Introduction via Self Debugging

Russell Harmon
reh5586@cs.rit.edu
Rochester Institute of Technology
Computer Science

1 Introduction

The omnipresent support for introspection in modern programming languages indicates the usefulness of the tool. [2, 3, 8, 9] Unfortunately, C, which is one of the most pervasive programming languages and the foundation of nearly every modern operating system, does not support introspection.

I propose to bring introspection to the C language via a novel application of an old tool: the debugger. Debuggers have long had access to the type and naming information which is needed for introspection. On most UNIX platforms, this is accomplished by the debugger reading any DWARF [4] symbols which may be present in the target binary. These symbols can be leveraged to gain the information which is needed to create a full-featured introspection API.

2 Rationale

One of the motivating factors for any language introducing introspection as a feature is the following use case:

> You are tasked with changing the save game format of a popular 1980s style terminal based game from a binary format composed of writing the structs which compose the game state to disk to a more flexible JSON format. After investigation, you discover that in order to do this, you can use the Jansson [7] C library to produce JSON. In order to do so, you invoke variants of the `json_object_set` function as given by the following prototype:
>
> ```
> int json_object_set(
>   json_t *object,
>   const char *key,
>   json_t *value
> );
> ```
>
> You notice that `json_object_set` takes as parameters the name and value of the field to be written necessitating the writing of a separate `json_object_set` call for every field of every struct which is to be written. After considering the literally thousands of fields across the nearly three hundred structs in the game you give up in frustration.

Clearly, it is a significant convenience to developers to be able to write code which is able to introspect upon data in a meta-programming style.

3 Introspection in Current Programming Languages

Introspection is found in many of the programming languages commonly used today including Java [8], Ruby [2], Python [9], Perl [3] and a limited form of introspection in C++ [13]. The various approaches to introspection differ in implementation detail; some receiving introspection as a direct consequence of the way they implement objects while some provide it as part of the standard library. Despite this, they all provide approximately the same set of features. It is by these features that introspection can be defined, rather than the details of how the features are implemented.

Two features are generally required in order to be considered introspection: *type* and *function enumeration*. In Java, the availability of *type enumeration* means that a program can retrieve the name of an object and enumerate the fields of that object, retrieving not only their values, but also strings representing their names and types, including the object's methods. *Function enumeration* can then be performed on the methods of an object giving access to information about the return type and argument types, including the names of the arguments.

Existing attempts to add introspection to C++ frequently require a separate description of the object to be implemented which is generated using either a separate parser [11] or using a complementary *metadata object* [1] and have the limitation that objects which come from external libraries cannot be introspected. Our implementation of introspection will neither have this library boundary limitation nor require external tools or additional objects in order to operate.

4 Debugging in C

There already exist a number of tools for interactive debugging of C programs. Some of the more well

known ones include GDB [5], WinDBG, Visual Studio's debugger and LLDB. [12] Traditionally, these debuggers have been used interactively via the command line where more recently debuggers such as the one embedded within Visual Studio integrate into an IDE.

An understanding of debugging in general, and about LLDB specifically are crucial to the understanding of this proposal, so some time will be spent explaining debugging.

The first step in debugging a program is the creation of a debuggable binary. This is accomplished by instructing the compiler to insert debugging symbols, commonly DWARF on UNIX platforms, into the binary. Interactive debugging is possible without these symbols, but becomes difficult.

The DWARF symbols provide LLDB with information about the variables and functions in the program and enables LLDB to determine the line number which produced a particular sequence of instructions.

4.1 LLDB

Figure 1 on the following page shows a simple debugging session using LLDB. In it, a test program is launched and the value of a stack-local variable is printed. Take note that LLDB is aware that the type of `foo.bar` is `char *`. In fact, most debuggers make available to their users nearly all of the type information which is available to the programmer writing the original source file.

While type information is useful while interactively debugging a program, the fact that LLDB is able to determine this information while the program is running suggests that if a program can itself control LLDB, it could potentially retrieve this type information about itself.

An important aspect of the type information which is available to LLDB is that this information is purely static. The debugger knows only the type of the variable being displayed, rather than the type of the data itself. This is in stark contrast with other introspective languages where the type information is carried with the data and can be recovered without any additional context. An example of the result of this is shown in Fig. 2 on the next page.

5 Related Work

A System for Runtime Type Introspection in C++ [1] discusses an approach to introspection for C++ whereby metadata objects are created using macros which are expected to be called at the definition of the object which is to be introspected.

The Seal C++ Reflection System [11] discusses an introspection system for C++ which uses a metadata generation tool to create descriptor files which contain the information needed for introspection.

Reflection for C++ [6] uses an approach very similar to the one proposed here, but instead of using a debugger to retrieve debugging information, it instead reads the debugging symbols directly. This limits the API to only leveraging information that it can retrieve from the debugging symbols themselves, as opposed to could be retrieved via interactive debugging. This possibility is discussed in Section 9.

