Alien GOO
A Lightweight C Embedding Facility

Jonathan Bachrach

MIT CSAIL

Quick Goo Intro

- Dynamic type-based object-oriented language
  - Interpreter semantics
  - Classes, multiple inheritance, multimethods
- Simpler, more dynamic, lisp-syntaxed Dylan ***

```
(defclass <packet> (<any>))
(defslot packet-name (<packet> => <str>))
(defgen add (l|<seq> x))
(defmet add (l|<lst> x) (pair x l))
```

- An object-oriented Scheme
- Dynamic C backend
  - Used for listener as well

*** *For the purposes of this talk, I expand definition names a bit*

How to interface C to Goo?

- Say you want multiprecision support

- Type and data definitions
- Memory management
- Variable references
- Call outs
- Call backs
- Automation mechanisms
  - Declarative definitions
  - Header parsing

Problems

- Syntactic mismatches
  - Infix versus prefix
- Type and object format mismatches
  - Tagged versus untagged
- Semantic mismatches
  - Pointers
  - Garbage collection

Alien Goo Idea

- Embed C code directly in host language
- Escape to host language as needed
- Rely on C for its type and data system
- Use only as much of library as needed
- Use macros for automation
- Write convenient interface in one step!

OUTLINE

- GOO intro
- Challenge, problems, and idea
- Previous work in Python ***
- Basics
  - Statements and expressions
  - GOO escapes
- Live demos
- Interplay with macros
  - Quasiquote and embedding C forms
  - Macro defining macros and layered interfaces
- Issues, future, and acknowledgements

C Extension Modules

- Wrap C functions in Python Module by hand
- C API for Python
  - Importing / exporting data
  - Reference counting
  - Error handling
  - Calling python from C
  - Abstract object layer
  - Low level functions
  - Defining new types
  - Registering modules

C Extension Mod Example

```
#include “Python.h”

int gcd (int x, int y) { … }

PyObject *spam_gcd(PyObject *self, PyObject *args) {
  int x, y, g;
  if (!PyArg_ParseTuple(args, “ii”, &x, &y))
    return NULL;
  g = gcd(x, y);
  return Py_BuildValue(“I”, g);
}

Static PyMethodDef spammethods[] = {
  {“gcd”, spam_gcd, METH_VARARGS},
  { NULL, NULL }
};

Initspam(void) {
  Py_InitModule(“spam”, spammethods);
}
```

Mo' Extending

- Python glue code: `setup.py`

```
# setup.py
From distutils.core import setup, Extension
Setup(name="spam", version="1.0",
      ext_modules=[Extension("spam", ["spam.c", "spamwrapper.c"])])
```

- Building extension module

```
> python setup.py build
```

- Using module

```
>>> import spam
>>> spam.gcd(63, 56)
7
```

C Extension Mod Problems

- Tedious
- Verbose
- No automation support

SWIG

- Language neutral
- Semi automatic C interface parser
- Produces C files
- Functions called in host language
- Variable referenced through function calls
- Performs run time type checking
- Users can tailor type mapping

SWIG EXAMPLE

```
%init sock

%{
#include <sys/types.h> …
struct sockaddr *new_sockaddr_in(short family, …) { … }
char *my_gethostbyname(char *hostname) { … }
%}

enum {AF_UNIX, AF_INET, … };
#define SIZEOF_SOCKADDR sizeof(struct sockaddr)
int socket(int family, int type, int protocol);
%name gethostbyname { char *my_gethostbyname(char *); }

%include unixio.i
```

MO' SWIGGIN

```
Unix> wrap -python socket.i
Unix> gcc -c socket_wrap.c -I/usr/local/include/Py
Unix> ld -G socket_wrap.o -lsocket -lnsl -o sockmodule.so
```

```
# Python script
from sock import *
PORT = 5000
sockfd = socket(AF_INET, SOCK_STREAM, 0)
…
close(sockfd)
```

SWIG Problems

- Produces clunky interfaces
- Produces big C files
- No easy extensibility

ctypes

- Imports dlls exposing namespace
- Manually specify type interfaces
  - Clone of C type system in python
  - Arg and res types
    - Res defaults to int
    - Automatic support for str, int, or unicode
- Call funs in python syntax
  - Extra mechanism for call by ref and callbacks
  - Values must be looked up through calls

ctypes examples

```
>>>print cdll.msvcrt.time(None)

>>>strchr = cdll.msvcrt.strchr
>>>Strchr.restype = c_char_p
>>>print strchr("abcdef", "d")
'def'
```

CTYPE PROBLEMS

- Large mirroring of C type system
- No automation mechanisms

PyInline

- Permits definition of C code snippets
- C code specified as python strings
- Works for other languages

PyInline Examples

```
m = pyinline.build(code="""
   double my_add(double a, double b) {
      return a + b;
   }
""", language="C")
print m.my_add(4.5, 5.5)
```

PyInline Problems

- Cumbersome C snippets
- No python escapes

PYREX

- Python dialect for producing C modules for python
- Intermix c and python
- Python mirror of C type system
- Vars can be typed by C types
- Optimized C code produced when all ref'd vars are c typed

