Fine-grained aspects of automatic refactoring in C2Eiffel

Report

Author(s):
Friedli, Adrian

Publication date:
2011

Permanent link:
https://doi.org/10.3929/ethz-a-007313685

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Technical report 741

Chair of
Software Engineering

Fine-grained aspects of automatic refactoring in C2Eiffel

Master Thesis

Adrian Friedli
ETH Zurich
adrianfriedli@student.ethz.ch

April 1, 2010 - September 30, 2010

Supervised by:
Marco Trudel
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

The C2Eiffel framework [1] translates C source code to Eiffel source code. It aims to generate code that looks natural to a programmer such that it is readable and maintainable. While this is already the case for most parts of common C programs, there are areas for improvements.

External C functions are currently wrapped in the generated Eiffel code since the used libraries are not translated. In this thesis I will replace some standard and often used C functions by the corresponding Eiffel equivalents. In the attempt to replace standard C string functions I will also try to replace the current translation of C strings to integer arrays with native Eiffel strings.

I will also remove jump statements like *break* that are often used in C programs but are not available in Eiffel. This will increase the readability of the code compared to the current implementation that emulates the jump mechanism in Eiffel and produces hard to read code.

Contents

Chapter 1

Introduction

The C2Eiffel framework [1] developed by Marco Trudel at ETH Zurich translates C source code to Eiffel source code. One of the goals of C2Eiffel is that the translated code looks natural to a programmer, that means the differences between the translated Code and manually written code should be as small as possible. The advantages of natural looking code is that is is more readable and easier to maintain. For most parts of common C programs C2Eiffel generates already such code.

The goal of this thesis is to improve the generated code even further. To measure the improvements we will test the generated code with the ten real world projects described in section 1.1. There are three main areas for improvements:

The first area is the translation of function calls to external C libraries. Since those libraries are not translated to Eiffel, the generated Eiffel code still calls those C functions directly. In this thesis I will replace some of those function calls with the corresponding Eiffel features. In chapter 2 we translate as many *printf* statements as possible to the Eiffel's counterpart *Io.put_string*. We will restrict us to the features already present in the Eiffel base library. Even with this restriction we can translate more than two thirds of the *printf* calls in the reference projects. In chapter 3 we translate some of the standard C string functions to the corresponding Eiffel features.

The second area is the translation of C strings and structs to Eiffel. At the moment C strings are translated to arrays of integers. In chapter 3 we try to replace them by Eiffel strings but we will see that this attempt will not be very successful. The current translation of structs to Eiffel classes uses embedded C code. The attempt to remove this C code in chapter 4 will unfortunately not be successful too.

The third area is the translation of jump statements like *break* or *return* that are not available in Eiffel. Currently those statements are handled in a generic way that is hard to read and maintain (see section 5). To improve this code I will remove *break*, *continue* and *return* statements by refactoring the C source code in chapter 5. This is done amongst others by adding *if* statements that allow to skip code when a jump statement was executed. Those code changes have only a small impact on the readability of the code compared to the original C source code as the analysis with the reference projects shows. Compared to the currently generated code it considerably improves readability.

1.1 Reference Projects

We will evaluate our efforts on the following real world projects:

- Hello World
  A simple hello world program that just prints *hello world* to the command line with the *printf* function.
- Micro_httpd [3]
  It's a small HTTP server with basic features and about 200 lines of code.
- Xeyes [4]
  A graphical program that shows two eyes which follow the mouse cursor.
- Less [2]
  A Unix program that shows the content of a file page by page in the terminal.
- Wget [6]
  A downloader that is part of the GNU Project.
- Links [5]
  A text based web browser.
- Vim [10]
  A feature rich text editor.
- Curl [7]
  A multiprotocol file transfer library.
- Gmp [8]
  The GNU multiple precision arithmetic library.
- Gsl [9]
  The GNU scientific library that provides a wide range of numerical routines.

Chapter 2

Translation of *printf* statements to Eiffel's counterpart

In this chapter we look at C's *printf* statements and which ones can be translated to Eiffel's *Io.put_string* feature and how.

2.1 The *printf* statement in C

The method signature of *printf* is the following:

```
int printf ( const char * format, ... );
```

The *format* string is the text to write to *stdout* and can contain format tags. Those tags get replaced by the additional arguments of the *printf* statement and formated according to the formating arguments inside the tags. A format tag has the following structure:

```
%[flags][width][.precision][length]specifier
```

where the parts in `[]` are optional. A format tag always starts with a `%` followed by a sequence of arguments:

specifier The specifier is the only required argument. It defines the type of the value that will replace this tag. The most common specifiers are *'c'* for characters, *'d'* or *'i'* for integers, *'f'* for floating-point values and *'s'* for strings.

flags The flags control the justification (*'-'*), signs before the value (*'+'* and *space*) and padding with zeros (*'0'*).

Table 2.1: Supported *printf* specifiers and format tags

specifier	format arguments
c	precision (has no effect)
d, i	width (incl. *)
f	width, precision (incl. *)
s	-
u	-

width The width specifies the minimal number of characters to be printed. If the representation of the value is shorter it will be padded with blanks.

precision The precision specifies the minimal number of digits to be printed for integer specifiers and the exact number of decimal digits for floating-point specifiers.

length The length specifies how the argument value is interpreted (as *short int*, *long int*, . . .).

The width and precision argument can be *'*'*, that means that the value is given as an additional argument before the argument that will be formated. If the *printf* statement is executed successfully the number of written characters, otherwise a negative number is returned.

2.2 Translatable *printf* statements

Since the return type of the Eiffel feature *Io.put_string* is *Void*, no *printf* statements can be translated where it is used as an expression, for example as the right hand side of an assignment. To make sure the translation works in all possible places, the replacement has to be a single statement too. If we restrict us to built in features of Eiffel the translatable *printf* statements only contain the specifiers and format arguments listed in table 2.1.

In addition to those *printf* statements we can support *fprintf* statements where the destination file stream is either *stdout* or *stderr*. In the case of *stdout* the *fprintf* and *printf* statements are equal and in the *stderr* case the output is redirected to the error stream.

2.3 Translation

In the following we look at five examples and how they are translated.

The first example (line 2 in listing 2.1) is a *printf* statement with a character specifier. In C2Eiffel characters are stored in an *INTEGER_8* and can be converted to a character with the feature *to_character_8*. The translated print statement is shown in listing 2.2 on line 2.

