EXPERIMENTS ON BUILDING HIGH LEVEL ABSTRACTIONS WITH C

PART 3: REFLECTION

Mirror mirror on the wall, am I not the prettiest of them all?

Franklin "Snaipe" Mathieu – 2017

4/12/2015

EPITA – GCONFS

WHAT IS REFLECTION?

DEFINITIONS

Reflection = _____ + _____

DEFINITIONS

Reflection = Introspection + _____

Introspection: The ability of a program to examine itself

DEFINITIONS

$$\text{Reflection} = \text{Introspection} + \text{Intercession}$$

Introspection: The ability of a program to examine itself

Intercession: The ability of a program to change its state and meaning

DEFINITIONS

$$\text{Reflection} = \text{Introspection} + \text{Intercession}$$

Introspection: The ability of a program to examine itself

Intercession: The ability of a program to change its state and meaning

Known implementors are Java, C#, Python, Ruby, PHP, ...

REFLECTION

To achieve reflection, we need sufficient information on various actors:

- Types
- Functions
- Variables

REFLECTION

To achieve reflection, we need sufficient information on various actors:

- Types
- Functions
- Variables

Thankfully, debuggers also need these informations to be useful.

IMPLEMENTING INSIGHT

DWARF

DWARF is a standardized debugging format, designed along with ELF.

Information is stored (and described) as a tree, but contains backreferences on nodes, making it effectively more like a graph.

DWARF

DWARF is a standardized debugging format, designed along with ELF.

Information is stored (and described) as a tree, but contains backreferences on nodes, making it effectively more like a graph.

libdwarf can be used to easily parse and iterate over the tree.

ANATOMY OF A SIMPLE C PROGRAM

test.c:

```
struct T { int field; double other_field; };
static struct T var;

int main(void) {}
```

ANATOMY OF A SIMPLE C PROGRAM

test.c:

```c
struct T { int field; double other_field; };
static struct T var;

int main(void) {}
```

DWARF information:

```
DW_TAG_compile_unit
  DW_AT_producer  GNU C11 5.2.0 -mtune=generic
                    -march=x86-64 -g
  DW_AT_language  DW_LANG_C99
  DW_AT_name      test.c
  DW_AT_comp_dir  /home/snaipe
  DW_AT_low_pc    0x004004b6
  DW_AT_high_pc   <offset-from-lowpc>11
  DW_AT_stmt_list 0x00000000
```

```
<0x2d> DW_TAG_structure_type
         DW_AT_name: ”T”
         DW_AT_byte_size: 0x10
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x1
<0x37>   DW_TAG_member
           DW_AT_name: ”field”
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x50>
           DW_AT_data_member_location: 0
<0x43>   DW_TAG_member
           DW_AT_name: ”other_field”
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x57>
           DW_AT_data_member_location: 8
```

```
<0x2d> DW_TAG_structure_type
         DW_AT_name: "T"
         DW_AT_byte_size: 0x10
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x1
<0x37>   DW_TAG_member
           DW_AT_name: "field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x50>
           DW_AT_data_member_location: 0
<0x43>   DW_TAG_member
           DW_AT_name: "other_field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x57>
           DW_AT_data_member_location: 8
```

```
<0x2d> DW_TAG_structure_type
         DW_AT_name: "T"
         DW_AT_byte_size: 0x10
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x1
<0x37>   DW_TAG_member
           DW_AT_name: "field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x50>
           DW_AT_data_member_location: 0
<0x43>   DW_TAG_member
           DW_AT_name: "other_field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x57>
           DW_AT_data_member_location: 8
```

```
<0x2d> DW_TAG_structure_type
         DW_AT_name: "T"
         DW_AT_byte_size: 0x10
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x1
<0x37>   DW_TAG_member
           DW_AT_name: "field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x50>
           DW_AT_data_member_location: 0
<0x43>   DW_TAG_member
           DW_AT_name: "other_field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x57>
           DW_AT_data_member_location: 8
```

```
<0x2d> DW_TAG_structure_type
         DW_AT_name: "T"
         DW_AT_byte_size: 0x10
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x1
<0x37>   DW_TAG_member
           DW_AT_name: "field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x50>
           DW_AT_data_member_location: 0
<0x43>   DW_TAG_member
           DW_AT_name: "other_field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x57>
           DW_AT_data_member_location: 8
```

```
<0x2d> DW_TAG_structure_type
         DW_AT_name: "T"
         DW_AT_byte_size: 0x10
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x1
<0x37>   DW_TAG_member
           DW_AT_name: "field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x50>
           DW_AT_data_member_location: 0
<0x43>   DW_TAG_member
           DW_AT_name: "other_field"
           DW_AT_decl_file: 0x1
           DW_AT_decl_line: 0x1
           DW_AT_type: <0x57>
           DW_AT_data_member_location: 8
```

```
<0x50> DW_TAG_base_type
         DW_AT_byte_size: 0x4
         DW_AT_encoding: DW_ATE_signed
         DW_AT_name: "int"
<0x57> DW_TAG_base_type
         DW_AT_byte_size: 0x8
         DW_AT_encoding: DW_ATE_float
         DW_AT_name: "double"
```

