C → Haskell, or Yet Another Interfacing Tool[⋆]

Manuel M. T. Chakravarty

School of Computer Science and Engineering
University of New South Wales, Sydney
`chak@cse.unsw.edu.au`
`www.cse.unsw.edu.au/~chak/`

Abstract This paper discusses a new method for typed functional languages to access libraries written in a lower-level language. More specifically, it introduces an interfacing tool that eases Haskell access to C libraries. The tool obtains information about the C data type definitions and function signatures by analysing the C header files of the library. It uses this information to compute the missing details in the template of a Haskell module that implements a Haskell binding to the C library. Hooks embedded in the binding file signal where, which, and how C objects are accessed from Haskell. The Haskell code in the binding file determines Haskell type signatures and marshaling details. The approach is lightweight and does not require an extra interface description language.

1 Introduction

The complexity of modern software environments frequently requires interoperability between software components that are coded in different programming languages. Thus, the lack of access to libraries and components implemented in another language severely restricts the scope of a programming language. It is, hence, not surprising that a number of methods have been proposed for integrating foreign code into functional programs, and in particular, for using imperative code with Haskell [13]—approaches range from simple inline-calls of C routines [12] to sophisticated COM support [6,5].

Interface generators in these previous approaches relied on an annotated interface in one of the two languages or used a dedicated language for the interface specification. Another interesting, yet unexplored approach is the *combined* use of interface specifications in both languages for expressing complementary information. Interestingly, this is especially attractive for the most frequent application of a *foreign language interface (FLI)*: access to existing libraries implemented in the language C. Interfaces to operating system services, as well as to system-level and higher-level libraries are usually available from C, which, due to its simplicity, makes adapting the interface to another language relatively

[⋆] This work was conducted, in large part, while working at the Institute of Information Sciences and Electronics, University of Tsukuba, Japan.

easy. The method presented here uses existing C header files in conjunction with additional high-level interface information—expressed in Haskell plus a small set of *binding hooks*—to generate a Haskell binding to a C library.

Experience with the interfacing tool C→HS shows that the outlined approach is practically feasible. By reading verbatim headers of a C library, C→HS has exactly the same information as the C compiler when translating C code using the same library. Therefore, it has sufficient knowledge to call the library's C functions and to read from or write to components of data structures; it can even access C structures while doing all of the address arithmetic in Haskell.

In summary, the original features of the presented approach are the following:

– Tool support based on the simultaneous and complementary use of an interface specification in the foreign and the host language.
– C→HS was the first tool using pristine C header files for Haskell access to C libraries.[1]
– Supports consistency checking of the Haskell interface specification against existing headers of the C library.
– Simple binding hooks and lightweight tool support—no complex interface language is required.

The tool C→HS is implemented and available for public use.[2]

The remainder of the paper is structured as follows. Section 2 introduces the concepts underlying the present approach and discusses related work. Section 3 details the binding hooks supported by C→HS. Section 4 describes the marshaling support provided by the library `C2HS`. Section 5 outlines the implementation and current experience with C→HS. Finally, Section 6 concludes.

2 Concepts and Related Work

Symmetric approaches to language interoperability like IDL/Corba [10] define interfaces in a special-purpose interface language and treat the interfaced languages as peers. In contrast, the present work is optimised for an *asymmetric* situation where libraries implemented in a lower-level language (called the *foreign* language) are accessed from a typed, high-level functional language (called the *host* language). In this asymmetric setting, the interface information provided by the foreign language library is generally insufficient for determining a corresponding host language interface. Assertions and invariants that are only informally specified in the foreign interface will have to be formalised for the host interface—a task that clearly requires user intervention. A second aspect of the asymmetry of this situation is that the foreign library including its interface specification usually exist before the host language interface is implemented; and furthermore, the foreign interface is usually developed further independent of and concurrent to the development of the host interface. This situation calls for an

[1] Apart from an early version of GreenCard that was never released.
[2] C→HS web page: `http://www.cse.unsw.edu.au/~chak/haskell/c2hs/`

approach where (a) the existing foreign interface is reused as far as possible and (b) the consistency between the two interfaces is checked automatically.

We achieve this by employing a tool that uses a foreign and a host language interface in concert—an approach that, to my knowledge, was not tried before. Let us call it the *dual language* approach (as opposed to the use of one special-purpose interface language or a restriction to the host language).

In comparison to symmetric approaches, a dual language approach trades generality for simplicity. In particular, a symmetric approach requires an extra language to describe interfaces (such as OMG IDL). From such a generic interface, a tool generates a host language interface, which then often requires another layer of code in the host language to provide a convenient interface for the library user. In contrast, in a dual language approach where the host interface is directly expressed in the host language, there is more freedom for the interface designer to directly produce an interface suitable for the library user.

