Automatic C Bindings Generation for Haskell

Travis Cardwell
Well-Typed LLP
London, United Kingdom
travis@well-typed.com

Sam Derbyshire
Well-Typed LLP
London, United Kingdom
sam@well-typed.com

Edsko de Vries
Well-Typed LLP
London, United Kingdom
edsko@well-typed.com

Dominik Schrempf
Well-Typed LLP
London, United Kingdom
dominik@well-typed.com

Abstract

Interfacing Haskell with C libraries is a common necessity, but the manual creation of bindings is both error-prone and laborious. We present `hs-bindgen`, a new tool that provides fully automatic generation of Haskell FFI bindings directly from C header files.

We introduce a novel method for describing bindings through *binding specifications*, providing a compositional method for using bindings for one library in bindings for another. Furthermore, we define a domain-specific language in Haskell to represent C expressions found in C macros, along with a corresponding type inference algorithm, to allow us to generate bindings for functions defined as C macros.

CCS Concepts: • **Software and its engineering → Source code generation**; *Macro languages.*

Keywords: binding generation, foreign function interface, ffi

ACM Reference Format:
Travis Cardwell, Sam Derbyshire, Edsko de Vries, and Dominik Schrempf. 2025. Automatic C Bindings Generation for Haskell. In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium (Haskell '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759164.3759350

1 Introduction

Haskell supports interaction with other programming languages through its foreign function interface or FFI [13, 17], part of the language specification since version 2010 [14].

Haskell '25, Singapore, Singapore

ACM ISBN 979-8-4007-2147-2/25/10
https://doi.org/10.1145/3759164.3759350

Probably the most important supported foreign language is C, with JavaScript perhaps coming in second place due to its importance for the WebAssembly [6] backend. In this paper we will focus exclusively on C, though some of the techniques we describe may apply in other contexts as well.

Haskell code can be made aware of functions written in C through a "foreign import" declaration. For example,

```
void print_coords(int x, int y) {
    printf("{%d, %d}\n", x, y);
}
```

can be imported as

```
foreign import ccall "print_coords"
  print_coords :: CInt → CInt → IO ()
```

and then called like any other function.

While C's primitive types such as `int` are supported out of the box, oftentimes C functions use compound types such as structs, enums and unions. For example, instead of `print_coords`, which takes separate x and y parameters, we might have

```
struct point { int x; int y; };

void print_point(struct point* p) {
    printf("{%d, %d}\n", p→x, p→y);
}
```

In cases like this we must define a Haskell type which corresponds to this C type, as well as a way to *serialise* the Haskell representation to the (in-memory) representation expected by C functions. We do this by giving an instance of the `Storable` type class, which describes how to marshall between Haskell types and C types; see fig. 1.

Tooling exists that can help with the technical details of such bindings, filling in information such as size and alignment (see section 7). However, as this small example already demonstrates, even with such tooling, writing bindings can be quite laborious—albeit not conceptually difficult.

Moreover, not all functions are supported; for example, there is no support for passing structs by value:

```
void print_byval(struct point p) {..}
```

```
data Point = Point {
    point_x :: CInt
  , point_y :: CInt
  }

instance Storable Point where
  sizeOf    _ = 8
  alignment _ = 4

  peek ptr =
    pure Point
      <*> peekByteOff ptr 0
      <*> peekByteOff ptr 4

  poke ptr (Point x y) = do
    pokeByteOff ptr 0 x
    pokeByteOff ptr 4 y

foreign import ccall "print_point"
  print_point :: Ptr Point → IO ()
```

Figure 1. Using Compound Types

Calling such functions usually involves writing a C wrapper that accepts the struct by reference, then writing a Haskell binding for the wrapper instead of the original function, making sure to deal with memory management. Again, not conceptually difficult, but tedious—especially when binding against large C libraries.

In this paper, we describe `hs-bindgen`, a new tool that can generate such bindings automatically, including foreign import declarations, marshalling code (`Storable` instances), and C wrappers. As it turns out, doing so in a completely automatic way requires careful design (section 2), and while it is of course not as difficult as compilation, we make use of several techniques from compiler design such as *use-decl* graphs (section 3.1), type inference (section 5.3), and others.

Consequently, `hs-bindgen` is structured like a compiler, with a frontend that processes the C source (section 3), which is followed by a backend generating Haskell code (section 4). We take special care to deal with C macros (section 5). In section 6 we introduce *binding specifications*, which enable us to re-use previously generated bindings, as well as guide code generation. Finally, we compare to related work in section 7 and conclude in section 8.

2 Design

Before we delve into the technical details, it is useful to take a moment to describe our overall design philosophy.

2.1 Design Principles

We start with the design principle that motivated this work:

> **Design principle 1: Automation.**
> We should be able to produce bindings without user input. Overrides are possible, but not required.

The principle of automation departs from most existing Haskell tools that aid in generating interface code (section 7), where it is the users who write bindings, using the tool to fill in specific details. In contrast, we sought to design a tool that automatically generates bindings, optionally allowing the user to override specific details.

Users can influence the generation process by changing global configuration parameters and through the use of (input) binding specifications (section 6.2). We also make `hs-bindgen` available as a Haskell library (in addition to the command line interface), so that in principle every aspect of generation can be tweaked. However, it should be possible for users to rely on sensible defaults, leaving `hs-bindgen` to produce bindings without *any* configuration. The aim of zero configuration motivates many of our decisions, including name generation (section 3.3), type inference for macros (section 5.3), and (external) binding specifications (section 6.1).

> **Design principle 2: Preservation of semantics.**
> The semantics of the C code, both explicit and implicit, should be preserved.

At some level this principle is obvious: clearly we need to ensure that the way we call C functions and the way that we marshall C types matches the C semantics. However, there are less obvious consequences also. For example, C allows the definition of type aliases through `typedefs`; we choose to translate such aliases to Haskell newtypes rather than expanding them (section 4.4), to preserve the implied semantics of the alias.

> **Design principle 3: Predictability.**
> The code generated by `hs-bindgen` should be clearly predictable from its corresponding C code.

Predictability has practical ramifications. For example, since C has anonymous types and Haskell does not, those anonymous types must be given a name (section 3.2). We could produce an arbitrary name, perhaps using an internal counter, but if we did that, minor changes to the C input could result in entirely different generated Haskell code. Since C bindings are not standalone, but are intended for integration into larger systems, this kind of instability of the generated bindings could rapidly make the maintenance of these systems intractable.