ANATOMY OF A SIMPLE C PROGRAM

```
<0x5e> DW_TAG_subprogram
         DW_AT_external: yes(1)
         DW_AT_name: ”main”
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x4
         DW_AT_prototyped: yes(1)
         DW_AT_type: <0x50>
         DW_AT_low_pc: 0x004004b6
         DW_AT_high_pc: <offset-from-lowpc>11
         DW_AT_frame_base: len 0x1: DW_OP_call_frame_cfa
<0x7b> DW_TAG_variable
         DW_AT_name: ”var”
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x2
         DW_AT_type: <0x2d>
         DW_AT_location: len 0x9: DW_OP_addr 0x006008c0
```

ANATOMY OF A SIMPLE C PROGRAM

```
<0x5e> DW_TAG_subprogram
         DW_AT_external: yes(1)
         DW_AT_name: ”main”
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x4
         DW_AT_prototyped: yes(1)
         DW_AT_type: <0x50>
         DW_AT_low_pc: 0x004004b6
         DW_AT_high_pc: <offset-from-lowpc>11
         DW_AT_frame_base: len 0x1: DW_OP_call_frame_cfa
<0x7b> DW_TAG_variable
         DW_AT_name: ”var”
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x2
         DW_AT_type: <0x2d>
         DW_AT_location: len 0x9: DW_OP_addr 0x006008c0
```

ANATOMY OF A SIMPLE C PROGRAM

```
<0x5e> DW_TAG_subprogram
         DW_AT_external: yes(1)
         DW_AT_name: "main"
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x4
         DW_AT_prototyped: yes(1)
         DW_AT_type: <0x50>
         DW_AT_low_pc: 0x004004b6
         DW_AT_high_pc: <offset-from-lowpc>11
         DW_AT_frame_base: len 0x1: DW_OP_call_frame_cfa
<0x7b> DW_TAG_variable
         DW_AT_name: "var"
         DW_AT_decl_file: 0x1
         DW_AT_decl_line: 0x2
         DW_AT_type: <0x2d>
         DW_AT_location: len 0x9: DW_OP_addr 0x006008c0
```

We need to process this tree and turn it into usable metadata for us.

