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

Diagram:

- Host Language
- Application
- FFI
- C Library
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
- 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

```c
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**

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

**After**

```c
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
A function call in C

```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)
```
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

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

Adapted from libarchive
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;
}
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
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;
}
Consider the standard C function

```c
char *strdup(const char *s)
```
Consider the standard C function

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

Compare with another standard C function

```c
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

Annotations \rightarrow C

\rightarrow Header Analysis

IR \rightarrow Code Generator

\downarrow (Python)
Approach

Current Approaches

Annotations → IR → C → Header Analysis → IR → Code Generator (Python)

Our Approach

Annotations → C → Source Analysis → IR (Inferred Annotations) → Code Generator (Python)
Static analysis of library source:
- Output parameters
- Array parameters
- Resource management functions

Our Approach

Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit
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)

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

- For each pointer parameter \( p \), the initial state is \( \perp \).
- The \( \text{join} \) operation for any statement \( s \) is

\[
\bigcup_{p \in \text{pred}(s)} \perp = \text{UNUSED}
\]
The transfer function for each statement $s$ using parameter $p$ depends on the syntactic form of $s$:

- $p = e$
- $p$
- $f(p)$
- Otherwise, $\text{exit}_s(p) = \text{entry}_s(p)$. 

$\top = \text{INOUT}$

$\perp = \text{UNUSED}$
We have recovered some programmer intent:

- Multiple Return Values
- Python uses tuples
- Example:

```c
int f_c(int x, int* y, int* z);
```
Expressing Output Parameters

\[
\text{int } f_c(\text{int } x, \text{int* } y, \text{int* } z);
\]

```python
def fpy(x):
```

Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit
Automatic Generation of Library Bindings Using Static Analysis
Expressing Output Parameters

\[
\text{int } f_c(\text{int } x, \text{int* } y, \text{int* } z);
\]

```python
def fpy(x):
    tmp_y = c_int()
    tmp_z = c_int()
```
Expressing Output Parameters

```python
def fpy(x):
    tmp_y = c_int()
    tmp_z = c_int()
    tmp_ret = fc(x, byref(tmp_y), byref(tmp_z))
```

```c
int fc(int x, int* y, int* z);
```
Expressing Output Parameters

int \( f_c(\text{int } x, \text{int* } y, \text{int* } z) \);

```python
def \( f_{py}(x) \):
    tmp_y = c_int()
    tmp_z = c_int()
    tmp_ret = \( f_c(x, \text{byref}(\text{tmp}_y), \text{byref}(\text{tmp}_z)) \)
    return (tmp_ret, tmp_y, tmp_z)
```
Compare calls to the \texttt{frexp} function in C and Python/ctypes

\begin{itemize}
  \item \texttt{int exp; double frac = frexp(x, &exp);}
  \item \texttt{exp = c_{int}() \quad frac = frexp(x, byref(exp))}
\end{itemize}
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

```
(frac, exp) = frexp(x)
```
Parameters used in **array contexts** can be treated as arrays. To find them:

Let

\[
\text{arrays} = \{ v | v = * (ptr + offset) \}
\]

and:

\[
\begin{align*}
    v_1 &= *(ptr + o_1) \rightarrow \ldots | v_2 | \ldots \\
    v_2 &= *(v_1 + o_2) \rightarrow N \\
    v_3 &= *(y + o_3) \rightarrow \ldots | z | \ldots
\end{align*}
\]
Parameters used in array contexts can be treated as arrays. To find them:

Let \( \text{arrays} = \{ v \mid v = \ast (ptr + offset) \} \) and:

1. Consider pairs
   \[ v_1 = \ast (ptr + o_1) \text{ and } v_2 = \ast (v_1 + o_2) \]

\[ v_1 = \ast (ptr + o_1) \rightarrow \ldots | v_2 | \ldots \]

\[ v_2 = \ast (v_1 + o_2) \rightarrow N \]

\[ v_3 = \ast (y + o_3) \rightarrow \ldots | z | \ldots \]
Parameters used in array contexts can be treated as arrays. To find them:

Let

\[ \text{arrays} = \{ v \mid v = \ast (ptr + \text{offset}) \} \]

and:

1. Consider pairs
   \[ v_1 = \ast (ptr + o_1) \text{ and } v_2 = \ast (v_1 + o_2) \]

2. Extend the base array of \( v_1 \) and create arrays’

\[ v_3 = \ast (y + o_3) \rightarrow \ldots |z| \ldots \]
Parameters used in array contexts can be treated as arrays. To find them:

Let

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

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:

\[
\text{arrays} = \{ v | v = \ast (ptr + offset) \}
\]

and:

1. Consider pairs
   \[
   v_1 = \ast (ptr + o_1) \quad \text{and} \quad v_2 = \ast (v_1 + o_2)
   \]
2. Extend the base array of \( v_1 \) and create arrays’
3. Find a fixed-point.
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`
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
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

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

```c
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

```c
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;
```
• A function $f$ finalizes pointer parameter $p$ if it passes $p$ to a finalizer on every path.
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 \texttt{finalized-or-NULL} for each $p$.

```c
if (!object) return;
free(object->field);
free(object);
```
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
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
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
We provided two manual annotations.

```c
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