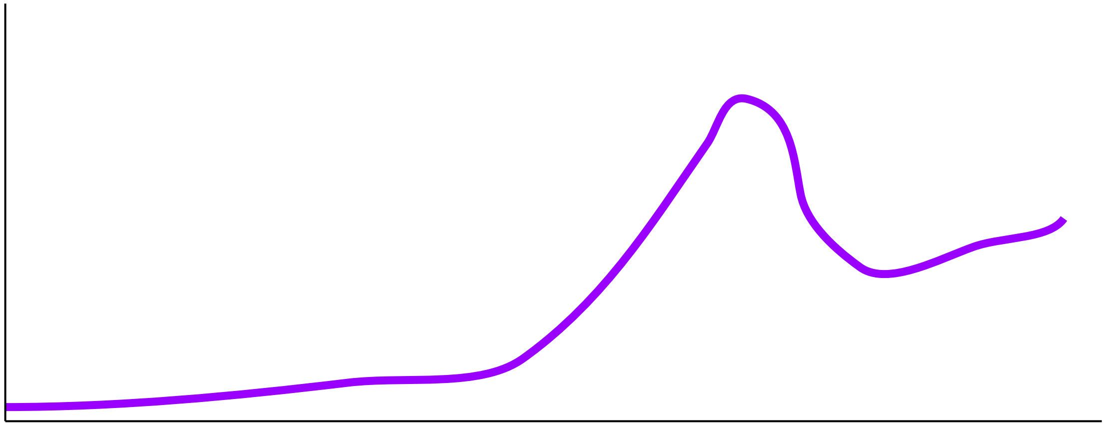
Rust & ZIG TOGETHER!

@rtfeldman

roc-lang.org

GitHub Repo Visitors



main roc / FAQ.md

Preview Code Blame 446 lines (318 loc) · 30.9 KB

Why does Roc use both Rust and Zig?

Rust

complex language

borrow checker

guarantees

ZIG

simple language

comptime

toolchain


Rust & ZIG
TOGETHER!

Outline

1. Why did we mix Rust and Zig?
2. Memory safety in practice
3. Where to draw the line?

The Storekeeper Bug

The Storekeeper Bug

```
Hello, traveler!
```

The Storekeeper Bug

Hello, traveler %Y$49fn tfr8xji

$Bbje359t3g894 845t$t 323rfal9

5i5tj4 i5j034 4$#@$ 3409jrj

f1061 0j025g4 er0-3304 4220

40w94t0wrkr094## J4o9IWJM#OI #$%T)#

Creating a Language

roc-lang.org

automatic memory management

“A **fast**, friendly, functional language.”

Building a fast compiler

Don't want to hit a language **performance ceiling**

Don't want to worry about **memory unsafety** bugs

```
Hello, traveler!@#$@%Y$49fn tfr8xji
$Bbje359t3g894  2459845t$t 323rfal9
5i5tj4 i5j034340jv 494$#@$ 3409jrj
```


no GC
concurrency
arena allocation
CPU memory cache
branch misprediction
SIMD instructions

How did it go?

Reminded me of learning Haskell

🤯 Learning curve

😵 Language complexity

💪 Confidence once I got it working

Roc “Builtins”

Basic language primitives (numbers, strings, lists, ...)

Partly implemented in Roc, partly **built into** compiler

Implemented with **manual LLVM calls** at first

Like writing **assembly** with much more **ceremony**

Rust

```rust
if len1 == len2 {
    // then-block
} else {
    // else-block
}
```

