Why Extension Programmers Should Stop Worrying About Parsing and Start Thinking About Type Systems

(or automatic extension building is harder than it looks, but that hasn't stopped anyone from trying to do it)

David M. Beazley
Department of Computer Science
The University of Chicago
beazley@cs.uchicago.edu

November 9, 2002

Background

I work on programming tools and applications

- SWIG project (www.swig.org), 1995-present.
- Scientific software (molecular dynamics, climate modeling)
- And some other projects.

I am not a programming languages researcher

- Which is why I am here

What Programmers Want

Access to the system

- Files and I/O
- Processes
- Threads
- Networking

Use of existing libraries

- Graphics
- Databases
- GUI

Mixing of languages

- C or C++ controlled by a high level language

Extension Programming

Most HLL languages support "extensions"

- Foreign function interface (FFI).
- Provides access to compiled code (C, assembler, etc.)
- However, you usually have to write wrappers

Example: Python

```
extern int gcd(int x, int y);
...
/* Python wrapper for gcd() */
PyObject *wrap_gcd(PyObject *self, PyObject *args) {
    int x, y, r;
    if (!PyArg_ParseTuple(args,"ii",&x,&y)) {
          return NULL;
    }
    r = gcd(x,y);
    return Py_BuildValue("i",r);
}
```

Extension Tools

Nobody likes writing wrappers

This has spawned an "extension tool" industry...

- SWIG
- Boost.Python
- CABLE
- h2xs/xsubpp
- Inline
- Vtk
- pyfort
- f2py
- gwrap
- Weave
- Pyrex
- SIP
- Matwrap
- tolua
- TclObj
- ModularTcl
- Itcl++
- GRAD
- Babel
- iVRS
- Wrappy
- ... and many others.

More extension tools than languages

- Or so it seems...

"Mondo" Extension Building

Mixed-language programming

- Application programmers are combining C/C++/Fortran with a HLL
- Examples: C++ driven by Python, Scheme, Perl, etc.

The utility of HLLs

- Control and configuration.
- User interfaces.
- Rapid prototyping.
- Debugging.
- Testing.
- Systems integration.

Examples of "mondo"

- 427 classes, 4351 methods, 220 templates (SWIG mailing list, Oct. 2002)
- 280 classes, 2500 methods, 45 templates (iVRS, USENIX Freenix'02)
- 200 classes (Vtk, USENIX 4th Tcl/Tk Workshop, 1996)
- 300 classes, 5750 methods/functions (SIP, PyQT extension).

Automation

The only practical way to wrap large applications

- Hand-written wrappers impossible to maintain.
- Limited programmer resources.

