Implementing GLib Collection Types in the Giraffe Library

Max Reeves
Abstract

Implementing GLib Collection Types in the Giraffe Library

Max Reeves

Only a few graphical user interface (GUI) toolkits for Standard ML are available to the public and they are in general old and do not seem to be actively maintained. The inability to create modern GUIs can be seen as one of the greater drawbacks of application development in Standard ML (SML). The Giraffe library provides a Poly/ML and MLton interface for GObject based C libraries that have GObject introspection repository information available, using language bindings. Development of the Giraffe library is far gone, but currently there is no support for GLib collection types. This thesis presents a simple implementation of support for the GSList collection type for the Giraffe library, as well as performance results of a number of different approaches to the conversion between GSLists and SML lists.
Acknowledgements

Special thanks to Phil Clayton for letting me work on this very interesting project, for taking the time to answer my questions about it, and for helping me set it up. Thanks to my supervisor, Tjark Weber, for all of the kind support provided during the course of this project.
## Contents

1 Introduction .................................................................................................................. 1

2 Contributions .................................................................................................................. 1

3 Background .................................................................................................................. 2
   3.1 Standard ML ............................................................................................................ 2
       3.1.1 MLton and Poly/ML ......................................................................................... 3
   3.2 Calling C functions from SML .................................................................................. 3
       3.2.1 The MLton FFI .............................................................................................. 3
       3.2.2 The MLton Finalizable structure .................................................................. 5
       3.2.3 The Poly/ML FFI ......................................................................................... 6
   3.3 GLib ....................................................................................................................... 8
       3.3.1 GObject ....................................................................................................... 8
       3.3.2 GObject introspection ................................................................................ 9
   3.4 Giraffe ................................................................................................................... 10
       3.4.1 The Giraffe FFI ........................................................................................... 11
       3.4.2 Use of phantom types .................................................................................. 13
       3.4.3 Object-oriented approach .......................................................................... 14

4 Implementation ............................................................................................................. 17
   4.1 SML representation of GSLists ............................................................................. 17
   4.2 Transfer scheme ..................................................................................................... 17
   4.3 The G_S_LIST and G_S_LIST_TYPE signatures .................................................. 19
   4.4 The GSList functor ............................................................................................... 21
       4.4.1 Support for return values ............................................................................ 23
       4.4.2 Support for value parameters .................................................................... 24
       4.4.3 Support for reference parameters ............................................................. 25
       4.4.4 The PolyML structure ............................................................................... 26
   4.5 Element type structures ......................................................................................... 26

5 Evaluation ..................................................................................................................... 28
   5.1 Testing .................................................................................................................... 28
   5.2 Performance evaluation ......................................................................................... 28
       5.2.1 Performance of the conversion functions .................................................... 28

6 Related work .................................................................................................................. 31

7 Conclusions and future work ....................................................................................... 32

8 References ..................................................................................................................... 33
A Implementation of conversion functions

A.1 Converting from GSList to SML list
   A.1.1 Pointer arithmetic
   A.1.2 Binding to the g_slist_next macro
   A.1.3 Intermediary C array

A.2 Converting from SML list to GSList
   A.2.1 Intermediary C array
   A.2.2 Binding to the g_slist_prepend function
1 Introduction

Standard ML (SML) [10] is a functional programming language; it treats computations as evaluations of mathematical functions. It is considered both statically and strongly typed, meaning that type information is obtained at compile-time, rather than at runtime, and that it does not allow implicit conversion between types.

There are mainly two approaches for supporting graphical user interfaces (GUIs) in SML: creating an entirely new toolkit designed solely for SML or creating a language binding to a library that already exists. A binding for an already existing GUI toolkit has the benefit of utilizing the power of an already existing framework and reducing the cost of implementation. But for such a binding to be possible, it is required that a foreign function interface (FFI) aimed at the targeted toolkit’s language exists. Fortunately most SML compilers support FFIs targeting the C programming language.

Gtk+ provides a modern and cross-platform GUI toolkit based on the GObject portable object system. GObject is written in C and is designed with cross-language interoperability in mind [16]. The creation of bindings to GObject based libraries is made even easier with the GObject introspection repository (GIR), which is a set of files describing the application programming interface (API) of GObject based libraries in a machine readable format [20]. A binding to Gtk+ and other GObject based libraries could therefore be a benefit to the SML community and a realistic goal.

The Giraffe library provides a SML interface compatible with Poly/ML and MLton for GObject based C libraries with GIR information available, for example GLib, Gtk+, GtkSourceView, Pango and VTE. The library started off as an extension of the mGTK project [12]: a “proof of concept” binding of the Gtk+ library targeting Moscow ML and MLton. It was eventually rewritten to generate code based on information from GIR files. The interface introduces abstractions of the C libraries to provide features expected of SML, such as garbage collection, type-safety, absence of uninitialized values, and portable source code. The encodings for class hierarchies of the code generated by Giraffe is an encoding of abstract types as described in OO Programming Styles in ML [1], which in turn is a particular use of phantom types [7].

While development of the Giraffe library is far gone, there has been no public release of the source code so far due to the level of library support. For example, there is no support for the Glib collection types GList, GSList or GHashTable yet. This report addresses problems related to implementing support for GLib collection types in the Giraffe library.

2 Contributions

This report presents a “proof of concept” implementation of support for the GSList collection type in the Giraffe library, targeting both Poly/ML and MLton. The implementation allows SML lists to be passed to and returned from targeted library functions in an apparently seamless manner that is type-safe, void of uninitialized values, and integrated with the garbage collection routines of the respective compilers.
The implementation includes a high-level FFI parameterized module written in SML for the GSList collection type, which fits the already existing framework and generated bindings and modules that implement the targeted type and can be used by the generator to create instances for different element types.

Finally, a number of different approaches to the conversion of GSLists to SML lists and vice versa are presented and compared performance-wise, across both compilers.

3 Background

This section elaborates on key concepts and technologies that are essential to the understanding of both the implementation of the Giraffe library, which is also outlined in this section, and the work described in subsequent parts of the report.

3.1 Standard ML

The module system of SML consists of structures, functors and signatures. A structure and a signature are combined to form a module in a way such that the structure contains the relevant declarations needed to implement the interface provided by the signature. Hansen and Rischel describe a functor as “a function from structures to structures” [10]. In other words, a functor takes structures as arguments and produces a new structure based on the structures passed as arguments. Functors can be used to implement polymorphic algorithms and data structures that require different operations over their type parameters.

SML has some imperative programming features, such as references, while loops, mutable arrays and other imperative data structures. It is important to have these features in SML for several reasons, e.g.: in order to allow input and output needed for interactive programs; for more efficient implementation of basic functional data types such as list, sets and tables [10]; and to enable implementation of object-oriented features such as classes using SML’s polymorphism [15].

One of SML’s characteristic features is type inference. Consequently the SML programmer does not have to describe the types of function arguments and other values to the compiler as they will be calculated during compilation. However, it is possible for the programmer to provide explicit type annotations. This can for example be useful to break polymorphism, to verify that values are of expected type and to describe interfaces between modules through signatures.

The Definition of Standard ML (Revised) [14], commonly known as SML ’97, contains the formal programming language semantic definition of SML as well as its grammar. The formal semantic describes the meaning of the language, and is formulated through rules using mathematical notation. What it describes is thus not how the language should be implemented. This allows for widely different implementations to share source code and still be expected to give the same result. Standard ML comes in many different flavors, all of which more or less strictly adhere to this definition.

The original, unrevised definition of SML, SML ’90, was published in 1990 and specified an initial basis, a set of primitive types such as int and string together with some related operations. This initial basis turned out to be too limited for application
programming. This caused most implementations of the language to extend the initial basis with collections of general-purpose libraries. Since the libraries spawned from different sources, they tended to be incompatible with other implementations of SML. The SML Basis Library [8] introduced with SML ’97 provides a standardized and rich collection of basic types and functions for the SML language that replaces the initial basis. It provides a basic toolkit for SML programmers that is intended to replace compiler-specific general-purpose libraries and make it easier to write applications that are not limited to a single SML compiler.

3.1.1 MLton and Poly/ML
MLton [4] and Poly/ML [13] are the two SML compilers currently supported by Giraffe. Both MLton and Poly/ML support the full SML ’97 language and Basis Library and are available for most popular architectures and operating systems. MLton is a whole-program optimizing compiler. Being a whole-program compiler it allows for a number of optimizations that reduce the cost of SML’s abstraction system, for example for functors, parametric polymorphism, and higher-order functions. Poly/ML on the other hand is both a compiler and a runtime system. It provides an interactive environment that lets the user load code from SML files and enter declarations and expressions.

3.2 Calling C functions from SML
Although the SML Basis Library contains functions suitable for many low-level programming tasks, it does not provide any tools purposed for calling C functions. In fact, there exists no compiler-independent SML library that can be used to call C functions, instead MLton and Poly/ML maintain FFIs with their own unique features and limitations.

The examples in this section are adapted versions of the one found in Section 16 of the Poly/ML Interface to the C Programming Language documentation [3].