Manual LLVM Calls

```rust
let then_block =
    ctx.append_basic_block(parent, "then");
let else_block =
    ctx.append_basic_block(parent, "else");

env.builder
    .build_conditional_branch(
        env.builder.build_int_compare(
            IntPredicate::EQ, len1, len2, "=="),
        then_block, // …later, mutate then_block
        else_block  // …later, mutate else_block
);
```

array1 == array2 in Manual LLVM Calls

```rust
fn build_list_eq_help<'a, 'ctx>(
    env: &Env<'a, 'ctx, '_>,
    layout_interner: &STLayoutInterner<'a>,
    layout_ids: &mut LayoutIds<'a>,
    parent: FunctionValue<'ctx>,
    element_layout: LayoutRepr<'a>,
) {
    let ctx = env.context;
    let builder = env.builder;

    {
        use inkwell::debug_info::AsDIScope;

        let func_scope = parent.get_subprogram().unwrap();
        let lexical_block = env.dibuilder.create_lexical_block(
            /* scope */ func_scope.as_debug_info_scope(),
            /* file */ env.compile_unit.get_file(),
            /* line_no */ 0,
            /* column_no */ 0,
        );

        let loc = env.dibuilder.create_debug_location(
            ctx,
            /* line */ 0,
            /* column */ 0,
            /* current_scope */ lexical_block.as_debug_info_scope(),
            /* inlined_at */ None,
        );
        builder.set_current_debug_location(loc);
    }

    // Add args to scope
    let mut it = parent.get_param_iter();
    let list1 = it.next().unwrap().into_struct_value();
    let list2 = it.next().unwrap().into_struct_value();

    list1.set_name(Symbol::ARG_1.as_str(&env.interns));
    list2.set_name(Symbol::ARG_2.as_str(&env.interns));

    let entry = ctx.append_basic_block(parent, "entry");
    env.builder.position_at_end(entry);

    let return_true = ctx.append_basic_block(parent, "return_true");
    let return_false = ctx.append_basic_block(parent, "return_false");
```

```rust
// first, check whether the length is equal

let len1 = list_len(env.builder, list1);
let len2 = list_len(env.builder, list2);

let length_equal: IntValue =
    env.builder
        .build_int_compare(IntPredicate::EQ, len1, len2, "bounds_check");

let then_block = ctx.append_basic_block(parent, "then");

env.builder
    .build_conditional_branch(length_equal, then_block, return_false);

{
    // the length is equal; check elements pointwise
    env.builder.position_at_end(then_block);

    let builder = env.builder;
    let element_type = basic_type_from_layout(env, layout_interner, element_layout);
    let ptr_type = element_type.ptr_type(AddressSpace::default());
    let ptr1 = load_list_ptr(env.builder, list1, ptr_type);
    let ptr2 = load_list_ptr(env.builder, list2, ptr_type);

    // we know that len1 == len2
    let end = len1;

    // allocate a stack slot for the current index
    let index_alloca = builder.build_alloca(env.ptr_int(), "index");
    builder.build_store(index_alloca, env.ptr_int().const_zero());

    let loop_bb = ctx.append_basic_block(parent, "loop");
    let body_bb = ctx.append_basic_block(parent, "body");
    let increment_bb = ctx.append_basic_block(parent, "increment");

    // the "top" of the loop
    builder.build_unconditional_branch(loop_bb);
    builder.position_at_end(loop_bb);

    let curr_index = builder
        .new_build_load(env.ptr_int(), index_alloca, "index")
        .into_int_value();

    // #index < end
    let loop_end_cond =
        builder.build_int_compare(IntPredicate::ULT, curr_index, end, "bounds_check");

    // if we're at the end, and all elements were equal so far, return true
    // otherwise check the current elements for equality
    builder.build_conditional_branch(loop_end_cond, body_bb, return_true);
```

```rust
{
    // loop body
    builder.position_at_end(body_bb);

    let elem1 = {
        let elem_ptr = unsafe {
            builder.new_build_in_bounds_gep(element_type, ptr1, &[curr_index], "load_index")
        };
        load_roc_value(env, layout_interner, element_layout, elem_ptr, "get_elem")
    };

    let elem2 = {
        let elem_ptr = unsafe {
            builder.new_build_in_bounds_gep(element_type, ptr2, &[curr_index], "load_index")
        };
        load_roc_value(env, layout_interner, element_layout, elem_ptr, "get_elem")
    };

    let are_equal = build_eq(
        env,
        layout_interner,
        layout_ids,
        elem1,
        elem2,
        element_layout,
        element_layout,
    )
    .into_int_value();

    // if the elements are equal, increment the index and check the next element
    // otherwise, return false
    builder.build_conditional_branch(are_equal, increment_bb, return_false);
}

{
    env.builder.position_at_end(increment_bb);

    // constant 1isize
    let one = env.ptr_int().const_int(1, false);

    let next_index = builder.build_int_add(curr_index, one, "nextindex");

    builder.build_store(index_alloca, next_index);

    // jump back to the top of the loop
    builder.build_unconditional_branch(loop_bb);
}
```