Listing 2.1: *C*: Five exemplary *printf* statements

```c
// c is of type char
printf("A character: %c\n", c);

// i is of type int
printf("An integer: %d\n", i);

// width is of type int
printf("An integer with variable width: %*i\n", width, i);

// d is of type double and precision is of type int
printf("A double with variable width and precision: %*.*f\n", width, precision, d);

// s is of type char*
printf("A string: %s\n", s);
```

Listing 2.2: *Eiffel*: The five translated *printf* statements

```eiffel
-- c is of type INTEGER_8
Io.put_string ("A character: " + (c).to_character_8.out + "%N")

-- i is of type INTEGER_32
Io.put_string ("An integer: " + i.out + "%N")

-- width is of type INTEGER_32
Io.put_string ("An integer with variable width: " + (create {FORMAT_INTEGER}.make (width)).formatted (i) + "%N")

-- d is of type REAL_64 and precision is of type INTEGER_32
Io.put_string ("A double with variable width and precision: "+ (create {FORMAT_DOUBLE}.make (width.max (precision), precision)).formatted (d) + "%N")

-- s is of type CE_POINTER [INTEGER_8]
Io.put_string ("A string: " + eif_string (s) + "%N")
```

The second example (line 5 in both listings) contains an integer specifier. The string representation of the integer is concatenated with the preceding string.

In the third example on line 8 we have the same with a variable width argument. In this case we use the *FORMAT_INTEGER* class which pads the integer with blanks if its representation is shorten than the specified width. The class offers more options to format the integer but for this we would need multiple instructions, which is not possible because the translation should be a single statement as discussed before.

On line 11 of the two listings we have a print statement that formates a floating-point value with variable width and precision. In the Eiffel code the *FORMAT_DOUBLE* class is used analogously to the integer formating. For the width we have to use the maximum of width and precision since the *make* feature requires that the width is greater or equal to the precision.

In the last example (line 14) we have a print statement with a string specifier. In the Eiffel code the array of characters (*CE_POINTER [INTETER_8]*) has to be converted to a *STRING*. This is done by the helper feature *eif_string*.

The translation of a *fprintf* statement with the error stream as first argument is the same as for the *printf* statement except that *Io.put_string* is replaced by *Io.error.put_string*.

For more translations see Appendix A with a list of 26 *printf* translations and the output of the executed print statements.

2.4 Translation in real examples

In order to see how many *printf* statements can be translated, we look at the reference projects listed in section 1.1 (Table 2.2). In those projects more than two-thirds of the *printf* statements can be translated in the above way to Eiffel.

Table 2.2: Number of translated *printf* statements

Project	Total *printf* statements	Replaced *printf* statements
hello world	1	1
micro_httpd	16	13
xeyes	6	6
less	0	0
wget	63	21
links	29	27
vim	6	4
libcurl	0	0
libgmp	16	10
libgsl	82	66
Total	219	148

Chapter 3

Translation of C strings to Eiffel Strings

In this chapter we look at C strings an how they can be translated to Eiffel strings. We will also see how the standard string function calls (from the *string.h* header file) can be replaced.

3.1 C Strings

In C a string is stored as an array of characters. Like for all arrays there are two possibilities to declare and allocate space for a string variable.

The first way is to declare the variable as character pointer as shown in listing 3.1 on line 1. With this declaration only space for the pointer is allocated, not for the content of the string. This is done with one of the memory allocation functions like *malloc*.

The second way to declare a string variable is as character array as shown in listing 3.1 on line 2. In this case also memory for N characters is allocated.

Besides the difference in memory allocation, there is no difference in the usage of the string variables.

Listing 3.1: *C*: Declaration of string variables

```
1   char *char_pointer;
2   char char_array[N];
```

3.2 Eiffel Strings

In Eiffel strings are also character arrays. Those strings have, in contrast to C strings, no terminating null character at the end. The length of the Eiffel string is stored within the *STRING* class. This difference has consequences when accessing individual characters of the string. We will discuss those later in this chapter.

3.3 Standard C string functions

The standard C string functions are defined in the *string.h* header file. The following functions are there defined:

- *char * strcat(char *str1, const char *str2);*
  Appends *str2* to *str1*, where the terminating null character of *str1* is overwritten. The function returns a pointer to *str1*.
- *char * strncat(char *str1, const char *str2, size_t n);*
  Same as *strcat* but appends at most *n* characters.
- *char * strchr(const char *str, int c);*
  Searches for the first occurrence of *c* in the string *str* and returns a pointer to that location.
- *char * strrchr(const char *str, int c);*
  Same as *strchr* but searches the last occurrence of *c*.
- *int strcmp(const char *str1, const char *str2);*
  Compares the two strings and returns zero if they are equal, less then zero if the first string is less than the second one and greater zero otherwise.
- *int strncmp(const char *str1, const char *str2, size_t n);*
  Same as *strcmp* but compares only the first *n* characters.
- *char * strcoll(const char *str1, const char *str2);*
  Same as *strcmp* but takes the *LC_COLLATE* setting into account.
- *char * strcpy(char *str1, const char *str2);*
  Copies *str2* to *str1* and returns a pointer to *str1*.
- *char * strncpy(char *str1, const char *str2, size_t n);*
  Same as *strcpy* but copies only *n* character.
- *size_t strspn(const char *str1, const char *str2);*
  Searches for the first sequence of characters in *str1* that does contain only characters from *str2* and returns the length of this sequence.
- *size_t strcspn(const char *str1, const char *str2);*
  Same as *strspn* but searches for a sequence that does not contain any characters from *str2*.

- *char * strerror(int errnum);*
  Returns a pointer to the corresponding error message.
- *size_t strlen(const char *str);*
  Returns the length of the string.
- *char * strpbrk(const char *str1, const char *str2);*
  Returns a pointer to the first character in *str1* that is also present in *str2*.
- *char * strstr(const char *str1, const char *str2);*
  Searches for the first occurrence of the entire string *str2* in *str1* and returns a pointer to the beginning.
- *char * strtok(char *str1, const char *str2);*
- *size_t strxfrm(char *str1, const char *str2, size_t n);*

3.4 Translatable string variables

The following types of C strings can potentially be translated to Eiffel strings:

- Global variables
- Function parameters and return types of internal functions
- Local variables

Supported operations on the string variables are:

- Assignment, where the right hand side of the assignment can be a string constant, another string variable or the return value of a function. This introduces a dependency that both sides of the assignment can be translated to Eiffel strings or none.
- Function calls to internal functions with a string variable as parameter. This introduces a dependency that both the declared and the actual parameter can be translated or none.
- Function calls to some external functions like *strcpy*, *strlen*, *printf* and others. The whole list of supported external functions is given in section 3.5.
- Equality or inequality check on two string variables. This introduces also a dependency between the two variables like an assignment.

