The Computer Scientist Nightmare

My Favorite Bug

Manuel Serrano(✉)

Inria Sophia Méditerranée, 2004 route des Lucioles - BP 93,
06902 Sophia Antipolis, Cedex, France
Manuel.Serrano@inria.fr

Hop has recently been used by a small French company, which we will call AIMM in the rest of this paper, to implement a new widget for a web multimedia application. This widget contained two parts: a music selector that lets end users browse a database of artists and music, and an HTML5 music player supporting on-demand conversion from one music format to another.

Figure 1 presents a screenshot of the widget. The music player is on the top, the database browser on the bottom. Clicking an artist name pushes a new panel presenting the songs that artist has produced. Clicking the small black arrow to the left of the name restores the previous panel. Graphical effects improve user experience. Depending on the speed of the platform and the web browser used, a new panel *slides* from left to right or *blends* into the previous one.

More than a real application, this was an endeavor, or an evaluation in order to understand how Hop fits in the context of realistic multimedia web application. The development was *a priori* easy because all the elements needed to implement the widget were provided by the Hop development kit off the shelf. I was in charge of the development that I estimated would take only a handful of days. With the AIMM engineers, we agreed on a common API that Hop could use to access the actual database; then I started to develop the widget. After a couple of days, everything went as we wished. The prototype was operational. Early tests showed that it was reliable enough to enter the second stage: to be tested on the AIMM server.

The AIMM server was a classical Linux Debian hosted by an x86/64 processor. A usual setting, almost the same as the one I used for developing the widget. After having installed additional Linux packages required by Hop, I installed the development kit and the prototype. Then I started to test the widget. At first, it seemed to be doing perfectly well. Everything seemed to be working as it should, until I realized that, on some browsers, clicking the artist names on the selector produced no effect. The new panel never showed up. Plagued with this erroneous behavior, the application was utterly useless. So began the battle I fought to understand and eliminate that bug.

1 A Multitier Bug

Mainly Firefox appeared to be affected by the disease. Other browsers, such as Chrome, Midori, or Opera, were doing well. At that early moment the logical