Goal: Get **Higher-Level**

Some languages can compile to **LLVM bitcode**

LLVM bitcode can mix with **Roc compiler output**

This includes **C**, **C++**, **Zig**, and **Rust**

```
if len1 == len2 {
    // then-block
} else {
    // else-block
}
```

Obvious First Choice: Rust!

`unsafe` needed all over the place

unsafe FFI means "you're on your own"

Rust's generated LLVM caused problems

Tooling and development build difficulties

Why not C?

The Storekeeper Bug Revisited

```
Hello, traveler!@#$@%Y$49fn tfr8xji
$Bbje359t3g894  2459845t$t 323rfal9
5i5tj4 i5j034340jv 494$#@$ 3409jrj
f1061 0j025g4rwKFK34k er0-3304 4220
40w94t0wrkr094## J4o9IWJM#OI #$%T)#
```

The Storekeeper Bug Revisited

Hello, traveler!@#$@%Y$49fn tfr8xji

$Bbje359t3g894 2459845t$t 323rfal9

5i5tj4 i5j034340jv 494$#@$ 3409jrj

f1061 0j025g4rwKFK34k er0-3304 4220

40w94t0wrkr094## J4o9IWJM#OI #$%T)#

Hello, traveler.

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	46	0

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	33	0

Hello, traveler!

Hello, traveler.

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	**46**	0

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	**33**	0

Hello, traveler!

```
str[str_length - 1] = '!';
```

Hello, traveler.

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	**46**	0

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	**33**	0

Hello, traveler!

```
str[str_length     ] = '!';
```

Hello, traveler.

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	**46**	0

72	101	108	108	111	44	32	116	114	97	118	101	108	101	114	46	**33**

Hello, traveler.!

```
str[str_length     ] = '!';
```

```
Hello, traveler.!@#$@%Y$49fn fr8xji
$Bbje359t3g894  2459845t$t 323rfal9
5i5tj4 i5j034340jv 494$#@$ 3409jrj
f1061 0j025g4rwKFK34k er0-3304 4220
40w94t0wrkr094## J4o9IWJM#OI%t 0
```

What if this memory had stored a **secret**?

