USENIX Association

Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

mGTK: An SML binding of Gtk+

Ken Friis Larsen
ken@friislarsen.net

Henning Niss
hniss@it.edu

Department of Innovation
IT University of Copenhagen
Denmark

Abstract

We describe mGTK, a Standard ML language binding for the Gtk+ toolkit. Gtk+ is a graphical toolkit for the X Window System, and provides an object-oriented C language API. Since Standard ML is a mostly-functional language without object types, constructing a binding to Gtk+ is not a trivial task. In mGTK, a single-inheritance class hierarchy is encoded using SML's type system. Most of the mGTK binding is machine generated, to best utilize the limited manpower of the project.

The goal of the mGTK project is "just" to present a type-safe interface to Gtk+ for SML programmers. This contrasts with GUI libraries for functional languages, which concentrate on producing good user interfaces: there are several SML graphical user interface libraries available for this task. With mGTK, SML applications have access to the mature, complete and familiar Gtk+ user interface.

1 Introduction

A good Standard ML (SML) binding to Gtk+ is an advantage for both the SML community and for the Gtk+ community. The SML community benefits in a couple of ways. First, SML programmers get access to a good general-purpose graphical user interface (GUI) library with a large range of modern widgets: the SML community is sorely missing such a library. Second, it is a step in providing full access to the whole GNOME platform. This will make it possible for SML programmers to make real applications without having to invent add-hoc solutions to many standard problems, such as database access. There are also a couple of advantages of an SML binding to the Gtk+ community. First, such a binding can help open up the small, but important, market of teaching languages. Second, as SML is a radically different language than C, an SML binding will test the "interfaceability" of Gtk+. This is an important design goal for the Gtk+ developers.

Standard ML

SML is a functional language with imperative features which is widely used for teaching and research. It is roughly on the same level of abstraction as Python and Scheme. In contrast to Python and Scheme, which are *dynamically typed*, SML is *statically typed*, like Java and C++. This means that type errors are detected at *compile time* rather than at *run time*. Despite static typing, it is not necessary for the SML programmer to explicitly provide type annotations in the program. SML features *type inference*: the compiler reconstructs type annotations as needed.

SML is one of the few languages with a formal definition. The definition of SML [10] consists of 93 pages of mathematical notation (a "big step" structured operational semantics, plus type inference rules) and English prose. The book is not meant as tutorial for the language. Rather, it provides an implementation independent formulation of SML. This formal definition means that it is possible to write substantial applications in SML that are not dependent on a specific compiler. There are also several mature SML implementations with widely different implementation strategies, ranging from byte-code interpreters with interactive *read–eval–print–loops* to aggressive whole-program optimizing native code compilers.

Gtk+

Gtk+, The GIMP Toolkit [14], is an LGPL-licensed [6] library for creating graphical user interfaces. It works on many UNIX-like platforms, on Windows, and on Linux framebuffer devices. Gtk+ is the graphical toolkit used in the GNOME desktop environment. Gtk+ is implemented in C, with an object-oriented hierarchy of user interface elements ("widgets"). From the beginning, the Gtk+ developers have paid attention to making it feasible and practical to develop "bindings" or "wrappers" permitting Gtk+ use by programs written in programming languages other than C. See [12] for a current list of bindings.

The Gtk+ library itself only contains widgets, but it is built on a number of useful libraries, with which it is often associated. Specific libraries used by Gtk+ are [13]:

GLib A general-purpose utility library, not specific to graphical user interfaces. GLib provides many useful data types, macros, type conversions, string utilities, file utilities, and so forth.

Pango A library for internationalized text handling. It provides the rendering engine for the widgets in Gtk+ that displays text.

ATK The Accessibility Toolkit. It provides a set of generic interfaces allowing accessibility technologies to interact with a graphical user interface. Gtk+ widgets have built-in support for accessibility using the ATK framework.

GDK The abstraction layer that allows GTK+ to support multiple windowing systems.

Our work, however, is concentrated on the widget set itself.

Overview of this paper

The rest of this paper is organized as follows. Section 2 gives a brief introduction to SML. Section 3 shows a "Hello World" example using mGTK. Section 4 explains how a single-inheritance class-hierarchy, in particular Gtk+'s, can be encoded in SML's type system, while retaining type safety. Section 5 describes the way that the mGTK library is built upon the mGTK infrastructure. Section 6 describes practical matters of mGTK, such as which SML compilers are supported. Finally, Section 7 lists related work and Section 8 gives some conclusions.