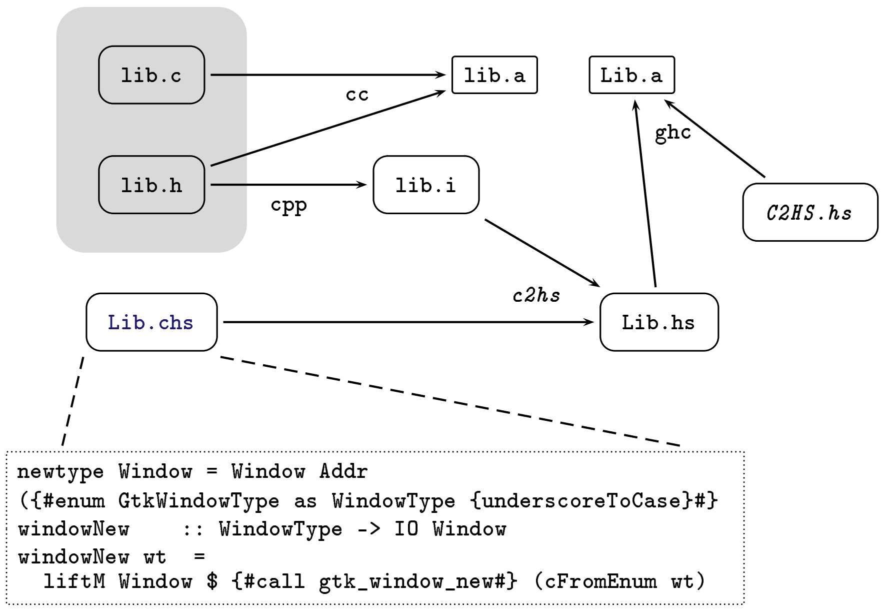
It is interesting to see how a dual language approach fits into the taxonomy given in [6, Section 2]. This paper makes an argument for adopting a third language, namely IDL, on the grounds that neither a specification that is exclusively in the host language (Haskell) nor one that is exclusively in the foreign language (C) is sufficient to determine the complete interface. It neglects, however, the possibility of using a host language specification in concert with a foreign language specification—which is particularly appealing if the foreign language specification does already exist and is maintained by the author of the library, which is usually the case when interfacing C libraries.

2.1 A Dual Language Tool for C and Haskell

In the following, we shall concentrate on the specific case of using C libraries from Haskell with the tool C→HS. This focus is justified as by its very nature, language interoperability has to handle a significant amount of language-specific technical detail, which makes a language-independent presentation tedious. In addition, C is currently the most popular low-level language; hence, most interesting libraries have a C interface. Despite our focus on Haskell, the discussed approach is appropriate for any typed functional language with basic FFI support.

C→HS generates Haskell code that makes use of the *foreign function interface (FFI)* [15,5] currently provided by the Glasgow Haskell Compiler (GHC) and soon to be supported by Hugs.[3] The FFI provides the basic functionality of calling C from Haskell, and vice versa, as well as the ability to pass primitive data types, such as integers, characters, floats, and raw addresses. Building on these facilities, C→HS automates the recurring tasks of defining FFI signatures of C functions and marshaling user-defined data types between the two languages. In other words, the full FLI consists of two layers: (a) basic runtime support for simple inter-language calls by the Haskell system's FFI and (b) tool support for the more complex aspects of data marshaling and representation of user-defined data structures by C→HS. The architecture of the latter is displayed in Figure 1.

[3] In my opinion, GHC's FFI is a good candidate for a standard Haskell FFI.



```
newtype Window = Window Addr
({#enum GtkWindowType as WindowType {underscoreToCase}#}
windowNew    :: WindowType -> IO Window
windowNew wt  =
  liftM Window $ {#call gtk_window_new#} (cFromEnum wt)
```

Figure1. C→HS tool architecture

The C library source (files `lib.h` and `lib.c`) in the grey box usually exists before the Haskell binding is implemented and will in most cases be concurrently and independently developed further. The header, `lib.h`, of the C library contains all C-level interface information. It is complemented by a C→HS *binding module,* `Lib.chs`, which contains the Haskell-side interface and marshaling instructions; the binding module specifies how the two interfaces inter-operate. The latter is achieved by virtue of *binding hooks*, which are expressions enclosed in `{#-#}` pairs, and marshaling instructions, which are denoted in plain Haskell. The figure contains a fragment of binding code including binding hooks that reference objects defined in the C header file, in this case `GtkWindowType` and `gtk_window_new`. In the binding module, all but the binding hooks is plain Haskell that either defines the Haskell interface (as for example, the type signature of `windowNew`) or details marshaling procedures (as for example, `cFromEnum`). The latter mostly consist of the use of marshaling routines that are predefined in the library `C2HS.hs`.

The interface generator, denoted by `c2hs`, reads the binding module together with the C header file, which it first pipes through the C pre-processor `cpp`. By exploiting the cross references from binding hooks to C objects and the corresponding definitions in the C header file, the interface generator replaces all binding hooks by plain Haskell code that exercises the FFI of the underlying Haskell system. In the figure, the resulting Haskell module is `Lib.hs`.; it makes use of the marshaling library `C2HS.hs`, which comes with C→HS.