S. Lindley et al. (Eds.): Wadler Festschrift, LNCS 9600, pp. 356–366, 2016.
DOI: 10.1007/978-3-319-30936-1_19

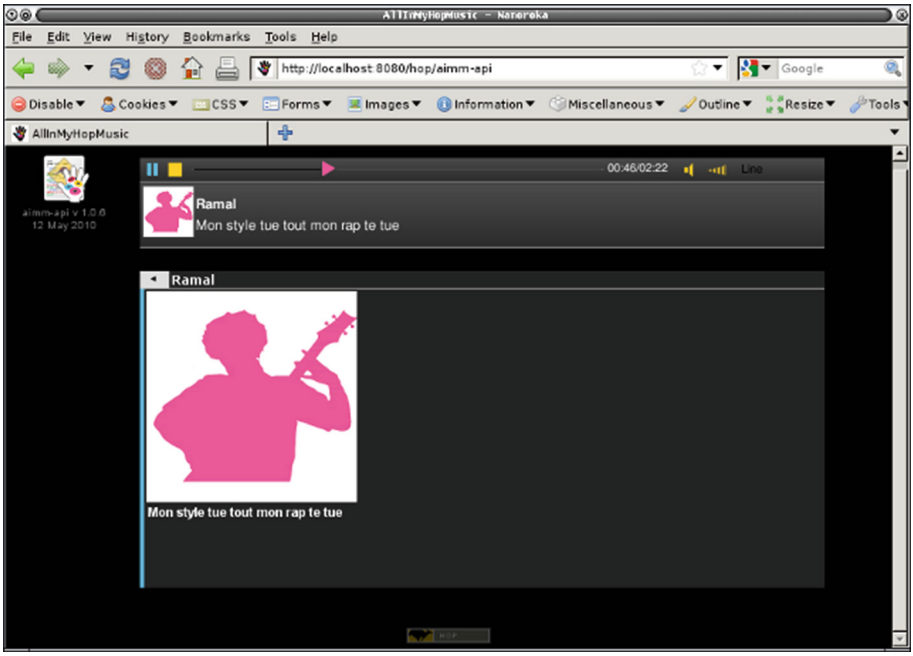

Fig. 1. The Hop web widget

suspicion was that something specific to Firefox prevented it from executing the application correctly. However, this first intuition was contradicted by an obvious observation: the same Firefox was doing well when the web page was served by a server running on my machine, that is, when the application was executed entirely locally. This was particularly shocking because the client-side code served by the two servers, the AIMM server and my local machine, were supposed to be identical. My first investigation drove me to inspect the implementation of the *client-side* part of the application in order to spot which Firefox specificity was breaking its execution.

Hop is a multitier language. A single formalism is used to program the server-side and the client-side of web applications. The web page *visualized* on a browser is first *elaborated* on a server as an abstract syntax tree that is eventually compiled on-the-fly into Html. Both ends of the application use the same *Document Object Model* to reify Html trees as first class values. In Hop, Html tags are standard functions. Hop extends the official set of Html tags with numerous new widgets that are implemented by composing elementary objects. The artist selector was implemented using one of these objects: the `SPAGE` widget.

A `SPAGE`, which stands for *sliding page*, is an Html container (*i.e.,* a box) augmented with a title and a stack of inner elements. Only the top-most element is visible at a time. A visual effect gives the feeling that new elements *slide* over

the previous ones. The source code using the SPAGE in the AIMM application was similar to:

```
(<SPAGE> :title "languages" :id "lang"
   (<DIV>
      (<SPAGE> :title "functional" :parent "lang"
         (<UL>
            (<LI> "Lisp")
            (<LI> "ML")
            (<LI> "Haskell"))))
   (<DIV>
      (<SPAGE> :title "sequential" :parent "lang"
         (<UL>
            (<LI> "Fortran")
            (<LI> "Pascal")
            (<LI> "C")))))
```

The functions <DIV>, <UL>, and <LI> are the Hop functions for the eponymous Html tags[1]. The function <SPAGE> is the constructor for the *sliding page*. In this example, the first page displays the words functional and sequential. When one is clicked, the corresponding page *slides* onto the screen and the associated enumeration is displayed. The default graphical configuration of SPAGE is defined by the following CSS rule set:

```
spage {
  background: white;
  border: 1px solid darkorange;
  font-size: 12px;
  border-radius: 0.2em;
  effect: auto;
}
```

The AIMM widget overrides these rules to use a gray background gradient and white texts, as shown in Fig. 1. The effect field lets applications choose the graphical effect they desire when a new element is pushed or popped. The four possible values are "fade", "slide", "auto", or "none". It defaults to "auto", which tells Hop to select the most suitable effect according to the characteristic of the browser. The CSS declaration is used by the client-side function spage-effect which implements the graphical effect on the browser. It is defined as:

[1] Note that in Hop, identifiers may contain the special characters "<", ">", "!", or "+". Hence, "<DIV>" is a regular identifier as is "string->integer", "remq!", or even "+".

```
(define (spage-effect node width step)
   (let ((e (node-computed-style-get node :effect)))
      (cond
         ((eq? e 'fade)
          (spage-fade node (if (> step 0) -0.3 0.2)))
         ((eq? e 'slide)
          (spage-slide node width step))
         ((eq? e 'none)
          (spage-none node (- step)))
         ((< (hop-config :browser-speed) 80)
          (spage-fade node (if (> step 0) -0.3 0.2)))
         (else
          (spage-slide node width step)))))
```

When the `effect` is "`auto`", it chooses the *best* effect according to the browser graphical animation speed. If the browser graphical animation is slow, it *fades* elements. Otherwise, it *slides* them. At the time that problem occurred, browsers based on Gecko (Firefox) were not able to animate smoothly the visual effect, while browsers based on Webkit (Google-Chrome, Midori, Safari) could do it. So, the configuration `effect: auto` of the `SPAGE` widget yield two different visual effects on the two browser families. This explained why Firefox and Google-Chrome behaved differently: they were simply not using the same code for the animation. A brief additional examination of the client-side code unveiled that the very call to `spage-fade` was the reason for the problem.