As soon as one of the following operation is applied to a string variable, this string variable can not be translated to an Eiffel string. Also all variables that have a dependency to this one introduced through an assignment or a function call can not be translated.

- The string is declared as a struct field.

- An array of strings is not supported.
- The string is used inside the signature of a function pointer or as parameter in a call of one.
- The string variable is used in an unsupported external function call (see section 3.5).
- The string variable is used with pointer arithmetic like $sp + 1$.
- The string variable is casted to another type. A check if the pointer is zero (includes a cast to unsigned integer) is supported.
- Accessing an individual character of a string variable (see end of this section for the discussion of the problem).
- Dereferencing a string variable (equal to accessing the first character of the string).

Now we look at a small meaningless C program with string variables (listing 3.2) and the translation to Eiffel (listing 3.3). The Eiffel code is slightly refactored to have everything in one class and to fit on one page.

Most of the translation is straight forward. The types *char ** and *char []* are replaced by *STRING_8*, assignments remain normal assignments and *printf* calls are replaced where possible like described in chapter 2. Interesting is the replacement of the standard C string function *strcpy* (line 24 in the C code). It is replaced by creating an identical copy of the string by calling *twin* and an assignment (line 47 in the Eiffel code). Also noticeable is the *printf* call that can't be translated (line 25 in the C code and line 48 in the Eiffel code). The string *s1* can still be translated to an Eiffel string but has to be converted before the call. In section 3.5 we will see more translations of standard string functions.

Array indexing problem

Now we discuss why accessing individual characters of a string can not be supported.

In C we allocate a given number of bytes for a string either by calling a memory allocation function or by declaring an array of characters with a given size. Then accessing any of those bytes with *s[n]* is valid even if n is greater or equal than the length of the string (the size returned by *strlen*). An example usage is when multiple strings are stored consecutively in one variable. In Eiffel you must not access any index greater than the length of the string. Since we can not decide on compile time if an access is within the bounds it can not be supported.

Listing 3.2: *C*: Program with string variables

```c
#include <stdio.h>
#include <string.h>

char *cp;
static char string[12] = "Hello World";

char* select(char *s, int i) {
  char *tmp;

  if (i != 0)
    tmp = s;
  else
    tmp = cp;

  return tmp;
}

int main()
{
  cp = "Test String";
  printf("%s\n", cp);

  char s1[100];
  strcpy (s1, cp);
  printf("%.4s\n", s1);

  char *s2 = select(string, 1);
  printf("%s\n", s2);

  return 0;
}
```

Listing 3.3: *Eiffel*: The translation of the C program in listing 3.2

```
class
  P_TEST_STRING_TRANSFORMER
inherit
  S_STDIO
create
  default_create

feature {NONE} -- Initialization
  default_create
    do
      string := "Hello World"
    end

feature {ANY} -- attributes
  cp: STRING_8 assign set_cp
  set_cp (a_cp: STRING_8)
    do
      cp := a_cp
    end

  string: STRING_8 assign set_string
  set_string (a_string: STRING_8)
    do
      string := a_string
    end

feature {ANY} -- routines
  select2 (a_s: STRING_8; a_i: INTEGER_32): STRING_8
    local
      l_tmp: STRING_8
    do
      if a_i /= 0 then
        l_tmp := a_s
      else
        l_tmp := cp
      end
      Result := l_tmp
    end

  main: INTEGER_32
    local
      l_s1: STRING_8
      l_s2: STRING_8
    do
      cp := "Test String"
      Io.put_string (cp + "%N")
      l_s1 := cp.twin
      printf ([ce_string ("%%.4s%N"), ce_string (l_s1)]).
          do_nothing
      l_s2 := (select2 (string, 1))
      Io.put_string (l_s2 + "%N")
      Result := 0
    end
end
```

3.5 Translation of standard C string functions to Eiffel features

From the standard C string functions listed in section 3.3 six can be replaced by Eiffel features. The functions and their replacements are listed in table 3.1.

Table 3.1: Standard C string functions and their Eiffel replacement

C string function	Eiffel code
strcat(s1, s2) *	s1.append(s2)
strncat(s1, s2, n) *	s1.append(s2.substring (1, s2.count.min(n)))
strcmp(s1, s2)	s1.three_way_comparison(s2)
strncmp(s1, s2, n)	s1.substring(1, s1.count.min(n)).three_way_comparison (s2.substring(1, s2.count.min(n)))
strcpy(s1, s2) *	s1 := s2.twin or s1 := s2 (if s2 is a string literal)
strlen(s)	s.count

The functions marked with a * store the string result in the first argument and also return a pointer to it. This is not possible in Eiffel so those functions can only be replaced if they are used as statements and not as expression. Unfortunately there is no feature in Eiffel to get a prefix of a string with at most *n* characters. So we have to use *substring* and we have to calculate the end position since *substring* returns an empty string if *n* is greater than the length of the string.

The replacement of the functions in table 3.1 is possible when at least the first string argument can be transformed to an Eiffel string. If the second string argument can't be transformed, then it can be converted on runtime with *eif_string* to an Eiffel string since it is only read.

In addition to the above replacements it is for most functions listed in section 3.3 possible to have one or more arguments as Eiffel string even if the function can not be replaced by a Eiffel feature. Those Eiffel strings are converted with *ce_string* before the call. This conversion is possible when the string is only read and no pointer to it is kept or returned by the function. Table 3.2 lists all standard string functions listed in section 3.3 and which arguments can be Eiffel strings.

3.6 Implementation of the analysis

The translation is done in two phases. In the first phase all string variables are collected and their usage in the code is analyzed. In the second phase the types are replaced by a new ast node called *AST_EIFFEL_STRING_TYPE*, the external functions are replaced like described in section 3.5 and some other adjustments are done.

In this section we will look at the first phase. The second phase is straight

Table 3.2: Eiffel strings as arguments to standard C string functions

C string function signature	Eiffel string as 1st arg.	Eiffel string as 2nd arg.
char * strcat(char *, const char *)	no	yes
char * strncat(char *, const char *, size_t)	no	yes
char * strchr(const char *, int)	no	-
char * strrchr(const char *, int)	no	-
int strcmp(const char *, const char *)	yes	yes
int strncmp(const char *, const char *, size_t)	yes	yes
char * strcoll(const char *, const char *)	yes	yes
char * strcpy(char *, const char *)	no	yes
char * strncpy(char *, const char *, size_t)	no	yes
size_t strspn(const char *, const char *)	yes	yes
size_t strcspn(const char *, const char *)	yes	yes
char * strerror(int)	-	-
size_t strlen(const char *)	yes	-
char * strpbrk(const char *, const char *)	no	yes
char * strstr(const char *, const char *)	no	yes
char * strtok(char *, const char *)	no	yes
size_t strxfrm(char *, const char *, size_t)	no	yes

forward and is not discussed here.