C++, along with all the other languages supported by Microsoft's CLR can be reflected upon by leveraging features exposed by the CLR. [10]

6 Approach & Design

By leveraging LLDB, I hope to create an API which allows a program to read the DWARF symbols of any object file, shared library or executable; including itself. The API which programmers will interact with should be complete enough that the presence of LLVM should be completely hidden from the programmer. This API should minimally allow introspection of scalar and pointer values, typedef'ed values, structs, functions and struct members; allowing access to the variable, struct member or function name, type name, size, struct member offset, function arguments and return type.

LLDB's API is not designed to require attaching to a running process in order to retrieve static debugging information which can be determined simply by examining the symbols in the binary. If information which is not available statically is needed, a running instance of the program can be started or attached to. For the initial implementation of the introspective API, attaching to the running program should not be necessary as all the introspective information needed should be retrievable directly from the executable file's debugging symbols. See Section 9 for future improvements that debugging the running program could provide.

The code shown in Fig. 4a is a simple example of a feature which will be enabled by this API. In Fig. 4a, a struct named `foo` is introspected upon in order to produce the output shown in Fig. 4b. Notice that although the API will not directly provide a method to access the introspected data's value, it is accessible indirectly via the *offset* and *size* fields which are available.

Figure 3 shows a tentative partial API complete enough for the example mentioned above and shown in

```
Current executable set to './a.out' (x86_64).
(lldb) breakpoint set -n main
Breakpoint created: 1: name = 'main', locations = 1
(lldb) run
Process 10103 launched: './a.out' (x86_64)
Process 10103 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f60 a.out`main + 16 at a.c:6
    frame #0: 0x0000000100000f60 a.out`main + 16 at a.c:6
   3    };
   4    int main() {
   5            struct foo foo;
-> 6            foo.bar = "Hello World!";
   7    }
(lldb) next
Process 10103 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f64 a.out`main + 20 at a.c:7
    frame #0: 0x0000000100000f64 a.out`main + 20 at a.c:7
   4    int main() {
   5            struct foo foo;
   6            foo.bar = "Hello World!";
-> 7    }
(lldb) print foo.bar
(char *) $0 = 0x0000000100000f66 "Hello World!"
```

Figure 1: Interactive Debugging with LLDB

```
Process 12066 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f64 a.out`main + 20 at a.c:3
    frame #0: 0x0000000100000f64 a.out`main + 20 at a.c:3
   1    int main() {
   2            void *baz = "Hello World!";
-> 3    }
(lldb) print baz
(void *) $0 = 0x0000000100000f66
```

Figure 2: Static Type Information in Debuggers

```
typedef struct {
  enum {
    TYPE_INT,
    TYPE_TYPEDEF,
    TYPE_POINTER,
    TYPE_STRUCT
  } type;
  const char *name;
  size_t size;
} type_t;

typedef struct {
  type_t type;
  type_t *real;
} typedef_t;

typedef struct {
  type_t type;
  type_t *referent;
} pointer_t;

typedef struct {
  const char *name;
  size_t offset;
  const type_t *referent;
} field_t;

typedef struct {
  type_t type;
  const char *name;
  size_t nfields;
  const field_t fields[];
} struct_t;

const type_t *get_type( const char *tynam );
void release_type( const type_t *typ );
```

Figure 3: Tentative Introspective API

Fig. 4 on this page. A program would call `get_type` in order to retrieve a `type_t` whose storage is managed by `get_type`. This `type_t` can then be used to determine information such as the name or size of the type which it represents by casting the `type_t` to the appropriate concrete type as indicated by the `type_t.type` field.

6.1 The LLDB API

Currently, LLDB is a relatively new project, and is only sparsely documented. It does however have example code of the use of it's API. From reading the sources to LLDB, there exists an `SBType` class which represents a type. These types include struct, function and primitive types. Using a `SBType`, all the information needed for a fully featured introspective API can be retrieved.

```
typedef char *string_t;
struct foo { string_t str; int i; };
void print_data( const type_t *type, const void *data )
{
  switch( type->type ) {
    case TYPE_TYPEDEF:
      assert(strcmp(type->name, "string_t") == 0);
      printf("(%s) \"%s\"\n", type->name,
             *((const string_t *) data));
      break;
    case TYPE_INT:
      printf("(%s) %d\n", t->name,
             *((const int *) data));
      break;
    case TYPE_STRUCT:
      const struct_t *s = (const struct_t *) type;
      printf("(%s) {\n", s->name);
      for( size_t i = 0; i < s->nfields; i++ ) {
        const field_t *f = &s->fields[i];
        printf("\t.%s = ", f->name);
        print_data(f->referent, data + f->offset);
      }
      printf("}\n");
      break;
    default:
      assert(false);
  }
}
int main()
{
  struct foo bar = {
    .str = "Hello World!",
    .i = 6666
  };
  const type_t *t = get_type("struct foo");
  print_data(t, &bar);
  release_type(t);
}
```

(a) Introspective code

```
(foo) {
  .str = (string_t) "Hello World!"
  .i = (int) 6666
}
```

(b) Output from introspective code

Figure 4: Introspection Using this API

7 Limitations

This style of introspection is slightly more limited than the classic style of introspection whereby an object carries it's own type information. Instead, the type of a *value* must be specified either by the type of it's *variable*, or explicitly by passing a string representation of it's type to the introspective API. The only limitation this *static introspection* should impose on the programmer is that an unknown **void** $*$ type cannot be introspected.