The central piece of the Hop development kit is the web broker. It is a full-fledged web server which embeds the Hop compilers. When a program runs on the broker, *i.e.,* the server-side of the application, it generates the program which is installed on the web browser, *i.e.,* the client-side of the application. This client-side part is compiled on-the-fly into JavaScript, the universal language of web browsers, by the Hop broker. As it turned out, the call to `spage-fade` was erroneously compiled into JavaScript. Instead of:

```
SpageFade( node, step > 0 ? -0.3 : 0.2 );
```

it was compiled into:

```
SpageFade( node, step > 0 ? -0.0 : 0.0 );
```

These zeroes were of course the reason for the problem: the step of the fading being 0, the element stayed invisible forever. So, the problem was not due to Firefox executing incorrectly the code it received but to Hop delivering this incorrect code! Firefox being innocent, the question remained: *what was so special with the compilation of the* `SpageFade` *call that drove Hop to generate these wrong zeroes?*

2 Read, Compile, Print

Although Hop and JavaScript share many similarities (they are both fully polymorphic functional languages, using dynamic type checking, and automatic memory management), the compilation from Hop to JavaScript is not straightforward. First, JavaScript is not fully lexically scoped. Hence, a closure analysis is needed to compile Hop closures into JavaScript closures. Second, JavaScript is not properly tail-recursive, so static analyses are needed to efficiently compile Hop tail recursion into JavaScript loops. Third, because on the web the client-side code first traverses the network, its size matters. The client-side Hop compiler uses fancy features to generate code as compact as possible. All in all, this makes the implementation of the JavaScript compiler about 17KLOC of Hop code (the system is fully bootstrapped). The reason for the erroneous generation of "`0.0`" was likely to be located somewhere in these lines.

After thorough investigations I ended up with the conclusion that the Hop reader, that is the *server-side function in charge of reading Hop files*, was at some point broken. At the beginning of the application, the reader was correct and then, after a while, it was returning 0 for all floating point numbers. The compiler was then probably not broken *per se*, only the reader was wrong.

The Hop syntax is defined by a regular language. It can then be parsed efficiently by a finite state automaton such as those generated by the Hop form "`regular-grammar`". The actual implementation of the Hop parser looks like:

```
(regular-grammar ((float (or (: (* digit) "." (+ digit))
                             (: (+ digit) "." (* digit))))
                  ...)
   ...
   ((: (* digit)
       (or letter special)
       (* (or letter special digit (in ))))
    (the-symbol))
   ((: (? (in "-+"))
       (or float
           (: (or float (+ digit))
            (in "eE") (? (in "+-")) (+ digit))))
    (the-flonum))
   ...)
```

The first part defines variables (`float` in the code snippet above) which bind regular expressions used in the *rules*. These rules bind regular expressions, defined in a infix syntax, to expressions that are executed when a pattern matches. Regular grammars are compiled on the fly into Hop procedures. The implementation of regular grammars is about 3.5KLOC of Hop code for the compiler, seconded by 1.1KLOC of native code for the IO layer.

Compiling the Hop reader produces a function such as:

```
(lambda (ip)
   (define (the-flonum::double)
      (rgc_buffer_flonum ip))
   (define (the-fixnum::long)
      (rgc_buffer_fixnum ip))
   (define (the-symbol::symbol) ...)
   ...
   (define (STATE0 ip) ...)
   (case (STATE0 ip)
      ...
      ((10) (the-flonum))
      ...))
```