For the analysis we use a dependency graph to store the dependencies between the string variables introduced through assignments or function calls (see section 3.4). Figure 3.1 shows the dependency graph for the C code in listing 3.2. The numbers next to the arrows indicate the line in the code that introduces the dependency. For example the assignment on line 11 introduces a dependency between the variables *tmp* and *s*.

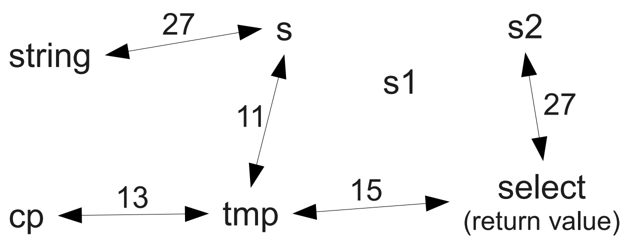

Figure 3.1: The dependency graph for the C code in Listing 3.2. The numbers next to the arrows indicate the line in the code where the dependency is introduced. The variable *s1* has no dependency to other variables.

While collecting those dependencies we also check the code for unsupported operations on the string variables (see section 3.4). In such situations the affected variables are marked as *bad* in the dependency graph. At the end of this first phase all variables that have a dependency to a bad one (possibly through other variables) are also marked bad. After that all variables that are not marked as bad can be translated to Eiffel strings.

To get a better idea why a C string can not be translated to an Eiffel string, we also set a flag while marking a variable as bad. The used flags and in what situations they are set is described below.

Bad_ref flag
: This flag is set while updating the dependency graph when a node has a dependency to a bad one and no other flag was set before (so it was not marked as bad before). Those variables could be translated to Eiffel strings if their dependencies wouldn't have been marked as bad.

Extern_function flag
: This flag is set when a variable is used as argument of an external function that can't be replaced.

Struct_member flag
: This flag is set when the variable is a struct field.

Pointer flag
: This flag is set if the pointer of the string variable is used, for example in pointer arithmetic like *cp++*.

Cast flag
: This flag is set when a string variable is casted to another type. Most of the time this is a cast to or from *void **.

Function_pointer flag
: This flag is set when a string variable is used inside the signature of a function pointer.

Assignment flag
: This flag is set when a variable is part of an unsupported assignment like *array[0] = a_string*.

Initializer flag
: This flag is set when a variable is initialized with something else than a string literal. In those cases the variable is mostly used as an array for small integers.

Array flag
: This flag is set when an individual character is accessed. See the array indexing problem at the end of section 3.4.

Other flag
: This flag is set when a variable is part of an *AST_EIFFEL_EXPRESSION*. This happens when another transformation step has already introduced Eiffel code into the *AST*.

3.7 Statistics

Table 3.3 shows how many strings in the reference projects (see section 1.1) can be translated to Eiffel strings. In average over all projects only 1.9 percent of the strings can be translated.

Table 3.3: Number of translatable strings in real projects

Project	Total strings	Translatable strings	Percentage
(1) hello world	0	0	-
(2) micro_httpd	31	6	19.4%
(3) xeyes	0	0	-
(4) less	508	6	1.2%
(5) wget	2274	10	0.4%
(6) links	246	2	0.8%
(7) vim	2315	27	1.2%
(8) libcurl	1036	2	0.2%
(9) libgmp	143	3	2.1%
(10) libgsl	263	74	28.1%
Total	6818	130	1.9%

Table 3.4 shows in more detail why the strings can not be translated. It lists for all flags introduces in section 3.6 for how many strings this flag was set. The projects *hello world* and *xeyes* are omitted, since they have no strings. Table 3.5 summarizes those numbers.

Table 3.4: Statistics for individual flags and projects

	Projects (numbers from table 3.3)							
Flags	(2)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
(Total strings)	31	508	2274	246	2315	1036	143	263
Bad_ref	8	196	713	58	376	300	26	61
Extern_function	15	96	785	31	399	506	56	26
Struct_member	0	14	113	6	35	136	5	21
Pointer	6	125	338	18	51	177	32	7
Cast	0	80	377	105	1612	236	36	8
Function_pointer	0	14	54	0	2	6	9	2
Assignment	2	24	37	1	13	89	7	2
Initializer	0	0	32	0	2	0	0	0
Array	6	166	620	80	86	235	54	87
Other	0	18	122	4	8	47	13	0
Bad marks	70	1278	5874	480	3348	2980	383	416
Average bad marks	2.3	2.5	2.6	2.0	1.4	2.9	2.7	1.6

The three most often set flags are the cast flag, the extern_function flag and the array flag. If we ignore the array indexing problem, more string variables could be translated (41 more in the examples (1) to (10) in total) but we can't guarantee that the generated code doesn't rise a precondition violation. But in most of the programs this wouldn't be the case.

Table 3.5: Summarized statistics for individual flags

Flags	Total	Percentage
(Total strings)	6816	
Bad_ref	1738	25.5%
Extern_function	1914	28.1%
Struct_member	330	4.8%
Pointer	754	11.1%
Cast	2454	36.0%
Function_pointer	87	1.3%
Assignment	175	2.6%
Initializer	34	0.5%
Array	1334	19.6%
Other	212	3.1%
Bad marks	14'829	
Average bad marks	2.2	

Chapter 4

Translation of C structs to Eiffel classes

The goal of this chapter is to replace C structs with simple Eiffel classes that only contain the attributes and setter features for them and no interfacing with C code.

4.1 Structs in C

In C a struct is used to group multiple variables together in a collection. It is a light version of a class like they are known from *C++* or *Eiffel*, since structs can only have fields and no functions.

The only operations allowed with structs are copying, assigning to another struct (this includes passing structs as arguments to functions), accessing the address with the `&` Operator or accessing the fields with the `.` or `->` Operators.

Listing 4.1 shows how structs can be declared.

