Automatic Generation of Library Bindings Using Static Analysis

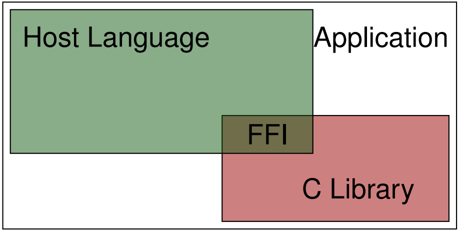
Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit
{travitch,sjackso,aderhold,liblit}@cs.wisc.edu

June 18, 2009

The Problem

Many development projects incorporate high-level languages. Often, they must use existing code written in other languages (typically C):

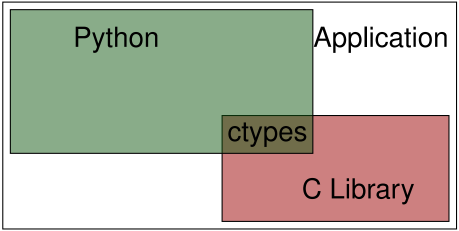
- Code is expensive to port
- Sharing code is often desirable
- Direct system access is uncommon in high-level languages



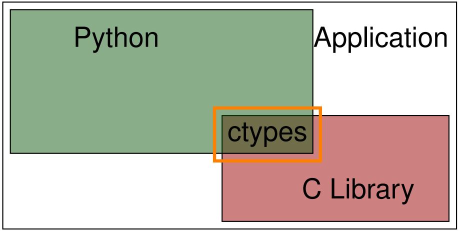
The Problem

Many development projects incorporate high-level languages. Often, they must use existing code written in other languages (typically C):

- Code is expensive to port
- Sharing code is often desirable
- Direct system access is uncommon in high-level languages



Our Goal



We want to automatically generate idiomatic C library bindings.

Current Solutions

- Most high-level languages have FFIs
- SWIG and related tools can scan library headers to generate bindings
- Library-specific binding generators that rely on convention

~~Current Solutions~~ More Problems

- Most high-level languages have FFIs
- SWIG and related tools can scan library headers to generate bindings
- Library-specific binding generators that rely on convention

Direct FFI Use is Error Prone

```
void pyg_register_pointer(GType pointer_type,
                          PyTypeObject *type)
{
  Py_TYPE(&type) = &PyType_Type;
  type->tp_base = &PyGPointer_Type;
}
```

```
void pyg_register_pointer(GType pointer_type,
                          PyTypeObject *type)
{
  Py_TYPE(type) = &PyType_Type;
  type->tp_base = &PyGPointer_Type;
}
```

Adapted from pygobject

Direct FFI Use is Error Prone

Before

```
void pyg_register_pointer(GType pointer_type,
                          PyTypeObject *type)
{
  Py_TYPE(&type) = &PyType_Type;
  type->tp_base = &PyGPointer_Type;
}
```

After

```
void pyg_register_pointer(GType pointer_type,
                          PyTypeObject *type)
{
  Py_TYPE(type) = &PyType_Type;
  type->tp_base = &PyGPointer_Type;
}
```

Adapted from pygobject

This was GNOME Bug 550463

Existing Binding Generators Require Annotations

A function call in C

```
int min_i;
int max_i;
gsl_stats_int_minmax(&min_i, &max_i, data, 1, 1);
```

Existing Binding Generators Require Annotations

A function call in C

```c
int min_i;
int max_i;
gsl_stats_int_minmax(&min_i, &max_i, data, 1, 1);
```

And in Python
...

```python
min_i = c_int()
max_i = c_int()
gsl_stats_int_minmax(byref(min_i), byref(max_i),
                     data, 1, 1)
```

Library-Specific Bindings

Great results, but effort does not translate to other libraries

- PyQt (about 2000)
- java-gnome (1998)
- tkinter (1995)

Why Annotations are Required

C function types are a lossy encoding of intent:

- Pointers are ambiguous
- Object ownership is implicit

Pointers Are Ambiguous

```
void __archive_check_magic(struct archive *a,
                           unsigned int magic)
{
  if (a->magic != magic) {
    diediedie();
  }
}
```

Adapted from libarchive

Pointers Are Ambiguous

```c
int prefix_w(const wchar_t *start,
             const wchar_t *end,
             const wchar_t *test)
{
  if(start == end) return 0;
  if(*start++ != *test++) return 0;

  while(start < end  &&  *start++ == *test++)
    ;

  if(start < end) return 0;

  return 1;
}
```

Adapted from libarchive

Pointers Are Ambiguous

```
double gsl_frexp(const double x, int *e)
{
  int ei = (int) ceil(log(fabs(x)) / M_LN2);
  double f = x * pow(2.0, -ei);

  while (fabs(f) >= 1.0) {
    ei++;
    f /= 2.0;
  }

  *e = ei;
  return f;
}
```

Adapted from GSL

Pointers Are Ambiguous

```
int BZ2_bzBuffToBuffCompress(char *dest, int *destLen)
{
   bz_stream strm;
   int ret;

   strm.next_out = dest;
   strm.avail_out = *destLen;

   ret = BZ2_bzCompress(&strm, BZ_FINISH);

   *destLen -= strm.avail_out;
   BZ2_bzCompressEnd(&strm);
   return BZ_OK;
}
```

Adapted from bzip2

Pointers And Resources

Consider the standard C function

```c
char *strdup(const char *s)
```

Pointers And Resources

Consider the standard C function

```
char *strdup(const char *s)
```

Compare with another standard C function

```
char *asctime(const struct tm *tm)
```

Our Goal (Again)

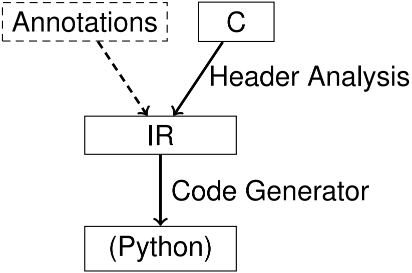
We want natural C library bindings. This means:

- Use multiple return values
- Convert native sequence types
- Integrate with the garbage collector

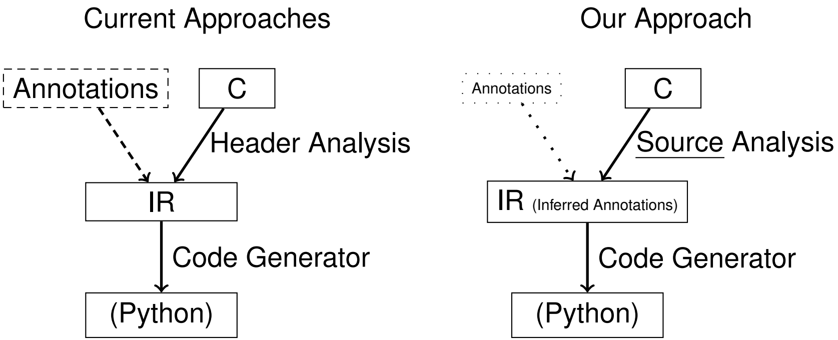
All as conveniently as possible (few to no annotations) without compromising safety.

Approach

Current Approaches



Approach

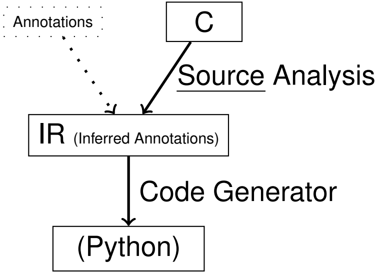

Current Approaches
Annotations
C
Header Analysis
IR
Code Generator
(Python)
Our Approach
Annotations
C
Source Analysis
IR (Inferred Annotations)
Code Generator
(Python)

Approach

Static analysis of library source:

- Output parameters
- Array parameters
- Resource management functions

Our Approach



Analysis Preliminaries

We assume a few preliminary transformations to input source code:

- Each function has a unique exit node
- The program is represented in SSA form (with global value numbering)

For simplicity of presentation, assume no pointer aliasing within functions

Output Parameters (What they look like)

```
double gsl_frexp(const double x, int *e)
{
  int ei = (int) ceil(log(fabs(x)) / M_LN2);
  double f = x * pow(2.0, -ei);

  while (fabs(f) >= 1.0) {
    ei++;
    f /= 2.0;
  }

  *e = ei;
  return f;
}
```

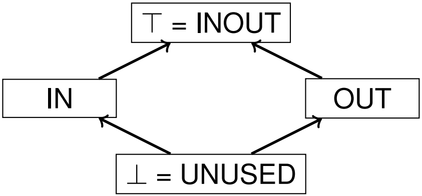
Out parameter, adapted from GSL

Finding Output Parameters

We formulate this as a dataflow problem tracking the uses of pointer parameters:

- For each pointer parameter p, the initial state is $\bot$
- The join operation for any statement s is

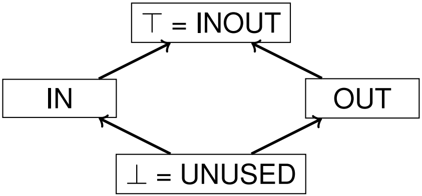
$$\bigsqcup_{p \in pred(s)}$$



Finding Output Parameters

The transfer function for each statement s using parameter p depends on the syntactic form of s:

- $*$p = e
- $*$p
- f(p)
- Otherwise, $exit_s(p) = entry_s(p)$.



Expressing Output Parameters

We have recovered some programmer intent:

- Multiple Return Values
- Python uses tuples
- Example:

```
int f_c(int x, int* y, int* z);
```

Expressing Output Parameters

```
int f_c(int x, int* y, int* z);
```

```
def f_py(x):
```

Expressing Output Parameters

```
int f_c(int x, int* y, int* z);
```

```
def f_py(x):
  tmp_y = c_int()
  tmp_z = c_int()
```

Expressing Output Parameters

```
int f_c(int x, int* y, int* z);
```

```
def f_py(x):
    tmp_y = c_int()
    tmp_z = c_int()
    tmp_ret = f_c(x, byref(tmp_y), byref(tmp_z))
```

Expressing Output Parameters

```
int f_c(int x, int* y, int* z);
```

```
def f_py(x):
  tmp_y = c_int()
  tmp_z = c_int()
  tmp_ret = f_c(x, byref(tmp_y), byref(tmp_z))

  return (tmp_ret, tmp_y, tmp_z)
```

And Then, Idiomatic Python

Compare calls to the `frexp` function in C and Python/ctypes

```c
int exp;
double frac = frexp(x, &exp);
```

```python
exp = c_int()
frac = frexp(x, byref(exp))
```

And Then, Idiomatic Python

Compare calls to the `frexp` function in C and Python/ctypes

```c
int exp;
double frac = frexp(x, &exp);
```

```python
exp = c_int()
frac = frexp(x, byref(exp))
```

Our generated wrapper is simpler

```python
(frac,exp) = frexp(x)
```

Identifying Array Parameters

Parameters used in array contexts can be treated as arrays. To find them:

Let

arrays $= \{v | v = *(ptr + offset)\}$

and:

arrays

$v_1 = *(ptr + o_1) \rightarrow \boxed{\ldots | v_2 | \ldots}$

$v_2 = *(v_1 + o_2) \rightarrow N$

$v_3 = *(y + o_3) \rightarrow \boxed{\ldots | z | \ldots}$

Identifying Array Parameters

Parameters used in array contexts can be treated as arrays. To find them:

Let

arrays $= \{v | v = *(ptr + offset)\}$

and:

1. Consider pairs
   $v_1 = *(ptr + o_1)$ and
   $v_2 = *(v_1 + o_2)$

arrays

$v_1 = *(ptr + o_1) \rightarrow \boxed{\ldots | v_2 | \ldots}$

$v_2 = *(v_1 + o_2) \rightarrow N$

$v_3 = *(y + o_3) \rightarrow \boxed{\ldots | z | \ldots}$

Identifying Array Parameters

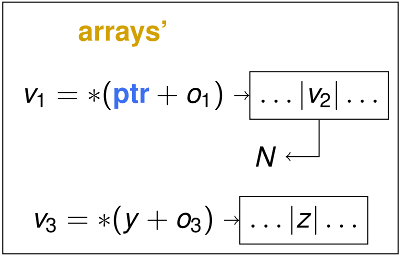
Parameters used in array contexts can be treated as arrays. To find them:

Let

arrays $= \{v | v = *(ptr + offset)\}$

and:

1. Consider pairs $v_1 = *(ptr + o_1)$ and $v_2 = *(v_1 + o_2)$
2. Extend the **base** array of v_1 and create **arrays'**



Identifying Array Parameters

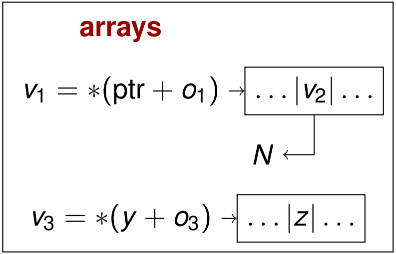
Parameters used in array contexts can be treated as arrays. To find them:

Let

arrays $= \{v | v = *(ptr + offset)\}$

and:

1. Consider pairs
   $v_1 = *(ptr + o_1)$ and
   $v_2 = *(v_1 + o_2)$
2. Extend the **base** array of v_1 and create **arrays'**
3. Find a fixed-point.



Identifying Array Parameters

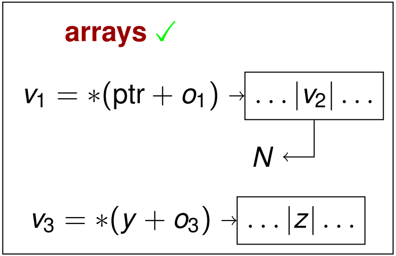
Parameters used in array contexts can be treated as arrays. To find them:

Let

arrays $= \{v | v = *(ptr + offset)\}$

and:

1. Consider pairs $v_1 = *(ptr + o_1)$ and $v_2 = *(v_1 + o_2)$
2. Extend the **base** array of v_1 and create **arrays'**
3. Find a fixed-point.



Sequence Mapping

We know which parameters are used as arrays; we want to automatically convert native sequences.

1. For each array parameter p of function f_C, generate a wrapper function f_{py} which checks the argument in that position before calling f_C
2. If p is a Python list, allocate a C array of the same dimensionality as p
3. Perform a shallow copy of the elements in p

Object Ownership

- In C, memory is typically managed manually
- Function prototypes do not describe allocation
- We employ an ownership model to recover this information

Object Ownership

- In C, memory is typically managed manually
- Function prototypes do not describe allocation
- We employ an ownership model to recover this information
    - Allocators create an object and give up ownership.
      Base allocators: `malloc` and `calloc`

Object Ownership

- In C, memory is typically managed manually
- Function prototypes do not describe allocation
- We employ an ownership model to recover this information
  - Allocators create an object and give up ownership. Base allocators: `malloc` and `calloc`
  - Finalizers assume sole ownership of objects. Base finalizer: `free`

Ownership Transfer

```
void add_property(component* c, property* p) {
  pvl_push(c->properties, p);
}

void component_free(component* c) {
  property* p;
  while((p=pvl_pop(c->properties)) != 0) {
      property_free(p);
  }
  pvl_free(c->properties);
  free(c);
}
```

An escaping parameter, adapted from libical

Identifying Allocators

For each function *f*, examine the unique exit node, *f* is an allocator if:

- All incoming values are the results of known allocators
- No incoming values escape

Identifying Allocators

For each function *f*, examine the unique exit node, *f* is an allocator if:

- All incoming values are the results of known allocators
- No incoming values escape

```
if(size > 0)
  return malloc(size * sizeof(int));
else
  return malloc(sizeof(int));
```

Identifying Allocators

For each function *f*, examine the unique exit node, *f* is an allocator if:

- All incoming values are the results of known allocators
- No incoming values escape

```
if(size > 0)
  return malloc(size * sizeof(int));
else if(size == 0)
  return malloc(sizeof(int));
else
  return NULL;
```

Identifying Allocators

For each function *f*, examine the unique exit node, *f* is an allocator if:

- All incoming values are the results of known allocators
- No incoming values escape

```
static void* lastAlloc = NULL;
if(size > 0)
  lastAlloc = malloc(size * sizeof(int));
else if(size == 0)
  lastAlloc = malloc(sizeof(int));
else
  return NULL;

return lastAlloc;
```

Identifying Finalizers

- A function *f* finalizes pointer parameter *p* if it passes *p* to a finalizer on every path.

Identifying Finalizers

- A function *f* finalizes pointer parameter *p* if it passes *p* to a finalizer on every path.
- We employ another dataflow analysis, tracking the dataflow fact finalized-or-`NULL` for each *p*.

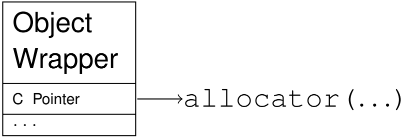
```
if (!object) return;

free(object->field);
free(object);
```

Host-Language Memory Management

We want the host-language runtime to do three things:

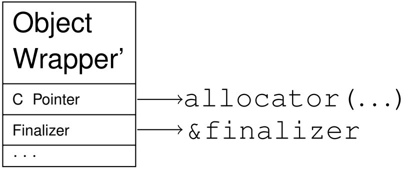
1. Take ownership of C objects returned by allocators
2. Automatically invoke finalizers when collecting C objects
3. Relinquish ownership of explicitly finalized C objects



Host-Language Memory Management

We want the host-language runtime to do three things:

1. Take ownership of C objects returned by allocators
2. Automatically invoke finalizers when collecting C objects
3. Relinquish ownership of explicitly finalized C objects



Evaluation

We generated Python bindings for four C libraries:

- The GNU Linear Programming Kit (GLPK)
- The GNU Scientific Library (GSL)
- libarchive
- libical

Evaluation

- Analyzed over 2500 functions
- Provided two manual annotations
- Infer annotations on about a third
  - 365 allocators
  - 421 output parameters
  - Over 1500 array parameters
- Running times: 15 minutes for GLPK, less than 5 minutes for the others

Manual Annotations

We provided two manual annotations.

```
void *xmalloc(int size) {
  LIBENV *env = lib_link_env();
  LIBMEM *desc;
  int sz = align(sizeof(LIBMEM));
  desc = malloc(size);

  desc->next = env->mem_ptr;
  env->mem_ptr = desc;

  return (void *)((char *)desc + sz);
}
```

These two manual annotations allowed us automatically infer 70 additional annotations.

Evaluating “Naturalness”

We compared our bindings against hand-written bindings for GLPK and (parts of) GSL:

- Our multiple return value transformation closely matches manual transformations in the GSL binding
- Hand-written bindings have more specific error handling code

We also identified several type errors in the GLPK bindings

Conclusion

We have:

- Identified high-level C idioms

Conclusion

We have:

- Identified high-level C idioms
- Demonstrated efficient recovery methods

Conclusion

We have:

- Identified high-level C idioms
- Demonstrated efficient recovery methods
- Created rich bindings

Conclusion

We have:

- Identified high-level C idioms
- Demonstrated efficient recovery methods
- Created rich bindings

All with minimal programmer effort.

Questions

Thanks