Much like a progress/soundness theorem in a more formal setting, principle 1, Automation, states that we can always generate *something*, and principles 2, Preservation, and 3, Predictability state that we generate something reasonable.

2.2 Platform (In)dependence

Suppose we have a C struct (record) definition containing just an integer value:

```
struct val { int x };
```

When we generate code for this, we immediately have to answer two (related) questions:

1. What is the size of this integer? 4 bytes, 8 bytes, more, less? *On which platform?*
2. Which Haskell type should we use to represent the integer: `CInt`, `Int32`, `Int64`, another type?

To answer these questions, it is useful to be precise. *Cross-compilation* is the compilation of a program on a machine with a given computer architecture for another machine with a different computer architecture. A typical use-case of cross-compilation is the compilation of programs for embedded devices which do not have the computation requirements to compile their own programs. We refer to a machine and its computer architecture as a *platform*.

When generating bindings, there is one more distinction to make: the platform on which `hs-bindgen` runs may not be the platform on which the generated bindings are compiled. Unfortunately, there is no standardised nomenclature in this area; we will refer to the platform running the tool generating bindings as the *host platform*, the platform compiling the generated code as the *compilation platform*, and the platform running the compiled code as the *target platform*. All of these designations are from the perspective of `hs-bindgen`; from the perspective of the compiler, *it* runs on the host, and compiles for a certain target, and from the perspective of the generated code, its "host" is simply the platform it runs on.

Orthogonally, we differentiate between the *interface* of the generated code and its *implementation*. The interface is the totality of the function signatures and type declarations that users make use of when writing code using generated bindings. The implementation refers to class instances, function wrappers (section 4.5), and other glue code.

Let us now return to our two questions.

2.2.1 Implementation.

In simple cases it might be possible to generate platform-independent code. For example, we could generate marshalling code for `val` that relies on `hsc2hs` (section 7.2) to fill in the details:

```
instance Storable Val where
  sizeOf _ = #size struct Val
  -- other members omitted
```

Unfortunately, use of the C pre-processor (CPP) to perform conditional compilation in C makes this difficult; for example, the C header might contain

```
#ifdef __aarch64__
...
#endif
```

Such conditionals can be arbitrarily complex; *perhaps* it would be possible to generate bindings that precisely mirror the conditions, but this seems a daunting task. It would also make it significantly more difficult to use tooling for working with C (section 3), which are not designed for this workflow.

We therefore opt to produce platform-specific code. This means that the distinction between the *host* platform and the *compilation* platform disappears, and going forward we will simply refer to the *host* platform (often this is also the target platform, as most users do not need cross-compilation).

Practically speaking, it means that `hs-bindgen` is designed to be run as part of compilation: the generated bindings should be treated as build artefacts. For `val`, the marshalling code will reflect a choice of platform:

```
instance Storable Val where
  sizeOf _ = 8
```

2.2.2 Interface.

Having decided that `int` is 8 bytes on the target platform, we can still choose which Haskell datatype we generate. The two most natural choices are

```
data Val = Val {
    x :: Int64
  }
```

or

```
data Val = Val {
    x :: CInt
  }
```

We opt for the latter, and choose `CInt`. We can motivate this choice using our design principles. First, `CInt` better preserves the semantics of the C code: `int` in C is machine-dependent, and so is `CInt` in Haskell. Choosing `Int64` would incorrectly suggest that the C interface is designed to *always* work with 64-bit values. Conversely, the use of `int` in the C library means that it promises that it can work across multiple architectures; that promise is lost if we use `Int64`.

Second, `CInt` is easier to predict. If the generated API is platform dependent, it is more difficult for users to know what it will be (predictability of the *implementation* is not important: it does not directly affect users). One important advantage of this predictability is that it minimises the risk that code written against the generated bindings on one platform does not compile on another. Unfortunately, this risk can only be reduced, not eliminated, due to conditional compilation.

3 Frontend

Writing a parser for the complete C language with its many extensions, as well as logic that determines the layout of structs, the types underlying `enum`s, etc.; and doing all of that across multiple platforms, is an enormous undertaking which is well beyond the scope of a tool such as `hs-bindgen`. Fortunately, we have an advantage that authors of previous tooling in the Haskell ecosystem (section 7) did not have: the availability of a modern C toolchain as a library: `clang` [12].

The first step in our processing pipeline uses `clang` to build a pure Haskell representation of `clang`'s internal abstract syntax tree (AST), extracting the information we need. During parsing we also construct a graph denoting which C headers include which other C headers. We use the include graph to resolve binding specifications (section 6).

We process this AST in several passes which incrementally annotate nodes of the tree with additional information. To achieve this, and to avoid needing many slightly different copies of the AST, we use a trees-that-grow style [15] design. We describe some passes in the remainder of this section, but postpone the description of more involved passes to dedicated sections:

1. Sort declarations with the *use-decl* graph (section 3.1)
2. Handle macros (section 5)
3. Rename anonymous declarations (section 3.2)
4. Resolve binding specifications (section 6)
5. Assign Haskell names (section 3.3)

We discuss some drawbacks of using `clang` in section 3.4.

3.1 *Use-decl* Graph

After constructing the AST, we build a *use-decl* graph. Nodes in the graph correspond to declarations, directed edges record use sites, and edge labels clarify exactly *how* a declaration is used. For example, consider this C declaration:

```
1 typedef struct {
2   struct {
3     struct { int a; } *innermost;
4   } inner1;
5   struct { int b; } inner2;
6   int c;
7 } *toplevel;
```

This example contains four anonymous struct declarations, on lines 1, 2, 3 and 5. The resulting *use-decl* graph is depicted in fig. 2; initially, anonymous declarations are assigned an identifier based on their location in the source.

A first use of this graph is to sort all declarations so that types are always declared before they are used (at least, before they are used by value; for pointers, the type declaration is not needed). This is *mostly* true for C, but inline type declarations like in this example (anonymous or not) are an exception, and we find it useful to enforce this invariant for *all* declarations.