The function `rgc_buffer_flonum` extracts from the input buffer the corresponding floating point number. It is part of the native runtime system. The Hop native compiler (73KLOC of Hop code) can produce either native code, JVM bytecode, or .NET bytecode. The function `rgc_buffer_flonum` thus exists for the three backends. C was the backend used for the AIMM experiment. Its implementation is:

```
#define rgc_buffer_flonum( ip ) \
  strtod( RGC_INPUT_PORT_BUFFER( ip ), 0 );
```

As the reader at some point was no longer able to read numbers correctly, either `RGC_INPUT_PORT_BUFFER` or the libc function `strtod` had to be wrong! This was very shocking. I could not believe that Hop IO buffers were wrong. The overall size of the Hop implementation is about 325KLOC of Hop code which are all read by the Hop reader. If the buffers were not correctly managed, the bootstrap which reads all the 12,371,080 characters composing the source code could not reasonably succeed! I could not either believe that `strtod` was wrong. The Gnu libc is too widely used and too many applications depend on it for such an obvious error to occur. However, the Linux version running on the AIMM server being rather old, I spent half an hour googling to check if that particular version of the library was known to have a wrong `strtod` implementation. It was not.

3 The Usual Suspects

At this point of the paper, it is probably useful to make a short point. I was chasing a bug that *(i)* only appeared with Firefox but Firefox was innocent; *(ii) appeared* in the client-side of the application but it was *located* in the server-side of the application; *(iii)* was due to an error in the server-side code in charge of producing the client-side; *(iv)* was due to server-side code that seemed to first behave correctly and then, at some point of the execution, started to behave badly; *(v)* was not observable when executed on my machine but systematic when

executed on the AIMM server. The obvious conclusion of all these observations was that something specific to the AIMM server was responsible for the wrong behavior.

When portability across a single operating system seems broken, the first idea that usually comes to mind is to check if there could be a potential confusion between pointers size. In our case we had an ideal suspect: my personal machine was using 32 bit pointers while the AIMM server was using 64 bit pointers! This suspicion was even strengthened by my previous investigation. As said before, I was more or less suspecting an IO buffer overflow and a confusion between 32 bit pointers and 64 bit pointers is very prone to buffer overflows. A quick test conducted on another x86/64 Linux machine of our own showed that Hop and the AIMM widget were running like a charm on that other 64 bit pointers computer. The problem was elsewhere.

The Hop broker (75KLOC) is deeply multi-threaded. When started, it spawns several preemptive threads (usually 20) that wait for connections. Each client request is handled in a separate thread. So, several client-side compilations may occur simultaneously. Multi-threaded applications are known to be a nightmare to program and an even worse nightmare to debug, in particular because multi-threading may break sequentially correct code. The C standard library is plagued with several non-thread-safe functions. In particular, all those that use static buffers, such as `strtok`. Would it be possible that `strtod` was also using a static buffer that made it non *thread-safe*? An excerpt of the Linux man page is given Fig. 2. It tells nothing about multi-threading and it proposes no thread-safe alternative. So, at least with the Gnu libc, it is very likely that `strtod` is thread safe. Another observation mostly invalidates the theory of a potential multi-threading problem. The bug was far too reproducible! The charm of multi-threading errors is their non-deterministic behavior. Here, the bug always occurred, at the same moment, on the same machine. Not something that could have happened with an error in the implementation of the parallelism.

After all, since the bug was reproducible, it was an ideal candidate for a debugger. I recompiled everything in debug mode and tried to learn more. No success. Firstly, the debugger was not easy to use because the error did not show up until the function had been called several thousand times. Secondly, it merely validated some already known facts: yes, the problem also showed up when the application was running inside the debugger; yes, after a while `the-flonum` was returning 0. Nothing else to learn. In addition to the debugger, I tried the read-eval-print loop Hop can spawn. Same result.

