Interfacing Python with Lisp

Tobias-Christian Rittweiler

November 3, 2009

Problem description

Existing Common Lisp application:

- generates computing model from geometric data describing parts of air engines
- computing model describes physical effects on these parts

Problem:

- people familiar with Common Lisp are in minority
- yet: people supposed to parametrize computing model

An idea...

Write a configuration DSL (Domain Specific Language)

Requirements:

- good notation for arithmetical expressions
- common number types, including:
  - dimensional numbers with convenient literal syntax (e.g. 12mm)
- integration/reuse of existing object hierarchy
- read/write access to existing persistent data layer
- runnable from command line
- beyond simple parametrization: (future objective)
  - influence of data / control flow (callbacks)

Python?

Why not use Python as a DSL?

- good notation for arithmetical expressions ✔
- common number types ✔
  - dimensional numbers with convenient literal syntax (e.g. 12mm) ✖
- integration/reuse of existing object hierarchy ✔ / ✖
- read/write access to existing persistent data layer ✔ / ✖
- runnable from command line ✔
- beyond simple parametrization ✔

CLPython!

CLPython is an implementation of (more or less) Python 2.3 in Common Lisp.

- (an implementation of the language, not of all modules written in C)[1]
- last point does not affect us: we want Python as an infix general-purpose DSL
- written in Lisp means integrating with Lisp comes rather easy
- last but not least: it's written in Lisp, so it's easily hackable...

[1] CLPython: batteries excluded.

Dimensional Numbers

Hacking the parser:

- extend syntax of numeric literals (translate $12m^2$ etc. to DimNumbers)
- do so optionally, that is in right context (if certain module is imported)

Plug DimNumber into Python's number tower:

- Example: `2cm + 13mm` => `2.3cm`
  `2cm + 13mm`^{2} => error

Because it's Lisp, this was mostly done in third-party contrib code. Only marginal changes on CLPython itself were needed.

Persistent data

Create Python object “Database”:

- internally interface it to existing Lisp code dealing with persistent storage
- specialize on `__enter__`, `__exit__` methods

Result:

```
with Database(‘‘/path/to/db’’) as db:
    ...
    ...
    db.commit()
```

Integration into existing CL stuff

- with CLPython, it's Lisp all the way down → straightforward to write an "F"FI
- Common Lisp, the programmable programming language:

```
 (define-wrapper-class CascadeParameter (Parameter)
     :wrapping turbine.design::CascadeParameter
   (nAirfoils :required t   :type integer)
   (parts     :required t   :type py-list
              :marshalling (py-list<->lisp-list))
   (flowpath  :required nil :type file))

# In Python:
cascade_parameter = CascadeParameter(
    nAirfoils = 53,
    flowpath  = ...,
    parts     = [ ..., ..., ...  ]
)
```

Integration (cont..)

`DEFINE-WRAPPER-CLASS` allows to conveniently

- make Lisp class available to the Python world via a wrapper class
- specify what slots Python code may access
- specify types of the slots (checked at construction / setting)
- specify required arguments for the constructor
- specify different slot names for Python and Lisp code
- specify marshalling if necessary (e.g. how to map strings in Python to symbols in Lisp)

Real life example

```
import MTU
import MTU.DimNumber
import MTU.Parasolid
import MTU.Aero
import MTU.Design
import MTU.Struct
import MTU.Common

# print MTU.getargs()
database = MTU.getoptval('-db')

with MTU.Database(database) as db:
  class Aero:
    airfoil_parameter = MTU.Aero.AirfoilParameter(
        airfoilType = "Blade",
        airfoil = MTU.File("aad468.xmt_txt",
                           "parasolid"),
        skeleton = MTU.Aero.SkeletonParameter(
            nSections = 10,
            nPoints = 10)
    )
```

Real life example (cont..)

```
module_parameter = MTU.Aero.ModuleParameter(
    moduleType = "LPC",
    flowpath = MTU.File("flowpath-aad454.xmt_txt",
                        "parasolid"),
    cascade = [ None,
                None,
                MTU.Aero.RotorParameter(
                    nAirfoils = 53,
                    airfoil = airfoil_parameter),
                None ]
)

module = MTU.Aero.run_LPC(module_parameter)
```