3.2 Naming Anonymous Declarations

Unlike C, Haskell does not support anonymous record declarations out of the box. There are libraries such as `vinyl`[1] and `large-anon`[2] that implement them, but each comes with their own set of drawbacks that not all users might be happy with (advanced types, use of a compiler plugin, etc.). The code produced by `hs-bindgen` therefore does not depend on these external packages, although in principle we could *optionally* make use of them.

This means that we must construct names for anonymous declarations, and as discussed in section 2.1, it is important that we do so in a predictable way. In some cases there is

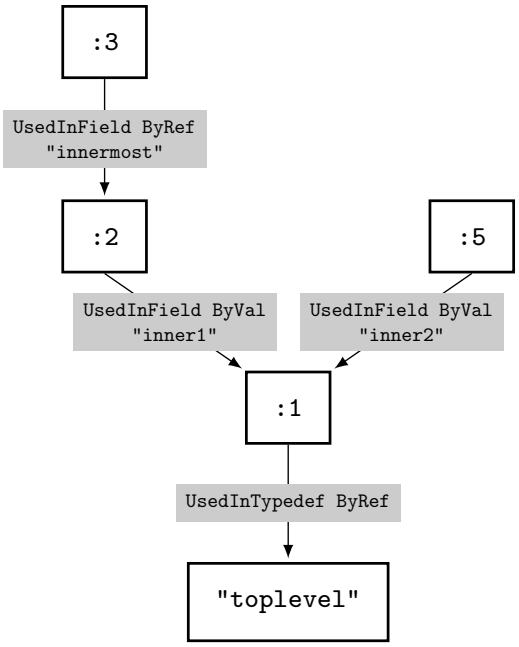

Figure 2. Example *use-decl* Graph

an obvious candidate, for example, a common pattern is a `typedef` around an anonymous `struct`:

```
typedef struct { .. } foo;
```

In this case, we can use the name of the `typedef` for the `struct` (and there is no need to distinguish between the name of the type defined by `typedef` and the name of the `struct` in the generated Haskell code).

In general, we employ the *use-decl* graph to analyse the use site of an anonymous declaration. Given an anonymous declaration and its use site[3], we apply the following rules:

1. When the anonymous declaration is used by value in a `typedef`, the `struct` gets the name of the `typedef`, and we do not generate a separate Haskell type for the `typedef`.
2. When the anonymous declaration is used by reference in a `typedef`, we append "`_Deref`" to the name of the `typedef`.
3. When an anonymous type is declared as part of a `struct` field, it is given the name which is the concatenation of the enclosing type (with the same naming rules applied recursively) and the name of the field.

For the running example (with *use-decl* graph in fig. 2), this results in the following Haskell names:

- `Toplevel`, corresponding to the C typedef
- `Toplevel_Deref`, for the outermost struct

[1] https://hackage.haskell.org/package/vinyl
[2] https://hackage.haskell.org/package/large-anon

[3] An anonymous declaration can have at most one use site, since there is no way to reference it. Anonymous declarations without *any* use sites we simply omit.

- `Toplevel_Deref_inner1` and `(..)_inner2` for the two "middle level" structs, and for the innermost struct `Toplevel_Deref_inner1_innermost`.

3.3 Generating Haskell Names

In order to produce valid Haskell code, we must "mangle" the C names so that they conform to Haskell's naming rules, the most important of which is that types must start with an uppercase letter, and values must *not* start with an uppercase letter.[4] This is not always trivial. For example, some Unicode characters considered to be "uppercase" do not have lowercase equivalents, and there are some other Unicode-related differences between C and Haskell to be taken care of also; these technical details are not particularly interesting and we refer the interested reader to the implementation instead.

In the context of name mangling, we seek to adhere to design principle 1, Automation, in that:

- we should always be able to generate a Haskell name, while allowing the user to override names (section 6);
- name mangling should not result in name clashes (resolving clashes would *require* user input) while still staying as close as possible to the original C names (design principle 3, Predictability).

3.4 Limitations of Using `clang`

Relying on `clang` has an important drawback: if the library we are generating bindings for is compiled with a different compiler than `clang` (such as `gcc`), certain details we get from `clang` (such as the precise layouts of `structs`, needed to generate marshalling code) might be incompatible with decisions made by the other compiler. Ultimately this would result in incorrect generated code.

Although the same problem arises whenever multiple C compilers are used together, it does mean that the use of `hs-bindgen` technically dictates the consistent usage of `clang`. This requirement may feel like a serious limitation, but fortunately in practice it is seldom an issue: the cases where divergence is possible are rare, and while we cannot remedy the situation when divergence does arise, we can at least attempt to detect it. As part of binding generation, `hs-bindgen` can generate a test suite that tests the generated marshalling code: we produce some values in Haskell, pass those values to a C function, which then asserts that the values it gets passed are correct and conform to the known expectations.

[4] This is the rule used by GHC, which is subtly different from the rule in the Haskell Report [14, Section 2.4, Identifiers and Operators], which requires that values must start with a lowercase letter. For letters from the Latin alphabet this is equivalent, but this is not true in general; for example, Chinese characters are considered to be neither uppercase nor lowercase.

4 Backend

The backend is responsible for generating Haskell code. In this section, we highlight key examples and illustrate some of its more intricate design decisions.

In some cases, the code produced by `hs-bindgen` requires specialised infrastructure, which we provide in a library designed for this purpose called `hs-bindgen-runtime`. We will introduce the relevant parts of this infrastructure where needed.

4.1 Structs

We saw in section 1 that the generated bindings for a simple C struct are a Haskell record datatype together with a `Storable` instance. In addition, we can also generate other instances such as `Eq`, `Show` etc.; we keep track of which instances are available for the types of the fields in order to determine which instances apply to the overall record.

However, some features of C structs must be addressed separately; we will turn our attention to these now.

4.1.1 Flexible Array Members.

A flexible array member (or FLAM) is an array of unspecified size, which must be the last field in a struct. For example:

```
struct surname {int len; char data[];};
```

In C, the **`sizeof`** operator, when applied to a structure with a FLAM, gives the size of the structure as if the FLAM were empty. In a similar spirit, the generated Haskell data type only contains the fixed-size members of the structure:

```
data Surname = Surname {
    surname_len :: CInt
  }
```

To recover the full datatype, `hs-bindgen-runtime` offers a type combinator called `WithFlexibleArrayMember`; values of such a struct-with-FLAM can be serialised, provided two instances are available:

- An instance of `HasFlexibleArrayMember` determines the offset of the FLAM within the struct. This instance is generated by `hs-bindgen`.
- An instance of `HasFlexibleArrayLength` determines the size of the FLAM, given the fixed-size part of the struct. This instance must be provided by the user.

