Interfacing C, .NET and other languages

Overview

- Language interoperability
- EiffelStudio execution
  - Garbage collection
- Interfacing C
  - Calling C from Eiffel
  - Calling Eiffel from C
- Eiffel to .NET translation

Interoperability problems

- The paradigm (I)
- The object model
  - Inheritance
  - "Pure" OO vs. value types
- Data abstraction
- Flow control
  - Exceptions
  - Threads
- Executional model
  - Memory allocation
  - Garbage collection

Client / Supplier

Using Wrappers
Inheritance / Subtyping

Eiffel Class

.NET Class

Eiffel Class

.NET Class

(More of that later)

Runtime Framework

Language A

Language B

FRAMEWORK

Runtime Framework (cont.)

Every runtime framework (even the CPU itself) implements a

CODE MODEL

(in the case of OO, this is called "object model").

The code model defines a program data type and an operational semantics for values of that type.
All languages have to "compile to" this code model.

Assembler Framework

Program data-type

- Sequence of assembler instructions
- Labels
- Declaration of variables / constants

Operational semantics

- Effect of executing instructions onto registers, memory, IO ports...

.NET Framework

Program data-type

- Byte-code assembly definition
  (CIL = Common Intermediate Language)

Operational semantics

- Execution model (CLR = Common Language Runtime)
  - Memory Management
  - Object creation
  - etc.
- Base Class Library (implicit semantics)

.NET Framework

Interface and Code Adaption

EiffelStudio

Eiffel

.NET CLR

C# compiler

C#
Overview

Interfacing C

Eiffel defines syntax to interface "foreign" languages:
- external keyword (ECMA 8.31.1)
- alias keyword
- External language definitions for C, C++, and DLLs (ECMA 8.31.10 ff)

Wrapping Functions

NAME
abs - compute the absolute value of an integer

SYNOPSIS
#include <stdlib.h>
int abs(int j);

feature -- Wrappers
abs_wrap (x: INTEGER): INTEGER is
  -- Wrapper for 'abs'
  external
  "C (int) : int | <stdlib.h>"
  alias
  "abs"
end

Wrapping Functions (cont.)

feature -- Wrappers

abs_wrap (x: INTEGER): INTEGER is
  -- Wrapper for 'abs'
  external
  "C [macro <stdlib.h>] (int) : int"
  alias
  "abs"
end

Can also be used for macros:
#define abs(x) ((x >= 0) ? x : (-x))

Inline C

Encapsulating:

A possible solution:

Anything wrong?
- Missing cast of $p$ to (const char *)
- Should it really be NATURAL?
- Missing cast for return instruction
Sample: encapsulating strlen (2)

A safer solution:

```c
extern
"C inline use <string.h>"
alias
"[
EIF_NATURAL_64 Result;
size_t val = strlen ((const char *) $p);
Result = (EIF_NATURAL_64) val;
assert (((size_t) Result) == val);
return Result;
]"
end
```

Sample: encapsulating strlen (3)

We forgot to equip the routine with contracts

```c
extern
"C inline use <string.h>"
alias
"[
EIF_NATURAL_64 Result;
size_t val = strlen ((const char *) $p);
Result = (EIF_NATURAL_64) val;
assert (((size_t) Result) == val);
return Result;
]"
end
```

Mapping Eiffel types

Each Eiffel basic type has a corresponding C type:

<table>
<thead>
<tr>
<th>Eiffel Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER_8</td>
<td>EIF_INTEGER_8</td>
</tr>
<tr>
<td>INTEGER_16</td>
<td>EIF_INTEGER_16</td>
</tr>
<tr>
<td>INTEGER_32</td>
<td>EIF_INTEGER_32</td>
</tr>
<tr>
<td>INTEGER_64</td>
<td>EIF_INTEGER_64</td>
</tr>
<tr>
<td>NATURAL_8</td>
<td>EIF_NATURAL_8</td>
</tr>
<tr>
<td>NATURAL_16</td>
<td>EIF_NATURAL_16</td>
</tr>
<tr>
<td>NATURAL_32</td>
<td>EIF_NATURAL_32</td>
</tr>
<tr>
<td>NATURAL_64</td>
<td>EIF_NATURAL_64</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>EIF_BOOLEAN</td>
</tr>
<tr>
<td>CHARACTER_8</td>
<td>EIF_CHARACTER</td>
</tr>
<tr>
<td>CHARACTER_32</td>
<td>EIF_WIDE_CHAR</td>
</tr>
<tr>
<td>REAL_32</td>
<td>EIF_REAL_32</td>
</tr>
<tr>
<td>REAL_64</td>
<td>EIF_REAL_64</td>
</tr>
<tr>
<td>POINTER</td>
<td>EIF_POINTER</td>
</tr>
</tbody>
</table>

Mapping Eiffel types (2)

- Unsized types CHARACTER, INTEGER and NATURAL use the C corresponding type of their mapping.
- Size of Eiffel types guarantees length of underlying Eiffel C type.
- Used to cast values obtained from calling C routines to expected Eiffel type.
- Used when calling Eiffel runtime routines from C code.

