High-Fidelity C Interoperability in Hylo

1. Contributions

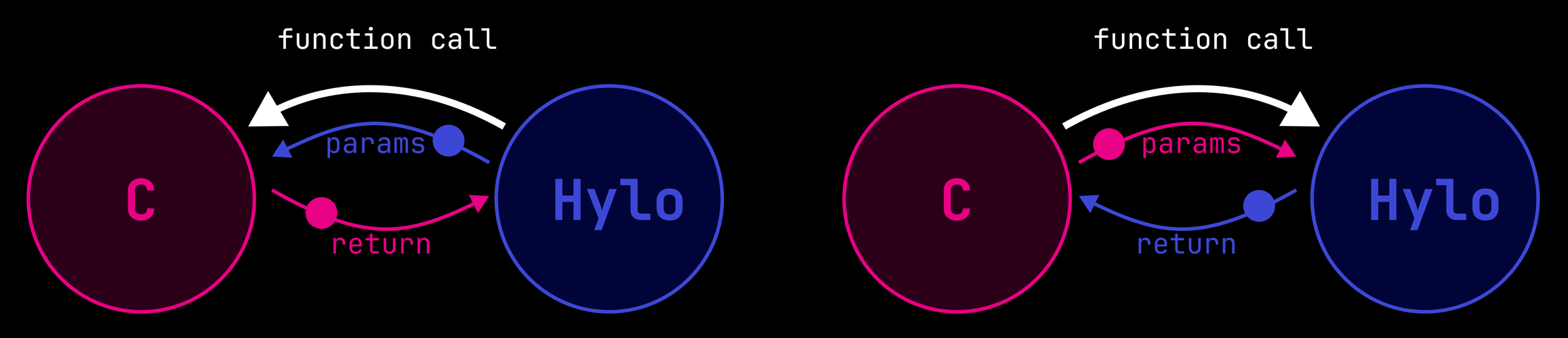
- design goals for a high-fidelity C interoperability
- novel, simple architectural design for capturing memory layout of C into Hylo
- specification for mapping C constructs to Hylo

2. Methodology

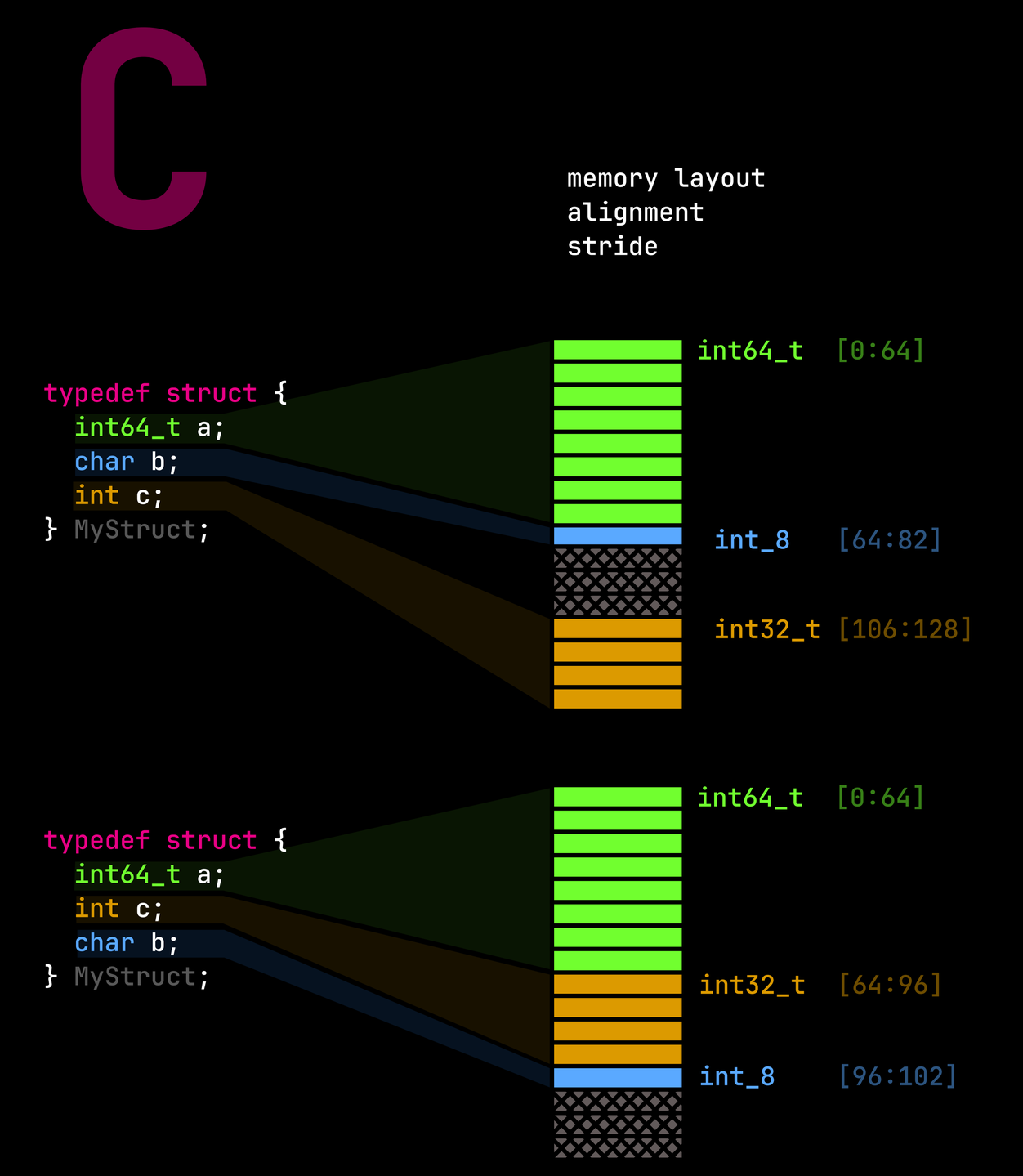
- Understand C and Hylo well
- Industry review - Rust Bindgen, Swift, Zig
- Academic literature review (scopus, Undermind)
- Personal interviews with PL experts and interop tooling developers
- Prototypes:
  - ABI explorer - abiexplorer.org
  - Explicit conversions from/to C integers
  - mapping prototypes: bit-fields, unions, flexible array members

