Stringrefs

Reference-typed strings in WebAssembly

26 April 2022 – Wasm CG

Andy Wingo | `wingo@igalia.com`

`https://github.com/wingo/stringrefs/`

Agenda

Motivation

Overview: Goals, Requirements, Design, Proposal

Open questions / feedback

`https://github.com/wingo/stringrefs`

Motivation

Three examples of suboptimality

- C++ on the web: double copies, memory capability
- Java on web: DOM access expensive, code duplication
- Component model: from single copy to zero copy

C++ on the web

Or Rust, or C, or CPython...

https://github.com/emscripten-core/emscripten/blob/main/src/preamble.js

```
function ccall(ident, returnType, argTypes, args, opts) {
  // For fast lookup of conversion functions
  var toC = {
    'string': function(str) {
      var ret = 0;
      if (str !== null && str !== undefined && str !== 0) { // null string
        // at most 4 bytes per UTF-8 code point, +1 for the trailing '\0'
        var len = (str.length << 2) + 1;
        ret = stackAlloc(len);
        stringToUTF8(str, ret, len);
      }
      return ret;
    },
    'array': function(arr) {
```

https://github.com/emscripten-core/emscripten/blob/main/src/preamble.js#L100

C++ on the web

Double-copy (first to stack then to where you need it)

`NUL` termination (have to scan again for length)

Can't represent `NUL` codepoints

Requires read/write capability on whole memory

Requires that users wrangle `malloc`

Requires JS

Similar problems in other direction

C++ on the web

Even experts get it wrong sometimes

https://github.com/emscripten-core/emscripten/blob/main/src/runtime_strings.js

```
    var u = str.charCodeAt(i); // possibly a lead surrogate
    if (u >= 0xD800 && u <= 0xDFFF) {
      var u1 = str.charCodeAt(++i);
      u = 0x10000 + ((u & 0x3FF) << 10) | (u1 & 0x3FF);
    }
```

https://github.com/emscripten-core/emscripten/issues/15324

Java on the web

JS strings are exactly what Java needs: immutable sequences of 16-bit code units

But all Java can do is GC array of u16 – GC allocation on Java/JS boundary

Penalizes access to DOM

Penalizes JS/Java interaction

Needlessly ships second string facility

Component model

Components are isolated

- Communication via abstractly-typed interfaces
- JIT compilation of adapters for concrete representations

Linear memory strings always copied at least once between components

Strings in GC arrays: same, if mutable, or if conversion needed

Immutable stringrefs: **zero copy**

General idea

Expose string facility from web platform to WebAssembly

Make it work for non-JS embedders too

Make it useful as a compile target – performance-positive

Humor me for a moment while we do details :)

Goals

- Enable programs compiled to WebAssembly to efficiently create and consume JavaScript strings
- Provide a good string implementation that many languages implemented on top of the GC proposal would find useful

Req’ts

- Zero-copy string passing between JS and Wasm
- No new string implementations on the web
- Allow WTF-8 or WTF-16 internal representations
- Allow WTF-16 code unit access
- Allow string literals in element sections

Design

The tension:

- Source languages: UTF-8 for Rust, WTF-16 for Java, codepoint access for Python...
- Implementations: WTF-16 for V8, UTF-8 for wasmtime...

Solve via common-denominator `stringref` plus encoding-specific `stringviews`

Proposal

`stringref` is new opaque reference-typed value, like `externref`

A `stringref` is a sequence of Unicode scalar values and isolated surrogates

Can obtain WTF-8, WTF-16, codepoint iterator “views” on a `stringref`

stringref

```
(string.new_wtf8 $memory ptr:address bytes:i32)
  -> str:stringref
(string.new_wtf16 $memory ptr:address codeunits:i32)
  -> str:stringref
(string.const contents:i32)
  -> str:stringref
(string.measure_utf8 str:stringref)
  -> bytes:i32
(string.measure_wtf8 str:stringref)
  -> bytes:i32
(string.measure_wtf16 str:stringref)
  -> bytes:i32
wtf8_policy ::= 'utf8' | 'wtf8' | 'replace'
(string.encode_wtf8 $memory $wtf8_policy
                    str:stringref ptr:address)
(string.encode_wtf16 $memory str:stringref ptr:address)
(string.concat a:stringref b:stringref) -> stringref
(string.eq a:stringref b:stringref) -> i32
(string.is_usv_sequence str:stringref)
  -> bool:i32
```