4.1.2 Bitfields.

Bitfields are fields of a struct with a size specified in bits. For example:

```
struct room {
  char window_id; int blinds_open : 1;
  char projector_id; int power_mode : 2;
};
```

For convenient access, we turn these bitfields into regular fields (though we should perhaps do a better job at giving them more precise types):

```
data Room = Room {
    room_window_id    :: CChar
  , room_blinds_open  :: CInt
  , room_projector_id :: CChar
  , room_power_mode   :: CInt
  }
```

The specific memory layout only becomes apparent in the generated `Storable` instance, which must of course respect the C layout.

4.2 Enums

C enums give names to values of an underlying type, usually (but not always) `unsigned int`. For example

```
enum HTTP_status {
    ok          = 200
  , bad_request = 400
  };
```

These values need not be consecutive or ordered, and the same value may even be assigned multiple names. Moreover, although `HTTP_status` can be used as a type, it is important to note that this is only a hint: it does *not* guarantee that the only possibles values are those that have been given a name.

4.2.1 Representation. We might expect an enum to be represented by an algebraic sum type with one constructor per name, but this is ruled out by the fact that we have values without a name, as well as values with multiple names.

Instead we use a newtype around the enum's underlying type, with a set of patterns synonyms [18] for the named values. For our running example, we generate

```
newtype HTTP_status = HTTP_status CUInt
```

as well as

```
pattern Ok, Bad_request :: HTTP_status
pattern Ok          = HTTP_status 200
pattern Bad_request = HTTP_status 400
```

Crucially, this set of pattern synonyms is *not* declared to be complete: other values are possible.

4.2.2 `Enum` and `Bounded` Instances. While we are on the topic of failed expectations, `hs-bindgen` does not provide an instance for the Haskell `Enum` type class by default. `Enum` requires us to define a successor relation, but it is not clear from the types what that relation should be: is the successor of value 200 ("`Ok`") 400 ("`Bad_request`"), or 201 (a value with no name)? Similar concerns apply to `Bounded`, which picks a minimum and maximum value of the type; should those be 200 and 400 respectively, or should they be the minimum and maximum value of the underlying integral type?

Instead, `hs-bindgen-runtime` provides a type class called `CEnum`. This type class has several members, but the most important one is a method that returns the set of names of the enum:

```
class CEnum a where
  declaredValues ::
    proxy a → DeclaredValues a
  -- other members omitted
```

The library offers deriving-via [1] support for users who *choose* to imbue the enum with a successor relation implied by the list of names (so that the successor of 200 is 400); alternatively, they can use newtype deriving, in which case the successor of 200 would be 201. Likewise, users can choose to derive `Bounded` via `CEnum`, or use newtype deriving to inherit the `Bounded` instance from the underlying type.

4.3 Unions

C unions do not carry a tag recording which alternative of the union is used. Indeed, some unions *don't* encode alternatives, but rather multiple views on the *same* data. This means they do not map onto Haskell sum types. Instead, we represent a union as an opaque type type with setters and getters. For example,

```
union occupation {
  struct student  {..} student;
  struct employee {..} employee;
};
```

is represented by

```
newtype Occupation  -- opaque
instance Storable Occupation where ..

get_occupation_student ::
  Occupation → Student
set_occupation_student ::
  Student  → Occupation → Occupation
```

and similarly for `Employee`. Unions are usually embedded in larger types (typically structs) which provide context for determining which alternative applies; since `hs-bindgen` has no way of reconstructing this logic, it is up to the users to use the appropriate getter and setter functions.

4.4 Typedefs

Typedefs in C are similar to type aliases in Haskell: they give a new name to an existing type, often used to provide additional semantic information. For example, the C standard library defines a type for the system time in `time.h`

```
typedef long clock_t;
```

As noted in section 2.1 when we discussed design principle 2, "preservation of semantics", it is important that we reflect these typedefs in the Haskell, rather than expanding them. That still leaves us with the choice to represent these as Haskell `type` aliases or as `newtypes`. We opt for the latter, as this gives users the option of providing different type class instances (for instance, perhaps the `Show` instance for

clock_t could show something more informative than just the integer value):

```
newtype Clock_t = Clock_t CLong
```

The use of newtype also results in the usual improved type safety over type aliases, requiring explicit coercion between Clock_t and its underlying representation.

4.5 Functions

We saw in section 1 that a function such as

```
void print_point(struct point* p);
```

can be imported in Haskell as

```
foreign import ccall "print_point"
  print_point :: Ptr Point → IO ()
```

This is however not quite what we do.

4.5.1 Userland CAPI. Instead, we generate a C wrapper which calls the original C function:[5]

```
void B_print_point(struct point *arg1) {
  print_point(arg1);
}
```

and then import the wrapper instead:

```
foreign import ccall "B_print_point"
  print_point :: Ptr Point → IO ()
```

This is precisely what the CAPI calling convention does in GHC (see section "The CAPI calling convention" of the GHC manual). We effectively reimplement this as a "userland CAPI" calling convention; this enables us to extend the set of functions for which we can generate wrappers (perhaps some of these can eventually make it into GHC itself).

4.5.2 Structs by Value. Unlike GHC, we can generate wrappers for functions that accept or return structs by value; such functions cannot be imported directly.
For example, for

```
struct point byval(struct point p);
```

we generate the wrapper

```
void B_byval ( struct point *arg
             , struct point *res ) {
  *res = byval(*arg);
}
```

which we import as

```
foreign import ccall safe "B_byval"
  byval_wrapper ::
    Ptr Point → Ptr Point → IO ()
```

Finally, we then define a Haskell-side wrapper function which recovers the original by-value semantics:[6]

```
byval :: Point → IO Point
byval p =
  with p $ \arg →
  alloca $ \res → do
    B.byval_wrapper arg res
    peek res
```

4.5.3 Safety. One aspect that we have glossed over so far is that an import of a C function can be declared "safe" or "unsafe": safe functions may indirectly invoke other Haskell functions, whereas unsafe functions may not but are faster [13]. The default is safe, and this is also the default in hs-bindgen.