Listing 4.1: *C*: Declaration of structs

```c
struct fpoint_2 {
  float x;
  float y;
};
struct fpoint_2 fp2;

struct fpoint_3 {
  float x;
  float y;
  float z;
} fp3;
```

Listing 4.2: *Eiffel*: The translation of the struct *fpoint_2* from listing 4.1

```
class
  FPOINT_2

feature {ANY} -- Eiffel getters

  x: REAL_32 assign set_x

  y: REAL_32 assign set_y

feature {ANY} -- Eiffel setters

  set_x (a_x: REAL_32)
    do
      x := a_x
    end

  set_y (a_y: REAL_32)
    do
      y := a_y
    end
end
```

On the lines 1 to 4 a struct with the name *fpoint_2* is declared. Then line 5 declares a variable with the name *fp2* of this struct type. The declaration of the struct and variables of this type can be combined. This is done on line 7 to 11 where a struct with the name *fpoint_3* and a variable with the name *fp3* is declared.

4.2 Translatable structs

Listing 4.2 show the translation of the struct *fpoint_2* from listing 4.1 to a simple Eiffel class that only has the attributes and the setter features.

Structs can be translated as long as no variable of this struct type is used in one of the following situations:

- An array of structs is not supported.
- The struct type is used inside the signature of a function pointer or as parameter in a call of one.
- A struct variable is used in an external function call.
- A struct variable is casted to another type. Casts between struct types can be supported if one can inherit from the other (see later in this section)
- The struct variable is used in embedded assembler code.

Listing 4.3: *C*: Inheritance for structs

```c
struct list {
  void *next ;
  void *prev ;
};

struct sub_list {
  sub_list *next ;
  sub_list *prev ;
  char *s;
};
```

Pointer to structs

In Eiffel and other object oriented languages variables are references to objects. When assigning one variable to the other only the reference is copied, the object itself is not. This is different for structs in C. They behave like simple data types, that means that in assignments and function calls the whole struct is copied. When we translate this to Eiffel we have to copy the whole class too. When the variable is declared as a pointer to a struct then the behavior is the same as with references in Eiffel.

In both cases we can replace the variable by the same Eiffel class but we have to handle assignments and function calls accordingly.

Inheritance between structs

When a struct extends another struct we can introduce an inheritance relationship. The necessary condition for such an inheritance is that the extending struct has all struct fields of the base struct with the same types or possibly subtypes. The names don't have to be the same. They can be renamed with the *rename* clause in Eiffel. In addition to these fields the extending struct can have more fields with any type. Casts between two structs that have an inheritance relation are supported in the translation to simple Eiffel classes. Listing 4.3 shows two structs, where the second one extends the first one.

4.3 Analysis Implementation

The translation works the same way as the translation of C strings to Eiffel strings described in section 3.6. But since the number of translatable structs (see section 4.4) is really small, the actual replacement of the structs is not implemented. For the analysis we use also a dependency graph like we did in the analysis for the translation of C string to Eiffel strings.

This time we don't translate the type of individual variables, now we want to replace the whole type implementation. This means that a struct type is replaced in the whole program or nowhere. As a consequence assignments or function calls don't introduce dependencies anymore. The only situation where a dependency is introduced is when a struct type is used inside the declaration of an other struct. This dependency is directed from the inner to the outer struct type, means that when the outer struct type is marked as bad so is the inner one.

In addition to the flags introduced in section 3.6 we need two more flags for the analysis of the structs:

Union flag
: In C2Eiffel unions and structs are threated equally, only a boolean value indicates the difference. So this flag is set for all unions because unions can not be translated to simple Eiffel classes.

Assembler flag
: This flag is set whenever a struct type is used inside embedded assembler code.

4.4 Statistics

Table 4.1 shows how many structs in the reference projects can be translated to simple Eiffel classes. The numbers marked with a * are upper limits. Since the translation is not implemented, the translation of those structs is not tested. So the actual number could be less if not all unsupported operations are recognized. In average over all projects only 6.5% of the structs can be translated (at most).

Table 4.1: Number of translatable structs in real projects

Project	Total structs	Translatable structs	Percentage
(1) hello world	0	0	-
(2) micro_httpd	6	0	0%
(3) xeyes	61	1	1.6%
(4) less	28	1	3.6%
(5) wget	86	1	1.2%
(6) links	149	0	0%
(7) vim	463	7*	1.5%
(8) libcurl	133	1	0.8%
(9) libgmp	36	5*	13.9%
(10) libgsl	303	66*	21.8%
Total	1265	82*	6.5%

The table 4.2 shows in more detail why the structs can't be translated. It lists for all flags from section 3.6 and 4.3 how many structs those flag have. Additionally it shows how often the variables were marked as bas in total and on average. The project *hello world* is omitted, since it has no structs. Table 4.3 summarizes those numbers.

Table 4.2: Statistics for individual flags and projects

	Projects (numbers from table 4.1)								
Flags	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
(Total structs)	6	61	28	86	149	463	133	36	303
Bad_ref	1	38	4	4	9	102	25	2	4
Extern_func.	3	11	7	17	17	89	13	2	1
Union	0	2	1	3	4	28	17	4	5
Pointer	3	6	17	47	78	174	69	15	43
Cast	0	3	10	54	98	190	55	16	200
Function_ptr.	1	8	0	2	20	52	6	3	13
Array	1	5	7	17	31	88	12	10	7
Assembler	0	0	0	1	1	2	1	1	0
Other	0	0	0	14	20	27	9	3	5
Bad marks	27	185	257	1030	5507	9848	2386	844	2295
Av. bad marks	4.5	3.0	9.2	12.0	37.0	21.3	17.9	23.4	7.6

Table 4.3: Summarized statistics for individual flags

Flags	Total	Percentage
(Total structs)	1265	
Bad_ref	189	14.9%
Extern_function	160	12.6%
Union	64	5.1%
Pointer	452	35.7%
Cast	626	49.5%
Function_pointer	105	8.3%
Array	178	14.1%
Assembler	6	0.5%
Other	78	6.1%
Bad marks	22'379	
Average bad marks	17.7	

Chapter 5

Removing break, continue and return statements

The C programming language allows to jump from one place in the code to another with jump instructions like *goto*, *break* and others. The purpose of this chapter is to remove some of those jump instructions because Eiffel doesn't support them. In particular the following jump instructions are removed:

break (only those belonging to a loop)
: When a *break* statement is executed, the execution immediately exits the current loop.

continue
: When a *continue* statement is executed, the execution skips the rest of the loop. In a *for* loop the execution continues with the increment block. For all other loops the execution continues with the checking of the loop condition.

return
: When a *return* statement is executed, the function terminates immediately. No further code is executed in this function.

Goto's are not removed by this transformation, they are handled by a later transformation step.