3. Required for Interop

ABI: Abstract Binary Interface

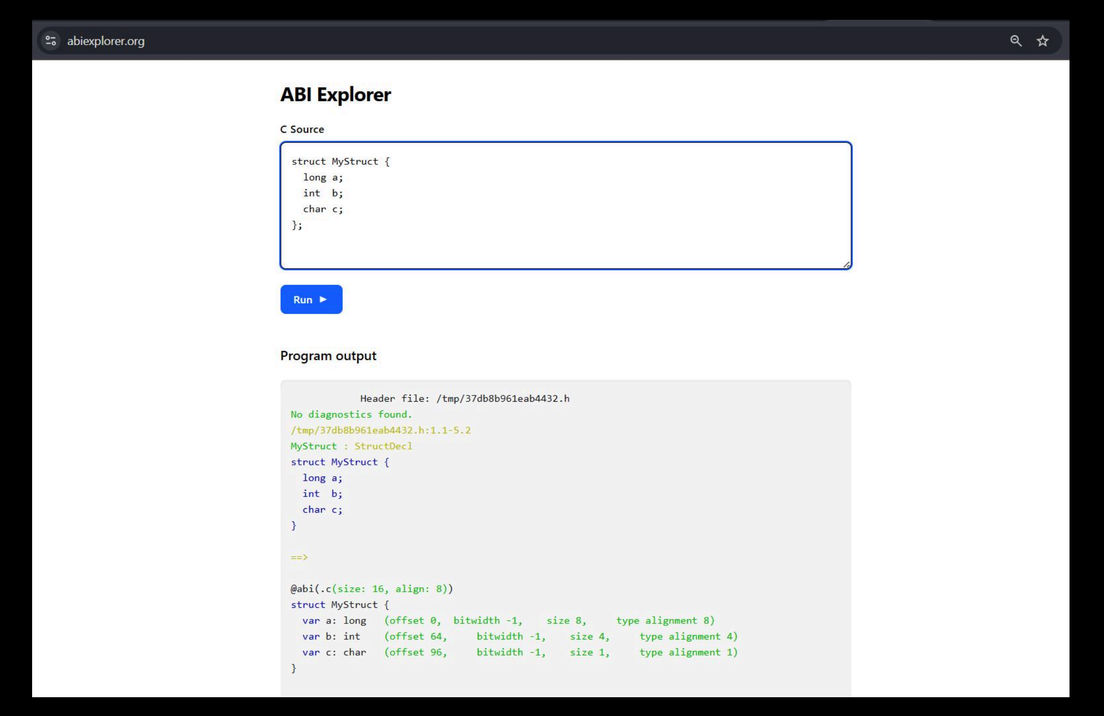


Two hard things:
- function calling conventions
- memory layout of passed data



Hylo

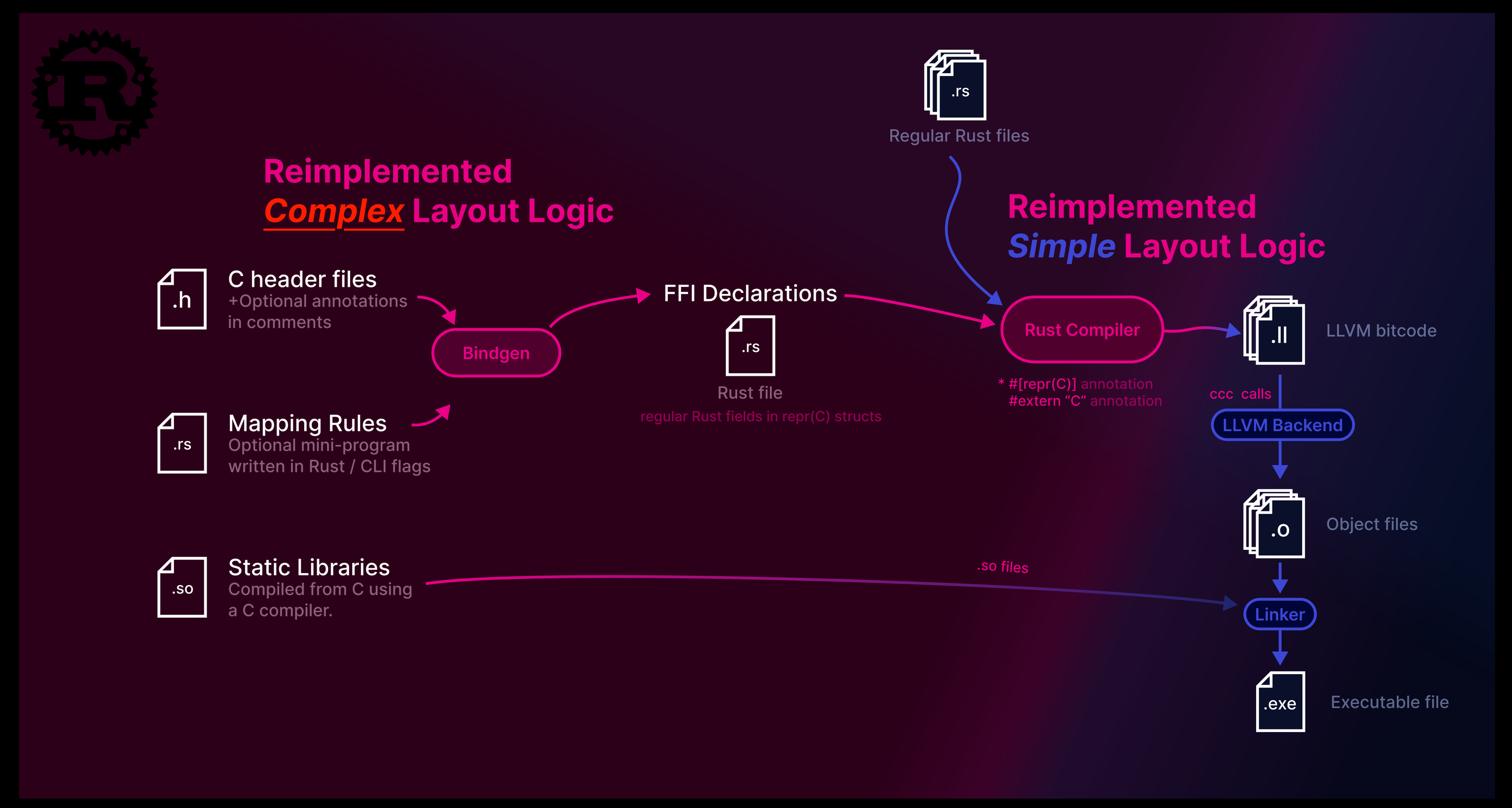
```
@alignment(8) @layout(transparent)
struct MyStruct {
  private var storage: Int8[16] = .init(all: 0)        // Do type
                                                        // punning here!
  public property a: Int64 {
    let {
      yield Int64(...)
    }
    inout {
      var temp = CChar(storage_byte_0)
      yield &temp
      &storage_byte_0 = UInt8(temp)
    }
    set (new_value) {
      &self.storage[...] = new_value
    }
  }                                                     <- Computed properties

  public property b: CChar {
    let {...}
    inout {...}
    set (new_value) {...}
  }                                                     <- Computed properties

  public property c: CInt {
    let {...}
    inout {...}
    set (new_value) {...}
  }                                                     <- Computed properties
}
```