4.6 Execution Mode

The most convenient way to run hs-bindgen is using Template Haskell (TH), which essentially makes it possible to "#include" a C header directly into a Haskell module, generating the bindings on the fly.

The alternative is to use hs-bindgen as a preprocessor. The most important use case for this is cross-compilation. Any code executed in a TH splice must run on the target platform; this means that if we run hs-bindgen in a TH splice, it as well as the entire C toolchain must run on the target, rather defeating the purpose of cross-compilation in the first place.

5 Macros

Some C libraries expose part of their public API as macros rather than functions. For example, a library for memory-mapped I/O might provide the offset of a register with respect to some base address as a macro defining a constant:

```
#define REG_OFFSET 12
```

Alternatively, the library might provide the same information as a macro *function* which adds the offset to a base pointer:

```
#define REG_PTR(ptr) ptr + 12
```

In order to produce complete Haskell bindings for such a library we must also generate Haskell bindings for these macros, at least for those that are intended to be part of the library's public interface (many libraries use macros as internal implementation techniques). This poses a problem, however: in C, there is no macro language per se. Macros are simply expanded by the C preprocessor, so macros are best thought of as consisting of *raw source tokens*—which is indeed how they are represented in the clang AST.

In practice, we have observed that macros used in user-facing APIs of libraries are largely covered by the two cases mentioned above: constants and functions. Furthermore, these macros are meant to be used *within C expressions*, so they should be imbued with the semantics of C code (which is not a priori obvious, since definitionally macros are just lists of raw tokens).

[5] The B_ prefix depends on the name of the Haskell module.
[6] Functions with, alloca and peek are all part of the FFI infrastructure in the Haskell base library.

Additionally, hs-bindgen also supports macro-defined *types*, which we treat similarly to C typedefs. This usage is however more rare, especially for public facing types[7], and we will not discuss it further in this paper.

5.1 Ingredients for Binding Macros

Consider the task of generating bindings for C expression-like macros. For simple constants such as:

```
#define N 8L
#define PI 3.14
```

hs-bindgen will generate the following bindings:

```
n :: CLong
n = 8
pi :: CDouble
pi = 3.14
```

This is straightforward so far: the type signatures can be immediately deduced from the literals. However, once one introduces functions, it becomes less obvious how to proceed:

```
#define F(X) (X)*((X)+1)
```

Generating a Haskell binding for F requires some form of operator overloading. The most obvious approach is to use standard Haskell type classes such as Num, but this is not general enough. For example, addition in C can be used to add an integer to a pointer, and get back another pointer; the REG_PTR function makes use of this. In general, the semantics of C arithmetic expressions introduces implicit conversions (§6.3.1 Arithmetic Operands in [8]). For example, consider

```
float fn(short x, float y){
  return (x + 2 * y);
};
```

Here, the expression x + 2 * y is typed from the leaves to the root using the C rules of integer promotion and arithmetic conversion:

- In 2 * y, 2 is an int which gets converted to a float before multiplication.
- Then, x is promoted to an int, and is then converted to a float before addition.

This can be naturally expressed in Haskell through class methods which compute the result type as a function of the argument types, e.g.

```
class Add a b where
  type AddRes a b
  (+) :: a → b → AddRes a b
class Mult a b where
  type MultRes a b
  (*) :: a → b → MultRes a b
```

This allows us to give the following polymorphic type for F:

```
f :: (Add a CInt, Mult a (AddRes a CInt))
  ⇒ a → MultRes a (AddRes a CInt)
f x = x*(x+1)
```

For REG_PTR we end up with

```
Add a CInt ⇒ a → AddRes a CInt
```

Importantly, this type can be instantiated to Ptr () →Ptr ().

To achieve this, we need two separate ingredients for binding macros:

- A library of overloaded functions, containing type classes and type class instances (such as Add and Mult) that define the target domain-specific language for C arithmetic expressions.
- A "mini macro compiler", which parses C expression-like macros, performs type inference on them, and generates code written in the above DSL.

The key observation here is that these two parts need to fit together: the "mini macro compiler" should be aware of the type class and type family instances that are defined in the DSL. This is necessary to keep the type signatures as simple as possible, e.g. when generating a binding for

```
#define TWO_PI 2 * PI
```

it is clearly preferable to generate

```
two_pi :: CDouble
two_pi = 2 * pi
```

rather than annotating it with the convoluted type

```
two_pi :: Mult CInt CDouble
       ⇒ MultRes CInt CDouble
```

5.2 c-expr: a DSL for C Expressions

c-expr is a Haskell library which provides a collection of type classes and type class instances that provide a DSL for C arithmetic expressions; we showed two examples Add and Mult in the previous section.

To define instances, c-expr defines value-level functions that implement C's arithmetic rules ([8], §6.3.1 Arithmetic Operands). For example:

```
binaryAddType
  :: Platform
  → CType → CType
  → Maybe (CType, ImplInfo)
```

Here, binaryAddType computes the return type of addition in terms of the types of the two arguments, together with additional implementation information which records the desired implementation of the type class instance. Note that this is platform-dependent, because the C rules for arithmetic conversion are platform-dependent.

We implement binaryAddType as a Haskell function for two main reasons:

[7] It is quite common for *internal* types, especially combined with conditional compilation. For example, MinGW defines the macro __PTRDIFF_TYPE__ as long long int or long int depending on the platform, and then defines the public type ptrdiff_t as a typedef in terms of __PTRDIFF_TYPE__.

- As discussed in section 5.1, type inference for macros (section 5.3) requires the ability to compute type family reduction. Exposing a Haskell function that implements this logic allows it to be imported and used in the implementation of `hs-bindgen`.
- It allows us to double-check the correctness of our implementation of C arithmetic conversion rules by comparing them against `clang`. [8]

With such Haskell functions in hand, we then use Template Haskell to generate corresponding Haskell instances for the target platform[9] in the `c-expr` DSL, such as

```
instance Add CInt CInt where
  type AddRes CInt CInt = CInt
  (+) = (Prelude.+)
instance Add CInt CFloat where
  type AddRes CInt CFloat = CFloat
  x + y = fromIntegral x Prelude.+ y
instance Add (Ptr a) CInt where
  type AddRes (Ptr a) CInt = Ptr a
  x + y = x `plusPtr` fromIntegral y
```

These type classes and instances are exposed by the `C.Expr` module of the `c-expr` library.