Existing translation

The existing handling of *break*, *continue* and *return* emulates jumps (also *goto*) in Eiffel. This is achieved with an inspect instruction inside a loop and an additional variable (*l_pos*) that stores the jump destination. The different inspect values mark the different jump destinations and a special value (-1) is used as an exit condition for the loop. After the inspect statement the *l_pos* variable is incremented. So when no jump statement has changed the variable, execution

Listing 5.1: *C*: Gcd Algorithm

```c
int gcd(int a, int b) {
  if (a == 0) {
    return b;
  }

  while (1) {
    if (b == 0) {
      break;
    }
    if (a > b) {
      a -= b;
    } else {
      b -= a;
    }
  }

  return a;
}
```

just continues with the directly following instructions. As an example listing 5.2 shows the transformation of the *gcd* algorithm from listing 5.1. This simple example shows also that the generated Eiffel code is not easy readable.

The advantage of this approach is that all jump statements can be handled at the same time. In the next section we will see an approach that is more readable but can't handle *break*'s in *switch* statements and *goto*'s. For the handling of those jump statements the existing approach is kept. The advantage of having both approaches at the same time is that the number of jump destinations is reduced and for many functions the emulation of jumps isn't needed anymore. Therefore the readability is improved.

5.1 Code transformation

The jump instructions that we remove in this chapter (*break*, *continue* and *return*) don't jump to programmer defined places in the code. They basically just skip the following instructions inside a loop or a function. This allows us to use *if* statements to implement those jumps by moving the following statements into a new *if* statement with an appropriate condition [11]. We will call those *if* statements *skip blocks*.

For this we introduce a new class *JUMP_STATE* (see listing 5.3) that tracks which jump statement was called and has features for the useage in the condition of the skip blocks. Every function that has a *break*, *continue* or *return* gets an additional local variable called *js* of type *JUMP_STATE*.

Then every *break* statement is replaced with the feature call *js.break* that sets the value of *break_called* to *True*. Analogously *continue* and *return* statements

Listing 5.2: *Eiffel*: Gcd Algorithm from listing 4.1

```
gcd (a_a: INTEGER_32; a_b: INTEGER_32): INTEGER_32
  local
    l_a: INTEGER_32
    l_b: INTEGER_32
    l_pos: INTEGER_32
  do
    from
      l_pos := 0
    until
      l_pos = -1 -- -1: return
    loop
      inspect l_pos

      -- <function body>
      when 0 then -- execution start:
        l_a := a_a
        l_b := a_b
        if l_a = 0 then
          Result := l_b
          l_pos := -1 + -1 -- return
        end
      when 1 then
        if False then
          l_pos := -1 + 4 -- skip loop
        end
      when 2 then -- loop body:
          if l_b = 0 then
            l_pos := -1 + 4 -- break
          end
      when 3 then
          if l_a > l_b then
            l_a := l_a - l_b
          else
            l_b := l_b - l_a
          end
          l_pos := -1 + 1 -- back to loop condition
      when 4 then -- loop done
        Result := l_a
        l_pos := -1 + -1 -- return
      end
      -- </function body>

      l_pos := l_pos + 1
    end
  end
```

Listing 5.3: *Eiffel*: Jump State class

```
expanded class
  JUMP_STATE

feature {ANY} -- return
  return_called: BOOLEAN
      -- "return" called

  return
    do
      return_called := True
    end

feature {ANY} -- loops
  break_called: BOOLEAN
      -- "break" called

  break
    do
      break_called := True
    end

  continue_called: BOOLEAN
      -- "continue" called

  continue
    do
      continue_called := True
    end

feature {ANY} -- queries
  loop_jumped: BOOLEAN
      -- "break", "continue" or "return" called
    do
      Result := break_called or continue_called or
          return_called
    end

  loop_left: BOOLEAN
      -- "break" or "return" called
    do
      Result := break_called or return_called
    end

feature {ANY}
  reset
    do
      break_called := False
      continue_called := False
    end

end
```

are replaced with the corresponding feature calls. The call *js.return* is omitted when the execution of the function ends anyway after the *return* statement. The other features of the class *JUMP_STATE* are:

- The feature *loop_jumped* indicates if the remaining statements of a loop should be skiped. This is the case if any of the three jump statements was executed. It is used for the conditions of the new skip blocks.
- The feature *loop_left* indicates if the execution should leave the current loop. This is the case if either *break* or *return* was called. It is used as additional condition for the loops.
- The feature *reset* resets the values of *break_called* and *continue_called* to *False*. It is used as first instruction of a loop to revert the setting of those values in the previous loop iteration. It is also used after every loop since those values are only valid within the loop they are called.

Listing 5.4 shows the transformation of the *gcd* algorithm from listing 5.1.

Nested loops

At first one would think that the value of *break_called* and *continue_called* could be changed by a nested loop and therefore one would need to have separate values for every nested loop. But whenever the execution enters a nested loop *break_called* and *continue_called* are *False*. Otherwise we would have skipped the loop in the first place.

There is one exceptional case: When the increment block of a *for* loop contains a loop. Then the *continue_called* can be *True*. But at this point we don't need the value anymore since the *continue* is already done and it doesn't influence the nested loop since the value is reset at the beginning anyway.

Also jumps with *goto* into or out of a loop are no problem. *continue_called* and *break_called* are always *False* when a *goto* is executed. This can be verified in the same way as above.

5.2 Optimization

Through the insertion of additional skip blocks the code indentation increases. This can affect readability when multiple skip blocks are nested and therefore the indentation level increases a lot. To counteract this incrementation two optimizations are implemented.

The first optimization tries to flatten the nesting of skip blocks. If a skip block is the last statement in another skip block and they have the same skip condition, it can be removed from the other skip block and inserted right after it. This reduces the indentation level of that skip block and all statements inside by one. Listings 5.5 and 5.6 illustrate this optimization.