Overall, we expect the following functionality from a tool like C→HS:

- Conversion of C enumeration types into Haskell
- Conversion of basic C types into corresponding Haskell types
- Generation of all required FFI declarations from C function prototypes
- Direct access to members of C structures from Haskell
- Library support for parameter and result marshaling
- Consistency check between the C and the Haskell interface

In contrast, we do not expect the following two features:

1. Generation of Haskell function signatures from C prototypes
2. Marshaling of compound C structures to Haskell values

On first sight, it may seem surprising that these two features are not included, but a closer look reveals that they are of secondary importance. Although the first feature 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). The second feature is more generally useful; however, often we do not really want to marshal entire C structures to Haskell, but merely maintain a pointer to the C structure in Haskell. The evaluation of the usefulness of the second feature is an interesting candidate for future work.

In summary, the use of pristine C header files for defining the low-level details of data representations simplifies interface specification significantly—no new interface language needs to be learned and the C header files are always up to date with the latest version of the C code, which allows the tool to check the consistency of the C header and Haskell interface. This, together with marshaling specified in plain Haskell and utilising a Haskell marshaling library, keeps the tool simple and flexible. The cost is a restriction of the foreign language to C.

2.2 Related Work

The absolute minimum requirement for an FLI is the ability to call out to C and pass arguments of primitive type (such as, integers and characters). If the interface is more sophisticated, it allows call backs from C to Haskell and contains some additional inter-language support from the storage manager [8]. As already stated, throughout this paper, we call such basic support a *foreign function interface (FFI)*—it does not constitute a full language binding, but allows merely for basic inter-language function calls. The first proposal for this kind of functionality in the Haskell compiler GHC was `ccall` [12], which made use of GHC's ability to compile Haskell to C. Recently, `ccall` was superseded by a new FFI [15, Section 3] that fits better into the base language and is more powerful as it allows call backs and functions to be exported and imported dynamically [5].

Green Card [9] is a tool that implements a full FLI on top of the basic FFI of some Haskell systems (GHC, Hugs, and NHC). Its input is a specification of Haskell signatures for foreign procedures augmented with declarations specifying various low-level details (data layout, storage allocation policies, etc.) that cannot be inferred from the signature. Its main disadvantage is the conceptual complexity of its interface specification, which arises from the need to invent a language that specifies all the low-level information that is not naturally present in a Haskell signature. As a result, part of the information that, when interfacing to C, is already contained in the C header file has to be re-coded in Green Card's own language and, of course, it has to be updated for new versions of the accessed C library. The goal behind Green Card and C→HS is the same; the essential difference in method is that C→HS reads the C header file to obtain the C-side interface and that it uses plain Haskell to express marshaling, instead of specialised *Data Interface Schemes.* Interestingly, the initial version of Green Card analysed C header files instead of using its own language (much like the tool SWIG discussed below), but this approach was later abandoned.

H/Direct [6] is a compiler for *Interface Definition Languages (IDLs)* that generates Haskell bindings from IDL interfaces. H/Direct can be used to generate a Haskell binding for a COM component or for a conventional C library. In addition, H/Direct supports the implementation of COM components in Haskell [5]. The special appeal of this symmetric approach is the use of a standardised interface language and the ability to mix Haskell code with code written in any other language that has COM IDL support—due to the proprietary nature of COM, the latter is currently restricted to the Windows platform; this restriction could be lifted by extending H/Direct to cover the open CORBA standard. Together with the generality, H/Direct also inherits the disadvantages of a symmetric approach: The programmer has to re-code and regularly update information already contained in existing C header files of a library; and furthermore, there is the additional overhead of learning a dedicated interface language.

Two methods were suggested for alleviating these disadvantages: first, automatic generation of an IDL specification from C headers, and second, direct processing of existing C headers by H/Direct. In both cases, the programmer has to manually supply additional information. In the first case, the programmer post-processes the generated IDL specification, and in the second case, the programmer supplies an additional file that contains annotations to the plain C declarations. The main conceptual difference between these methods and C→HS is that H/Direct generates a fixed Haskell interface from the input, whereas C→HS allows the programmer to determine the Haskell interface. For simple interfaces, the fixed output may be sufficient, but for more complicated interfaces (like GTK+ [7,3], see the marshaling in Section 4.1), H/Direct's approach requires another layer of Haskell code to provide a suitable interface for the user.

A second difference is that H/Direct marshals entire C structures to Haskell, whereas C→HS allows access to individual fields of C structures without marshaling the whole structure. Again, for simple structures like time descriptors or geometric figures it is usually more convenient to marshal them in their entirety, but in the case of complicated structures like widget representations individual access to structure members is preferable (see Section 3.5).

Finally, SWIG [1] should be mentioned—although, there is no Haskell support at the time of writing. SWIG generates a binding to C or C++ code from an annotated C header file (or C++ class definition). SWIG works well for untyped scripting languages, such as Tcl, Python, Perl, and Scheme, or C-like languages, such as Java, but the problem with typed functional languages is that the information in the C header file is usually not sufficient for determining the interface on the functional-language side. As a result, additional information has to be included into the C header file, which leads to maintenance overhead when new versions of an interfaced C library appear. This is in contrast to the use of pristine C header files complemented by a separate high-level interface specification as favoured in C→HS.