3.2.1 The MLton FFI
With MLton’s FFI it is possible to import C symbols, such as functions and variables, that are exported from dynamically linked C libraries. It allows implicit conversion between standard SML types, such as int, char, bool, and vector, and their C counterparts.

The structure MLton.Pointer is designed for pointer handling. It contains the type MLton.Pointer.t, representing a C pointer, and a wide range of functions for pointer manipulation. MLton’s FFI documentation [6] contains a table of the mappings between SML and C types.

The syntax for writing a function binding in MLton is straightforward, as the following implementation of a binding for the C standard library function qsort shows.

The qsort function is an implementation of the quicksort algorithm and takes an array of unspecified type as argument, together with the number of items in the array, the size of the element type and a pointer to a function for comparison of array elements.
void qsort(void *base,
    size_t nitems,
    size_t size,
    int (*compar)(const void *, const void *))

Listing 1: Declaration of the qsort C standard library function.

This example uses an array of integers, and defines a function compar which takes
pointers to two integer variables and returns either true or false after comparing the
two.

int compar (const void* pkey, const void* pele) {
    return *((int*) pkey) > *((int*) pele);
}

Listing 2: The compar C function.

The definition of the function qsort_ exemplifies the extension of SML’s syntax
designed for importing C functions in MLton: _import "C function name" attr...
: cFuncTy; where attr... denotes an optional sequence of attributes that is ana-
logous to C’s __attribute__ specifier [6]. The syntax extension makes use of the already
present method for explicit type annotations to define parameter and return types of
imported symbols, as can be seen in the declaration of the function qsort_.

fun qsort list =
    let
        val qsort_ = _import "qsort" :
            int array * int * int * MLton.Pointer.t -> unit;
        val comparPtr = _address "compar" : MLton.Pointer.t;
        val array = Array.fromList list
        val nitems = Array.length array
        val size = 4 (* Assuming integers are 4 bytes *)
    in
        (qsort_ (array, nitems, size, comparPtr);
        Array.foldr (op ::) [] array)
    end;

Listing 3: Example of a binding of the qsort function in MLton.

Note that it is not possible to pass a SML list directly to a C function, since there is
no corresponding standardized type in C to convert to. In this example the problem is
overcome by converting the list to an array.

In order to pass a C function as argument to a different C function you first need
its address. The syntax for getting an address of a C function or variable is similar to
importing, with the difference that the type always is a pointer and _import is replaced
with _address.

MLton has no built in support for handling C structures. It is therefore necessary
for structures to be handled on the C side. It can for example be done by defining a C
function that allocates memory for a structure and initializes it with values passed as arguments, which can later be modified with get and set functions.

3.2.2 The MLton Finalizable structure

MLton implements the Finalizable structure [5] that allows functions to be run on a value after the garbage collector has deemed the value unreachable. More precisely, the structure allows the creation of a finalizable value, which is a container to which functions may be attached. These function are called finalizers. A finalizer takes the contained value as argument and is run at some point after the garbage collector considers the finalizable, and consequently the value it contains, unreachable.

Since data allocated by a C function typically requires explicit freeing, the concept of finalizers is useful when handling C allocated data in SML. By attaching a finalizer (which could be a binding to a C function such as free or similar) to a value containing a C pointer, it is possible to include the freeing of C data in the SML garbage collection procedure.

Listing 4 shows the signature for the Finalizable structure. A new finalizable value is created with the function new. It returns a container of type 'a Finalizable.t: a polymorphic type where the type variable 'a is the type of the original value. Once a finalizable value has been created, a finalizer can be attached with the function addFinalizer. As the type of the function indicates, the finalizer can be any SML function that takes as argument a value of the same type contained in the finalizable it is being attached to and returns unit. Sometimes it is necessary to apply a function to a finalizable value. The way to do this is to use the function withValue. It applies a function to the value contained in a finalizable and returns the return value of the function application, with the assurance that finalizers attached to the finalizable will not run until the function call is completed. Calling finalizeBefore (v1, v2) will ensure that the finalizers of v1 will be finalized before those of v2, while touch will ensure that the finalizers of the argument will not run before the call to touch.

highlight

signature MLTON_FINALIZABLE =
  Sig
    type 'a t

    val addFinalizer: 'a t * ('a -> unit) -> unit
    val finalizeBefore: 'a t * 'b t -> unit
    val new: 'a -> 'a t
    val touch: 'a t -> unit
    val withValue: 'a t * ('a -> 'b) -> 'b

end

Listing 4: Signature for the Finalizable structure.

5
3.2.3 The Poly/ML FFI

In Poly/ML, C function bindings can be made by using the CInterface structure [3]. It allows loading of dynamic libraries and extraction of symbols from loaded libraries. C values are represented as SML values of type vol, which is short for volatile, and can be created and converted to and from native SML values using built in functions, as explained below. Volatiles are garbage collected, just like regular SML values.

The data encapsulated by a volatile is not kept in the SML heap together with other SML values but in a separate memory space. There are two reasons for this: values stored on the heap can change address during garbage collection and a pointer into the SML heap has the potential to corrupt other data [3].

CInterface contains the polymorphic type a Conversion. A conversion contains a value of type CType, representing a C type, and functions that allow conversions between C values and SML values. CInterface comes with a set of predefined, basic conversions corresponding to basic data types found in C. For example, there is the DOUBLE Conversion. It is composed of the functions toCDouble and fromCDouble and the value CDouble, which can all be found in the CInterface structure. The to and from functions allow conversion between SML values of type real and C values of type double. The return value of toCDouble is of type vol as all C data is represented by volatiles in Poly/ML.

structure C = CInterface
val libcomp = C.get_sym "compar.so";
val libc = C.get_sym "libc.so.6";
val toCarray = List.foldl (fn (e, p) => (C.assign C.Cint p (C.toCint e);
  C.offset 1 C.Cint p));
fun fromCarray p n =
  List.tabulate (n, (fn n => C.fromCint (C.offset n C.Cint p)));
fun qsort list =
  let
    val qsort_ = C.call4 (libc "qsort")
      (C.POINTER, C.INT, C.INT, C.POINTER) C.VOID
    val comparPtr = C.volOfSym (libcomp "compar")
    val nitems = List.length list
    val arrayPtr = C.alloc nitems C.Cint
    val size = C.sizeof C.Cint
  in
    (toCarray arrayPtr list;
     qsort_ (C.address arrayPtr, nitems, size, comparPtr);
     fromCarray arrayPtr nitems)
  end;

Listing 5: Binding of the qsort C standard library function in Poly/ML.
CInterface allows making and breaking of conversions using the functions mkConversion and breakConversion, respectively. Making new conversions can be useful in order to differentiate volatiles containing abstract types from normal volatiles [3].

breakConversion takes a conversion and breaks it down to its elements: 'a Conversion -> (vol -> 'a) * ('a -> vol) * Ctype.

Writing function bindings using Poly/ML’s CInterface is not as straightforward as in MLton. The example in Listing 5 shows a Poly/ML implementation of the same qsort binding found in Section 3.2.1.

The call to get_sym returns a function that requires the name of a symbol in the library specified in the call to get_sym. The functions libcomp and libc can thus be used to load symbols from the dynamic libraries compar.so and libc.so.6. The functions returned by these calls have type string -> CInterface.sym, where sym represents the symbol named by the string. Calling libc causes the library and symbol to be reloaded at every invocation, which may sound very inefficient if subsequent calls are made. The reloading may however be made faster through caching [3].

Listing 6: The type of CInterface.call4.

<table>
<thead>
<tr>
<th>CInterface.sym -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a CInterface.Conversion * 'b CInterface.Conversion *</td>
</tr>
<tr>
<td>'c CInterface.Conversion * 'd CInterface.Conversion -&gt;</td>
</tr>
<tr>
<td>'e CInterface.Conversion -&gt; 'a * 'b * 'c * 'd -&gt; 'e</td>
</tr>
</tbody>
</table>

The function call4 is a member of a family of callN functions, where 0 ≤ N ≤ 9, that can be used to create bindings to N-parameter C functions. In the example, a function qsort_ is constructed using call4 such that it requires a four-tuple parameter CInterface.vol * int * int * CInterface.vol. Note that the conversion POINTER yields a parameter of type vol, so a C pointer is represented by a volatile in SML.

With CInterface all conversions between SML and C types have to be done explicitly, either by the use of conversions or CInterface functions and C type values, and it is not possible to pass an SML array to a C function directly at the call. First memory has to be allocated for the array. This can be done using the alloc function, which returns a vol containing the allocated memory. As mentioned earlier, vols are garbage collected and so is the memory allocated by alloc. Another approach is to use C’s malloc function through a binding, in which case the allocated memory would not be garbage collected and a vol containing a pointer to the allocated memory would be returned. In this case the function address from the CInterface structure is used to get a pointer to an allocated C array.

In order to import a pointer to the comparison function, a call to volOfSym is made. It returns a vol containing a pointer to the compar function. The array can then be written to and read from using the functions toCarray and fromCarray, respectively.

val LINK = CInterface.STRUCT2 (CInterface.POINTER, CInterface.DOUBLE);

Listing 7: Creating a new conversion LINK of a C struct using CInterface.
Unlike MLton’s FFI, CInterface allows the creation of C structures in the SML domain using a family of STRUCT functions. We can, e.g., create a structure of a pointer and a double, possibly representing a link in a singly linked list, as in Listing 7, and get a new conversion describing the corresponding C structure.