Listing 5.4: *Eiffel*: Gcd Algorithm from listing 4.1 (second version)

```
gcd (a_a: INTEGER_32; a_b: INTEGER_32): INTEGER_32
  local
    l_a: INTEGER_32
    l_b: INTEGER_32
    js: JUMP_STATE
  do
    l_a := a_a
    l_b := a_b
    if l_a = 0 then
      Result := l_b
      js.return
    end
    if not js.return_called then
      from
      until
        js.loop_left
      loop
        js.reset
        if l_b = 0 then
          js.break
        end
        if not js.loop_jumped then
          if l_a > l_b then
            l_a := l_a - l_b
          else
            l_b := l_b - l_a
          end
        end
      end
      js.reset
      Result := l_a
    end
  end
```

Listing 5.5: *Eiffel*: Nested skip blocks

```
if not js.loop_left then
  if foo then
    js.break
  end
  if not js.loop_left then
    bar
  end
end
```

Listing 5.6: *Eiffel*: Nested skip blocks after optimization

```
if not js.loop_left then
  if foo then
    js.break
  end
end
if not js.loop_left then
  bar
end
```

Listing 5.7: *Eiffel*: Merged skip block and *if* statement

```
if not js.loop_left and then foo then
  js.break
end
if not js.loop_left then
  bar
end
```

The second optimization combines a skip block with an *if* statement inside under the condition that the *if* statement is the only statement inside the skip block and it has no *else* part. The combination is done by combining the condition of the skip block with the condition of the *if* statement with an *and then*. This ensures that the second condition is only evaluated when it would also be evaluated in the original code. This optimization also reduces the indentation level of the *if* statement and all statements inside by one. Listing 5.7 shows the result of this optimization for the code from listing 5.6.

5.3 Statistics

In the following we will see how much the transformation described in this chapter changes the code of the reference projects (see section 1.1).

At first we look at how many jump statements could be removed. The remaining jump statements are *break* statements that belong to a *switch* or *goto* statements. Table 5.1 shows how many jump statements are in the projects before and after the refactoring. In the last column is the number of added skip blocks. Over all projects more than 80 percent of the jump statements could be removed.

Functions that have no jump statement after the transformation don't need the emulation of jumps described in the beginning of this chapter. Table 5.2 lists how many functions have jump statements before and after the transformation. The number of such functions decreases clearly.

As discussed in section 5.2 the transformation changes the indentation level of the code. Table 5.3 shows the maximal and average code indentation before

Table 5.1: Removed jump statements

	jump statements				added
	before	after	removed	removed [%]	skip blocks
micro_httpd	38	1	37	97.4	22
xeyes	21	4	17	81.0	0
less	1426	376	1050	73.6	602
wget	2636	566	2070	78.5	885
links	3650	1072	2578	70.6	1372
vim	18085	5023	13062	72.2	7597
libcurl	3369	974	2395	71.1	1499
libgmp	5483	367	5116	93.3	733
libgsl	19404	1796	17608	90.7	10118
Total	54112	10179	43933	81.2	22828

Table 5.2: Functions with jump statements

	functions with jump statements			
	before	after	less	less [%]
micro_httpd	7	1	6	85.7
xeyes	17	2	15	88.2
less	375	70	305	81.3
wget	599	100	499	83.3
links	994	199	795	80.0
vim	4113	875	3238	78.7
libcurl	640	137	503	78.6
libgmp	580	91	489	84.3
libgsl	5073	515	4558	89.8
Total	12398	1990	10408	83.9

and after the transformation. It shows that the change is acceptably small and therefore doesn't have a negative influence on the readability.

The transformation also introduces additional statements. Table 5.4 shows that the number of statements increases on average by 15 percent. Since the jump statements have to be modeled in Eiffel somehow, additional statements can not be avoided.

Table 5.3: Change in code indentation

	maximal nesting		average nesting	
	before	after	before	after
micro_httpd	5	5	0.90	1.10
xeyes	5	5	0.46	0.45
less	13	13	0.75	1.01
wget	18	18	0.92	1.14
links	22	25	0.86	1.11
vim	96	96	1.32	1.59
libcurl	20	26	1.02	1.33
libgmp	17	18	1.14	1.37
libgsl	19	20	1.00	1.25

Table 5.4: Number of statements

	before	after	difference	difference [%]
micro_httpd	234	284	50	21.37
xeyes	246	249	3	1.22
less	7’019	8’263	1’244	17.72
wget	17’971	20’382	2’411	13.42
links	26’957	29’396	2’439	9.05
vim	138’274	152’477	14’203	10.27
libcurl	19’371	22’765	3’394	17.52
libgmp	30’670	35’937	5’267	17.17
libgsl	123’059	148’884	25’825	20.99
Total	363’801	418’637	54’836	15.07

Chapter 6

Conclusions

6.1 Conclusions

The C and Eiffel programming languages are quite different and therefore translating C strings or structs to the Eiffel counterpart is not very successful. The main reasons for this is the usage of pointers and external libraries in most C programs. The number of translatable strings is for most of the projects too small to justify the usage of two different strings representations. Those would raise difficulties when the code is changed later and string variables with a different representations are used together for example in an assignment.

In contrast to the translation of strings and structs the translation of *printf* statements to the Eiffel feature *Io.put_string* is possible for the common usages. The refactoring of the code to remove jump statements like *break* and *continue* always works and is a considerable improvement over the emulation of jumps like it is described in chapter 5. It enhances the readability with an acceptable increase in the code size.

6.2 Future Work

Possible future topics are the translation of C arrays to Eiffel arrays and the iteration over those from pointers to Eiffel iterators. Also the file handling could be translated to Eiffel's own file handling. But after the results of the string and struct translations the success of those is uncertain.

Appendix A

Additional translations of *printf* calls

Listing A.1: *C*: test_printf_transformer class

```c
#include <stdio.h>

/* Different printf statements to test the
    PRINTF_TRANSFORMER */
int main(int argc, char* argv[])
{
  // supported printf statements
  char c = 'A';
  int code = 66;
  printf("A character: %c\n", c);
  printf("A character from ASCII code: %c\n", code);
  printf("A constant character: %c\n", 'C');
  printf("A constant character from ASCII code: %c\n", 68);

  int i = -123;
  long int l = 1234567l;
  long long int ll= 123456789000ll;
  unsigned int ui = 123;
  unsigned short us = 123;
  int width = 6;
  printf("An integer: %d\n", i);
  printf("An integer with width: %5i\n", i);
  printf("An integer with variable width: %*i\n", width, i);
  printf("A long integer: %d\n", l);
  printf("A long long integer: %d\n", ll); // will be
      truncated
  printf("An integer from unsigned int: %i\n", ui);
  printf("An integer from unsigned short: %i\n", us);
  printf("A constant integer: %d\n", 1234);

  float f = 123.45678; // will be truncated
  double d = 123.45678;
  int small_width = 1;
  int big_width = 12;
  int precision = 3;
  printf("A float: %f\n", f);
  printf("A float with precision: %.1f\n", f);
  printf("A float with variable precision: %.*f\n", precision, f);
  printf("A float with (to) small width: %5f\n", f);
  printf("A float with variable width: %*f\n", big_width, f);
  printf("A double with width and precision: %1.3f\n", d);
  printf("A double with variable width and precision: %*.*f\n", small_width, precision, d);
  printf("A double with variable width and fix precision: %*.3f\n", big_width, d);
  printf("A double with fix width and variable precision: %12.*f\n", precision, d);
  printf("A constant float: %f\n", 1.234);

  char* s = "A variable String";
  printf("A string: %s\n", s);
  printf("A constant string: %s\n", "Hello World");

  printf("An unsigned integer: %u\n", ui);
  printf("An unsigned integer from int: %u\n", i); // to test conversion

  return 0;
}
```