2 Brief Introduction to SML

This section gives a brief overview of some of the main features of SML. It is not sufficient to serve as a standalone user programming guide for the language. However, it should be sufficient to get an understanding of the examples in the rest of the paper. For more information about SML, we refer the interested reader to one of the many fine textbooks [8, 11] available.

SML is a two-level language. It consists of a *core language* for programming in the small (that is, functions, data structures and algorithms), and a *module language* for programming in the large.

Figure 1 shows a small stack library implemented in SML. This example shows most of the important features of SML. The library consists of two parts: an interface description, which is called a *signature* in SML (Figure 1(a)), and an implementation module, which is called a *structure* in SML (Figure 1(b)). Informally speaking, a signature is the "type" of a structure. It specifies the declarations of the structure that are to be externally visible.

The signature is named `STACK`. Its extent is delimited by `sig` ... `end`. It contains five specifications: one type specification, one exception specification, and three value specifications.

The type specification `type 'a stack` states the constraints on a module that satisfies (implements) the signature `STACK`. Such a module must declare a type named `stack`: this type is parameterized. The `'a` is a *type variable*: Type variables can be instantiated to other types. This is what is meant by a parameterized type. Thus, the type of a stack of integers is `int stack`, the type of a stack of integer stacks is `int stack stack`, and so on. Type variables are at the core of *parametric polymorphism* (similar to generics in, for example, C++ and Java; see [7] for a comparison of programming languages with support for parametric polymorphism). Note that the type specification does not say anything about how a stack must be implemented.

The exception specification states that an exception named `EmptyStack` must be declared.

The first value specification says that a constant named `empty` must be declared and that this constant must have type `'a stack`. That is, `empty` is a *polymorphic* value: it can be used in contexts where an `int stack` is needed or in contexts where a `int stack stack` is needed. The next value specification states that a function named `push` must be implemented. This function takes two arguments, an element and a stack, and returns a stack. Again, we see how type variables are used to specify that `push` must work with stacks, whereas the elements can have any type. The last value specification states that a function named `pop` must be implemented, and that `pop` takes a stack as argument and returns an element and a new stack.

Figure 1(b) shows the code of the implementation, in the form of a structure declaration. The declarations states that the structure is named `Stack`, that `Stack` satisfies the signature `STACK`, and that `Stack` does not reveal any implementation details not revealed by `STACK` (the latter connoted by `:>`). The extent of a structure is delimited by `struct` ... `end`.

The parameterized type `stack` is implemented by an algebraic data type described by a `datatype` declaration. This declaration says that an `'a stack` is either the constant `Empty`, or is built by applying the constructor `Stack` to an element and a stack. (Constants declared by a `datatype` declaration such as `Empty` are known as constructors).

The exception declaration `exception EmptyStack` declares an exception.

The next declaration states that `empty` is bound to `Empty`. The function `push` just applies the constructor `Stack` to its arguments. The function `pop` is more interesting. This function takes a stack as argument and then uses a `case` expression to analyze its argument. (Here we have reused the name `stack`. Types and values uses different name spaces: thus, the same name can be used for both a type and a value.) If the argument is the empty stack (the constant `Empty`) then the exception `EmptyStack` is raised (thrown). Otherwise, the ar-

```
signature STACK = sig
  type 'a stack
  exception EmptyStack
  val empty : 'a stack
  val push  : 'a * 'a stack -> 'a stack
  val pop   : 'a stack -> 'a * 'a stack
end
```

(a) interface

```
structure Stack :> STACK =
struct
  datatype 'a stack =
           Empty
         | Stack of 'a * 'a stack
  exception EmptyStack
  val empty = Empty
  fun push(elem, stack) =
      Stack(elem, stack)
  fun pop stack =
      case stack of
          Empty => raise EmptyStack
        | Stack(top, rest) =>
          (top, rest)
end
```

(b) implementation

Figure 1: Simple stack library implemented in SML.

gument has been constructed by applying `Stack` to the arguments `top` and `rest`, and then a pair consisting of `top` and `rest` is returned.

Users of this library can call functions from the structure `Stack` by using "dot-notation": for example, `Stack.pop mystack`.

This small example illustrates one of the cornerstones in functional programming: new values are constructed by analyzing, composing, and sharing old values. This is in contrast to imperative and object-oriented programming, where values are copied and modified. (A new trend in object-oriented programming is to simulate a functional style. See for example [2, Item 13 and 14].)