The above discussion largely centres around the various FLIs available for the Glasgow Haskell Compiler. This is not to say that other Haskell implementations do not have good FLI support, but GHC seems to have enjoyed the largest variety of FLIs. More importantly, there does not seem to be any approach to FLI design in any of the other systems that was not also tried in the context of GHC. The same holds for other functional languages (whose inclusion was prevented by space considerations).

3 Interface Specification

As discussed in Section 2.1, binding hooks establish the link between objects defined in the C header and definitions in the Haskell binding module (i.e., the `.chs` file). The tool C→HS replaces these binding hooks by interfacing code computed from the C declarations. Binding hooks are enclosed between `{#` and `#}` and start with a keyword determining the type of hook, of which there are the following six,

- `context`: Specifies binding information used throughout the module
- `type`: Computes the Haskell representation of a C type
- `enum`: Maps a C enumeration to a corresponding Haskell data type definition
- `call`: Calls out to a C function
- `get` and `set`: Allows to read and write a components of C structures

A context hook, if present, has to be the first hook occurring in a module. The following subsections discuss the structure and usage of binding hooks; Appendix A contains a formal definition of their grammar.

3.1 Context hooks

A context hook may define the name of a dynamic library that has to be loaded before any of the external C functions may be invoked and it may define a prefix to use on all C identifiers. For example,

```
{#context lib="gtk" prefix="gtk"#}
```

states that the dynamic library called `gtk` (e.g., `libgtk.so` on ELF based Linux systems) should be loaded and that the prefix `gtk` may be safely omitted from

C identifiers. All identifiers in the GTK+ library start with `gtk` or `Gtk`—in a kind of poor man's attempt at module name spaces in C. The above prefix declaration allows us to refer to these identifiers while omitting the prefix; so, we can write `WindowType` instead `GtkWindowType`. Matching the prefix is case insensitive and any underscore characters between the prefix and the stem of the identifier are also removed, so that we can also use `new_window` instead of `gtk_new_window`. Where this leads to ambiguity, the full C name can still be used and has priority. To simplify the presentation, the following examples do not make use of the prefix feature.

3.2 Type hooks

C→HS's marshaling library defines Haskell counterparts for the various primitive C types. A type hook, given a C type name, is expanded by C→HS into the appropriate Haskell type. For example, in

```
type GInt   = {#type gint#}
type GFloat = {#type gfloat#}
```

the type `gint` may be defined to represent `int` or `long int` in the C header file; the hook `{#type gint#}` is then replaced by `CInt` or `CLInt`, respectively, which will have the same representation as a value expected or returned by a C function using the corresponding C type.

3.3 Enumeration hooks

An enumeration hook converts a C `enum` declaration into a Haskell data type. It contains the name of the declaration in C and, optionally, a different name for the Haskell declaration. Furthermore, a translation table for mapping the names of constructors may be defined. For example, given the C declaration

```
typedef enum
{
  GTK_WINDOW_TOPLEVEL,
  GTK_WINDOW_DIALOG,
  GTK_WINDOW_POPUP
} GtkWindowType;
```

and the hook

```
{#enum GtkWindowType as WindowType {underscoreToCase}#}
```

C→HS generates

```
data WindowType = GtkWindowToplevel
                | GtkWindowDialog
                | GtkWindowPopup
               deriving (Enum)
```

The C identifier mentioned in the hook can be a type name referring to the enumeration, as in the example, or it can be the tag of the `enum` declaration itself. Optionally, it is possible to give the Haskell definition a different name than the C type—in the example, `WindowType`. The last argument of the hook, enclosed in braces, is called a *translation table*. When it contains the item `underscoreToCase`, it specifies that C's common `this_is_an_identifier` (or `A_MACRO`) notation is to be replaced in Haskell by `ThisIsAnIdentifier` (or `AMacro`). Whether or not `underscoreToCase` is used, explicit translations from C into Haskell names can be specified in the form *cname* `as` *HSName* and always take priority over the `underscoreToCase` translation rule.

In the above example, the values assigned by C and Haskell to the corresponding enumerators are the same.[4] As C allows us to explicitly define the values of an enumerator, whenever any of the values is explicitly given, C→HS generates a customised `Enum` instance for the new data type, instead of using a derived instance. This guarantees that, whenever the C library is updated, re-generating the binding with C→HS will pick up all changes in enumerations.

C libraries occasionally define enumerations by a set of macro pre-processor `#define` statements, instead of using an `enum` declaration. C→HS also provides support for such enumerations. For example, given

```
#define GL_CLAMP          0x2900
#define GL_REPEAT         0x2901
#define GL_CLAMP_TO_EDGE 0x812F
```

from the OpenGL header, we can use a hook like

```
{#enum define Wrapping {GL_CLAMP         as Clamp,
                        GL_CLAMP_TO_EDGE as ClampToEdge,
                        GL_REPEAT        as Repeat}#}
```

to generate a corresponding Haskell data type definition including an explicit `Enum` class instance, which associates the specified enumeration values. C→HS implements this variant of the enumeration hooks by generating a C enum declaration of the form

```
enum Wrapping {
  Clamp       = GL_CLAMP,
  ClampToEdge = GL_CLAMP_TO_EDGE,
  Repeat      = GL_REPEAT};
```

and then processing it as any other enumeration hook—including pre-processing the definition with the C pre-processor to resolve the macro definitions.