Under current plans, the programmer will have to provide the desired type as a string to the `get_type` function. It is however desirable to be able to use any expression as the argument to `get_type`. In order to accomplish this, a running debugger will need to be attached to the program in order to determine the result type of the expression. Future work may provide this feature.

Although not strictly a limitation, many programmers will likely want to introspect strings as such, rather than as the **char** $*$ type. Unfortunately, since there is no typing difference between a C string and a pointer to one or more chars, this introspective API will be unable to determine the difference between the two. The example code shown in Fig. 4a on the previous page works around this issue by creating a typedef of **char** $*$ to string_t.

8 Timeline & Completion

Beyond design and implementation of the introspective API, completion of this project would be measured by the following criteria: *a*) code which is similar in structure to Fig. 4a on the preceding page is able to produce the output shown in Fig. 4b on the previous page *b*) a proof of concept `abort_with_stacktrace()` function which when invoked prints a complete stack trace to standard error and exits and *c*) the release of the source code in a usable, fully documented form on the internet.

The work to create an introspective API is expected to take approximately three to four months. This is expected to occur between the months of April and August.

As shown in Fig. 5 on this page, I expect the actual programming portion of the project to take until July and the writing of the thesis to take until August. I expect to be able to defend by the beginning of the first academic semester of 2013.

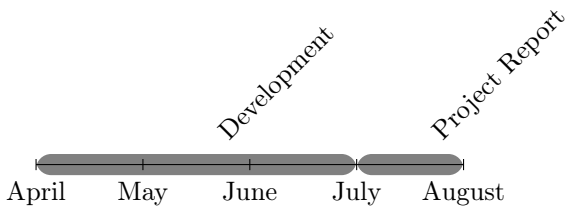

Figure 5: Completion Timeline

9 Future Work

In order to enable the passing of an expression to `get_type`, it should be possible to spawn a *debugger control thread*, which receives requests from other threads. These requests would be serviced by attaching to the requesting thread, manipulating the debugger to retrieve the type of the passed in variable, and finally passing that information back to the requesting thread. This design is chosen due to the anticipated limitation in LLVM that you will be unable to attach to your own thread.

It should be possible to extend this API to operate with other debuggers (e.x. `gdb`) possibly even on other platforms (e.x. `windbg` on Windows). Currently, efforts will be focused on features of the introspective API before portability is considered.

References

[1] Maximilien de Bayser and Renato Cerqueira. "A System for Runtime Type Introspection in C++". In: *Proceedings of the 16th Brazilian conference on Programming Languages*. SBLP'12. Natal, Brazil: Springer-Verlag, 2012, pp. 102–116. ISBN: 978-3-642-33181-7. URL: http://dx.doi.org/10.1007/978-3-642-33182-4_9.

[2] *Class: Object (Ruby 1.9.3)*. URL: http://ruby-doc.org/core-1.9.3/Object.html#method-i-instance_variables (visited on 2/19/2013).

[3] *Class::MOP::Class*. URL: http://search.cpan.org/dist/Class-MOP/lib/Class/MOP/Class.pm (visited on 2/19/2013).

[4] DWARF Standards Committee. *The DWARF Debugging Standard*. URL: http://dwarfstd.org/ (visited on 10/20/2012).

[5] *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb/ (visited on 2/19/2013).

[6] K. Knizhnik. *Reflection for C++*. URL: http://www.garret.ru/cppreflection/docs/reflect.html (visited on 2/19/2013).

[7] Petri Lehtinen. *Jansson*. URL: http://www.digip.org/jansson/ (visited on 2/7/2013).

[8] *Package java.lang.reflect*. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html (visited on 2/19/2013).

[9] *Python 2.7.3 » Documentation » The Python Standard Library*. Built-in Functions. URL: http://docs.python.org/2/library/functions.html#dir (visited on 2/19/2013).

[10] *Reflection in C++*. URL: http://msdn.microsoft.com/en-us/library/y0114hz2(v=vs.80).aspx (visited on 2/19/2013).

[11] S. Roiser and P. Mato. "The Seal C++ Reflection System". In: *Proceedings of CHEP 2004*. CHEP04 (Sept. 27–Oct. 1, 2004). CERN. Interlaken, Switzerland, 2004. URL: http://indico.cern.ch/getFile.py/access?contribId=222&resId=0&materialId=paper&confId=0 (visited on 2/19/2013).

[12] LLVM Team. *The LLDB Debugger*. URL: http://lldb.llvm.org/ (visited on 10/11/2012).

[13] "Working Draft, Standard for Programming Language C++". In: *ISO/IEC 14882:2011* (2011), 99, §5.2.8.