Just like MLton, Poly/ML supports finalizers. However, there is no use of finalizable containers but a finalizer has to be attached directly to a volatile. Another difference is that only C functions can be used as finalizers. Moreover, finalizers can only be attached to C data. The function setFinal from the CInterface structure is used to attach a finalizer to a volatile and takes as argument the targeted volatile of type vol and an imported C function of type sym. The type of setFinal is CInterface.sym -> CInterface.vol -> unit.

3.3 GLib

The GLib library provides to the C programmer both abstractions of primitive data types as well as implementations of composite data structures that are normally included by default in higher level languages, such as memory chunks, linked lists, hash tables, queues and trees. More than that, it provides many different utility functions for memory management, message logging, debugging, threads, and exception handling as well as macros and a generic main event loop, etc. The abstractions and utilities provided by GLib simplify ports between platforms and are handy for many different programming tasks.

Since GLib’s primitive types abstract the basic data types found in C in order to make them more portable between different platforms, programmers using GLib are encouraged to use these types rather than those inherent to C [16]. These types can be broken down into four groups: new types that are not present in standard C; integer types that have the same size independent of platform; types that have better usability than their C counterparts, perhaps most notably the untyped pointers gpointer and gconstpointer which can be used for type abstraction; and types equal to their C counterparts. In GLib it is the functions that use untyped pointers that are responsible for verifying the type, not the programmer [16]. This is significant in the ability to write introspectable GObject based libraries discussed later in this section.

3.3.1 GObject

Object-oriented programming is mainly a way of thinking, and the idea is that programming should be modeled around objects; that data types, variables of these types, and their algorithms belong together. In practice, an object is a data structure stored in memory and it is an instance of a class. A new object is created by invoking the constructor of a class – a process known as instantiation. Algorithms are implemented as methods that belong to a class and are included in any instance of that class. Programming languages that adopt this paradigm, like C++ and Java, make object-oriented programming much easier by providing tools built into the language itself, to help with the organizational details of the object-oriented system. The C programming language does not provide an object-oriented programming system, but does provide all the nec-
ecessary means to implement one. The GLib Object System [19], more commonly known as GObject, provides an object-oriented framework for C.

The GObject system is thus largely a set of rules governing how to think of objects in C, but provides several features that make this task easier and more powerful. Most importantly, it provides a generic type system, GType, that allows registering of inherited types and manages creation, initialization, and memory management of object and class structures. It also provides a fundamental base object type, the GObject type, to base object hierarchies upon; every object belongs to a class and at the core is the GObject base class from which all classes inherit [16]. Only single inheritance is possible, meaning that each class may only inherit from at most one other class. The GObject system also allows the declaration of interfaces derived from the GTypeInterface structure, which is a non-instantiable interface type. These work similarly to Java’s interfaces [16]: they describe a common programming interface that several classes can implement.

GObject implements reference counting in order to support garbage collection. Objects are handled through object references. Since an object may have several references to it, the number of references to an object is kept track of by a reference counter. Objects have to be referenced and unreferenced with special functions that increment and decrement the reference counter. Once the reference count is decremented to zero it means that an object will no longer be used and can be freed from memory, so what happens is that: GObject disposes of all references held by the no longer used object to other objects; finalizes it, which means that the memory is marked as reusable; and lets a garbage collector free the memory when appropriate [16]. However, not all types in the GObject system are reference counted, some types require that the memory they use is freed manually, but a type either is or is not reference counted.

### 3.3.2 GObject introspection

The GObject system is designed to allow easy access to C objects and methods from other programming languages. The GObject introspection toolkit [20] is a set of applications that act as middleware between GObject based C libraries and language bindings. It can collect and extend metadata for these libraries with the purpose to describe all information about the APIs needed to write language bindings. The introspection metadata is then presented in a machine readable format. As a result of this, GObject introspection, together with GObject based libraries, makes a large body of code available to developers using high-level languages, such as SML.

There are several details in GObject code that are needed by language bindings but are not captured by the code itself, for example reference counting semantics and the type of elements in collection types. GObject introspection solves this by putting all of the metadata inside the GObject library itself, using comments with special annotation derived from the GTK-Doc format [23]. (GTK-Doc is a tool used to extract API documentation from C code.) This report only concerns about the annotation for function parameters and return values that are collection types, like lists and hash tables. It is therefore unnecessary to describe the entire annotation repertoire and better to focus on explaining the most relevant parts. For the full list of annotations available, see the official GObject-Introspection Annotations documentation [21].

In general terms the *transfer mode* annotation manages the memory and life-cycle of values. More precisely it specifies the transfer ownership for parameters and return values. The transfer ownership indicates whether a parameter or return value is owned by the recipient or not, and thus whether it needs to be either copied or referenced by the recipient. Types that are reference counted are owned by the recipient when it owns a reference to a value, and types that are not reference counted are owned by the recipient when it owns the entire value – a unique copy in memory. There are four different transfer modes to choose from: none, container, full, and floating. None simply means that the value is not owned by the recipient. The floating mode is similar to none, but is intended for floating objects: objects that are not specifically owned by any code portion, but referenced by a “floating” reference that needs to be “sunked”. The modes container and full are only meaningful for container types. Container means that the recipient owns only the container but not its elements, while full means that both the container and elements are owned by the recipient.

Function parameters can be of three different kinds as suggested by the annotations in, out, and inout. In is the annotation for parameters that are only read by the function. Since C functions are only capable of returning a single value, it is common practice to use reference parameters for functions that need to return several values. Reference parameters in C are simulated by passing a pointer, holding the address of a value, that can be read or written to. Out and inout parameters are both reference parameters that return some value to the callee, with the difference that inout parameters are not only written to but are also used for input to the function. The annotations nullable and not nullable indicate whether `NULL` is accepted as a value for a parameter. For the most part not nullable is the default unless anything else is stated, although there are exceptions for, e.g., `gpointer` types and closures [21]. Another type of parameter that accepts `NULL` as a value are the optional out and inout parameters. In this case `NULL` would indicate that the output should be ignored.

The GObject introspection repository (GIR) is the extensible markup language (XML) format that describes the introspection metadata collected for libraries. The GIR files generated from the source code can be parsed and analyzed by any code generator that aims to produce bindings for a second language. Information contained in the GIR files include the annotations from the annotated comment blocks as well as information about namespaces and class hierarchies.

3.4 Giraffe

The Giraffe library provides a Poly/ML and MLton interface for an number of GObject based libraries that have GIR information available. At the time of writing this report the following libraries are supported: GLib 2.0, Pango 1.0, Gtk+ 3.0, GtkSourceView 3.0, VTE 2.90, and VTE 2.91. Later versions of these libraries are expected to work, but there is no support for features introduced in later versions. The interface introduces abstractions of the C libraries to provide features expected of SML, such as garbage collection, type-safety, no uninitialized values and portable source code. The code is licensed under the GNU Lesser General Public License (LGPL), just like the GObject based libraries mentioned.
Giraffe can roughly be split into two parts: the generator and the generated library. The latter is what is referred to as the Giraffe library. For the most part Giraffe generates language bindings based on the information from GIR files. However, since different SML compilers do not share a common library needed to call foreign functions, the Giraffe library implements its own FFI built on top of those available for the supported compilers. This FFI is split into two levels: a high-level FFI that is shared by both the Poly/ML and MLton compilers and a low-level FFI that makes use of the compiler-specific FFI libraries. The high-level FFI is introduced for converting between abstract types and concrete C types and for controlling ownership of C allocated memory. This separation allows the same application built using the Giraffe library to be compiled by both Poly/ML and MLton.

The Giraffe library aims to adapt to some SML conventions. Code portability is an important factor in the usability of the Giraffe library: more compilers supported means more potential users and uses. As SML is a garbage collected language, it is reasonable for a programmer who uses the bindings provided by the Giraffe library to expect that values returned by functions will be garbage collected. Not only should this be expected but it also simplifies the programming task, considering that SML was not designed for manual garbage collection. Making sure that C data is automatically garbage collected is not a simple task, but fortunately the GObject system has been designed to make this easier, and with the use of finalizers it should be manageable even in SML. Another challenge is maintaining SML type-safety. When handling C data in SML it is important that, e.g., types of objects are accurately described and that a method of a class can only be applied to instances of that class. This also means that no uninitialized values can be allowed. These topics are discussed in more detail in the following sections.

3.4.1 The Giraffe FFI

The high-level FFI consists mostly of code generated from the GIR data and provides the interface that can be directly used by the programmer to create bindings to supported library functions. In order to make the code generation easier, most of the complexity is hidden in the low-level FFI. The high-level FFI also implements some syntactical elements that make it possible to reuse much of the code for different compilers, thus reducing the complexity of the code generator even further.

The outline of a function generated by Giraffe can take on many different forms, but they all have a similar structure and for the purpose of this report it is unnecessary to show all possibilities. The example in Listing 8 generalizes the form of a typical class method.