3.4 Call hooks

Call hooks specify calls to C functions. For example,

[4] We understand the value of an enumerator in Haskell to be the integer value associated with it by virtue of the `Enum` class's `fromEnum` method.

```
{#call gtk_radio_button_new#}
```

calls the function `gtk_radio_button_new`. GHC's FFI requires that each external function has a foreign declaration. This declaration is automatically added to the module by C→HS. If the function is defined as

```
GtkWidget* gtk_radio_button_new (GSList *group);
```

in the C header file, C→HS produces the following declaration:

```
foreign import ccall "libgtk.so" "gtk_radio_button_new"
  gtk_radio_button_new :: Addr -> IO Addr
```

We assume here that the call is in the same module as our previous example of a context hook; therefore, the dynamic library `libgtk.so` (assuming that we are compiling for an ELF-based Linux system or Solaris) is added to the declaration. In this declaration, the identifier exclosed in quotes specifies the name of the C function and the following identifier is the bound Haskell name. By default they are equal, but optionally, an alternative name can be given for the Haskell object (this is necessary when, e.g., the C name would not constitute a legal function identifier in Haskell). Moreover, C→HS infers the type used in the foreign declaration from the function prototype in the C header file. As the argument and result of `gtk_radio_button_new` are pointers, `Addr` is used on the Haskell side. So, there is clearly some more marshaling required; we shall come back to this point in Subsection 4.

By default, the result is returned in the `IO` monad, as the C function may have side effects. If this is not the case, the attribute `fun` can be given in the call hook. For example, using `{#call fun sin#}`, we get the following declaration:

```
foreign import ccall sin :: Float -> Float
```

Furthermore, the attribute `unsafe` can be added to C routines that cannot reenter the Haskell runtime system via a call back; this corresponds to the same flag in GHC's FFI.[5]

3.5 Get and set hooks

Get and set hooks are related and have, apart from the different keyword, the same syntax. They allow reading from and writing to the members of C structures. The accessed member is selected by the same access path expression that would be used in C: It consists of the name of a structure followed by a series of structure reference (*s*`.`*m* or *s*`->`*m*) and indirection (`*`*e*) operations. Given a C header containing

[5] The attribute is called `unsafe`, as the call *does not have to play safe.* This naming scheme is taken from GHC's FFI.

```
typedef struct _GtkAdjustment GtkAdjustment;
struct _GtkAdjustment {
  GtkData data;
  gfloat  lower;
  ... /* rest omitted */
};
```

the binding file might, for example, contain a hook

```
{#get GtkAdjustment.lower#}
```

By reading the C header, C→HS has complete information about the layout of structure fields, and thus, there is no need to make a foreign function call to access structure fields—the necessary address arithmetic can be performed in Haskell. This can significantly speed up access compared to FLIs where the information from the C header file is not available. As with enumeration hooks, it is sufficient to re-run C→HS to adjust the address arithmetic of get and set hooks when the C library is updated.

Get and set hooks expand to functions of type `Addr -> IO` *res* and `Addr -> ` *res* ` -> IO ()`, respectively, where *res* is the primitive type computed by C→HS for the accessed structure member from its definition in the C header file. Marshaling between this primitive type and other Haskell types is discussed in the next subsection.

C→HS allows for some flexibility in the way a hook may refer to a structure definition. Above, we used a type name associated with the structure via a C `typedef` declaration; but we could also have used the name of the tag of the structure declaration, as in `{#get _GtkAdjustment.lower#}`. Finally, if there had been a type name for a pointer to the `_GtkAdjustment` structure, we could also have used that. This flexibility is important, as C libraries adopt varying conventions as to how they define structures and we want to avoid editing the C header to include a definition that can be used by C→HS.

As already mentioned in Section 2.2, for complex data structures (like GTK+'s widgets), it is often preferable to access individual structure members instead of marshaling complete structures. For example, in the case of `GtkAdjustment`, only a couple of scalar members (such as `lower`) are of interest to the application program, but the structure itself is rather large and part of an even larger linked widget tree.

4 The Marshaling Library

When using call, set, and get hooks, the argument and result types are those primitive types that are directly supported by the FFI—e.g., in the example where we called `gtk_radio_button_new`, the result of the call was `Addr`; although, according to the C header (repeated here),

```
GtkWidget* gtk_radio_button_new (GSList *group);
```

the function returns a `GtkWidget*`. There is obviously a gap that has to be filled. It is the task of the library `C2HS`, which contains routines that handle storage allocation, convert Haskell lists into C arrays, handle in-out parameters, and so on. However, the library provides only the basic blocks, which have to be composed by the programmer to match both the requirements specified in the C API and the necessities of the Haskell interface. The library covers the standard cases; when marshaling gets more complex, the programmer may have to define some additional routines. This is not unlike the pre-defined and the user-defined data interface schemes of Green Card [9], but entirely coded in plain Haskell.