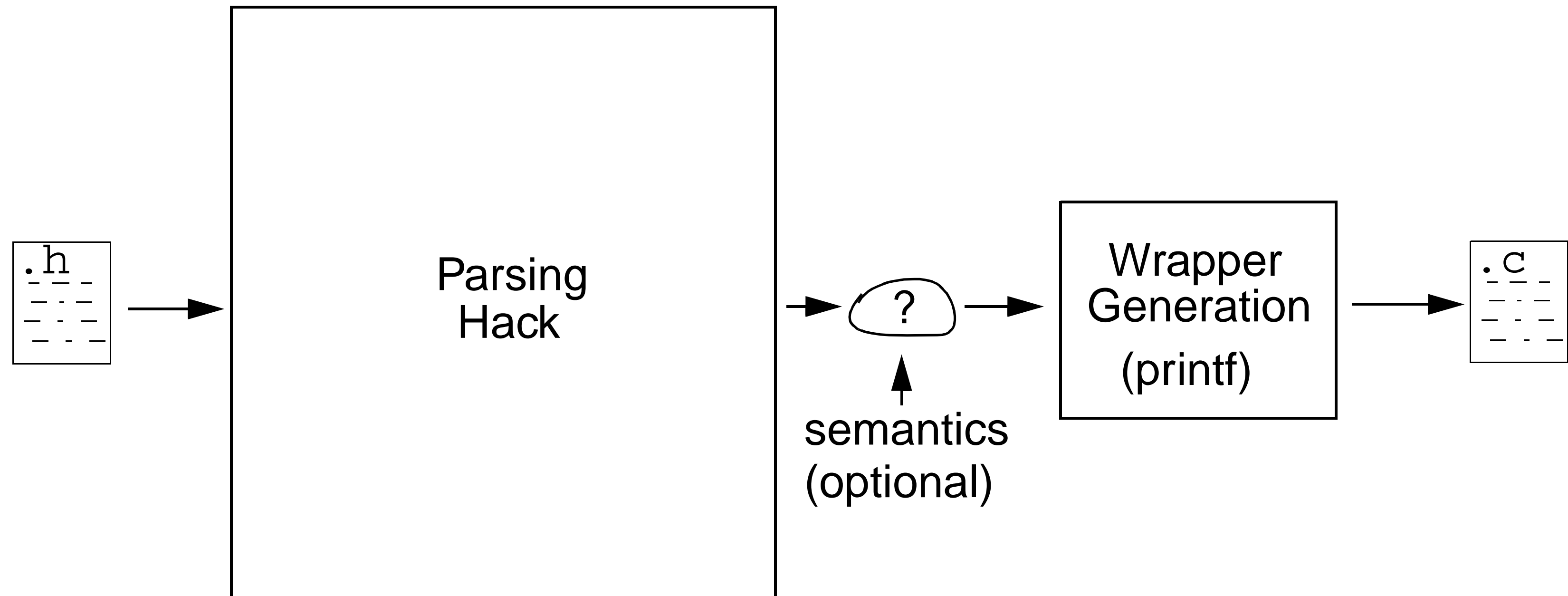
A common approach: Header file parsing

- Header files define interfaces in C/C++ (separate compilation)
- A natural starting point in the eyes of many application programmers.
- Variations of header parsing used in many different tools.

Extension tools are *trying* to do more than you think

- Automatic wrapping of C++ class hierarchies.
- Overloaded methods, functions, and operators.
- Templates, STL integration.
- Smart pointers, memory management.
- Exception handling
- Mixed-language polymorphism (terminology?)

A Prototypical Extension Tool



Tools mostly target the common case

- Can generate useful wrappers with only a superficial approach
- Simple string matching, regex matching, reduced grammars
- Some heuristics
- And some customization features (a.k.a, "workarounds")
- Ignore the hard corner cases.

Parsing header files doesn't seem *that* hard

Death by Corner Cases

Do any of these tools *actually* work?

- Well, it depends on your definition of "works."
- Tools kind of work "well enough," but they could be a lot better.
- The problem is very difficult to generalize.
- Wrapper generation is different than compiling.

PL: Parsing header files will *never* work

- Too hard (or bogus). Don’t do it.
- Have to reimplement the C++ compiler (or instrument an existing one)
- Not enough information in header files.

Users: Parsing header files *might* work

- It seems to work pretty well in a lot of specific applications.
- Programmers really like it (well, when it works).
- Have *viable* alternatives been offered?

Parsing is the *Wrong* Focus

Header file parsing is only a convenient user-interface

- Yes, users like it.

Extension building is not a parsing problem

- Marshalling of types (representation).
- Creating wrappers around objects (C++ types).
- In fact, you are gluing two type systems together (in a practical sense)
- Wrappers and parsing are merely implementation details.

Parsing alone is not enough to understand C++

- Classes introduce "types"
- The members of a class determine its behavior.
- Templates/namespaces/typedef introduce very complicated relationships.
- (Then again, maybe C++ can’t be understood by anything)

The Classic Parsing Problem

Not enough information in headers:

```
void blah(int *x, int n);
```

- An input value?
- An array?
- An output value?
- An input/output value?
- Is it somehow related to n?

Common workarounds (modifiers, naming conventions)

```
void blah(%out int *x, int n);
void blah(int *out_x, int n);
```

This has never been a major obstacle

Breaking The Common Case

What is this?

```
extern binop_t  add, sub, mul, div;
```

A declaration of 4 global variables (obviously).

Breaking The Common Case

What is this?

```
typedef int binop_t(int, int);
extern binop_t  add, sub, mul, div;
```

~~A declaration of 4 global variables (obviously).~~

Well, actually, it’s a declaration of 4 functions.

- Okay, it’s perverse.

Question: what will a tool do?

- A lot of tools will break.

==> You can’t just look for simple text patterns.

Breaking Typenames

Consider:

```
namespace A {
    typedef int Integer;
}
class B {
public:
    typedef A::Integer  type;
};
template<class T> struct Foo {
    typedef typename T::type Footype;
};
using A::Integer;

extern Foo<B>::Footype spam(Integer x, B::type y, int z);
```

Question: Can a tool figure out the types?

- Not if it looks for simple names like "int"

==> Types aren't strings and more than a table lookup.