stringview_wtf8

```
(string.as_wtf8 str:stringref)
  -> view:stringview_wtf8
(stringview_wtf8.advance view:stringview_wtf8
                         pos:i32 bytes:i32)
  -> next_pos:i32
(stringview_wtf8.encode $memory $wtf8_policy
    view:stringview_wtf8 ptr:address pos:i32 bytes:i32)
  -> next_pos:i32, bytes:i32
(stringview_wtf8.slice view:stringview_wtf8
                       start:i32 end:i32)
  -> str:stringref
```

stringview_wtf16

```
(string.as_wtf16 str:stringref)
  -> view:stringview_wtf16
(stringview_wtf16.length view:stringview_wtf16)
  -> length:i32
(stringview_wtf16.get_codeunit view:stringview_wtf16
                               pos:i32)
  -> codeunit:i32
(stringview_wtf16.encode $memory view:stringview_wtf16
                         ptr:address pos:i32 len:i32)
(stringview_wtf16.slice view:stringview_wtf16
                        start:i32 end:i32)
  -> str:stringref
```

stringview_iter

```
(string.as_iter str:stringref)
  -> view:stringview_iter
(stringview_iter.cur view:stringview_iter)
  -> codepoint:i32
(stringview_iter.advance view:stringview_iter
                         codepoints:i32)
  -> codepoints:i32
(stringview_iter.rewind view:stringview_iter
                        codepoints:i32)
  -> codepoints:i32
(stringview_iter.slice view:stringview_iter
                       codepoints:i32)
  -> str:stringref
```

Perf

Rust, C/C++, CPython

❧ Constant factors via toolchain, big wins with code changes

Java, Dart, Kotlin

❧ Big wins for access to e.g. JS RegExp

Component model

❧ Significant for GC, minimal for linear-memory

Verify with prototype

Secondary benefits

Interoperation MVP

Less poking at memories

Smaller binaries

Less duplication

Less platform fragmentation (modules, WASI vs web)

Relation to GC

Not dependent on GC MVP

Same family though

Best “like `externref`” formulation is in terms of `heaptype`, from typed function references

Will want `array u16`, `array u8` read/write

That’s the proposal!

Now let’s criticise it

Y u spoil my ISA

How to choose what goes where? Core, libraries, components, Web, WASI, toolchains, compilers, run-times…

Goal: overall system perf & simplicity

Often by going low-level (`i64.add` vs JavaScript add)

Sometimes by abstracting: call stack, `struct.new`, `table.init`

Abstractions assume a run-time – e.g. expose GC from JavaScript

Strings can fit here?

Alternatives

Could we imagine strings any other way?

GC arrays?

This works when targetting GC

Can be slow interoperating with host

Surprisingly thorny case: RegExp

Conjecture: using stringref faster; results expected in H2 2022

Library?

Idea: `externref`, imported functions

Maybe compile-time imports

Duplication: Host already has strings

Duplication: Avoid library per module or component

Betting the farm on adaptive optimization

Perf challenge: accessing string contents

We should try and see?

Pitch

Stringref could be a nice addition to the WebAssembly platform

Could improve performance – will know in H2

Could have nice platform network effects https://github.com/wingo/wasm-jit/blob/main/interp.py

```python
def write_string(string):
    utf8 = string.encode('utf-8')
    ptr = interplib.allocateBytes(len(utf8) + 1)
    ptr = interplib.allocateBytes(len(utf8) + 1)
    dst = interplib.memory.data_ptr(wasmtime.loader.store)
    for i in range(0,len(utf8)):
        dst[ptr + i] = utf8[i]
    dst[ptr + len(utf8)] = 0
    return ptr
```

Next steps

Phase 1 poll for general interest?

Move repo to WebAssembly org

Q2-Q3: Prototyping in V8

Q3-Q4: Toolchain: GC-targetting languages, LLVM, Binaryen

`https://github.com/wingo/stringrefs/`

`wingo@igalia.com`