All interface functions in the libraries generated by Giraffe are curried, which means that the syntax for function arguments is a sequence of values separated by space characters rather than several values in a single argument tuple. The function with name <functionName> thus has at least one parameter, self, that is the current class instance, followed by any number of in parameters separated by the space character.
fun <functionName> self <functionInParameters> = 
  let
    val <functionOutParameters> = 
      ( <withFunctions> ---> <fromFunctions> )
          <functionName>_ (<argValues>)
  in
    <retExp>
  end

Listing 8: The typical form of a class method that takes an instance of its own class as argument.

The value <functionName>_ is a binding to a C function made using the FFI provided by the compiler, and is therefore created differently depending on which compiler the function is generated for. In MLton the binding is created as described in Section 3.2.1, while Poly/ML uses a customized call function that works similarly to the callN functions described in Section 3.2.3, but allows a variable number of parameter and return values.

val <functionName>_ = 
  call <functionSym> ( <functionInConversions> ---> <functionOutConversion> )

Listing 9: The form of a function binding using the call function.

Listing 9 shows the outline of bindings generated using the call function. Here <functionSym> is a function symbol of type CInterface.sym and <functionInConversions> is a list of conversions separated by the infix operator &>&, which in turn is separated from the conversion <functionOutConversion> by the infix operator --&.

The <functionOutParameters> value collects the list of values returned by the bound C function. This includes not only the actual return value, but also values returned by reference parameters. It is a set of nested pair arguments ending with the actual return value. It can be constructed using the & character, which is an infix constructor for the pair type. A <functionOutParameters> could, e.g., be x & y & z, where x and y would hold a values returned by reference and z would hold the actual return value of the C function. Similarly <argValues> is a set of & separated values passed to the C function. Usually <functionOutParameters> is a subset of <argValues>, because there are cases when function parameters are ignored by passing the NULL pointer to the C function or when some other standard value is used. The <retExp> is a tuple of return values, usually a subset of those in <functionOutParameters>.

In order to convert SML values to C values and vice versa, converting between abstract types, and to control ownership of C allocated memory before and after function calls, Giraffe implements type-specific “with” and “from” functions. In the example above, <withFunctions> and <fromFunctions> are sequences of “with” and “from” functions, respectively. The elements are separated by the infix &>& operator. The infix --& operator separates the two sequences, e.g., u &>& x --& y &>& z for a function with two arguments and return values. Listing 10 shows their implementation.
\begin{verbatim}
infixr 0 --->
infixr 1 &&&>

structure HighLevelFFI =
  struct
    fun (f &&&> g) h (a & b) = f (fn a' => g (fn b' => h (a' & b'))) b) a
    fun (f ---> g) x = g o (f x)
  end;

Listing 10: Definitions of the high-level FFI operators.

The semantics of these “with” functions is similar to the \texttt{withValue} function found in the MLton \texttt{Finalizable} structure, but they are more specialized because they can also handle conversion from a specific SML type to a corresponding C type, including procedures needed for memory management such as addition of finalizers. Every SML type that can be passed to a C function in the Giraffe library has at least one “with” and at least one “from” function. The “from” functions work in a similar way but in the other direction: the conversion is from C to SML. The implementation of these functions is compiler-specific and can for the most part be found in the structure \texttt{FFI.<Type>.C} in the source code for both supported compilers. The \texttt{FFI} structure contains the low-level FFI mechanisms for most non-object types, while for object types they are implemented in the class structures, e.g., \texttt{<Type>Class.C} and \texttt{<Type>Record.C}.

For a working example of how the Giraffe FFI can be used in practice, refer to Section 3.4.3.

\textbf{3.4.2 Use of phantom types}

The Giraffe library uses phantom types \cite{7} to represent C pointers and to encode the class hierarchies of the targeted libraries. The latter is described in Section 3.4.3 while this section focuses on the use of phantom types to represent C pointers.

A simple explanation of a phantom type is a parameterized type with a superfluous type variable that can be used to encode extra type information. Take for example the phantom type \texttt{‘a t} in the structure \texttt{Pointer} in Listing 11. Here a phantom type is used to represent C pointers that either \texttt{may be} or \texttt{are not} NULL: \texttt{unit Pointer.t} and \texttt{nonnull Pointer.t}, respectively. In this example a pointer is represented by a volatile, as a C pointer would be when using the Poly/ML FFI.

\begin{verbatim}
structure Pointer :>
  sig
    type ‘a t
    type notnull
    val null : unit t
    val deref : notnull t -> unit t
    val address : ‘a t -> notnull t
  end =
\end{verbatim}

\end{verbatim}
Listing 11: Example of a structure using a phantom type.

With this definition of the structure `Pointer`, calling `Pointer.deref Pointer.null` will result in a compiler error warning about incompatible types, indicating that the `deref` function expected a `nonnull` but received a `unit` pointer. If this type restriction had not been made, the code could have resulted in a runtime error. This is because the compiler would not have been able to catch that a `NULL` pointer was dereferenced. (CInterface actually allows dereferencing `NULL` pointers, but then a runtime error could have occurred elsewhere.) Note that removing the signature would remove this type restriction, as the compiler would be able to infer that the type `nonnull` is indeed the same as `unit`.

The example in Listing 11 illustrates how the Giraffe library maintains type-safety for C pointers. This includes pointers to objects in particular, as the actual SML value of an object will be a pointer to some C data. It allows for type-safe operations on pointers and helps to catch errors at compile-time rather than runtime.

### 3.4.3 Object-oriented approach

The difference between the modular design practice of SML and an object-oriented class hierarchy lies in the way data structures are enriched. For example, in SML a polymorphic data structure, such as list, is enriched by instantiating it with an element type: `'a list` is enriched by `int list`. While in object-oriented programming, a data structure is typically enriched by extending it with additional constituents. For example, an existing ball structure may be enriched by an additional color attribute. So the components available at a certain level of specification are only known to methods at the same level and above, while the addition of information preserves the structure of existing components.

Giraffe uses the class encoding scheme described in Section 2.1 in *OO Programming Styles in ML* [1] to encode the class hierarchies of the targeted libraries. The encoding builds on the concept of phantom types, rather than subtyping (which is not provided by SML), and supports GObject’s single inheritance class hierarchies.

In the Giraffe library, an object returned by a class method is essentially represented by a pointer to the data allocated for the object in C memory space, and its methods are bindings to the C functions belonging to that class. This means that at its core, each class is derived from the type `'a Pointer.t` and the enrichment is merely a redefinition of this basic pointer type that permits the application of methods on objects at the same level of inheritance and above, while restricting the application on objects lower in the class hierarchy.
type base = unit;

structure BaseClass :>
  sig
    type 'a t
    structure C :
      sig
        type notnull
        type 'a p
        val fromPtr : notnull p -> 'a t
        val withPtr : (notnull p -> 'b) -> 'a t -> 'b
      end
    structure PolyML :
      sig
        val PTR : C.notnull C.p CInterface.Conversion
      end
  end =
    struct
      type 'a t = Pointer.notnull Pointer.t
      structure C =
        struct
          type notnull = Pointer.notnull
          type 'a p = 'a Pointer.t
          val fromPtr = I
          fun withPtr f x = f x
        end
      structure PolyML =
        struct
          val PTR = Pointer.PolyML.PTR
        end
    end;

structure Base :>
  sig
    type t = base BaseClass.t
    val new : int -> base BaseClass.t
    val get : 'a BaseClass.t -> int
  end =
    struct
      type t = base BaseClass.t
      val new_ = call (load_sym libbase "base_new")
        (FFI.Int.PolyML.VAL --> BaseClass.PolyML.PTR)
      val get_ = call (load_sym libbase "base_get")
        (BaseClass.PolyML.PTR --> FFI.Int.PolyML.VAL)
      fun new i = (FFI.Int.PolyML.withVal ---> BaseClass.C.fromPtr) new_ i
      fun get self =
        (BaseClass.C.withPtr ---> FFI.Int.PolyML.fromVal) get_ self
    end;

Listing 12: Example of a base class.
The examples in Listing 12 and 13 illustrate the basic implementation of class structures in the Giraffe library. They show how a base class and a derived class can be implemented in Poly/ML. (An implementation in MLton would be similar but a bit simpler due to the FFI differences, but the general principle is the same.) Just like in the Giraffe library, the implementation is split into a class structure that implements the actual class encoding and eventual low-level FFI mechanisms and an interface structure that implements the class methods. In this case the functions withPtr and fromPtr are very simplified, because the conversion from an object to a pointer is unnecessary and because this example does not implement a memory management routine.

As can be seen in Listing 12 implements the type `'a BaseClass.t as a nonnull pointer, indicating that a base object may not be uninstantiated. More so, the implementation of the `'a BaseClass.t type is not known outside of this structure which requires the use of the use of the withPtr and fromPtr functions to manipulate the underlying pointer type of base objects and objects of derived classes.