4.1 Library-specific Marshaling

In the case of the GTK+ library, a radio button has the C type `GtkRadioButton`, which in GTK+'s widget hierarchy is an (indirect) instance of `GtkWidget`. Nevertheless, the C header file says that `gtk_radio_button_new` returns a pointer to `GtkWidget`, not `GtkRadioButton`. This is perfectly ok, as GTK+ implements widget instances by means of C structures where the initial fields are identical (i.e., the C pointer is the same; it is only a matter of how many fields are accessible). There is, however, no way to represent this in Haskell. Therefore, the Haskell interface defines

```
newtype RadioButton = RadioButton Addr
```

and uses type classes to represent the widget hierarchy. As a result, the marshaling of the result of `gtk_radio_button_new` has to be explicitly specified. Moreover (for reasons rooted in GTK+'s specification), the argument of type `GSList *group` is of no use in the Haskell interface. Overall, we use the following definition in the binding file:

```
radioButtonNew :: IO RadioButton
radioButtonNew  =
  liftM RadioButton $ {#call gtk_radio_button_new#} nullAddr
```

The function `liftM` is part of Haskell's standard library; it applies a function to the result in a monad. The constant `nullAddr` is part of the FFI library `Addr` and contains a null pointer.

The important point to notice here is that complex libraries are built around conventions that usually are only informally specified in the API documentation and that are not at all reflected in the formal C interface specification. No tool can free the interface programmer from the burden of designing appropriate marshaling routines in these cases; moreover, an elegant mapping of these API constraints into the completely different type system of Haskell can be the most challenging part of the whole implementation of a Haskell binding. The design decision made for C$\rightarrow$HS at this point is to denote all marshaling in Haskell, so that the programmer has the full expressive power and abstraction facilities of Haskell at hand to solve this task.

4.2 Standard Marshaling

The library C2HS, which comes with C→HS, provides a range of routines, which cover the common marshaling requirements, such as bit-size adjustment of primitive types, marshaling of lists, handling of in/out parameters, and common exception handling. Unfortunately, a complete discussion of the library is out of the scope of this paper; thus, we will only have a look at two typical examples.

Conversion of primitive types. For each primitive Haskell type (like `Int`, `Bool`, etc.), C2HS provides a conversion class (`IntConv`, `BoolConv`, etc.), which maps the Haskell representation to one of possibly many C representations and vice versa.

For example, in the case of the get hook applied to the struct `GtkAdjustment` in Subsection 3.5, we have to provide a pointer to a `GtkAdjustment` widget structure as an argument to the get hook and marshal the resulting value of C type `gfloat` to Haskell. We implement the latter using the member function `cToFloat` from the class `FloatConv`.

```
newtype Adjustment = Adjustment Addr

adjustmentGetLower :: Adjustment -> IO Float
adjustmentGetLower (Adjustment adj) =
  liftM cToFloat $ {#get GtkAdjustment.lower#} adj
```

The interaction between the interface generator and Haskell's overloading mechanism is crucial here. As explained in Subsection 3.5, the get hook will expand to a function of type `Addr -> IO` *res*, where *res* is the Haskell type corresponding to the concrete type of the C typedef `gfloat`—as computed by the interface generator from the C header. For the overloaded function `cToFloat`, the Haskell compiler will select the instance matching *res* `-> Float`. In other words, every instance of `FloatConv`, for a type t, provides marshaling routines between t and `Float`. This allows us to write generic marshaling code without exact knowledge of the types inferred by C→HS from the C header files. This is of particular importance for integer types, which come in flavours of varying bit size.

Compound structures. `GtkEntry` is again a widget (a one line text field that can be edited), and the routine

```
void gtk_entry_set_text (      GtkEntry *entry,
                         const gchar    *text);
```

requires in its second argument marshaling of a `String` from Haskell to C (there is no direct support for passing lists in GHC's FFI). C2HS helps here by providing support for storage allocation and representation conversion for passing lists of values between Haskell and C. The classes `ToAddr` and `FromAddr` contain methods to convert Haskell structures to addresses referencing a C representation of the given structure. In particular, `stdAddr` converts each type for which there is an instance of `ToAddr` into the type's C representation.

```
newtype Entry = Entry Addr

entrySetText :: Entry -> String -> IO ()
entrySetText (Entry ety) text =
  {#call gtk_entry_set_text unsafe#} ety `marsh1_`
                                          (stdAddr text :> free)
```

Each member of the family of functions `marsh`*n* marshals *n* arguments from Haskell to C and back. The conversion to C is specified to the left of `:>` and the reverse direction to its right. The routine `free` simply deallocates the memory area used for marshaling. The `marsh`*n*`_` variants of these functions discard the values returned by the C routines. In addition to marshaling strings to and from C, these routines can generally be used to handle in/out arguments.