POINTERs

- Any pointer definition of C is mapped to the built-in data-type POINTER.
- Access to elements of type POINTER is opaque. They cannot be accessed in Eiffel.
- You can get the pointer to any Eiffel object using the $ operator:
  - `my_pointer := $my_string`

BEWARE OF THE GARBAGE COLLECTOR

Mark / Sweep Garbage Collection

[Diagram of garbage collection with objects and the stack]
Eiffel Hector: Adopt / Access / Wean

**EIFF_OBJECT**

```c
void eif_adopt (EIFF_OBJECT r)

C wants to keep the reference
```

**EIFF_REFERENCE**

```c
void eif_access (EIFF_OBJECT h)

Access the object given by handle h
```

```c
void eif_wean (EIFF_OBJECT h)

Allow the GC to reclaim the object pointed to by h
```
Proxy Objects

Shared structure

Helper classes

EiffelBase provides some classes to facilitate interfacing with C code:

- **C_STRING**
- **MANAGED_POINTER**
- **MEMORY_STRUCTURE**

C_STRING

Convert C string to Eiffel string and vice versa

```eiffel
create c_string_make ("TEST")
io.put_string (c_string_make (c_string.item).out)

create c_string_make_empty (512)
-- Some code that fills 'c_string'
io.put_string (c_string_string)
```

MANAGED_POINTER

Automatic memory management of memory
Allow random access with read/write of any basic type

```eiffel
c_area: MANAGED_POINTER
val: INTEGER_32
create c_area_make (512)
c_area.put_integer_32 (val, 1)
c_area.put_integer_32_1s (val, 1)
c_area.put_integer_32_be (val, 1)
val = c_area.read_integer_32 ($) val = c_area.read_integer_32 ($) val = c_area.read_integer_32 ($)```

MEMORY_STRUCTURE

Deferred class: recommended as ancestor to all classes wrapping a C structure

```eiffel
class POINT inherit MEMORY_STRUCTURE
create new
feature -- Measurement
structure_size: INTEGER
external "C inline svc 'point_h'"
alias "return sizeof(struct point)";
end

feature -- Access
x: INTEGER do Result := c_x (item) end
feature [NONE] -- Implementation
end
require p_not_null: p /= default_pointer
external "C inline svc 'point_h'"
alias "return ((struct point *) $p)->x;";
end
```
Callbacks

```
class CALLBACK
  feature --- Action
    callback (val: INTEGER) do ... end
  feature --- Callback initialization
    callback_setup do
      c_callback_setup [Current, $callback]
    end
  feature (NONE) --- Implementation
    c_callback_setup (obj: ANY; fnptr: POINTER)
    external "C inline use "eif_eiffel.h"
    alias
      "[/callback_object = eif_adapt (obj)];
      callbackroutine = (CALLBACK_PROC) $fnptr;
    "
  end
end
```

Do not forget to call `eif_wean` on `callback_object`

CECIL

"C-to-Eiffel Call-In Library"
- Library to access features of Eiffel objects
- Compile your Eiffel system to an own library
  - make cecil
  - builds `lib<system>.a` / `.lib`

Using CECIL

```
#include "eif_setup.h"
#include "eif_eiffel.h"

int main(int argc, char *argv, char *envp)
  /* Please respect this signature: 'argc', 'argv' and 'envp' are used in EIF_INITIALIZE. */
  {
    /* declarations of variables */
    /* Initialize the Eiffel run-time. */
    EIF_INITIALIZE(failure)
    /* body of your "main" function */
    /* Reclaim the memory allocated by the Eiffel run-time. */
    EIF_DISPOSE_ALL
  } 
```

Using CECIL (cont.)

```
Eiffel: print(obj)

Print (EIF_OBJECT obj)
{
  EIF_PROCEDURE ep;
  EIF_TYPE_ID tid;
  tid = eif_type_id("ANY");
  ep = eif_procedure("print", tid);
  (ep) (eif_access(obj), eif_access(obj));
}
```

Overview

Interfacing C

Interfacing .NET
**Similarities between .NET and Eiffel**

- Object-oriented
- Imperative code
- Strong type system
- Type can have many ancestors
- Garbage collector

**Eiffel vs. .NET object model**

- Multiple Inheritance
- Free Generics
- Expanded Types
- Clusters
- Once
- Call by Value
- Restricted Exceptions
- Single Inheritance
- Restricted Generics
- Value Types
- Packages, Assemblies
- Static
- Call by Value or Reference
- Free use of try/catch

**Mapping Inheritance**

```
A -- B -- C
   
D
```

**Mapping Inheritance**

```
A
   
IMPL.A
   
IMPL.B
   
IMPL.C

B
   
IMPL.B
   
IMPL.D
   
IMPL.A

C
```

**Browsing .NET assemblies**

**Open Problems**

- Eiffel for .NET design is based on .NET 1.1 (work with 2.0)
- No link between Eiffel and .NET Genericity
- Contracts are compiled into the code
- Lots of name mangling
- Cumbersome Interface
Summary

Compiling to C
- Language support integrated into Eiffel
- Well-defined interface
- Call-back Eiffel features using CECIL

Compiling for .NET
- Everything is fine if we only execute Eiffel on .NET
- Matching the model works, incl. multiple inheritance
- Some Eiffel concepts cannot be mapped to .NET and vice versa