5.3 Type Inference

In order to perform type inference for C arithmetic expressions, `hs-bindgen` internally defines a Hindley–Milner-like type system with data constructors, multi-parameter type classes, and type families:

$$\tau = \text{ty-var} \mid \tau \to \tau \mid \text{data-con}\ \{\tau\} \mid \text{ty-fam}\ \{\tau\}$$
$$\theta = \text{class}\ \{\tau\} \mid \tau \sim \tau$$
$$\sigma = \forall\ \{\text{tyvar}\}.\ \{\theta\} \Rightarrow \tau$$

We can think of this type system as being a sub-type system carved out of the Haskell type system, consisting of the constructs that are used in `c-expr`'s DSL. This type system is used to perform type inference for C expressions, following the classic Damas–Milner Algorithm W [5] extended to deal with class constraints using the generate-then-solve approach of Pottier–Remy ([19], §10 The Essence of ML Type Inference). Each class comes with a collection of instances, together with defaulting assignments to handle ambiguous type variables (as in [9]).

One key challenge is to relate type inference for this internal type system with the type class and type family instances of the target language of the bindings. For example, recall that the `AddRes` type family should follow the C rules for integer promotion and arithmetic conversion (§6.3.1 Arithmetic Operands in [8]), which are platform-dependent. As per section 5.1 , we import the `c-expr` library in the implementation of type inference in `hs-bindgen`. This allows us to e.g. call the `binaryAddType` to compute, during type inference, how the `AddRes` type family reduces. This is the key to ensure that we infer types that `GHC` will accept; in particular, this ensures that the type family reduction rules implemented in `hs-bindgen`'s type inference engine agree with the type family instances available in the target DSL.

6 Binding Specifications

As Haskell programmers we value compositionality: how can we ensure that if we break a problem into subproblems, that we can then combine the solutions to those subproblems to solve the overall problem? Types are an obvious example: when we solve a task using several functions, their signatures help us put them back together.

In a similar spirit, we introduce *binding specifications*: mappings from C types to the corresponding Haskell types, along with properties of those Haskell types; the abstract syntax[10] is shown in fig. 3. In section 6.1 we will see how binding specifications ensure compositionality of bindings generated by `hs-bindgen`, and in section 6.2 we will see how we use the same concept to guide the generation of bindings in the first place.

6.1 Compositionality

Suppose we have a C library for networking, providing

```
struct socket;
void open(struct socket *s);
```

The generated bindings might look something like this[11]

```
data Socket
foreign import ccall safe "open"
  open :: Ptr Socket → IO ()
```

Other C libraries that build on this networking library will have headers that `#include` the network one. For example, a library for making HTTP requests might look like

```
#include "network.h"
void http(struct socket* s, char* req);
```

Unless told otherwise, `hs-bindgen` will generate bindings for *all* definitions used in this HTTP library, which would include the `Socket` type. This works well for standalone libraries that are implemented in a single Haskell module, but is not at all composable: the `Socket` type used in `http` would be a different type from the one in `network`, and functions such as `open` would not be usable with the `Socket` type generated from `http.h`.

Binding specifications solve this problem. As mentioned, they map C types to Haskell types; in this particular case,

[8] These functions *could* be implemented by simply calling `clang`, but this has undesirable consequences due to the way `clang` aggressively expands macros and `typedefs`.

[9] Note that `hs-bindgen`'s target platform is the same as `c-expr`'s host platform: `c-expr` is a Haskell library, which only cares about the platform on which it is executed.

[10] The concrete syntax uses YAML or JSON.

[11] For simplicity we omit the "userland CAPI" we describe in section 4.5.

```
BindingSpec  ::= CIdent ↦ HsInfo
CIdent       ::= (Name, Set Header)
HsInfo       ::= omit | require HsSpec
HsSpec       ::= (ModuleName?, HsName?, TypeClass ↦ InstanceInfo)
InstanceInfo ::= omit | require InstanceSpec
InstanceSpec ::= (Strategy?, Set Constraints)
```

Figure 3. Binding Specification Abstract Syntax

mapping the C type "socket" to the Haskell type "Socket". Alongside the generated bindings, hs-bindgen can also produce a binding specification for the network library, though it is also possible to write the specification by hand, which is useful for bindings that are not produced by hs-bindgen. If we point hs-bindgen to the network specification when we generate bindings for the HTTP library, it will reuse the previously generated Socket type (any number of such "external" binding specifications can be used).

In addition to recording *which* Haskell type corresponds to a particular C type, binding specifications can also record properties *of* that Haskell type, such as which type class instances it has (along with any superclass constraints of those instances). This too is necessary for compositionality; for example, if we know that Socket has a Storable instance but no Show instance, then if we encounter another type which encapsulates a Socket we will know to equip it with Storable but not Show instances.

One minor difficulty with binding specifications is that we need a precise way to refer to specific C identifiers. On the Haskell side, a module and identifier name suffice[12], but C does not have a "module" concept. We can instead refer to a header file, but a declaration may actually exist in an internal header file that is included by the public API, and the same declaration may be provided by multiple public headers[13]. We therefore use a set of headers, and we interpret a reference to a particular header to also mean any of its transitive includes, which we can check using the include graph we build during parsing (section 3).

6.2 Guiding Generation

Various parameters in hs-bindgen influence generation: predicates select a subset of the C declarations for which we want bindings, translation configuration allows specifying desired type class instances, etc.

When we use a binding specification *of* a set of bindings in order to *generate* those bindings in the first place, we obtain a more fine-grained way to guide binding generation. For example, we re-interpret the mapping from a C name to a Haskell name as a declaration of the *desired* name of the Haskell type; we re-interpret the set of instances as a declaration of the *desired* instances to generate (and allow to specify a strategy to do), etc. Overrides like this must be respected; for example, if a binding specification states that a type must be omitted, then it is an error if that type is used; if a binding specification states that a certain type class instance must be generated, then it is an error when that instance cannot be generated (e.g., because it depends on other instances that are not available).

