Integrating Foreign Code in Common Lisp

(was: Naturalizing Foreign Libraries)

Rudi Schlatte

rudolf.schlatte@joanneum.at

Why foreign code?

- Speed
- Effort to reimplement
- Reference implementation of evolving standard
- Over-the-wire protocol not documented, but library provided

The easy way out: the C API

```
(let ((world (librdf_new_world)))
  (librdf_world_open world)
  (let ((uri (librdf_new_uri
               world
            "http://example.com/")))
    (prog1
        (librdf_uri_to_string uri)
      (librdf_free_uri uri)
      (librdf_free_world world))))
```

This is ugly!

- Foreign data is just opaque pointers
- No introspection, debuggability
- Hunt for memory leaks, just like in the old times

What do we want?

- The application programmer should not be able to tell whether the package was implemented in Lisp or not

Checklist

- Use wrapper objects
- Use designators
- Simplify resource handling

Use wrapper objects

- Enable method dispatch
- State in wrapper object vs state in foreign library
  - don‘t duplicate state, it will hurt
  - state on Lisp side can be inspected / modified

Use designators

- Seamless integration between Lisp datatypes and your classes
  - (pathname „/home/rudi/foo.txt“)
  - (uri „http://example.com/“)

Simplify resource handling

- Provide (with-...)-style macros for lexical scope
- Integrate with garbage collector for indefinite scope
  - <audience getting restless here>
  - Provide explicit close method, use GC as safety net

```
(let ((world (librdf_new_world)))
  (librdf_world_open world)
  (let ((uri (librdf_new_uri
              world
           "http://example.com/")))
    (prog1
        (librdf_uri_to_string uri)
      (librdf_free_uri uri)
      (librdf_free_world world))))
```

The new version

```
(uri-to-string
  (uri "http://example.com/"))
```

Miscellaneous

- Implement a condition hierarchy
- Implement print-object, describe-object
- Make it asdf-installable