A derived class inherits from the base class by enriching the base class type with a type `'a DerivedClass.derived_object, as can be seen in Listing 13, which implementation is hidden to classes further down the inheritance chain.

```
structure DerivedClass =>
  sig
    type 'a derived_object
    type 'a t = 'a derived_object BaseClass.t
  end =
  struct
    type 'a derived_object = unit
    type 'a t = 'a derived_object BaseClass.t
  end;

structure Derived =>
  sig
    type t = base DerivedClass.t
    val new : int -> base DerivedClass.t
    val get : 'a DerivedClass.t -> int
  end =
  struct
    type t = base DerivedClass.t
    val new_ = call (load_sym libderived "derived_new")
      (FFI.Int.PolyML.VAL --> BaseClass.PolyML.PTR)
    val get_ = call (load_sym libderived "derived_get")
      (BaseClass.PolyML.PTR --> FFI.Int.PolyML.VAL)
    fun new i = (FFI.Int.PolyML.withVal ---> BaseClass.C.fromPtr) new_ i
    fun get self =
      (BaseClass.C.withPtr ---> FFI.Int.PolyML.fromVal) get_ self
  end;
```

Listing 13: Example of a derived class.
4 Implementation

The implementation of the support for GSLists follows the same SML conventions as the rest of the the Giraffe library. This means that the objective is to make sure that there is no need to explicitly free memory, as the memory management of GSLists and their elements are integrated into the SML garbage collection routine; no uninitialized values exist; optional values have an optional type, meaning that a NULL pointer can not be dereferenced because it is represented as an optionally NULL pointer; and that the same SML application code works across all supported compilers.

Rather than requiring the user to handle GSLists in SML domain, the implementation ensures that every function that returns or takes a GSList as argument makes the necessary conversions to a corresponding SML lists. This section describes the implementation and how these requirements have been met.

4.1 SML representation of GSLists

The GSList consists of the C structure defined in Listing 14 and a number of functions and macros that implement typical list operations, such as appending and prepending elements, removing elements, sorting, and searching. [22] The empty GSList is represented by the NULL pointer, and each element is an instance of the GSList struct where the member data holds the element’s data and next holds the link to the next element in the list. The base of a non-empty GSList will be a pointer to its first element and the last element will always point to NULL. The data held by an element can be any integer value (that fits in the size of a gpointer) or, more commonly, a pointer to any kind of data.

```
struct GSList {
    gpointer data;
    GSList *next;
};
```

Listing 14: The GSList C struct.

In SML a list is a parameterized datatype ‘a list that is capable of holding elements of a single type. For example, int list is the type of a list of integers and (string * int list) list a list of tuples containing one string and one integer list. Similarly to the GSList, the empty SML list is represented by the nil (or []) value and the base of the list will be its first element unless it is empty. [10] Converting a GSList to an equivalent SML list and vice versa is therefore a straightforward affair.

4.2 Transfer scheme

Great care has to be taken to ensure that a GSList that is passed or return as well as its elements is freed, and when it is, at an appropriate time. As mentioned in Section 3.3.2, the memory management of a GSList parameter value is mainly determined by three things: the transfer mode of the parameter, what kind of parameter it is (in, out, or
inout), and the type of its elements. The same thing holds for return values of type GSList, except that they are treated the same as out parameters.

Each function call that takes a list as argument will have to convert an SML list to an equivalent GSList. Likewise, a function call that returns a GSList, either through a parameter or as a return value, will have to convert the returned list to an equivalent SML list. In the case of an inout parameter, both conversions still have to be made. It is during these conversions that the library takes action to make sure that the memory of the lists is handled correctly. The conversion procedure for in parameters can be broken down into the following three general steps. The full transfer scheme for in parameters of type GSList is specified in Table 1.

1. **Element management** For value elements this means converting each element to a gpointer, for reference counted elements it could mean calling g_object_ref_sink on each element in order to pass a reference to the called function, and similarly, for elements that require manual memory management it could mean creating a copy of the element.

2. **Conversion** This step involves the creation of a new GSList using the elements from the previous step.

3. **Freeing the list** In some cases it might be necessary to free the new GSList after the function call. This means that g_list_free would be added as a finalizer to the list itself.

<table>
<thead>
<tr>
<th>Element</th>
<th>Transfer</th>
<th>Free list</th>
<th>Element management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data in pointer</td>
<td>Full</td>
<td>No</td>
<td>Convert to gpointer</td>
</tr>
<tr>
<td>Container</td>
<td>No</td>
<td>Yes</td>
<td>Convert to gpointer</td>
</tr>
<tr>
<td>Reference counted data</td>
<td>Full</td>
<td>No</td>
<td>Take reference</td>
</tr>
<tr>
<td>Container</td>
<td>No</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>None</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Manually managed data</td>
<td>Full</td>
<td>No</td>
<td>Copy element</td>
</tr>
<tr>
<td>Container</td>
<td>No</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>None</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 1: Transfer scheme for in parameters.
In a similar way, the procedure for out parameters and return values can be broken down into the following three general steps which are concretized in Table 2.

1. **Conversion**  Create a new SML list with the elements from the returned GSList.

2. **Element management**  Value elements need to be converted from `gpointer` to the a value of appropriate SML type, reference counted elements may need a `g_object_unref` finalizer and in some cases a reference needs to be taken by a call to `g_object_ref_sink` or similar, and elements that require manual memory management need have an appropriate free finalizer attached to them and in some cases need to be copied in whole.

3. **Freeing the list**  Once step 1 and 2 have been performed, it might be necessary to call `g_list_free` on the returned GSList.

<table>
<thead>
<tr>
<th>Element</th>
<th>Transfer</th>
<th>Free list</th>
<th>Element management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data in pointer</td>
<td>Full</td>
<td>Yes</td>
<td>Convert to SML value</td>
</tr>
<tr>
<td></td>
<td>Container</td>
<td>Yes</td>
<td>Convert to SML value</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>No</td>
<td>Convert to SML value</td>
</tr>
</tbody>
</table>

| Reference counted data   | Full     | Yes       | Add unreference finalizer |
|                          | Container| Yes       | Take reference, add unreference finalizer |
|                          | None     | No        | Take reference, add unreference finalizer |

| Manually managed data    | Full     | Yes       | Add free finalizer |
|                          | Container| Yes       | Copy element, add free finalizer to the copy |
|                          | None     | No        | Copy element, add free finalizer to the copy |

Parameters of the inout kind will use some combination of the behavior for in and out parameters. In libraries that take advantage of GObject introspection annotation, the transfer mode annotation of an inout parameter will normally only refer to the out part, while the in part defaults to none. However, there are cases of library functions where this annotation is interpreted differently.

### 4.3 The `G_S_LIST` and `G_S_LIST_TYPE` signatures

The signature `G_S_LIST` in Listing 15 defines the higher-level FFI support for GSLists. The type `t` represents GSLists that can be used directly by C functions and is defined as an optionally NULL pointer.
The “with” and “from” wrapper functions that provide support for the high-level FFI and the types that are required in library type signatures are contained in the structure C. The wrapper functions constructs high-level foreign functions that contain GSLists in their signature. The functions toList and fromList could be used to convert to and from SML lists if required. The two boolean values passed to the “with” and “from” functions specify the transfer mode of the parameter or return value for which they are used, where the first value specifies whether ownership of the list should be transferred and the second specifies the same for the elements of the list. That is, false false, true false and true true translates to the transfer modes none, container and full, respectively. (The combination false true has no meaning.)

```
signature G_S_LIST =
sig
  type t
  type 'a elem
  val toList : t -> 'a elem list
  val fromList : 'a elem list -> t

structure C :
  sig
    type notnull
    type 'a p
    type 'a out_p
    val fromPtr :
      bool -> bool -> 'a out_p -> 'a elem list

    type 'a in_p
    val withPtr :
      bool -> bool -> ('a in_p -> 'r) -> 'a elem list -> 'r

    type ('a, 'b) r
    val withNullRef :
      (('a, 'b) r -> 'r) -> unit -> 'r
    val withRefPtr :
      bool -> bool -> (('a, 'b) r -> 'r) -> 'a elem list ->
      ('b out_p, 'r) pair
    val withRefPtrOptPtr :
      bool -> bool -> (('a, 'b) r -> 'r) -> 'a elem list option ->
      ('b out_p, 'r) pair
  end
end
```

Listing 15: The G_S_LIST signature.

The signature G_S_LIST_TYPE in Listing 16 defines the functionality for GSList element types. The purpose of this signature is to allow the creation of functors that can be instantiated for different element types, in order to allow code reuse and a uniform behavior for the part of the higher-level FFI support that is shared by all lists.
signature G_S_LIST_TYPE =
  sig
    type elem
    structure C :
      sig
        type notnull
        type 'a p
        val withPtr : bool -> elem -> unit p
        val fromPtr : bool -> unit p -> elem
      end
  end

Listing 16: The G_S_LIST_TYPE signature.

4.4 The GSList functor

The core functionality is contained in the functor GSList, which takes as argument a structure known as GSListType that implements the G_S_LIST_TYPE signature. Only the Poly/ML implementation of the GSList functor is shown here, but the MLton implementation is similar to the extent that any important differences can, and will, be pointed out. The code is split between the listings in Sections 4.4.1, 4.4.2, 4.4.3, and 4.4.4, in order of appearance, starting with Listing 17 below.

In Listing 17, the type 'a elem obtained from the G_S_LIST signature is instantiated to 'a GSListType.elem by a so-called type realization, allowing the element type to be determined by the implementation of the GSListType structure.