Listing A.2: Output from the code in listing A.1

```
A character: A
A character from ASCII code: B
A constant character: C
A constant character from ASCII code: D
An integer: -123
An integer with width:  -123
An integer with variable width:   -123
A long integer: 1234567
A long long integer: -1097262584
An integer from unsigned int: 123
An integer from unsigned short: 123
A constant integer: 1234
A float: 123.456779
A float with precision: 123.5
A float with variable precision: 123.457
A float with (to) small width: 123.456779
A float with variable width:   123.456779
A double with width and precision: 123.457
A double with variable width and precision: 123.457
A double with variable width and fix precision:      123.457
A double with fix width and variable precision:      123.457
A constant float: 1.234000
```

```
A string: A variable String
A constant string: Hello World
An unsigned integer: 123
An unsigned integer from int: 4294967173
```

Listing A.3: *Eiffel*: Translated test_printf_transformer class

```
class
  TEST_PRINTF_TRANSFORMER

inherit
  TEST_PRINTF_TRANSFORMER_DATA

feature {ANY} -- routines

  main (argc: INTEGER_32; argv: CE_POINTER [CE_POINTER [
      INTEGER_8]]): INTEGER_32
    local
      c: INTEGER_8
      code: INTEGER_32
      i: INTEGER_32
      l: INTEGER_32
      ll: INTEGER_64
      ui: NATURAL_32
      us: NATURAL_16
      width: INTEGER_32
      f: REAL_32
      d: REAL_64
      small_width: INTEGER_32
      big_width: INTEGER_32
      precision: INTEGER_32
      s: CE_POINTER [INTEGER_8]
    do
      create s.make

      c := (('A').code).to_integer_8
      code := 66
      Io.put_string ("A character: " + (c).to_character_8.
          out + "%N")
      Io.put_string ("A character from ASCII code: " + (code
          ).to_character_8.out + "%N")
      Io.put_string ("A constant character: C%N")
      Io.put_string ("A constant character from ASCII code:
          D%N")
      i := -123
      l := 1234567
      ll := {INTEGER_64} 123456789000
      ui := {NATURAL_32} 123
      us := (123).to_natural_16
      width := 6
      Io.put_string ("An integer: " + i.out + "%N")
      Io.put_string ("An integer with width: " + (create {
          FORMAT_INTEGER}.make (5)).formatted (i) + "%N")
      Io.put_string ("An integer with variable width: " + (
          create {FORMAT_INTEGER}.make (width)).formatted (i)
```

```
           + "%N")
      Io.put_string ("A long integer: " + l.out + "%N")
      Io.put_string ("A long long integer: " + ll.
          to_integer_32.out + "%N")
      Io.put_string ("An integer from unsigned int: " + ui.
          to_integer_32.out + "%N")
      Io.put_string ("An integer from unsigned short: " + us
          .out + "%N")
      Io.put_string ("A constant integer: 1234%N")
      f := (123.45677999999999).truncated_to_real
      d := 123.45677999999999
      small_width := 1
      big_width := 12
      precision := 3
      Io.put_string ("A float: " + (create {FORMAT_DOUBLE}.
          make(6, 6)).formatted (f) + "%N")
      Io.put_string ("A float with precision: " + (create {
          FORMAT_DOUBLE}.make(1, 1)).formatted (f) + "%N")
      Io.put_string ("A float with variable precision: " + (
          create {FORMAT_DOUBLE}.make((1).max(precision),
          precision)).formatted (f) + "%N")
      Io.put_string ("A float with (to) small width: " + (
          create {FORMAT_DOUBLE}.make(6, 6)).formatted (f) +
          "%N")
      Io.put_string ("A float with variable width: " + (
          create {FORMAT_DOUBLE}.make(big_width.max(6), 6)).
          formatted (f) + "%N")
      Io.put_string ("A double with width and precision: " +
           (create {FORMAT_DOUBLE}.make(3, 3)).formatted (d)
          + "%N")
      Io.put_string ("A double with variable width and
          precision: " + (create {FORMAT_DOUBLE}.make(
          small_width.max(precision), precision)).formatted (
          d) + "%N")
      Io.put_string ("A double with variable width and fix
          precision: " + (create {FORMAT_DOUBLE}.make(
          big_width.max(3), 3)).formatted (d) + "%N")
      Io.put_string ("A double with fix width and variable
          precision: " + (create {FORMAT_DOUBLE}.make((12).
          max(precision), precision)).formatted (d) + "%N")
      Io.put_string ("A constant float: " + (create {
          FORMAT_DOUBLE}.make(6, 6)).formatted (1.234) + "%N"
          )
      s.ce_assign (ce_string ("A variable String"))
      Io.put_string ("A string: " + eif_string (s) + "%N")
      Io.put_string ("A constant string: Hello World%N")
      Io.put_string ("An unsigned integer: " + ui.out + "%N"
          )
      Io.put_string ("An unsigned integer from int: " + i.
          to_natural_32.out + "%N")
      Result := (0)
    end
end
```

List of Tables

List of Figures

Listings

Bibliography

[1] C2eiffel. http://c2eiffel.origo.ethz.ch.

[2] Less (version 382), 2 2004. http://www.gnu.org/software/less.

[3] Micro_httpd, 12 2005. http://www.acme.com/software/micro_httpd.

[4] Xeyes (version 1.0.1), 1 2006. http://xorg.freedesktop.org/.

[5] Links (version 1.0), 12 2007. http://www.jikos.cz/~mikulas/links.

[6] Wget (version 1.12), 9 2009. http://www.gnu.org/software/wget.

[7] Curl (version 7.21.2), 10 2010. http://curl.haxx.se.

[8] Gmp (version 5.0.1), 2 2010. http://gmplib.org.

[9] Gsl (version 1.14), 3 2010. http://www.gnu.org/software/gsl.

[10] Vim (version 7.3), 8 2010. http://www.vim.org.

[11] Marco Trudel. Java sourcecode to eiffel sourcecode compiler. Master's thesis, ETH Zurich, 2008. Section 4.28 http://jaftec.origo.ethz.ch.