Breaking Classes

Consider a class:

```
class Foo : public Bar {
public:
      int blah(int x);
};
```

Question: Can you create a Foo?

- Can you create a wrapper that constructs a Foo?
- Well, that depends. What is Bar?

Breaking Classes

Consider a class:

```
class Foo : public Bar {
public:
      int blah(int x);
};
```

Question: Can you create a Foo?

- Can you create a wrapper that constructs a Foo?
- Well, that depends. What is Bar?

Examples:

```
class Bar {
private:
    Bar();
};
```

```
class Bar {
public:
   virtual void spam() = 0;
};
```

==> Can’t look at classes in isolation

Overloading

What happens here?

```
void foo(char *s);      // A
void foo(double x);     // B
void foo(int x);        // C
```

Consider a language like Tcl

- Typeless. All objects are strings.

Yet, some tools actually seem to work

```
% foo 3          # Invokes C
% foo 3.0        # Invokes B
% foo hello      # Invokes A
```

A type hierarchy imposed on a typeless target (somehow)

==> Definitely more than parsing

Breaking Your Head

Advanced C++ idioms. Example: smart-pointers

```
struct Foo {
    int x;
    int blah(int x);
};
template<class T> class SmartPtr {
    ...
    T *operator->();
    ...
};
extern SmartPtr<Foo> create_foo();
```

Now, in Python:

```
>>> f = create_foo()
>>> f.x                    # Implicit dereference through
3                          # operator->()
>>> f.blah()
```

==> More than parsing (and possibly a bad idea).

Stepping Back

Wrapping C++ is nontrivial

- You knew this.
- Many corner cases.
- Dark and obscure parts of the standard.
- Even more complicated than C wrapping (which is already hard).

Parsing it is not enough

- This, I am absolutely sure about.

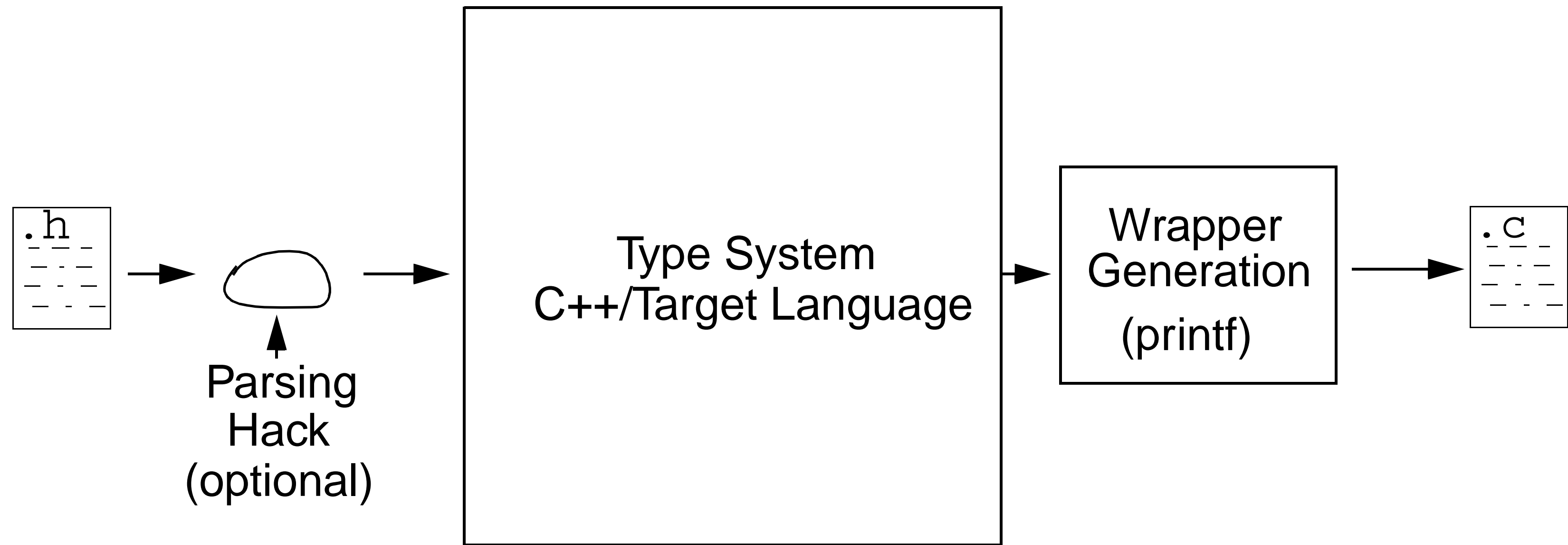
Non-trivial interaction of several components

- C++ types, target language, semantics, user-driven customization features.