As mentioned in Section 3.4.2, a pointer that may or may not be NULL is represented by the type unit Pointer.t while a pointer that is definitely not NULL is represented by notnull Pointer.t. Since the empty GSList is represented by a NULL pointer, it means that every pointer to a GSList has to carry the type unit Pointer.t in SML domain in order to enforce the type-safety constraints.

The Poly/ML implementation requires a workaround addFinalizer function. The reason stems from the fact that the Poly/ML FFI does not allow passing SML functions as finalizers and the fact that a pointer passed to Finalizable.addFinalizer has to be of type notnull while a GSList object is represented as a possibly NULL (viz. unit) pointer. It works because the call to Finalizable.new returns the same underlying volatile, but it breaks the abstraction of finalizables. Ideally Finalizable.addFinalizer should be used by passing a SML function that uses Option.app and Pointer.toOpt to only apply the finalizer function if the pointer is not NULL, for example the function Option.app free_ o Pointer.toOpt, but this is not possible since the Poly/ML implementation of Finalizable.addFinalizer only allows C function symbols as finalizers. This is not a problem in MLton.

21
functor GSLList (GSListType : 
  G_S_LIST_TYPE
  where type C.notnull = CPointer.notnull
  where type 'a C.p = 'a CPointer.t) :

sig
  include 
  G_S_LIST 
  where type 'a elem = 'a GSLListType.elem
end =

struct
  type t = unit CPointer.t Finalizable.t
  type 'a elem = 'a GSLListType.elem

structure C =
  struct
    structure Pointer = CPointer
    type 'a p = 'a Pointer.t
    type notnull = Pointer.notnull
    structure Type = GSLListType.C
  end

local
  open CInterface
  open Pointer
  open PolyML
in
  val giraffe_g_slist_next_sym = load_sym libglib "giraffe_g_slist_next"
  val g_slist_prepend_sym = load_sym libglib "g_slist_prepend"
  val g_slist_free_sym = load_sym libglib "g_slist_free"

  val next_ : notnull t -> unit t =
    call1 giraffe_g_slist_next_sym POINTER POINTER
  val prepend_ : unit t * unit t -> unit t =
    call2 g_slist_prepend_sym (POINTER, POINTER) POINTER
  val free_ : notnull t -> unit =
    call1 g_slist_free_sym POINTER VOID
end

fun addFinalizer f (t : t) : unit =
  let
    open Finalizable
    val ptr = withValue (t, I)
    val add = fn ptr => addFinalizer (new ptr, f)
  in
    Option.app add (Pointer.toOpt ptr)
  end

fun map_rev f l = foldl (fn (x,a) => (f x)::a) [] l;

Listing 17: The GSLList functor: parameter definitions, function bindings, and workaround addFinalizer function.
4.4.1 Support for return values

The only difference between the Poly/ML and MLton code to support lists as return values is the implementation of the function `getDataPtr`, which simply dereferences a pointer to a list element in order to get a pointer to the contained data.

```
fun getDataPtr (p : notnull p) : unit p = 
  Pointer.MLton.fromOptPointer (Pointer.MLton.getPointer (p, 0))
```

Listing 18: MLton implementation of `getDataPtr`.

```
type 'a out_p = 'a p

local
  val getDataPtr : notnull p -> unit p = 
    Pointer.PolyML.fromVol o Pointer.PolyML.derefToVol

fun fromClist (list : unit p list) = 
  fn
    SOME ptr =>
      let
        val data = getDataPtr ptr
        val next = next_ptr ptr
      in
        fromClist (data :: list) (Pointer.toOpt next)
      end
    | NONE => list

fun toList p = fromClist nil (Pointer.toOpt p)

fun copyPtrToList e p = 
  let
    val list = toList p
  in
    map_rev (Type.fromPtr e) list
  end

fun fromTransferPtr cont p = 
  if cont then Option.app free_ (Pointer.toOpt p) else ();
in
fun fromPtr cont e p = 
  let
    val q = Pointer.toOptNull p
  in
    copyPtrToList e q before fromTransferPtr cont q
  end
end
```

Listing 19: The `GSList` functor: support for return values.
MLton does not provide a simple dereference function like Poly/ML. Instead it provides the function `MLton.Pointer.getPointer`, which expects a pointer to dereference and the index of the value. It is part of a collection of `getX` functions where `X` specifies the type of the returned value, e.g., Pointer, Int32, Real32, or Word32. However, these functions leave up to the programmer to guarantee that the returned value is of the expected type.

### 4.4.2 Support for value parameters

An example of where the workaround function `addFinalizer` has to be used can be found in `withTransferPtr`. Here a function that does a shallow free of the allocated memory of a GSList is optionally added as finalizer to the input argument according the transfer rules. Listing 20 shows how it is implemented in MLton and what it ideally should look like.

```ml
fun withTransferPtr cont f t =  
  (if cont then () else  
    Finalizable.addFinalizer (t, Option.app free_ o Pointer.toOpt);  
    Finalizable.withValue (t, f))
```

*Listing 20: MLton implementation of `withTransferPtr`.*

```ml
local

fun prepend (data : unit p, p : unit p) : unit p =  
  prepend_ (p, data)

val toClist : unit p list -> unit p =  
  List.foldl prepend Pointer.null

fun withTransferPtr cont f t =  
  (if cont then () else addFinalizer g_slist_free_sym t;  
    Finalizable.withValue (t, f))

fun fromList (list : unit p list) : t =  
  Finalizable.new (toClist list)
in

fun copyListToPtr e list =  
  let  
    val ptrList = map_rev (Type.withPtr e) list  
  in  
    fromList ptrList  
  end

fun withPtr cont e f list =  
  withTransferPtr cont f (copyListToPtr e list)

in
```

*Listing 21: The `GSList` functor: support for value parameters.*
The idea with adding a finalizer to the input list is that the memory allocated for the list should not be freed until after the targeted C function has received and processed it. In this case it would be when the function has finished executing and the SML reference to the temporarily allocated C data becomes unreachable.

4.4.3 Support for reference parameters

The type of an reference pointer differs slightly between the two implementations due to the ability to pass references to values as parameters instead of pointers to values for C function parameters in MLton’s FFI. In MLton the reference pointer is therefore defined as unit p ref rather than unit p. The implementations of the function withRef differ for the same reason.

```ml
fun withRef f t = 
  let
    val a = ref (Pointer.toOptNull t)
    val r = f a
  in
    ! (Pointer.MLton.unsafeRefConv a) & r
  end
```

Listing 22: MLton implementation of withRef.

The function unsafeRefConv lets us convert from unit p ref to notnull p ref and the other way around. This of course breaks NULL type-safety, but is meaningful when we want to specify that a pointer reference parameter is optionally NULL before and not NULL after a function call. This can not be done without a function such as unsafeRefConv because, although the underlying type is unchanged, it is not possible to have a different SML type for before and after values due to the way reference parameters are typed.

```ml
type ('a, 'b) r = unit p
local
  val null = Pointer.null
fun withValue f t = Finalizable.withValue (t, f)
fun withRef f t = 
  let
    val v = Pointer.PolyML.toVol (Pointer.toOptNull t)
    val a = Pointer.toOptNull
      (Pointer.PolyML.addressFromVol v)
    val r = f a
  in
    (Pointer.PolyML.fromVol v) & r
  end
```

25
fun withTransferRefPtr transfer f t =  
  (if transfer then () else addFinalizer g_slist_free_sym t; 
   withValue (withRef f) t) 
in
fun withNullRef f () = f null
fun withRefPtr cont e f list = 
  withTransferRefPtr cont f (copyListToPtr e list)
fun withRefPtrOptPtr transCont transElem f = 
  fn  
    SOME list => withRefPtr transCont transElem f list 
  | NONE    => withValue (withRef f) 
       (Finalizable.new Pointer.null) 
end
end

Listing 23: The GSLList functor: support for reference parameters.

4.4.4 The PolyML structure

As the name implies, the PolyML structure within the GSLList functor is only meaningful in Poly/ML. It defines the conversions for the GSLList return values and parameters needed by the Poly/ML FFI. These are no different from the default CInterface.POINTER conversion.

fun toList t = Finalizable.withValue (t, C.fromPtr true true) 
fun fromList l = Finalizable.new (C.withPtr true true I l)
structure PolyML = 
  struct 
  val OUTPTR : 'a C.out_p CInterface.Conversion = 
    C.Pointer.PolyML.POINTER 
  val INPTR : 'a C.in_p CInterface.Conversion = 
    C.Pointer.PolyML.POINTER 
  val INOUTREF : ('a, 'b) C.r CInterface.Conversion = 
    C.Pointer.PolyML.POINTER 
  end
end

Listing 24: The GSLList functor: the PolyML structure.

4.5 Element type structures

Structures implementing the G_S_LIST_TYPE signature can be easily implemented to create instances of the GSLList functor. Listing 25 shows the implementation of the basic GObject object list type in the GObjectObjectClass structure. The resulting nested GSLList structure can be used for all GSLList parameters where the element type inherits from the GObject object class. SML can infer the correct element type due to
the class encoding scheme described in Section 3.4.3. Similar type structures are defined 
for objects that inherit from classes different from the GObject object class.