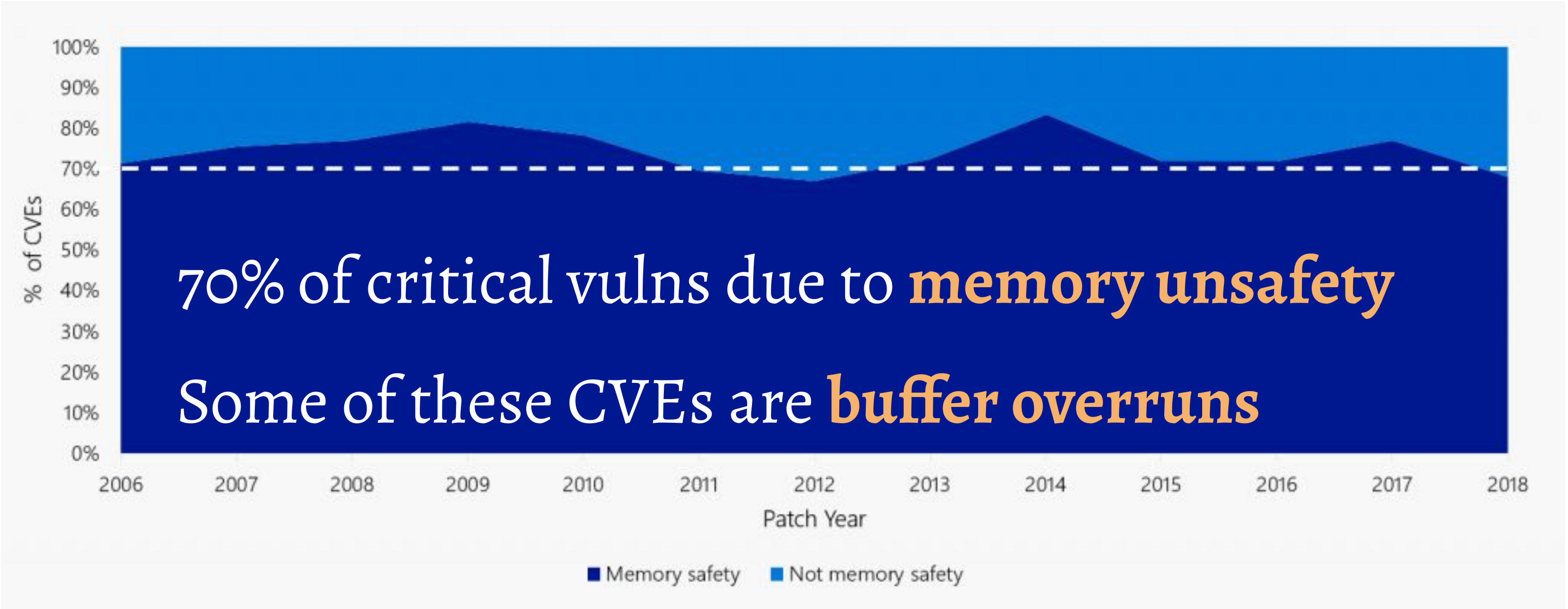
Memory Safety

```
str[str_length] = '!';
```

silently overwrite whatever's there

Error: Index out of bounds

Error: Index out of bounds



Bounds checks prevent buffer overrun vulns

There are **other types** of memory unsafety!

Memory Safety

```
array = malloc(321);

// do a bunch of stuff

free(array);

// use `array` again
```

use-after-free

Memory Safety

```
array = malloc(321);

// do a bunch of stuff

free(array);
// do other things
free(array);
```

use-after-free
double-free

Memory Safety

```
array = malloc(321);

defer free(array);

// do a bunch of stuff
```

ZIG

use-after-free
double-free

Memory Safety

```
array = malloc(321);

defer free(array);

// do a bunch of stuff
```

much less likely to have a use-after-free

double-free

Memory Safety

```
array = malloc(321);

defer free(array);

// do a bunch of stuff
```

much less likely to have a use-after-free
much less likely to have a double-free

Memory Safety

```
array = malloc(321);

// do a bunch of stuff

// (`array` gets freed)
```

Rust

cannot have a use-after-free

cannot have a double-free

Memory Safety

```
array = malloc(321);

// do a bunch of stuff

// (`array` gets freed)
```

Rust

unless you use the **unsafe** keyword

cannot have a use-after-free*

cannot have a double-free*

“I don’t understand why, in this day and age, anyone would use a **memory-unsafe language**.”

Potential Memory Unsafety	App Code	Dependencies
"memory-safe language"		
JavaScript	none	constant number
JS	FFI	every dependency
GO	FFI + "unsafe"	every dependency
asm [ObjC]	anywhere	every dependency

“I don’t understand why, in this day and age, anyone would use a **memory-unsafe language**.”