My view:

- Tool builders have been working on the problem at the *wrong* level
- Reflects my experience with the SWIG project.

An Alternative View



A Better Approach

- Build extension tools around "type" systems, not parsers.

It makes sense

- Wrapper generation is a fundamentally a problem in type systems.
- C++ is only understood through its type system.
- It makes sense *even if you don't parse C++*.

A Tale of Two Tools

Two type-based extension building tools

- Boost.Python (http://www.boost.org/libs/python/doc)
- SWIG (http://www.swig.org)

Both of these tools

- Support a large subset of C++
- Support a variety of advanced features (overloading, templates, etc.)
- Both tools are primarily built around the *C++ type system*.
- Both unconventional in approach.

Boost.Python

Wrapper generation using the C++ compiler itself

```
struct Foo {
    double x;
    Foo(double);
    int blah(int x, int y);
};

BOOST_PYTHON_MODULE_INIT(foomodule) {
    class_<Foo>("Foo")
        .def(init<double>())
        .def("blah", &Foo::blah)
        .def_readwrite("x",&Foo::x);
}
```

What is this doing?

- Using C++ template machinery to automatically generate wrappers to C++.
- Template meta-programming.
- There is no "tool" here. The C++ compiler is the tool.

C++ type system used to wrap C++ types.

Boost.Python

This is very cool

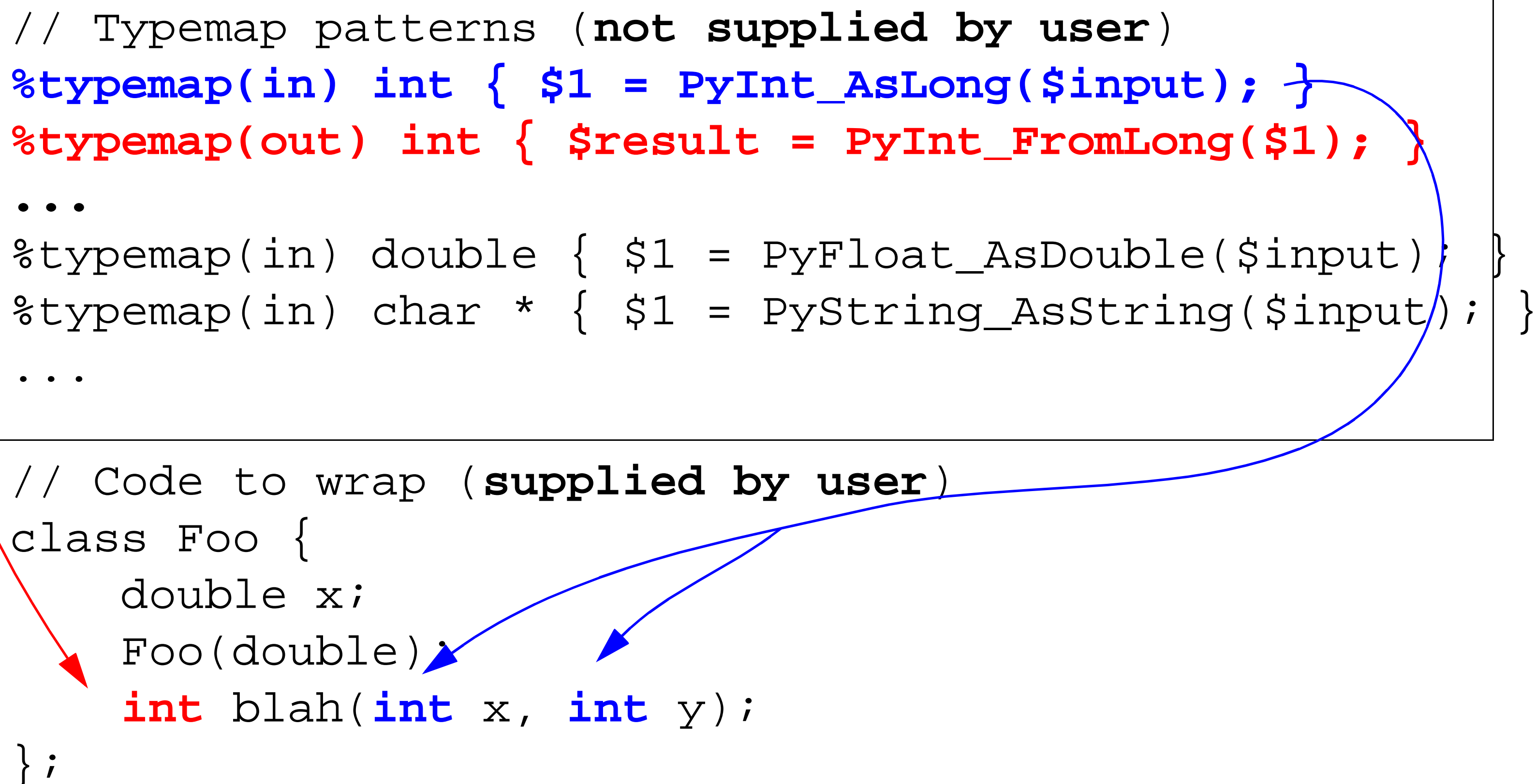
- It is able to handle *very* complicated C++
- Especially templates, namespaces, other advanced features.
- Didn't require an implementation of a C++ parser or a C++ type system.