One interesting detail here is what to do when the user insists that an instance must be generated, say a Show instance for a struct, but we lack an instance for one of the types of the fields of the struct. In this case we generate an instance with a superclass constraint:

```
instance Show Field ⇒ Show Record where ..
```

A concrete superclass constraint like this is unusual, but in this case quite useful; it allows the code produced by hs-bindgen to depend on an instance written by the user.

When an "input" binding specification does not mention a particular C type, we choose to interpret such *absence of information* as meaning that we should proceed with the default behaviour. Consequently, following design principle 1, "Automation", users only need to provide necessary overrides. However, it is occasionally useful to have *information about absence*; the syntax of binding specifications therefore also makes it possible to explicitly state that we should *not* produce any Haskell type for a given C type, that a Haskell type should *not* have a certain instance, etc.

7 Related Work

Automatic generation of bindings is a challenging problem that necessitates careful design and advanced techniques, often drawing parallels with compiler construction. This research also highlights a notable opportunity within the academic community. While many tools exist, there is a relative scarcity of scientific literature exploring the complex challenges of automatic binding generation as evidenced by the reliance on links to publicly available software rather than peer-reviewed articles.

7.1 Rust

The primary inspiration for our work is rust-bindgen [23], a command line client and Rust library that automatically

[12] Since binding specifications are associated with a single Haskell package, the package can be left implicit.

[13] For example, uint64_t may be declared via inttypes.h or stdint.h in the standard library.

generates Rust bindings to C and some C++ libraries. Like `hs-bindgen`, `rust-bindgen` uses `libclang` [24] to parse C headers, translating the structured information provided by `libclang` into Rust source code. The generated bindings can be customised through configuration parameters; we can include or exclude types, functions or global variables, prevent generation of certain traits (type classes), or set structure or union field visibility. Of course, many difficulties in binding generation arise from the target language, with Rust and Haskell offering different challenges; for example, Rust has C-style unions, making their translation more direct.

Macro support of `rust-bindgen` is rudimentary: bindings are generated for constants, but not for functions. In contrast, `hs-bindgen` defines a small macro expression language, and uses type inference to assign a most general type to macro functions. For macros that we cannot parse or typecheck we do not generate any bindings.

Further, the bindings generated by `rust-bindgen` are not (easily) composable. When library `C` depends on library `P`, we can generate bindings for `P` or the union of both libraries, but we can not generate bindings for library `P`, and then use those bindings when generating bindings for `C`. By contrast, `hs-bindgen` will generate a binding *specification* for P along with its bindings, which can then be used when generating the bindings for `C`. We consider bindings specifications an important contribution of our work.

7.2 Haskell

The commonly used tools for creating Haskell bindings to C code are `hsc2hs` [21] and `c2hs` [3, 22], which we will discuss separately. We will discuss some other tools in section 7.2.3.

7.2.1 `hsc2hs`. The development of `hsc2hs` has been tightly coupled to `GHC`: it is distributed with GHC, and documented in the `GHC` User's Guide. When writing C bindings by hand, `hsc2hs` can be used to integrate small C snippets into Haskell code; we saw a small example in section 2.2.1. This can be used to query the C compiler about the offset of fields, the size of a type, etc. It is not a full C parser; the C snippets are evaluated by generating, compiling, and executing a C program, whose output forms the resulting Haskell code. While `hsc2hs` is certainly useful, it does not provide any automation: the bulk of the work is done by the programmer.

7.2.2 `c2hs`. While `c2hs` is a closer in spirit to `hs-bindgen` than `hsc2hs`, its focus is different. Indeed, its design goals [3, §2.1] explicitly *exclude* generation of Haskell function signatures from C prototypes and marshalling of compound C structures, the pair of which could reasonably be described as `hs-bindgen`'s raison d'être. It is worth taking a moment to take a closer look at this difference in perspective.

Chakravarty acknowledges that marshalling compound values is useful, but notes: "often we do not really want to marshal entire C structures to Haskell, but merely maintain a pointer to the C structure in Haskell". This is reasonable, but highly application dependent. In cases where this is true, `hs-bindgen` may indeed not be optimal. Like `hsc2hs`, `c2hs` does provide support for querying the C compiler technical details such as the offsets of fields; like `hs-bindgen`, it uses a C toolchain to parse and analyse the C header to do so.

In regards to functions, Chakravarty writes: "Although [function signature generation] seems very convenient for a couple of examples, we generally cannot derive a Haskell signature from a C prototype (a C `int` may be an `Int` or `Bool` in Haskell)". We do not disagree necessarily, but feel that there is value in exposing the function signature *without* making the choice between `Int` or `Bool`, and choosing the low-level `CInt` instead: the higher-level wrapper can then be defined in terms of the low-level function, using the full power of Haskell to do so.

Much of `c2hs`' focus (especially in the tool as it exists today [22]) is on providing a domain-specific language for making it more convenient to call C functions, automatically marshalling inputs and outputs, mapping single arguments on the Haskell side to multiple arguments on the C side, etc. This is not something we offer at all in `hs-bindgen`, though our hope is we will be able to build this on top of the low-level API we describe in this paper. We will come back to this when discussing future work (section 8.1).

7.2.3 Other Tools. C header files do not contain much meta-information; to cite one simple example, an argument of type `char*` might be intended as pointer to one character, as a pointer to a string, or indeed as an *output* parameter that the function will write to. A plethora of domain specific "interface description languages" have been developed to remedy this problem and describe C APIs at a higher-level of abstraction. Some of these have corresponding Haskell tools; this includes `fficxx` [10], `apiGen` [25], `haskell-gi` [26], `amazonka` [2], `godot-haskell` [20], `glgen` [11], `vulkan` [7] and `vulkan-api` [4]. It is an interesting question if we could extend `hs-bindgen` with hooks to enable it to combine its internal logic with this kind of externally provided meta information.

There is also some research into interfacing Haskell with other languages, such as the use of multi-parameter type classes for interfacing with object-oriented languages [16]. A survey of this research is beyond the scope of this paper.

8 Conclusions

Interfacing Haskell with C libraries is a common necessity, but the manual creation of bindings is both error-prone and laborious. This paper introduced `hs-bindgen`, a novel tool designed to automate the generation of Haskell FFI bindings directly from C header files.