Pyrex Example

```
cdef extern from "cups/cups.h":
  ctypedef struct cups_option_t:
    char *name
    char *value
  …
  int cupsGetDests
        (cups_dest_t **dests)
  ctypedef cups_option_t
  …
def get_dests():
  cdef cups_dest_t *dests
  cdef cups_dest_t currDest
  numDests = cupsGetDests(&dests)
  retval = []
  for i in range(numDests):
    currDest = dests[i]
    retval.append(currDest.name)
  return retval
```

PYREXING

```
Unix> python2.2 pyrexc pyxcups.pyx
Unix> gcc -c -fPIC -I/usr/include/python2.2/pyxcups.c
Unix> gcc -shared pyxcups.o -lcups -o pyxcups.so
```

```
#python script
import pyxcups
for printer in pyxcups.get_dests():
  print printer
```

Pyrex problems

- Mirror of C type system
- Whole other python dialect

Weave Inline

- Allows inclusion of C code within python
- Can reference Python vars from C code

Weave Example

```
a = 'string'
weave.inline(r'printf("%d\n", a);', ['a'])

def c_int_binary_search(seq, t):
  code = """
         int val, m, min = 0;
         int max = seq.length() - 1;
         PyObject *py_val;
         for (;;) {
           if (max < min) {
             return_val = Py::new_reference_to(Py::Int(-1));
             break;
           } …
         }
        """
   return inline(code, ['seq', 't'])
```

Weave Inline Problems

- Somewhat cumbersome
- Limited python escapes
  - Have to resort to Python's C interface
- No automation mechanisms

Pros/Cons

Name	*Pros*	*Cons*
SWIG	Declarative, language neutral	Heavyweight and limited extensibility
Ctypes	Loads dlls	Mirrored c types
Pyinline	Lighter weight	Awkward and no python escapes
Pyrex	Integrated	Another Python dialect
Weave	Even lighter weight	Still awkward, limited python escapes, limited extensibility

Summary

- Previous solutions are either too heavy or complicated
  - Space speed
  - Amount of extra C code
- Complicated or nonexistent customization
- Weave is most similar but
  - Has limited python escapes
  - Is a bit long winded
  - Provides no extensibility

Alien GOO

- Embed C code directly in GOO
  - No awkward syntax
  - No displacement
- Escape to GOO as needed
  - Variable references
  - Arbitrary GOO expressions
- Rely on C for its type system and data
- Customize with macros
- Write interface in one step!

C Statements

- Consider construction of simple opengl layer on top of GOO
- Simplified initialization

```
(defmet gl-setup () #{ glutInitWindowSize( 640, 480 ); })
```

- C statement form `#{ … }` form
  - escapes to C
  - executes a series of C statements
  - evaluates to false
  - reader macro for `(c-ment #" … "#)`

GOO ESCAPES

- Next we define a drawing function

```
(defmet gl-vertex (x|<int> y|<int>)
  #{ glVertex3i($x, $y); })
```

- Where $ operator escapes back into GOO evaluating the following GOO sexpr
  - `#{ … }` reader macro for `(c-ment [c-snippet | form]+)`
- Can also be used to
  - Assign back to GOO variables

```
#{ $x = f($y); }
```

  - Create pointers to GOO objects

```
#{ f(&$x); }
```

C EXPORTS

- But x and y must first be exported to C

```
(defmet gl-vertex (x|<int> y|<int>)
  #{ glVertex3i($(to-c x), $(to-c y)); })
```

- Where to-c converts GOO object to C format
  - Predefined for `<log> <int> <chr> <str>`
    - But, flo's must be treated specially ***
  - User extensible
- Provide @ shorthand

```
(defmet gl-vertex (x|<int> y|<int>)
  #{ glVertex3i(@x, @y); })
```

C Expressions

- Often need to get values back from C functionally
- Introduce C expression `#ex{ … }`
- Same as C statement except
  - Value is value of enclosed C expression
  - Modifier x specifies interpretation

    `i` for `<int>`, `f` for `<flo>`, `s` for `<str>`,
    `c` for `<chr>`, `b` for `<log>`, `l` for `<loc>`
- For example, can define constant

```
(dv $gl-line-loop #ei{ GL_LINE_LOOP })
```

Top Level C Code

- Top level C code can be defined at GOO top level with #{ ... }
- In order to define a callback

```
#{ int gl_idle(int x) { $(gl-idle); } }
(defmet gl-idle () …)
```

- Can use this for typedefs, structure definitions, and includes

```
#{ #include <gl.h> }
```

- Can link libraries as follows

```
(use/library glut)
```

LIVE DEMOS

- printf

```
(df f () #{ printf("goo sucks\n"); })
(df f (x) #{ printf("give me %d bucks\n", @(+ x 9)); }
```