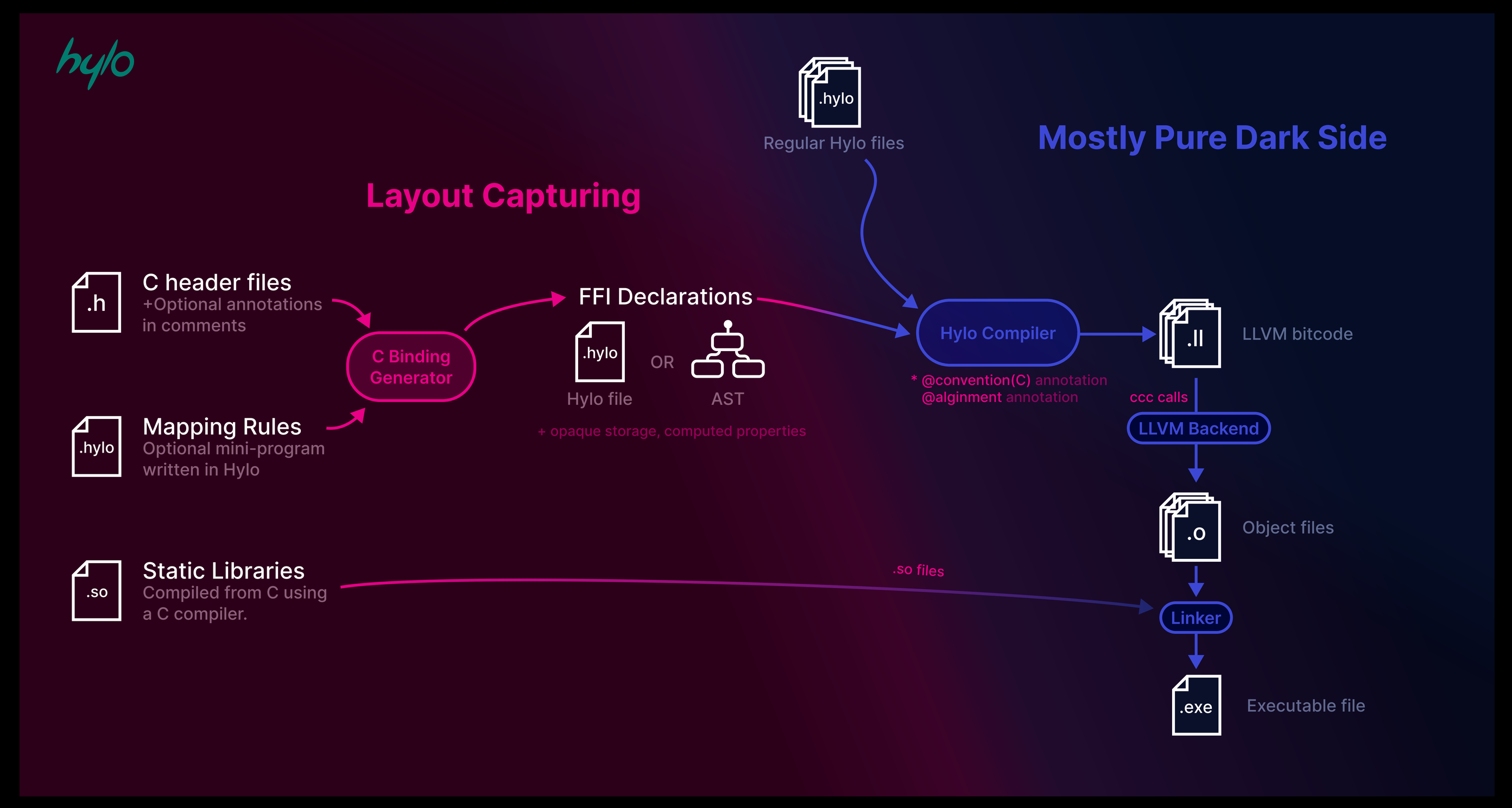


4. Other Requirements

- High coverage of C constructs
- Flexible and portable use (dialects)
- Maintainable and robust interop tooling
- Control and customizability
- Cross-Language LSP support
- Build system integration

Architecture





Type Mapping: Integers

// C Declaration	// C Meaning
`char x;`	"size = 1 byte, at least 8 bit, either signed or unsigned"
`int x;`	"signed, at least 16 bit sizeof(short) ⩽ sizeof(int) ⩽ sizeof(long)"

- avoid conversions when not needed
- preserve values
  - guarantee at compile-time or runtime
- encourage maximal portability - no reliance on accidentally matching types

```
                    #if
int x;   -/->   UInt32 x;

         --->   CInt x;

let x1 = UInt32(truncating_if_needed: x)
let x2 = UInt32(trap_on_loss: x)
let x3 = UInt32(non_narrowing: x)
```

Future Work

- macro translation
- full implementation
- design and implement the customization library details