3 "Hello World" in mGTK

Figure 2 shows a deliberately simple "Hello World" example using mGTK. It illustrates (1) how to get the toolkit initialized using `GtkBasis.init` (from a module containing basic Gtk+ functionality not related to specific widgets), (2) how to construct new widgets (using module `Window` for the Window widget, and `Button` for the Button widget), and (3) how to connect signals to widgets (using module `Signal`).

Even this small example shows some of the main advantages of combining SML with Gtk+. There are no type annotations in the program source. Nonetheless, the program is statically type-checked by the compiler: type errors are found and reported at compile time rather than at runtime. In the figure we use a SML construct not explained earlier: the expression `fn _ => false` denotes an *anonymous function* that returns `false` regardless of what argument is given (you can use the *wildcard pattern* `_` (underscore) to ignore an argument to a function). Anonymous functions are often handy for simple callbacks, such as this one.

The construct `let val` *x* = *exp* declares the identifier *x* to be bound to the value obtained by evaluating the expression *exp*. If the only reason for evaluating *exp* is for its side effect, one can use the wildcard pattern `_` instead of *x*. Expressions evaluated only for their side effects can also be sequentialized using `;`. The value `()`, the nullary tuple of type `unit`, can be used as the return value of purely side-effecting functions. Such syntacic conveniences improve the readability and quality of code.

Finally, in SML, the double-colon `::` denotes the cons operation on lists. That is, to add an element *x* to the beginning of a list *xs* we write *x*`::`*xs* (in contrast to C++ where double-colon is the module operator). Built-in data types such as lists make GUI programming more convenient.

4 Encoding of Classes

As described in Section 1, SML is a functional language without object-oriented features, while Gtk+ is designed as an object-oriented library. This mismatch makes constructing an SML interface to Gtk+ tough . The most difficult problem is how to represent the subtype relations defined by a class hierarchy in SML's type system. In this section, we discuss how to present a type-safe SML interface to the Gtk+ class hierarchy. By *type-safe* we mean that when an SML application programmer using our library makes a type-error in calling into Gtk+ (calling a undefined method on object, for instance) the SML compiler should give a type error at compile time.

```
structure HelloWorld = struct
  fun hello _ = print "Hello World\n"

  fun main _ =
      let val _ = GtkBasis.init(CommandLine.name()::CommandLine.arguments())
          val window = Window.new ()
          val button = Button.new_with_label "Hello World"
      in  Signal.connect window (Widget.delete_event_sig (fn _ => false))
        ; Signal.connect window (Widget.destroy_sig GtkBasis.main_quit)
        ; Signal.connect button (Button.clicked_sig hello)
        ; Container.add window button
        ; Widget.show_all window
        ; GtkBasis.main()
      end
end

val _ = HelloWorld.main()
```

Figure 2: Hello World in mGTK.

Throughout, we shall think of class hierarchies mainly as definitions of subtype relationships. This "confusion" of classes and types is intentional: it is standard practice in types for object oriented programming. We are able to take advantage of two properties of Gtk+ and SML. First, Gtk+ implements only a single-inheritance class system. Second, SML's type system is expressive enough to express the subtype relations of single-inheritance class hierarchies.

We present a general method of taking a given object oriented class hierarchy and encoding it in the SML type system. The properties of the resulting encoding are: each class type has a corresponding SML type; the encoding is *complete* (all typings allowed by the class hierarchy is also allowed by the encoding); and the encoding is *sound* (all typings that is disallowed by the class hierarchy is also disallowed by the encoding). The last property is also called "type safety". In type-theoretic jargon, the trick is to use parametric polymorphism and *existential types* to encode inheritance subtyping. In particular we use *phantom types* to encode the inheritance path.

Figure 3 shows a small class hierarchy with just four classes. Label and Container are subclasses of the class Widget and Window is a subclass of Container. For the specific class hierarchy in Figure 3, the properties we want to enforce with our type encoding into SML's type system are: that we should be allowed to call the show method on all kinds of arguments whether they are of class type Widget, Label, Container, or Window; that we should be allowed to call the add method on Containers and Windows, but not Widgets and Labels; that we should only be allowed to call set_title on Windows; and so on.