Since the debugger and the interactive loop were mostly useless, I instrumented the application to log all the calls to `strtod` and I wrote another program for replaying the logs. Surprisingly, this new program ran well, as all the calls to `strtod` completed correctly. So, if `strtod` was broken, it was broken in a weird context-dependent way. This experiment taught me another important fact: by observing the log player running correctly, I realized that many Hop instances were running *well* on the AIMM server. Something specific to the AIMM widget made it run incorrectly on the AIMM server!

```
STRTOD(3)                  Linux Programmer's Manual                  STRTOD(3)


NAME
       strtod, strtof, strtold - convert ASCII string to floating-point number

SYNOPSIS
       #include <stdlib.h>

       double strtod(const char *nptr, char **endptr);
       ...

DESCRIPTION
       The strtod(), strtof(), and strtold() functions convert the initial
       portion of the string pointed to by nptr to double, float, and long
       double representation, respectively.

       The expected form of the (initial portion of the) string is optional
       leading white space as recognized by isspace(3), an optional plus ('+')
       or minus sign ('-') and then either (i) a decimal number, or (ii) a
       hexadecimal number, or (iii) an infinity, or (iv) a NAN (not-a-number).

       A decimal number consists of a nonempty sequence of decimal digits pos-
       sibly containing a radix character (decimal point, locale-dependent,
       usually '.'), optionally followed by a decimal exponent. A decimal
       exponent consists of an 'E' or 'e', followed by an optional plus or
       minus sign, followed by a nonempty sequence of decimal digits, and
       indicates multiplication by a power of 10.

       ...

CONFORMING TO
       C89 describes strtod(), C99 describes the other two functions.
```

Fig. 2. The Linux man page for strtod

4 The Solution

At a moment, for no particular reason, by luck, or because I was unconsciously examining all the possible directions, the small detail that I had in front of my eyes since the beginning and that I had neglected so far attracted my attention: if `strtod` was wrong by returning 0.0 how come it returned −0.0 and not just 0.0 for "`-0.3`"? Where did this "`-`" come from? After all, it seemed that something was correct in the result since the sign was correct! A further experiment showed that actually the result was only wrong for the decimal part of the numbers. That is "`0.2`" was read as 0.0 but "`1.2`" was read as 1.0. After this observation everything went fast and easy.

As stated in the Linux man page (see above), `strtod` is locale-dependent. In French, the radix character is "`,`", contrary to English that uses "`.`".

Hence, parsing `-0.3` in English *must* return -0.3 while parsing it in French *must* return -0.0, because the french parsing of the number stops after parsing the characters `-` and `0`, ignoring the rest of the string. My computer and the AIMM server were both located in France but my machine was using an English setting while the AIMM server was using a French setting. On the AIMM server, the Linux global environment variable `LOCALE` was set to "`fr_FR`". This was the reason for the bug. The only problem yet to be solved was to understand why the behavior of `strtod` changed during the execution. Why did it start with an English setting and at some point switch to French?

The Hop server starts in a bare minimal core image. When an application is loaded it dynamically loads the libraries it depends on. Many multimedia Hop applications depend on the widely used Gstreamer library to manipulate multimedia material. In the AIMM application, it was used to encode MP3 into OGG on the fly (while Google-Chrome supports MP3, Firefox only supports OGG). So, when the AIMM application started it loaded Gstreamer.

The Gstreamer documentation specifies that an application should first call `gst_init` to initialize the library but it says nothing about the locale. However inspecting the source code, and in particular, the file gst/gst.c, I spotted the following:

```
static gboolean
init_pre (GOptionContext * context, GOptionGroup * group,
          gpointer data, GError ** error)
{
  ...
#ifdef ENABLE_NLS
  setlocale (LC_ALL, "");
  bindtextdomain (GETTEXT_PACKAGE, LOCALEDIR);
  bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
#endif /* ENABLE_NLS */
 ...
}
```

This clearly meant that Gstreamer allowed itself to change the locale of the application, with the side effect of changing the behavior of `strtod`, and in that case, breaking the Hop client-side compiler. The fix was straightforward and it took exactly two lines of code. It merely required saving the current locale before calling the `gst_init` function and to restore it afterward as in:

```
char *locale = setlocale( LC_ALL, 0 );
gst_init( &argc, &argv );
setlocale( LC_ALL, locale );
```