We need to process this tree and turn it into usable metadata for us.

```
typedef struct insight_type_s   *insight_type_info;
typedef struct insight_field_s  *insight_field_info;
```

We need to process this tree and turn it into usable metadata for us.

```
typedef struct insight_type_s   *insight_type_info;
typedef struct insight_field_s  *insight_field_info;

insight_type_info insight_typeof_str(const char *);
insight_type_info insight_typeof_addr(void *);
```

We need to process this tree and turn it into usable metadata for us.

```
typedef struct insight_type_s   *insight_type_info;
typedef struct insight_field_s  *insight_field_info;

insight_type_info insight_typeof_str(const char *);
insight_type_info insight_typeof_addr(void *);

const char *insight_type_name(insight_type_info);
size_t insight_type_size(insight_type_info);
```

We need to process this tree and turn it into usable metadata for us.

```
typedef struct insight_type_s   *insight_type_info;
typedef struct insight_field_s  *insight_field_info;

insight_type_info insight_typeof_str(const char *);
insight_type_info insight_typeof_addr(void *);

const char *insight_type_name(insight_type_info);
size_t insight_type_size(insight_type_info);

insight_field_info insight_field(insight_type_info,
                                 const char *);
void insight_field_set(insight_field_info,
                       void *instance, void *data,
                       size_t size);
void insight_field_get(insight_field_info,
                       void *instance, const void *data,
                       size_t size);
```

INTRODUCING TYPE_OF

we need a practical way of getting the type of anything.

INTRODUCING TYPE_OF

we need a practical way of getting the type of anything.

GNU C `__typeof__` is the perfect candidate.

```
#define type_of(Thing) ({ \
    static __typeof__(Thing) *insight_typeof_dummy = NULL; \
    insight_typeof_addr(&insight_typeof_dummy); \
  })
```

INTRODUCING TYPE_OF

we need a practical way of getting the type of anything.

GNU C `__typeof__` is the perfect candidate.

```
#ifdef __GNUC__
# define type_of(Thing) ({ \
    static __typeof__(Thing) *insight_typeof_dummy = NULL; \
    insight_typeof_addr(&insight_typeof_dummy); \
  })
#else
# define type_of(Thing) (insight_typeof_str(#Thing))
#endif
```

SOME SHORT EXAMPLES

```
insight_type_info void_t = type_of(void);
insight_type_info int_t = type_of(int);
insight_type_info struct_t = type_of(struct foo);
```

SOME SHORT EXAMPLES

```
insight_type_info void_t = type_of(void);
insight_type_info int_t = type_of(int);
insight_type_info struct_t = type_of(struct foo);

insight_type_info int_t = type_of(1);
insight_type_info long_t = type_of(1L);
insight_type_info double_t = type_of(3.14);
insight_type_info char_arr_t = type_of("Foo");
```

SOME SHORT EXAMPLES

```
struct T {
  int i;
};

int main(void) {
  struct test_struct t = {24};

  insight_type_info type = type_of(struct T);
  insight_field_info field = insight_field(type, "i");

  printf("t.i = %d\n", t.i);
  int i = 42;
  insight_field_set(field, &t, &i, sizeof (i));
  printf("t.i = %d\n", t.i);

  return 0;
}
```

What about C++?

We can build a similar API for C++:

```
class StructInfo : virtual public TypeInfo,
                   virtual public Container {
public:
  virtual Range<MethodInfo> methods() const = 0;
  virtual MethodInfo& method(std::string) const = 0;
  virtual Range<FieldInfo> fields() const = 0;
  virtual FieldInfo& field(std::string name) const = 0;
  virtual WeakRange<StructInfo> supertypes() const = 0;
  virtual StructInfo& supertype(std::string) const = 0;
  virtual bool is_supertype(const TypeInfo&) const = 0;
  virtual bool is_ancestor(const TypeInfo&) const = 0;
};
```

```
class TestClass {
public:
    TestClass(int f);
    int get_field();
private:
    int field;
};
```

SOME C++ EXAMPLES

```
class TestClass {
public:
    TestClass(int f);
    int get_field();
private:
    int field;
};

Insight::StructInfo& type = type_of(TestClass);
```

```
class TestClass {
public:
    TestClass(int f);
    int get_field();
private:
    int field;
};

Insight::StructInfo& type = type_of(TestClass);

int& field = type.field("field").get<int>(instance);
field = 55;

type.method("get_field").call<int>(instance)
```

USING REFLECTION FOR FOREIGN INTERFACES

We could:

- load an external library and automatically discover what's in it.