The functions withPtr and fromPtr only need to specify the behavior for the transfer 
mode of the list elements, the conversion between C and SML data, and SML type of 
the elements. For object types this is quite simple as they all follow similar patterns 
described above, but for integer types that can be stored as pointers and types that have 
a corresponding type in SML but do not fit in a C pointer, e.g. large number formats or 
string types, more elaborate conversions are required. This, however, does not appear 
to be much of an issue since the targeted libraries do not contain such parameter types, 
except from strings.

```ml
structure GSListType :>
G_S_LIST_TYPE
  where type 'a elem = 'a t
  where type Cnonnull = notnull
  where type 'a C.p = 'a p =
  struct
    type 'a elem = 'a t
  structure C =
    struct
      type notnull = notnull
      type 'a p = 'a p
    local
      open CPointer
    in
      fun withPtr transfer x =
        if transfer
          then C.withPtr (toOptNull o refSink_) x
          else C.withPtr toOptNull x
      fun fromPtr transfer ptr =
        C.fromPtr transfer (toNotNull ptr)
    end
  end
structure GSList = GSList (GSListType)
```

Listing 25: Implementation of GObject element types.
5 Evaluation

5.1 Testing

The implementation has been tested using unit tests on bindings to a custom C library that makes use of the transfer modes in all possible combinations, as well as testing nested function calls by passing return values directly as arguments to other functions. A special debug finalizer has been used to monitor and verify that the garbage collection of allocated memory works as expected between test cases.

5.2 Performance evaluation

This section evaluates the performance differences of various methods for converting between GSLists and SML lists. The same methods have been implemented in both Poly/ML and MLton and are all designed to be viable and interchangeable options in the final implementation. These methods are evaluated and compared in terms of time and space efficiency as well as platform independence. The methods tested for converting from GSLists to SML lists are the following.

- **Using pointer arithmetic**  The built in functions for pointer manipulation provided by respective compiler’s FFI are used to fetch the value of the next link member in the GSList struct.

- **Using a binding to the `g_slist_next` macro**  The value of the next link member is fetched by using a C function wrapper for the `g_slist_next` macro provided by GLib.

- **Using an intermediary C array**  The GSList is passed to a C function that converts the list to a C array, which is then converted to an SML list.

For converting from SML lists to GSLists, the following methods have been tested.

- **Using a binding to the `g_slist_prepend` function**  A new GSList is created, element by element, through repeated calls to the `g_slist_prepend` function provided by GLib.

- **Using an intermediary C array**  The SML list is converted to an array, which is then passed to a C function that creates a GSList containing the elements of the array.

Appendix A contains a detailed description of how these methods have been implemented.

5.2.1 Performance of the conversion functions

The following benchmarks have been performed on a computer with a Dual-Core Intel Pentium 987 1.50GHz 32-bit CPU and 4GB RAM. Time has been measured using the `Timer` structure from the SML basis library. All measurements show the time spent by
the code in user space, and does not include the time spent by the operating system during the conversions.

Each measurement is the result of dividing a list of $N$ number of elements into smaller lists of equal length $M$ and calculating the sum of the times it took to convert the smaller lists. There is a significant difference in time between the results of Poly/ML and MLton, which is why $N = 10^6$ for Poly/ML and $N = 10^7$ for MLton in the benchmarks. All times are the results of a single run, since the time difference between repeated runs proved to be insignificant. The benchmarks were compiled using the default settings for both compilers.

**Figure 1: Performance for Poly/ML when converting $10^6$ list elements.**

(a) From C to SML.

(b) From SML to C.

The results for Poly/ML in Figure 1 show that the approach using an intermediate array outperforms the other two methods as the number of list elements increase, but suffer for smaller number of list elements. The results also show that using pointer arithmetic instead of a function binding will roughly halve the conversion time from C to SML. Figure 1b shows that there is almost always and improvement in conversion speed from SML to C when an intermediate array is used instead of a function binding.
The results for MLton do not show the same difference in performance. Figure 2a shows a slight improvement for smaller list when an intermediate array is used in the conversion from C to SML. However, these differences are small and the runtimes proved to depend a lot on the execution order. This was thought to be related to garbage collection, but forcing the garbage collector to run between each timing instance had no observed effect on the runtimes.

In all, there are improvements in performance when calls to foreign functions are avoided in Poly/ML. Most apparent is the difference when an intermediate array is used. There are a couple of drawbacks in using an intermediate array: it requires extra memory to be allocated and may be slower when the number of list elements is very small. The first drawback may not be such a large problem since each data element is represented by a pointer. That is, the size of the extra memory needed to allocate an array is independent of the type of the data.

The final consideration is how the speed improvements compare to the increased code complexity of using pointer arithmetic or an intermediate array in the list conversions. For MLton the method using a binding to a foreign function has proved to compare well to the other methods and is generally fast and easy to implement. Poly/ML on
the other hand may benefit from the performance increase. Since the method using
pointer arithmetic is easy to implement and roughly halves the runtime in all cases in
the conversion from C to SML, it might be the most suitable option. Likewise, the
substantial performance increase obtained for larger lists when an intermediate array is
used in the conversion from SML to C is non-negligible.

6 Related work

Many different languages, supporting many different programming paradigms, have
bindings to Gtk+. However, only few have bindings to the complete set of GNOME
libraries that are supported by the GObject introspection framework, which is the tar-
get of the Giraffe library. This section will be focused on Gtk+ binding to languages
that most closely related to SML. These are all statically and strongly typed functional
languages with automatic type inference.

The mGTK [12] project, a Gtk+ binding that targeted the Moscow ML and MLton
compilers, was the original starting point of the Giraffe library before it was completely
rewritten to generate code from GObject introspection data – where the mGTK project
used .defs files, a less expressive alternative to GIR XML, to automatically generate
the bindings. Although the code was completely rewritten, the encoding of classes, as
described in Section 3.4.3, remained the same. The latest version of mGTK as of writing
this was released in 2013 and the project appears to no longer be maintained.

OCaml is another member of the ML family of programming languages. One of
the main differences compared to SML is that OCaml is extended with object-oriented
programming constructs and can therefore make use of Gtk+’s object-oriented hierarchy
without any fuss. LablGtk [9] is a Gtk+ 1.2 and 2.x binding for OCaml, also with
automatic memory management using GObject’s reference-counting mechanism.

Haskell is perhaps one of the most popular functional languages available at this time
and shares many features with SML. For Haskell you have the projects Gtk2Hs [17] and
haskell-gi [18]. Gtk2Hs is a binding to Gtk+ 2.x that is being increasingly replaced by
the haskell-gi project, which, similarly to the Giraffe library, is a code generator built
on GObject introspection that provides bindings to a large number of libraries.

A nearly complete encoding of the C type system in SML types, modeled with
very limited compiler support, was introduced with the No-Longer-Foreign Function
Interface [2] (NLFFI). (The only missing part is variable-argument functions.) It builds
on the basic idea of data-level interoperability, that is the ability of inspecting and
manipulating C data structures from SML code. It also includes a tool that allows
generation of SML glue code from C header files. NLFFI was originally written for
Standard ML of New Jersey and has later been ported to MLton.

During the early stages of this project Poly/ML introduced a new FFI defined in
a structure called Foreign. The Giraffe library has as of end of writing of this report
been modified to use Foreign instead of CInterface.
Conclusions and future work

This report shows that it is practically possible to implement support for linked lists (GSLists) in the Giraffe library, using a simple interfacing functor that can be instantiated for different element types and which integrates with the Poly/ML and MLton garbage collection routines.

Measurements of different conversion functions show that performance can be improved in Poly/ML by avoiding repeated function calls through the FFI, and instead use the FFI’s built in functions for pointer manipulation or intermediate arrays during the conversion. In MLton these methods show no improvement in performance. Contrary, the use of pointer arithmetic matches the performance of calling functions through the FFI, which is quite remarkable. The measurements also show that the MLton implementation outperforms that of Poly/ML by far.

Certain element types may require different behavior, which means that a separate structure defining the correct behavior may have to be created for each element type. These structures have been written manually, but there is an overlap in the behavior of certain types, such as those derived from the GObject object class, which allows a great deal of code reuse. Furthermore, another consideration is whether these element structures could be reused in the implementation of support for other collection types. This would of course have to be proved in practice, but the idea has always been in focus during the implementation process and should not pose a problem.

The implementation does not include integration with the code generator due to lack of time. Tests show that the manual integration works, and due to the simplicity of the necessary syntax extension it should not be problematic to generate the code automatically. Within this context the work described in this report should therefore be considered a “proof of concept” rather than a full featured implementation.

More work remains to implement support for remaining collection types. Adding support for doubly linked lists (GLists) could be achieved easily by reusing most of the code used for singly linked lists, assuming that the corresponding SML type would be `a list. Support for other collection types will require different approaches, considering that there is no obvious SML representation of, e.g., GHashTable or GLib’s N-ary Trees combined with possible information loss and performance issues during conversion, among other reasons.
8 References


A Implementation of conversion functions

This appendix describes the implementation of the conversion functions evaluated in Section 5.2.