5 Implementation and Application of C→HS

The interface generator C→HS is already implemented and publicly available (the link was given in Section 1). The following provides a rough overview over the current implementation and reports on first experiences with the approach to interfacing described in this paper.

5.1 Implementation

The interface generator is entirely implemented in Haskell and based on the *Compiler Toolkit* [2]. It makes heavy use of the toolkit's self-optimising parser and lexer libraries [14,4]; in particular, a full lexer and parser for C header files is included. The Haskell binding modules are, however, not fully analysed. The lexer makes use of the lexer library's meta actions to distinguish whether it reads characters belonging to standard Haskell code or to a binding hook. Haskell code is simply collected for subsequent copying into the generated plain Haskell module, whereas binding hooks are fully decomposed and parsed according to the rules given in Appendix A.

After the header and the binding module have been read, C→HS converts all binding hooks into plain Haskell code, and finally, outputs the resulting Haskell module. During expansion of the hooks, the definitions in the C header file referenced by binding hooks are analysed as far as this is required to produce binding code—however, in general, the tool does not recognise all errors in C definitions and does not analyse definitions that are not directly referred to in some binding hooks; thus, the header file should already have been checked for errors by compiling the C library with a standard C compiler (if, however, errors are detected by the binding tool, they are properly reported). This lazy strategy of analysing the C definitions makes a lot of sense when considering that a pre-processed C header file includes the definitions of all headers that it directly or indirectly includes—in the case of the main header `gtk.h` of the GTK+ library, the C pre-processor generates a file of over 230kB (this, however, contains a significant amount of white space).

The analysis performed on C declarations is standard in the sense that it is a subset of the semantic analysis performed in a normal C compiler. Hence, a detailed discussion would not lead to any new insights. Details of how this information is used to expand the various forms of binding hooks, while interesting, would exceed the space limitations placed on this paper. However, C→HS's source code and documentation is freely available and constitutes the ultimate reference for all questions about the implementation.

5.2 Application

The idea for C→HS arose in the context of the implementation of a Haskell binding [3] for the GTK+ graphical user interface toolkit [7,11]. Naturally, the GUI toolkit is an important application of the binding generator. The Haskell binding of GTK+ was originally coded directly on top of GHC's new FFI and is currently rewritten to use C→HS. The resulting code is more compact and cross checking consistency with the C headers is significantly improved by C→HS.

The libraries of the Gnome [11] desktop project include a C library implementing the HTTP 1.1 protocol, called `ghttp`. A Haskell binding for `ghttp` was implemented as a first application of C→HS to a library, which is structured differently than GTK+. The library is relatively small with four enumeration types, one structure declaration, and 24 functions that have to be interfaced. The Haskell binding module `Ghttp` is 153 lines (this excludes empty lines and lines containing only comments) and is expanded by C→HS to a 276 line plain Haskell module. The latter is almost exactly the code that would have been written manually by a programmer using GHC's FFI. Thus, the use of C→HS reduced the coding effort, in terms of lines of code, by 45% (assuming that even when the binding had been coded manually, the marshaling library `C2HS` would have been available). Judging from the experience with GTK+, the amount of saved work is, however, smaller when the library and its interface is more complex, because there is more library-specific marshaling required.

6 Conclusions

In many respects, C→HS builds on the lessons learned from Green Card. It avoids the complexity of a new interface specification language by re-using existing C interface specifications and by replacing data interface schemata with marshaling coded in plain Haskell. The latter is simplified by providing a comprehensive marshaling library that covers common marshaling situations. Green Card pioneered many of the basic concepts of C-with-Haskell interfacing and C→HS definitely profited from this.

C→HS demonstrates the viability of dual language tools, i.e., it demonstrates that interface specifications in the two languages concerned can be jointly used to bridge the gap between languages as different as C and Haskell. The advantages of this approach are that the binding hooks necessary to cross-reference complementary definitions in the two interfaces are significantly simpler than dedicated interface languages and existing library interfaces can be reused in their pristine form. The latter saves work and allows consistency checks between the two interfaces—this is particularly important when the interfaced library already exists and is independently developed further. H/Direct's recent support for C headers is another indication for the attractiveness of this approach.

C→HS has so far proved valuable in developing a Haskell binding to the GTK+/Gnome libraries [11,3]. More experience is, however, required for a thorough evaluation.

In my experience, GHC's new FFI provides a very nice basic interface to foreign functions in Haskell. Thus, I would highly recommend its inclusion into the next Haskell standard. After all, Haskell's value as a general purpose language is severely limited without good foreign language support—such an important aspect of the language should definitely be standardised!

Future Work. The functionality of C→HS was largely motivated by the requirements of GTK+. As the latter is a large and complex system, it is to be expected that most of the interesting problems in a binding are encountered throughout the implementation of a Haskell binding for GTK+. However, the conventions used in different C libraries can vary significantly, so further extensions may become attractive with added experience; furthermore, C→HS allows the programmer direct access to the FFI of the Haskell system, where this seems more appropriate or where additional functionality is required. In fact, there are already a couple of areas in which extensions seem desirable: (1) support for accessing global C variables is needed; (2) the tool should help generating the signatures for call back routines; (3) sometimes the marshaling code for functions might be generated automatically; (4) better type safety for address arguments and results; and (5) marshaling of complete structures, as in H/Direct, is sometimes convenient and currently has to be done in a mixture of set/get hooks and dedicated Haskell code.