5 Concluding Remarks

This article is *unacademic* on purpose. It is written in a first-person narrative mode. it contains no citations, no mathematical formulae of any kind, and no algorithms, but it does contain code! Readers will eventually judge, but I think that in spite of containing no theorem, it demonstrates a flaw in the *principles* we use to design our languages and to write our programs because the bug that is described in this paper could probably occur in many different contexts and with most programming languages. *How many readers of this paper can be sure that none of their programs are affected by a similar problem?*

This paper tells a story of a program that apparently broke none of our commonly accepted rules but that still went wrong. The program suffered no memory leak. It was type safe, with respect to the common understanding of what a type should be. It suffered no data race. But still it went wrong!

Who is to blame for the error then? This is a legitimate question. Someone has to be blamed! Obviously I have my share of responsibility. It might be argued that the Hop implementation of `the-flonum` is incorrect and this is the reason for the whole problem. It definitively is and the function should be re-written. My knowledge of the C function `strtod` was too imprecise and it had yielded to the problem described in the paper. I'm the main culprit. However, it is inconceivable to re-implement all the low level functions used by Hop. It is likely that some other functions rely on a hidden state that might be changed externally, in particular amongst the functions used to implement the network and DNS APIs. However, re-writing these functions would take an unreasonable amount of work and independently of the feasibility of the task it is also worth wondering if this approach makes any sense. The purpose of Hop is to provide an infrastructure to help re-using code written elsewhere by other people, maybe using other formalisms. Re-writing all the core APIs in Hop would then contradict the overall goal of the whole system.

If I'm not the only one to blame, who else? Clearly the C functions `strtod` and `atof` have a peculiar design that is partly responsible for the problem. First, relying on external information, the locale, they make programs that use them unsealed. Second, using a weak definition for the external representation of numbers, they are prone to incorrectly parse numbers. Even worse, they offer no means to isolate the code relying on them. The locale used by this function is *always* a kind of oracle that cannot be controlled by the program.

In my opinion, the Gstreamer library also shares a part of the responsibility. First, it might be argued that no API should ever change the global context in which programs that use them execute. Second, if such a modification cannot be avoided, then it should be stated very explicitly in the documentation. In that particular case, Gstreamer seemed to call `setlocale` for no particularly good reason (parsing ID3 tags potentially expressed in the locale of the host is a weak motivation). Second, as reported, the documentation of the `gst_init` function does not even mention the change of the locale. This might be the biggest fault.

What tool could have helped to prevent the error or to detect it? Usual debuggers were of no help because, first, the execution was correct although with an

unintended behavior (after all, `strtod` was configured to return -0.0), second, the lines of code that were incorrect were not implemented in the main language but in a hidden implementation language that the system tries hard to make invisible to the end programmer.

It is frequently argued that strong static type checking is a powerful tool that helps detecting errors soon. In this particular case it would have been of no help. With respect to the type system found in general purpose languages, everything in the execution was correct. Maybe this tells us that the types we are used to are inadequate? Maybe this tells that we should rely on types that denote concrete entities and not abstract mathematical things? Maybe the type of `strtod` should not be a function that takes a *string of characters* and that returns a real but a function that takes *a floating point number represented as a string of characters* and that returns a real? Maybe the type of `gst_init` should not only be *a function that accepts an integer and an array of strings and that returns an integer* but also something that denotes *the effect on the current locale*?

I think the real reason for the error presented in this paper is that the languages we have designed so far, and maybe even the way we think a program should be written, do not allow programs to compose safely. The error presented in this article came from an apparently innocuous function of a widely used library that was permitted to modify the whole behavior of the application. As long as the systems we design will be liable to such behaviors this kind of error will keep occurring. I think this story shows the limit of the current direction we are following in designing our languages and programs. I also think that it shows that we should explore different paths where safe composition is the starting point of the design. In the era we are to enter where programs need to use heterogeneous sets of resources and data, this will be vital for the reliability of our yet to be invented brave new applications.