USING REFLECTION FOR FOREIGN INTERFACES

We could:

- load an external library and automatically discover what's in it.
- Use functions and types without their definitions

USING REFLECTION FOR FOREIGN INTERFACES

We could:

- load an external library and automatically discover what's in it.
- Use functions and types without their definitions
- Call functions in language A from language B

USING REFLECTION FOR FOREIGN INTERFACES

We could:

- load an external library and automatically discover what's in it.
- Use functions and types without their definitions
- Call functions in language A from language B
- Manipulate concepts foreign to the current language (e.g. C++ classes in C code)

USING REFLECTION FOR FOREIGN INTERFACES

We could:

- load an external library and automatically discover what's in it.
- Use functions and types without their definitions
- Call functions in language A from language B
- Manipulate concepts foreign to the current language (e.g. C++ classes in C code)
- Monkey-patch our types and vtables for profit

USING REFLECTION FOR FOREIGN INTERFACES

We could:

- load an external library and automatically discover what's in it.
- Use functions and types without their definitions
- Call functions in language A from language B
- Manipulate concepts foreign to the current language (e.g. C++ classes in C code)
- Monkey-patch our types and vtables for profit

USING REFLECTION FOR FOREIGN INTERFACES

We could:

- load an external library and automatically discover what's in it.
- Use functions and types without their definitions
- Call functions in language A from language B
- Manipulate concepts foreign to the current language (e.g. C++ classes in C code)
- Monkey-patch our types and vtables for profit

These are not (yet) implemented, but are entierly possible.

IMPLEMENTING ANNOTATIONS

ANNOTATIONS FOR CLASS METADATA

Java:

```
@Entity
@Table(name = "events")
public class Event {
    @Id
    private long id;

    private String name;
}
```

ANNOTATIONS FOR CLASS METADATA

C++:

```
$(Entity)
$(Table, .name = "events")
struct event {
    $(Id)
    size_t id;

    std::string name;
};
```

O_o

$ is a nonstandard albeit valid identifier on the GNU compiler family.

$ is a nonstandard albeit valid identifier on the GNU compiler family.

We define the following:

```
#define $(Name, ...)                                  \
  static const struct Name <unique_id> = {  \
      __VA_ARGS__                                     \
  };
```

ANNOTATIONS FOR CLASS METADATA

$ is a nonstandard albeit valid identifier on the GNU compiler family.

We define the following:

```
#define $(Name, ...)                                  \
  static const struct Name <unique_id> = {  \
      __VA_ARGS__                                     \
  };
```

The static constant holds the actual metadata.

ANNOTATIONS FOR CLASS METADATA

$ is a nonstandard albeit valid identifier on the GNU compiler family.

We define the following:

```
#define $(Name, ...)                                  \
  static const struct Name <unique_id> = {  \
      __VA_ARGS__                                     \
  };
```

The static constant holds the actual metadata.

The constant is elected as an annotation when traversing the DWARF tree.

It affects the entry directly below it.

ANNOTATIONS FOR CLASS METADATA

$ is a nonstandard albeit valid identifier on the GNU compiler family.

We define the following:

```
#define $(Name, ...)                                  \
  static const struct Name <unique_id> = {  \
      __VA_ARGS__                                     \
  };
```

The static constant holds the actual metadata.

The constant is elected as an annotation when traversing the DWARF tree.

It affects the entry directly below it.

For portability purposes, we may choose to conditionally define $ and use a more standard identifier.

What's next?

Get out of alpha and release v1.0!

(Logo designed by @pbouigue on twitter)

Any sensible contribution is welcome.

`https://snai.pe/git/insight`

`franklinmathieu+insight@gmail.com`

`Snaipe` on `#criterion-dev` @ `irc.freenode.net`

Questions?