“memory-safe language” ≠ “memory unsafety cannot happen here”

“memory-unsafe language” ≠ “everything will definitely explode”

these are about **where potential memory unsafety** can be found

> "I don't understand why, in this day and age, anyone would use a **memory-unsafe language**."

If we **cannot have memory safety** in this code...

...and Rust code would need **`unsafe`** everywhere

then why not **optimize for other things**?

> “I don’t understand why, in this day and age, anyone would use a **memory-unsafe language**.”

TigerBeetle

Never deallocates

Zig has bounds checks

~~use after free~~

~~double free~~

~~buffer overrun~~

What **help** do I get?

```
str[str_length] = '!';
```

silently overwrite whatever's there

Error: Index out of bounds

Error: Index out of bounds

What **help** do I get?

Bounds Checks

`Drop` in Rust

`defer` in Zig

RAII in C++

ARC in ObjC/Swift

Tracing GC

Zig testing allocators

Address Sanitizer

UBSan

Miri in (non-FFI) Rust

Memory Safety isn't all-or-nothing

Rust's borrow checker is **useful tool**

Rust's `unsafe` is a **useful tool**

Zig's `defer` is a **useful tool**

Zig's testing allocator is a **useful tool**

These tools all have different **tradeoffs**

Why not C? (...when Zig is an option)

More prone to **memory unsafety** (e.g. no `defer`)

More **gotchas and footguns** (e.g. silent conversions)

Less **ergonomic features** (e.g. no `comptime`)