Our primary contribution is a fully automated approach to binding generation. We have shown how we can address the more difficult aspects of C-Haskell interoperability, such as

name assignment for anonymous types, representing unions, and calling functions that take or return structs by value.

A key innovation is the concept of binding specifications, which provides a compositional method for using bindings for one library in bindings for another. Furthermore, we defined a domain-specific language in Haskell to represent C arithmetic expressions found in C macros, along with a corresponding type inference algorithm. This allows for the type-safe handling of (a subset of) C macros.

8.1 Future Work

The bindings generated by `hs-bindgen` are low-level. For example, consider

```
void g(char*);
```

The code that we generate has signature

```
g :: Ptr CChar → IO ()
```

Depending on the intended use of `g`, however, any of the following signatures (or indeed others) might be preferable:

```
g :: String → IO ()
g :: Text   → IO ()
g :: Char   → IO ()
g ::          IO Char
```

This kind of *high-level bindings* is not limited to functions; for example, we might think about recognising *tagged* unions (so that we can translate them to a proper Haskell sum type rather than the opaque type described in section 4.3).

We plan to tackle high-level bindings in two ways:

- Through a set of Haskell combinators that make it more convenient to (manually) write such high-level bindings *in terms of* the low-level bindings. This is similar in spirit to what `c2hs` does (section 7.2.2), but in Haskell itself, rather than in a DSL, so that we can take advantage of the full power of abstraction that Haskell offers.
- Through heuristics that can generate the high-level bindings automatically. Unlike the low-level bindings, this *will* require user input. It would be useful to add support for existing interface description languages to guide this process (see also section 7.2.3).

We structured `hs-bindgen` like a compiler, giving us a solid foundation for further development. For example, while we currently allow users to provide predicates selecting which declarations in a C header to generate bindings for, these predicates are currently quite simplistic. However, we could for example make use of the *use-decl* graph (section 3.1) to allow users to select a function "and any of the types it depends on", akin to program slicing [27]. Similarly, while experience will need to make clear over time if the macro expression language we chose (section 5) is sufficient, the foundation we have provided here will make extensions to this language relatively easy.

Acknowledgments

We would like to thank Oleg Grenrus for his contributions to the implementation of `hs-bindgen`. This work has been supported by Anduril Industries Inc.

References

[1] Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving via: or, how to turn hand-written instances into an anti-pattern. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 55–67. doi:10.1145/3242744.3242746

[2] Brendan Hay. Accessed: 2025-06-02. A comprehensive Amazon Web Services SDK for Haskell . https://github.com/brendanhay/amazonka.

[3] Manuel M. T. Chakravarty. 2000. C→HASKELL, or Yet Another Interfacing Tool. In *Implementation of Functional Languages*, Pieter Koopman and Chris Clack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 131–148. doi:10.1007/10722298_8

[4] Artem M. Chirkin. Accessed: 2025-06-02. Low-level, low-overhead Haskell bindings to the Vulkan API. https://github.com/achirkin/vulkan.

[5] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. doi:10.1145/582153.582176

[6] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. doi:10.1145/3140587.3062363

[7] Ellie Hermaszewska. Accessed: 2025-06-02. Haskell bindings for Vulkan. https://github.com/expipiplus1/vulkan.

[8] ISO. 1999. *ISO C Standard 1999*. Technical Report. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf ISO/IEC 9899:1999 draft.

[9] Mark P Jones. 1999. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*.

[10] Ian-Woo Kim. Accessed: 2025-06-02. Haskell-C++ Foreign Function Interface Generator. https://github.com/wavewave/fficxx.

[11] Edward Kmett. Accessed: 2025-06-02. Complete raw OpenGL bindings for Haskell. https://github.com/ekmett/gl.

[12] Chris Lattner. 2008. LLVM and Clang: Next Generation Compiler Technology. Presentation at BSDCan 2008: The BSD Conference.

[13] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. 2004. Extending the Haskell foreign function interface with concurrency. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/1017472.1017479

[14] Simon (Editor) Marlow. 2010. Haskell Language Report.

[15] Shayan Najd and Simon Peyton Jones. 2017. Trees that grow. *Journal of Universal Computer Science (JUCS)* 23 (January 2017), 47–62. doi:10.3217/jucs-023-01-0042

[16] André T. H. Pang and Manuel M. T. Chakravarty. 2005. Interfacing Haskell with Object-Oriented Languages. In *Implementation of Functional Languages*, Phil Trinder, Greg J. Michaelson, and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35. doi:10.1007/978-3-540-27861-0_2

[17] Simon Peyton Jones. 2001. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*. IOS Press, 47–96. https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/

[18] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern synonyms. *SIGPLAN Not.* 51, 12 (Sept. 2016), 80–91. doi:10.1145/3241625.2976013

[19] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press. doi:10.7551/mitpress/1104.003.0020

[20] SimulaVR. Accessed: 2025-06-02. Haskell bindings for GdNative. https://github.com/SimulaVR/godot-haskell.

[21] The GHC Team. Accessed: 2025-05-30. Writing Haskell interfaces to C code: `hsc2hs`. https://downloads.haskell.org/ghc/latest/docs/users_guide/utils.html.

[22] The `c2hs` Team. Accessed: 2025-05-30. `c2hs` is a pre-processor for Haskell FFI bindings to C libraries. https://github.com/haskell/c2hs.

[23] The `rust-bindgen` Team. Accessed: 2025-05-30. `bindgen` automatically generates Rust FFI bindings to C and C++ libraries. https://rust-lang.github.io/rust-bindgen/.

[24] The LLVM Project. Accessed: 2025-06-02. `libclang` (part of the LLVM Project). https://clang.llvm.org/docs/LibClang.html.

[25] The `gtk2hs` Team. Accessed: 2025-06-02. Semi-automatically generate binding modules complete with Haddock format documentation for all GTK modules. https://github.com/gtk2hs/gtk2hs/tree/master/tools/apiGen.

[26] The `haskell-gi` Team. Accessed: 2025-06-02. Generate Haskell bindings for GObject-Introspection capable libraries . https://github.com/haskell-gi/haskell-gi.

[27] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 439–449. doi:10.1109/TSE.1984.5010248

Received 2025-06-08; accepted 2025-07-17