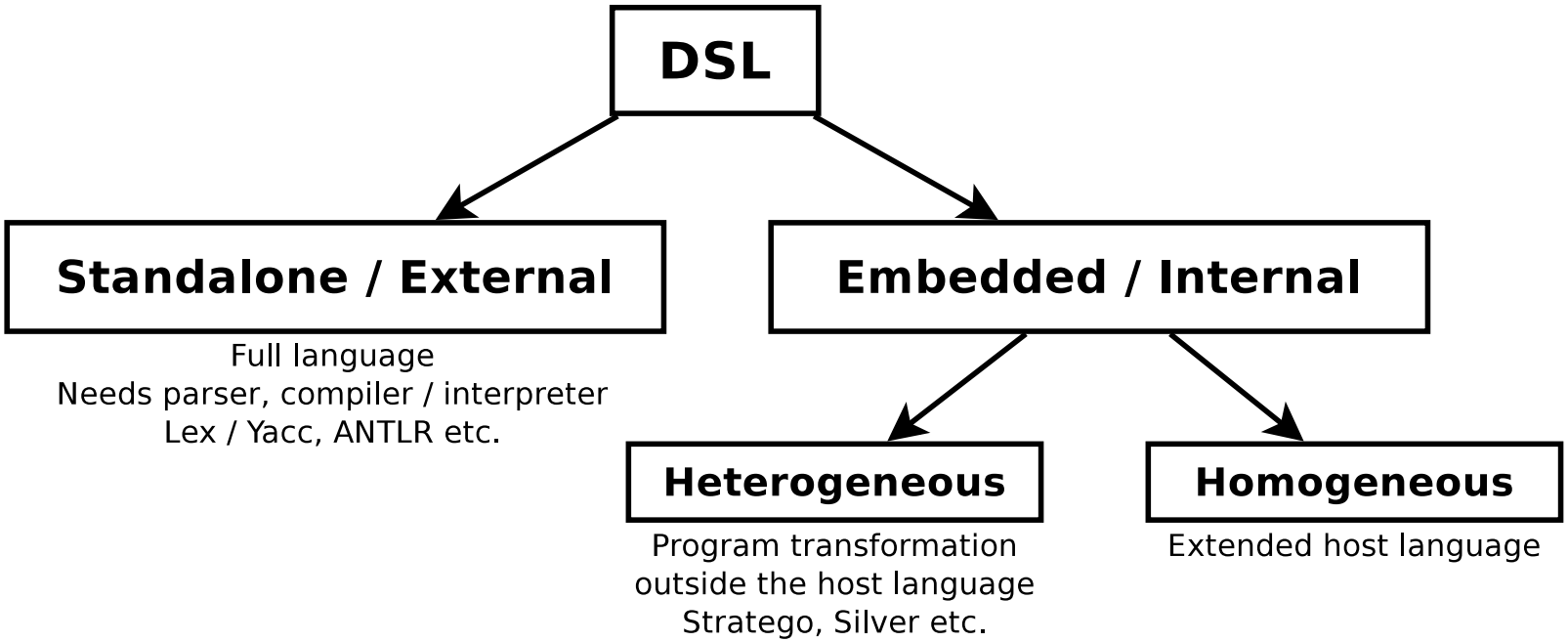
A case study in Common Lisp

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

DSLDI 2013 – Monday, July 1st

Taxonomy of DSLs
[Fowler, 2005, Tratt, 2008]



Example
Command-line options highlighting


```
Usage: advanced [-hd] [+d] [OPTIONS] cmd [OPTIONS]

Available commands: push pull.
Use 'cmd --help' to get command-specific help.
  -h, --help                    Print this help and exit.
  -(+)d, --debug[=on/off]       Turn debugging on or off.
                                Fallback: on
                                Environment: DEBUG
```

- Properties (bold, underline, foreground color...)
- Faces (localized property set)
- Themes (face trees)

Example
Underlying implementation

The `face` class

```
(defclass face ()
  ((name :initarg :name)
   ;; Properties:
   (foreground :initarg :foreground)
   (background :initarg :background)
   (boldp :initarg :bold)
   ;; etc.
   (subfaces :initarg :subfaces)))
```

Example
A DSL for theme customization

How do we go from this...

```
;;; default.cth --- Personal default theme for Clon

:background black
:face { option :foreground white
               :face { syntax :bold t :foreground cyan }
               :face { usage :foreground yellow }
      }
```

...to that?

```
(setq default-theme
      (make-instance 'face :name 'toplevel
        :background 'black
        :subfaces (list (make-instance 'face :name 'option
                          :foreground 'white
                          :subfaces (list (make-instance 'face :name 'syntax
                                            :bold t
                                            :foreground 'cyan)
                                          (make-instance 'face :name 'usage
                                            :foreground 'yellow))))))
```

Step 1
Hook into the Lisp parser: reader macros

- *readtable*: currently active syntax extensions table
- *macro character*: special syntactic meaning
- *reader macro*: implements macro character behavior

Let's do it!