Zig community is **helpful and beginner-friendly**

Zig Tooling ❤️

vendr

vendr.com/careers

`zig cc` makes cross-compiling C code easy

At work, we use this for Node.js ↔ Roc interop

array1 == array2 in Manual LLVM Calls

```
fn build_list_eq_help<'a, 'ctx>(
    env: &Env<'a, 'ctx, '_>,
    layout_interner: &STLayoutInterner<'a>,
    layout_ids: &mut LayoutIds<'a>,
    parent: FunctionValue<'ctx>,
    element_layout: LayoutRepr<'a>,
) {
    let ctx = env.context;
    let builder = env.builder;

    {
        use inkwell::debug_info::AsDIScope;

        let func_scope = parent.get_subprogram().unwrap();
        let lexical_block = env.dibuilder.create_lexical_block(
            /* scope */ func_scope.as_debug_info_scope(),
            /* file */ env.compile_unit.get_file(),
            /* line_no */ 0,
            /* column_no */ 0,
        );

        let loc = env.dibuilder.create_debug_location(
            ctx,
            /* line */ 0,
            /* column */ 0,
            /* current_scope */ lexical_block.as_debug_info_scope(),
            /* inlined_at */ None,
        );
        builder.set_current_debug_location(loc);
    }

    // Add args to scope
    let mut it = parent.get_param_iter();
    let list1 = it.next().unwrap().into_struct_value();
    let list2 = it.next().unwrap().into_struct_value();

    list1.set_name(Symbol::ARG_1.as_str(&env.interns));
    list2.set_name(Symbol::ARG_2.as_str(&env.interns));

    let entry = ctx.append_basic_block(parent, "entry");
    env.builder.position_at_end(entry);

    let return_true = ctx.append_basic_block(parent, "return_true");
    let return_false = ctx.append_basic_block(parent, "return_false");
```

```
// first, check whether the length is equal

let len1 = list_len(env.builder, list1);
let len2 = list_len(env.builder, list2);

let length_equal: IntValue =
    env.builder
        .build_int_compare(IntPredicate::EQ, len1, len2, "bounds_check");

let then_block = ctx.append_basic_block(parent, "then");

env.builder
    .build_conditional_branch(length_equal, then_block, return_false);

{
    // the length is equal; check elements pointwise
    env.builder.position_at_end(then_block);

    let builder = env.builder;
    let element_type = basic_type_from_layout(env, layout_interner, element_layout);
    let ptr_type = element_type.ptr_type(AddressSpace::default());
    let ptr1 = load_list_ptr(env.builder, list1, ptr_type);
    let ptr2 = load_list_ptr(env.builder, list2, ptr_type);

    // we know that len1 == len2
    let end = len1;

    // allocate a stack slot for the current index
    let index_alloca = builder.build_alloca(env.ptr_int(), "index");
    builder.build_store(index_alloca, env.ptr_int().const_zero());

    let loop_bb = ctx.append_basic_block(parent, "loop");
    let body_bb = ctx.append_basic_block(parent, "body");
    let increment_bb = ctx.append_basic_block(parent, "increment");

    // the "top" of the loop
    builder.build_unconditional_branch(loop_bb);
    builder.position_at_end(loop_bb);

    let curr_index = builder
        .new_build_load(env.ptr_int(), index_alloca, "index")
        .into_int_value();

    // #index < end
    let loop_end_cond =
        builder.build_int_compare(IntPredicate::ULT, curr_index, end, "bounds_check");

    // if we're at the end, and all elements were equal so far, return true
    // otherwise check the current elements for equality
    builder.build_conditional_branch(loop_end_cond, body_bb, return_true);
```

```
{
    // loop body
    builder.position_at_end(body_bb);

    let elem1 = {
        let elem_ptr = unsafe {
            builder.new_build_in_bounds_gep(element_type, ptr1, &[curr_index], "load_index")
        };
        load_roc_value(env, layout_interner, element_layout, elem_ptr, "get_elem")
    };

    let elem2 = {
        let elem_ptr = unsafe {
            builder.new_build_in_bounds_gep(element_type, ptr2, &[curr_index], "load_index")
        };
        load_roc_value(env, layout_interner, element_layout, elem_ptr, "get_elem")
    };

    let are_equal = build_eq(
        env,
        layout_interner,
        layout_ids,
        elem1,
        elem2,
        element_layout,
        element_layout,
    )
    .into_int_value();

    // if the elements are equal, increment the index and check the next element
    // otherwise, return false
    builder.build_conditional_branch(are_equal, increment_bb, return_false);
}

{
    env.builder.position_at_end(increment_bb);

    // constant 1isize
    let one = env.ptr_int().const_int(1, false);

    let next_index = builder.build_int_add(curr_index, one, "nextindex");

    builder.build_store(index_alloca, next_index);

    // jump back to the top of the loop
    builder.build_unconditional_branch(loop_bb);
}
```

Goal: Get Higher-Level

Some languages can compile to **LLVM bitcode**

LLVM bitcode can mix with **Roc compiler output**

This includes **C**, **C++**, **Zig**, and **Rust**

```
if len1 == len2 {
    // then-block
} else {
    // else-block
}
```

Obvious First Choice: Rust!

`unsafe` needed all over the place

unsafe FFI means "you're on your own"

Rust's generated LLVM caused problems

Tooling and development build difficulties


Rust & ZIG
TOGETHER!

Calling between Rust and Zig

Both can compile to **C-compatible** binary libraries

Both can import **C-compatible** binary libraries

Overhead is same as **using a C library** (e.g. `libssl`)

Sharing type defs requires duplication or code gen

Why not use Zig in Roc's compiler too?

We've discussed it!

Main appeal: **compile times**

Zig's **allocators** are a natural fit

Some parts of code base could be **simplified**

Why not use Zig in Roc's compiler too?

We already have 300K LoC of Rust

Sharing code would complicate the build (even more)

`unsafe` is very nice for auditing new contributions

Costs seem to outweigh benefits for the compiler itself

Why else might one mix Rust and Zig?

Large code base with lots of **mandatory-unsafe** code

Also lots of things with tricky **lifetimes** to get right

Might want access to Zig toolchain **and** Rust crates

Rust's concurrency checking, but also Zig's **`comptime`**


Rust & ZIG
TOGETHER!

I host a podcast!

roc-lang.org

software-unscripted.com