A.1 Converting from GSList to SML list

Listing 26 shows the function called when converting a GSList to a SML list for both Poly/ML and MLton. It takes a pointer \( p \) to a GSList and returns a SML list with elements of type \( \text{elem} \), the SML equivalent of the element type of the converted GSList. This function in turn calls the auxiliary function \( \text{fromClist} \) with \( p \) as argument. The following subsections describe different implementations of \( \text{fromClist} \).

\[
\text{fun toList } (p : \text{unit out_p}) : \text{elem list} = \\
\text{List.rev } (\text{fromClist nil } \text{(Pointer.toOpt p)})
\]

Listing 26: The function used to call the auxiliary conversion functions from C to SML in Poly/ML.

A.1.1 Pointer arithmetic

The function \( \text{fromClist} \) in Listing 27 traverses each link in the argument GSList and executes a function \( \text{getLinkContent} \) on each link, returning a the data element, \( \text{elem} \), and a pointer to the next link, \( \text{next} \).

\[
\text{fun fromClist } (\text{list : elem list}) = \\
\text{fn } \text{NONE } \Rightarrow \text{list} \\
| \text{SOME } p \Rightarrow \\
\text{let} \\
\text{val } (\text{elem, next}) = \text{getLinkContent } p \\
\text{in} \\
\text{fromClist } (\text{elem :: list} ) \text{(Pointer.toOpt next)} \\
\text{end}
\]

Listing 27: The implementation of \( \text{fromClist} \) that uses pointer arithmetic to convert a GSList to a SML list.

A pointer to the beginning of a C struct always points to the first member of the struct [11]. In the case of the GSList struct it is the \text{data} member. The link data is extracted by dereferencing the link pointer using a function \text{ptrToData}.
fun getLinkContent (p : notnull p) : int * unit p =  
let  
  open CInterface  
  val elemVol = Pointer.PolyML.derefToVol p  
  val nextVol = offset nextOffset Cchar elemVol  
  val nextPtr = Pointer.PolyML.fromVol nextVol  
in  
  (ptrToData elemVol, nextPtr)  
end

Listing 28: Poly/ML version of getLinkContent.

Listing 28 shows the Poly/ML implementation. A pointer to the next link in the list is extracted by incrementing the data pointer by nextOffset number of bytes using the offset function from the CInterface structure. The value nextOffset holds the number of bytes from the beginning of the struct to the member next.

fun getLinkContent (p : notnull p) : int * unit p =  
let  
  val elemPtr = Pointer.MLton.fromOptPointer  
     (Pointer.MLton.getPointer (p, 0))  
  val nextPtr = Pointer.MLton.fromOptPointer  
     (Pointer.MLton.getPointer (p, nextOffset))  
in  
  (ptrToData elemPtr, nextPtr)  
end

Listing 29: MLton version of getLinkContent.

The corresponding MLton function in Listing 29 works in a similar way. However, the MLton Pointer structure is limited in that it only provides the getPointer function for dereferencing pointers, where the offset argument is interpreted as the multiple of the size of a pointer in bytes.

A.1.2 Binding to the g_slist_next macro

This version of getLinkContent uses a binding to the g_slist_next macro from the GSLList structure, or rather a function that encapsulates this macro — which is necessary because it is not possible to create macro bindings using the Poly/ML and MLton FFIs. It uses the same approach to extract the data held by a link as in the previous section, but extracts the pointer to the next link in the GSLList by calling the function next_, which is a binding to g_slist_next.
fun getLinkContent (p : notnull p) : int * unit p =
  let
    val elemVol = Pointer.PolyML.derefToVol p
    val nextPtr = next_ p
  in
    (ptrToData elemVol, nextPtr)
  end

Listing 30: Poly/ML version of getLinkContent that uses a binding to the g_slist_next macro to convert a GSList to a SML list.

Listing 30 shows the Poly/ML implementation of this function and Listing 31 the implementation for MLton. There is no difference between these implementations apart from the FFIs used and the representation of pointers.

fun getLinkContent (p : notnull p) : int * unit p =
  let
    val elemPtr = Pointer.MLton.fromOptPointer
    (Pointer.MLton.getPointer (p, 0))
    val nextPtr = next_ p
  in
    (ptrToData elemPtr, nextPtr)
  end

Listing 31: MLton version of getLinkContent that uses a binding to the g_slist_next macro to extract next link from a given GSList link.

A.1.3 Intermediary C array

In this approach the implementation of fromClist allocates an array arr to temporarily hold the GSList link data. The data is then written to arr by calling the function listToArray_, which is a binding to a C function list_to_array. A SML list containing the link data is then generated using a tabulator to iterate over arr and convert the data using the function ptrToData described earlier.
fun fromClist list ptr : elem list =
case ptr of
  NONE => list
| SOME p =>
  let
    open CInterface
    val len = length_ p
    val arr = alloc len voidStar
  in
    listToArray_ (p, Pointer.PolyML.addressFromVol arr);
    List.tabulate
    (len, (fn n => ptrToData (offset (len-n-1) voidStar arr)))
  end

Listing 32: Poly/ML version of fromClist that uses an intermediate array to convert a GSList to a SML list.

In the Poly/ML implementation in Listing 32, the value of arr is a volatile holding a C array allocated using the alloc function from CInterface, while it is a SML array in the MLton implementation in Listing 33. As explained in Section 3.2.1, SML arrays are mutable objects and in MLton they can be passed to and used directly by C functions.

fun fromClist list ptr : elem list =
case ptr of
  NONE => list
| SOME p =>
  let
    val len = length_ p
    val arr = Array.array (len, Pointer.null)
  in
    listToArray_ (p, arr);
    List.tabulate
    (len, (fn n => (ptrToData (Array.sub(arr, len-n-1)))))
  end

Listing 33: MLton version of fromClist that uses an intermediate array to convert a GSList to a SML list.

A.2 Converting from SML list to GSList

Listing 34 shows the function fromList, which is the function called when converting a SML list to a GSList for both Poly/ML and MLton. It takes as parameter a SML list, list, with elements of type elem and returns a finalizable pointer to a GSList. This function calls the auxiliary function toClist with list as argument. The following subsections describe different implementations of toClist.
fun fromList (list : elem list) : t =
    Finalizable.new (toClist list)

Listing 34: The function used to call the auxiliary conversion functions from SML to C.

A.2.1 Intermediary C array

The Poly/ML function toClist in Listing 35 allocates an intermediary C array \texttt{arr}. The function updateArr is used to write the data of the input list \texttt{list} to \texttt{arr}. The element data is converted to a C data pointer using the function dataToPtr, which has the inverted functionality of \texttt{ptrToData}. The function arrayToList is a binding to a C function \texttt{array\_to\_list}, which creates and returns a GSList with the data contained in \texttt{arr}.

fun toClist (list : elem list) : unit p =
    let
        open CInterface
        val len = List.length list
        val arr = alloc len voidStar
        fun updateArr (e, p) =
            (assign voidStar p (dataToPtr e); offset 1 voidStar p)
    in
        List.foldl updateArr arr list;
        arrayToList_ (len, Pointer.PolyML.addressFromVol arr)
    end

Listing 35: Auxiliary function that uses an intermediary C array and a C function \texttt{array\_to\_list} in the conversion.

Similarly, the MLton function in Listing 36 generates a SML array \texttt{arr} containing the element data contained by \texttt{list} converted to C data pointers, which is then passed to arrayToList_.

fun toClist (list : elem list) : unit p =
    let
        val len = List.length list
        val ptrList = List.map dataToPtr list
        val arr = Array.fromList ptrList
    in
        arrayToList_ (len, arr)
    end

Listing 36: Auxiliary function that uses an intermediary C array and a C function \texttt{array\_to\_list} in the conversion.
A.2.2 Binding to the g_slist_prepend function

This implementation uses a binding, prepend_, to the g_slist_prepend function in the GSList C structure to generate a new GSList from a given SML list. prepend_ takes as arguments a pointer to a GSList and a pointer to the data which is to prepended to the GSList. It returns a pointer to the new head of the GSList.

val toClist : elem list -> unit p =
  List.foldl prepend Pointer.null o List.rev

Listing 37: Auxiliary Poly/ML function that uses a binding to the g_slist_prepend function to dynamically allocate new GSList links.

The function toClist in Listing 37 is the same for both the Poly/ML and MLton implementations. What differs is the implementations of the function prepend in Listing 38 and 39 for Poly/ML and MLton, respectively.

fun prepend (elem : elem, listHd : unit p) : unit p =
  let
    val elemVol = dataToPtr elem
    val elemPtr = Pointer.PolyML.fromVol elemVol
    val newHd = prepend_ (listHd, elemPtr)
  in
    newHd
  end

Listing 38: Auxiliary Poly/ML function that uses a binding to the g_slist_prepend function to dynamically allocate new GSList links.

fun prepend (elem : elem, listHd : unit p) : unit p =
  let
    val elemPtr = dataToPtr elem
    val newHd = prepend_ (listHd, elemPtr)
  in
    newHd
  end

Listing 39: Auxiliary Poly/ML function that uses a binding to the g_slist_prepend function to dynamically allocate new GSList links.