- Make the `{}` characters active
- Read a list of tokens until the closing brace
- Push the symbol `define-face` on top of that list

Note: RTMP (Read-Time Meta-Programming)

Step 1
Hook into the Lisp reader

This is how we go from this...

```
;;; default.cth --- Personal default theme for Clon

:background black
:face { option :foreground white
               :face { syntax :bold t :foreground cyan }
               :face { usage :foreground yellow }
      }
```

...to that:

```
;;; default.cth --- Personal default theme for Clon

:background black
:face (define-face option :foreground white
                  :face (define-face syntax :bold t :foreground cyan)
                  :face (define-face usage :foreground yellow))
```

Step 2
Hook into the Lisp compiler: macros

- Ordinary Lisp functions
- Work on chunks of code (as data)
- Transform expressions into new expressions
- Control over evaluation

Let's make `define-face` a macro!

- Quoting its key arguments, except for the `:face` ones
- Generating a call to `make-face`

Note: CTMP (Compile-Time Meta-Programming)

Step 2
Hook into the Lisp compiler: macros

This is how we go from this...

```
;;; default.cth --- Personal default theme for Clon

:background black
:face (define-face option :foreground white
                  :face (define-face syntax :bold t :foreground cyan)
                  :face (define-face usage :foreground yellow))
```

...to that:

```
;;; default.cth --- Personal default theme for Clon

:background 'black
:face (make-face 'option :foreground 'white
                 :face (make-face 'syntax :bold t :foreground 'cyan)
                 :face (make-face 'usage :foreground 'yellow))
```

Step 3
A couple of wrappers

Lambda-list manipulation / 1st class functions

```
(defun make-face (name &rest args &key &allow-other-keys)
  (apply #'make-instance 'face :name name args))

(defun make-theme (&rest args)
  (apply #'make-face 'toplevel args))
```

And while we're at it...

```
(defmacro define-theme (&rest args)
  `(define-face toplevel ,@args))
```

Step 3
A couple of wrappers

This is how we go from this...

```
;;; default.cth --- Personal default theme for Clon

:background 'black
:face (make-face 'option :foreground 'white
                 :face (make-face 'syntax :bold t :foreground 'cyan)
                 :face (make-face 'usage :foreground 'yellow))
```

...to that:

```
;;; default.cth --- Personal default theme for Clon

:background 'black
:face (make-instance 'face :name 'option :foreground 'white
         :face (make-instance 'face :name 'syntax :bold t :foreground 'cyan)
         :face (make-instance 'face :name 'usage :foreground 'yellow))
```

Step 4
Hook into the object system: the MOP

The CLOS Meta-Object Protocol (MOP)

- **CLOS *itself* is object-oriented**
  - The CLOS MOP: a *de facto* implementation standard
  - The CLOS components (classes, methods *etc.*) are (meta-)objects of some (meta-)classes

Generic functions, methods

```
(defmethod func ((arg1 class1) arg2 ...)
  body)
```

Methods are *outside* the classes (ordinary function calls)

Multiple dispatch (multi-methods)

Step 4
Hook into the object system: the MOP

- Object instantiation (`make-instance`) is a protocol
- Slot initialization (`initialize-instance`) is a generic function

Let's extend it!

- Provide our own method for the `face` class
- Collect all `:face` arguments
- call the next (standard) method with a new `:subfaces` initarg

Step 4
Hook into the object system: the MOP

This is how we go from this...

```
;;; default.cth --- Personal default theme for Clon

:background 'black
:face (make-instance 'face :name 'option :foreground 'white
       :face (make-instance 'face :name 'syntax :bold t :foreground 'cyan)
       :face (make-instance 'face :name 'usage :foreground 'yellow))
```

...to that:

```
;;; default.cth --- Personal default theme for Clon

:background 'black
:subfaces (list (make-instance 'face :name 'option :foreground 'white
                 :subfaces (list (make-instance 'face
                                  :name 'syntax :bold t :foreground 'cyan)
                                 (make-instance 'face
                                  :name 'usage :foreground 'yellow))))
```

Wrap Up
Using the DSL externally

Mostly a matter of `read`, `compile` *etc.*

```
(defun read-user-theme ()
  (with-open-file (stream (merge-pathnames ".faces" (user-homedir-pathname)))
    (read (make-concatenated-stream (make-string-input-stream "(define-theme ")
                                    stream
                                    (make-string-input-stream ")")))))

(defmacro make-user-theme (&optional compile)
  (if compile
      `(funcall (compile nil (lambda () ,(read-user-theme))))
      (read-user-theme)))
```

Wrap Up
Using the DSL internally

Mostly a matter of. . . Just-Do-It™

```
(setq default-theme
      (define-theme
          :background black
          :face { option :foreground white
                         :face { syntax :bold t :foreground cyan }
                         :face { usage :foreground yellow }
                }
        ))
```

Conclusion

- Impact of GPL on DSL design and implementation
- Key GPL aspect: extensibility
- Embedded homogeneous approach
  - A single language
  - DSL infrastructure smaller
  - DSL both internal and external
- Common Lisp
  - Functional, Imperative, Object-Oriented
  - MOP
  - CTMP (macros)
  - RTMP (reader macros)
  - `read`, `eval`, `compile`