Why it's interesting

- Very unconventional (the only tool I know that works like this)
- Operates completely within the C++ type system domain.
- A good example of the *right problem focus*.

SWIG

Wrappers generated through pattern matching rules



Patterns built into the C++ type system.

- The patterns provide code fragments and other details.

SWIG - Pattern Matching

```
%typemap(in)  int { $1 = PyInt_AsLong($input); }
%typemap(out) int { $result = PyInt_FromLong($1); }

namespace A {
    typedef int Integer;
}
class B {
public:
    typedef A::Integer  type;
};
template<class T> struct Foo {
    typedef typename T::type Footype;
};
using A::Integer;

extern Foo<B>::Footype spam(Integer x, A::Integer y, int z);
```

Pattern matching is more than string matching

SWIG - Pattern Matching

Patterns may incorporate more than types

- Argument names, typedef names, argument sequences

```
%typemap(in) int positive { ... }
%typemap(in) dnonzero_t { ... }
%typemap(in) (char *str, int len) {
     $1 = PyString_AsString($input);
     $2 = PyString_Size($input);
}
...
typedef double dnonzero_t;
extern int factorial(int positive);
extern double inverse(dnonzero_t);
extern int find(char *str, int len, char *pat);
```

There's more to this, but that's a different talk

- Patterns used for specifying behavior (in, out, arrays, checking, etc.)
- Typemaps can be incorporated into classes, templates, etc.
- Declaration annotation.

Big Picture

By shifting the focus to "types"...

- It solves really hard problems (templates, namespaces, overloading, etc...)
- It eliminates vast numbers of corner cases.
- Greater reliability and simplified maintenance
- It allows tool builders to focus on more important problems
- Users *really* like it, but they don't quite know why (it just "works").

But this work is incomplete and ongoing....

- No tool is perfect or free of bugs.
- There is definitely no "silver bullet."
- Still fighting a very complicated, possibly intractable, but practical problem.

Where do we go from here?

There are hard problems that need to be resolved

- Extension building tools are only starting to play with type systems.
- I don't think that anyone has quite figured it out.

Connection to programming languages

- This is clearly a programming language issue (at least I think so)
- However, it's hard for us to make a precise connection to prior work.
- If not type systems, what is the right connection?
- Also strong ties to software engineering.

Obstacles

- Cultural divide. Extension tools primarily built by applications programmers.
- Language designers have better things to worry about.
- A lot of work in FFIs primarily focused on C, representation issues.
- "Wasn't this problem dismissed/solved in 1971?"

In Closing

To extension tool builders...

- Parsing is not the most important problem.
- Think about the C/C++ type system--that's where the real action is.
- You *will* like it.

To the programming languages community...

- A lot of people are building extension tools. Pay attention.
- This is an important practical problem to application programmers.
- They don't want to reimplement existing code and libraries.
- There is an opportunity to make connections (and have impact).
- Metaobject protocol? AOP? FFI's? Type systems?

Acknowledgements

- David Abrahams
- William Fulton, Matthias Koeppe, Jason Stewart, Lyle Johnson, Richard Palmer, Luigi Ballabio, Sam Liddicott, Art Yerkes, Thien-Thi Nguyen, Marcelo Matus, Loic Dachary, Masaki Fukushima, and others.