- getpid

```
(df f () #ei{ getpid() })
```

- goo loop

```
(for ((i (below 10)))
  #{ printf("hey %d\n", @i); } )
```

Large GOO Interfaces

- Want to define a GOO layer to a large and regular C library, say gmp for bignums ***
- Could just start by defining functions

```
(use/library gmp)
#{ #include "gmp.h" …
   static inline mpz_ptr bignum_to_mpz(P obj) { … }
   … }
(defmet + (x|<bignum> y|<bignum> => <int>)
  (let ((res 0))
    #{ mpz_t z; mpz_init_zero(z);
       mpz_add(z, bignum_to_mpz($x), bignum_to_mpz($y));
       $res = mpz_to_goo(z); }
       res))
```

*** *Actually used for bignum support in latest GOO*

Macros

- But going to be defining a bunch so want macros to ease the burden
- Start by making returning values easier

```
(defmac with-returning (,res ,@body)
  `(let ((,res #f)) ,@body ,res))
```

- Making original look as follows

```
(defmet + (x|<bignum> y|<bignum> => <int>)
  (with-returning res
    #{ mpz_t z; mpz_init_zero(z);
       mpz_add(z, bignum_to_mpz($x), bignum_to_mpz($y));
       $res = mpz_to_goo(z); } ))
```

Body Defining Macros

- But many bignum method bods have similar form
  - Gmp variable initialization
  - GOO specific body
  - Conversion back to GOO
- Can make body defining macro

```
(defmac with-gmp-returning (,z ,body)
  (let ((res (gensym)) (zc (to-str z)))
    `(with-returning ,res
       #{ mpz_t $,zc; mpz_init_zero(z);
          $,body
          $,res = mpz_to_goo($,zc); })))
```

- Note quasiquote's unquote within C form
  - Turns back on GOO evaluation
  - If it evaluates to a string it's consider more C code

Body Mac Usage and Beyond

- Original addition definition becomes

```
(dm + (x|<bignum> y|<bignum> => <int>)
  (with-gmp-returning z
    #{ mpz_add(z, bignum_to_mpz($x), bignum_to_mpz($y));
  } ))
```

- Many GOO wrapper methods have this form
  - Differ only in gmp arithmetic function called

Declarative GMP

- Can make method defining macro

```
(defmac def-b-b (,name ,c-fun)
  `(dm ,name (x|<bignum> y|<bignum> => <int>)
     (with-gmp-returning z
       #{ $,c-fun(z, bignum_to_mpz($x), bignum_to_mpz($y)); })))
```

- Now can define wrapper more declaratively

```
(def-b-b + "mpz_add")
```

- Can also define macros for other types

```
(def-b-b * "mpz_mul")
(def-b-i * "mpz_mul_si")
(defmet * (x|<fixnum> y|<bignum> => <int>) (* y x))
```

Even More Declarative

- Moving forward

```
(defmac def-log-ops (,name ,c-fun)
   `(seq (def-b-b ,name ,c-fun)
         (defmet ,name (x|<fixnum> y|<bignum> => <int>)
           (,name (to-bignum x) y))
         (defmet ,name (x|<bignum> y|<fixnum> => <int>)
           (,name x (to-bignum y)))))

(def-log-ops & "mpz_and")
(def-log-ops ^ "mpz_xor")
```

Callbacks Revisited

- Callbacks were

```
#{ int gl_idle(int x) { $(gl-idle); } }
(defmet gl-idle () …)
```

- Could define callback macro

```
(defmac (def-c-callback ,name (,@sig) ,@body)
  (let ((c-name    (gensym))
        (arg-names (map arg-name (sig-args sig))))
    `(seq #{ P ,c-name (,@arg-names) {
                 return $(,name ,@arg-names); } }
          (defmet ,name (,@args) ,@body))))
```

- Callbacks become

```
(def-c-callback gl-idle () …)
```

Layered Interfaces Recap

- Showed how macros interoperate with embedded C forms
- Define a layer of automation macros for
  - Returning values
  - Defining bodies
  - Defining wrapper methods
  - Callbacks
- Can use the appropriate level for given job
- Defines the conversion and glue code in one step producing a convenient lightweight interface

Conclusion

- Alien GOO is a lightweight, powerful, and extensible C interface mechanism
- Embeds C directly in GOO
- Allows escapes back and forth GOO
- Interoperates seamlessly with macros

- Makes
  - Simple C call outs and backs easy
  - GOO interfaces to C libraries manageable

Limitations

- No error checking
- Relies on conservative GC
- Still not entirely happy with to-c mechanism

Applicability

- Could work for other host languages but relies on C backend and C compiler
- Could work for languages other than C
- Range of possibilities
  - Embed C directly
  - Direct escapes to host language
    - Variables
    - Arbitrary expressions
  - Macros

Future Work

- Semi automatic C interface macros
- Error checking
- Non pointer sized returning C expressions
- Other host languages
- Other embedded languages

Acknowledgements

- Andrew Sutherland
  - Wrote GOO SWIG backend
  - Wrote GOO x GTK interface
    - Many megabytes of C code
    - Still required lots more glue code
- James Knight
  - Thought there had to be a better way
  - Suggested embedding C code directly

Questions

- Send me mail

  jrb@ai.mit.edu
- GOO is GPL

  www.googoogaga.org