Internal *vs* External DSLs
[Kamin, 1998, Czarnecki et al., 2004]

- Sub-optimal syntax **ok but...**
  Not ok:
  - [Fowler, 2010]: “external DSLs have their own custom syntax and you write a full parser to process them”
  - [Kamin, 1998, Czarnecki et al., 2004]: “a prerequisite for embedding is that the syntax for the new language be a subset of the syntax for the host language”
  - BTW, same disagreement at the semantic level (MOP)
- Poor error reporting
  - Research: [Tratt, 2008]
  - Lisp: ? (but Cf. condition system & restarts)

Controversial aspects of extensibility

- Dynamic typing
  - pros: end-user friendly
  - cons: run-time type errors / checking
  - Research: [Taha and Sheard, 1997]
  - Hybrid languages (Cf. Racket)
- Lazy Evaluation
  - pros: infinite data structures, new control primitives *etc.*
  - cons: pure functional languages only
  - Lisp: laziness through macros (not as straightforward), but side-effects for free, and still functional.

The root of (Lisp) extensibility

- Reflection
  - Introspection
  - Intercession
- Implementation
  - By API
  - Inherent: “homoiconicity” [McIlroy, 1960, Kay, 1969]
- Further distinction [Maes, 1987, Smith, 1984]
  - Structural Reflection (program)
  - Behavioral Reflection (language)

Bibliography I

Ahson, S. I. and Lamba, S. S. (1985).
The use of Forth language in process control.
Computer Languages, 10:179–187.

Czarnecki, K., O Donnell, J., Striegnitz, J., and Taha, W. (2004).
DSL implementation in MetaOCaml, Template Haskell, and C++.
In Lengauer, C., Batory, D., Consel, C., and Odersky, M., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin / Heidelberg.

Bibliography II

Denert, E., Ernst, G., and Wetzel, H. (1975).
Graphex68 graphical language features in algol 68.
Computers & Graphics, 1(2-3):195–202.

van Deursen, A., Klint, P., and Visser, J. (2000).
Domain-specific languages: an annotated bibliography.
SIGPLAN Notices, 35:26–36.

Elliott, C. (1999).
An embedded modeling language approach to interactive 3D and multimedia animation.
IEEE Transactions on Software Engineering, 25:291–308.

Bibliography III


Fleutot, F. and Tratt, L. (2007).
Contrasting compile-time meta-programming in Metalua and Converge.
In *Workshop on Dynamic Languages and Applications*.


Fowler, M. (2005).
Language workbenches: The killer-app for domain specific languages?


Fowler, M. (2010).
Domain Specific Languages.
Addison Wesley.

Bibliography IV

Hudak, P. (1998).
Modular domain specific languages and tools.
In *Proceedings of the 5th International Conference on Software Reuse*, ICSR'98, pages 134–142, Washington, DC, USA. IEEE Computer Society.

Kamin, S. N. (1998).
Research on domain-specific embedded languages and program generators.
In *Electronic Notes in Theoretical Computer Science*, volume 14. Elsevier.

Kay, A. C. (1969).
The Reactive Engine.
PhD thesis, University of Hamburg.

Bibliography V


Maes, P. (1987).
Concepts and experiments in computational reflection.
In *OOPSLA*. ACM.


McIlroy, M. D. (1960).
Macro instruction extensions of compiler languages.
Commun. ACM, 3:214–220.


McNamara, B. and Smaragdakis, Y. (2000).
Functional programming in C++.
SIGPLAN Not., 35:118–129.


Pagan, F. (1979).
ALGOL 68 as a metalanguage for denotational semantics.
The Computer Journal, 22(1):63–66.

Bibliography VI

Prud'homme, C. (2006).
A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations.
Journal of Scientific Programming, 14:81–110.

Rompf, T., Sujeeth, A. K., Lee, H., Brown, K. J., Chafi, H., Odersky, M., and Olukotun, K. (2011).
Building-blocks for performance oriented DSLs.
In *DSL'11: IFIP Working Conference on Domain-Specific Languages*.

Skalski, K., Moskal, M., and Olszta, P. (2004).
Meta-programming in Nemerle.
Technical report.

Bibliography VII

Smith, B. C. (1984).
Reflection and semantics in Lisp.
In *Symposium on Principles of Programming Languages*, pages 23–35. ACM.

Taha, W. and Sheard, T. (1997).
Multi-stage programming with explicit annotations.
In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '97, pages 203–217, New York, NY, USA. ACM.

Bibliography VIII

Tratt, L. (2008).
Domain specific language implementation via compile-time meta-programming.
ACM Transactions on Programming Languages and Systems, 30:31:1–31:40.

Tratt, L. (2005).
Compile-time meta-programming in a dynamically typed OO language.

Vasudevan, N. and Tratt, L. (2011).
Comparative study of DSL tools.
Electronic Notes in Theoretical Computer Science, 264:103–121.