Regarding Point (3), for functions with simple signatures, the marshaling code is often obvious and could be generated automatically. This would make the code a bit more concise and easier to maintain. Regarding Point (4), all pointer arguments of C functions are mapped to type `Addr` in Haskell, which makes it impossible for the Haskell compiler to recognise errors, such as, exchanged arguments. It would be interesting to use a variant of `Addr` that gets an additional type argument, namely, the name of the type referred to by the address. Even for abstract types, a type tag can be generated using a Haskell newtype declaration. This would allow C→HS to generate different instances of the parametrised `Addr` type for different C types, which would probably significantly enhance the consistency checks between the C and the Haskell interface.

Acknowledgements. I am grateful to Michael Hobbs for our discussions about the GTK+ binding; they were part of the motivation for starting to think about C→HS. Furthermore, I like to thank Gabriele Keller, Sven Panne, Alastair Reid, Michael Weber, and the anonymous referees for their helpful comments and suggestions.

References

1. David M. Beazley. SWIG and automated C/C++ scripting. *Dr. Dobb's Journal*, February 1998.
2. Manuel M. T. Chakravarty. A compiler toolkit in Haskell. `http://www.cse.unsw.edu.au/~chak/haskell/ctk/`, 1999.
3. Manuel M. T. Chakravarty. A GTK+ binding for Haskell. `http://www.cse.unsw..edu.au/~chak/haskell/gtk/`, 1999.
4. Manuel M. T. Chakravarty. Lazy lexing is fast. In Aart Middeldorp and Taisuke Sato, editors, *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
5. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming.* ACM Press, 1999.
6. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 153–162. ACM Press, 1998.
7. Eric Harlow. *Developing Linux Applications with GTK+ and GDK.* New Riders Publishing, 1999.
8. Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stables names in Haskell. In *Proceedings of the International Conference on Functional Programming*, 1999.
9. T. Nordin, Simon L. Peyton Jones, and Alastair Reid. Green Card: a foreign-language interface for Haskell. In *Proceedings of the Haskell Workshop*, 1997.
10. The common object request broker: Archictecture and specification, rev. 2.2. Technical report, Object Management Group, Framingham, MA, 1998.
11. Havoc Pennington. *GTK+/Gnome Application Development.* New Riders Publishing, 1999.
12. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages*, pages 71–84. ACM Press, 1993.
13. Haskell 98: A non-strict, purely functional language. `http://haskell.org/definition/`, February 1999.
14. S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer-Verlag, 1996.
15. The Haskell FFI Team. A primitive foreign function interface. `http://www.dcs.gla.ac.uk/fp/software/hdirect/ffi.html`, 1998.

A The Grammar of Binding Hooks

The grammar of binding hooks appearing in Haskell binding modules is formally defined in Figure 2. Here *string* denotes a string literal and *ident* a Haskell-style variable or constructor identifier (this lexically includes C identifiers). If `underscoreToCase` occurs in a translation table, it must be the first entry.

hook	$\rightarrow$	`{#` *inner* `#}`	(binding hook)
inner	$\rightarrow$	`context` *ctxopts*	(set context)
	$\mid$	`type` *ident*	(type name)
	$\mid$	`enum` *idalias trans*	(map enumeration)
	$\mid$	`call` [`fun`] [`unsafe`] *idalias*	(call a C function)
	$\mid$	`get` *apath*	(read a structure member)
	$\mid$	`set` *apath*	(write to a structure member)
ctxopts	$\rightarrow$	[`lib =` *string*] [`prefix =` *string*]	(context options)
idalias	$\rightarrow$	*ident* [`as` *ident*]	(possibly renamed identifier)
apath	$\rightarrow$	*ident*	(access path identifier)
	$\mid$	`*` *apath*	(dereferencing)
	$\mid$	*apath* `.` *ident*	(member selection)
	$\mid$	*apath* `->` *ident*	(indirect member selection)
trans	$\rightarrow$	`{` $alias_1$ `,` ... `,` $alias_n$ `}`	(translation table, $n \geq 0$)
alias	$\rightarrow$	`underscoreToCase`	(standard mapping)
	$\mid$	*ident* `as` *ident*	(associate two identifiers)

Figure2. Grammar of binding hooks.

Generally, it should be noted that in the case of an enumeration hook, the referenced C object may either be an enum tag or a type name associated with an enumeration type using a typedef declaration. Similarly, in the case of a set/get hook, the name of the C object that is first in the access path may be a struct or union tag or a type name associated with a structure type via a typedef declaration; a pointer to a structure type is also admitted. All other identifiers in an access path need to be a member of the structure accessed at that level. A type hook always references a C type name.