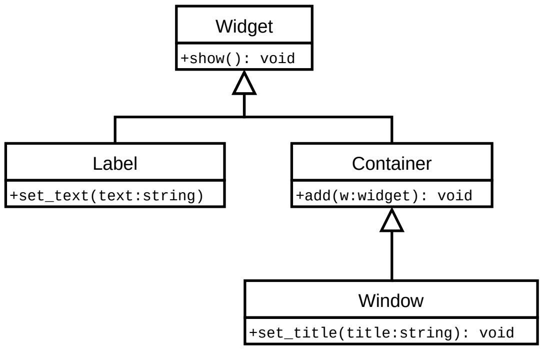

Figure 3: Small example class hierarchy

We now describe the details of the encoding of a class. Throughout this description we only present the SML specifications, that is, the parts that go into the signatures. The parts that go into the structures are not as interesting: that is simply a matter of calling into the C runtime.

Class types: A base class like Widget in Figure 3 is encoded as an abstract parameterized type:

```
type 'path widget
```

(We follow the convention suggested by the Standard ML Basis Library and spell type-names in lowercase, with underscores if needed.) The type variable 'path will be used to hold the inheritance path for subclasses.

Subtyping/Inheritance: For a subclass like Label we need to encode two things: the existence of the class (type), and the subtype relation to the parent class. To do this, we declare two new SML types: an abstract parameterized type, and a type abbreviation for specifying the inheritance.

```
type 'path label_t
type 'path label =
        'path label_t widget
```

We call the abstract type (here `label_t`) the *witness type* because it witnesses that the class exists. Similar, the type `label` is the type abbreviation that specifies that Label inherits from Widget. In the declaration of `label` we see that the type variable `'path` in the declaration of the type `widget` has been instantiated with the type expression `'path label_t`, which contains a new type variable (also named `'path`). In the rest of the paper we shall use the convention that witness types ends with `_t`.

This is the juicy bit of the encoding, because this is really what makes it possible to encode single-inheritance class hierarchies in SML. Unfortunately, this is also the hardest part of the encoding to understand. Our experience is that you have to work a bit with some code to really comprehend the trick.

Methods: Because SML is not an object-oriented language we shall model methods with ordinary functions. We use the usual convention that the first argument is the object on which the method is called. (Gtk+ also uses this convention.)

We can now write the type for the method `add` in Container:

```
val add : 'path container -> 'a widget
                              -> unit
```

This specification says that `add` takes two arguments, an object of type Container and a widget, and that it returns `unit` as result. Similarly, the method `set_title` from class Window has the type:

```
val set_title : 'path window -> string
                              -> unit
```

That is, `set_title` takes two arguments, an object of type Window and a string, and it returns `unit`.

Constructors: We have to be a bit careful with constructors. If we return a value with a polymorphic type-variable `'path` that holds an inheritance path that has not yet been "plugged", then we could accidentally use a super-class constructor to construct values that can be instantiated to the type of a sub-class. Hence, we introduce the abstract dummy type `base` and use that to plug the type variable. Thus, the type of the constructor for Label is:

```
type base
val new : unit -> base label
```

The convention in Gtk+ is that constructors are named `new`.

Fields: We are not able to handle fields directly, because we keep the representation of objects completely opaque. Thus, all inspections of and changes to fields must be done through accessor methods.

We then wrap all parts of the encoding of a class into a signature/structure pair of its own. That is, for the class Window in Figure 3 the SML signature is:

```
signature Window =
sig
  type 'path window_t
  type 'path window =
      'path window_t Container.container
  val new : unit -> GtkBasis.base window
  val set_title : 'path window -> string
                               -> unit
end
```

We see that this signature relies on two structures: `Container` for the class Container, and `GtkBasis` for the dummy type `base`. In addition to the signature `Window` we also need a structure called `Window` that implements the actual calls to the relevant Gtk+ C functions.

Does this encoding really allow all the things that the Gtk+ class hierarchy allows? Yes. For example, the function `Container.add` has type:

```
val add : 'p1 container -> 'p2 widget
                               -> unit
```

From this type we can see that, if `label` is a value of type `base label` and `window` is a value of type `base window` then the application `Container.add window label` is well-typed because: (1) the type of `window` is just a abbreviation for `base window_t container`, thus, the type variable `'p1` can be instantiated to `base window_t`, and (2) the type of `label` is just an abbreviation of `base label_t widget`, thus, the type variable `'p2` can be instantiated to `base label_t`.

Consider now the function `set_title` that only works on Windows:

```
val set_title : 'p window -> string -> unit
```

If we, by mistake, attempt to use this function to set the text of the label `label` (with type `base label`) as in expression `Window.set_title label "New text"`, we get a (compile-time) type error saying (essentially) that the Label widget is not a subclass of the Window widget because the inheritance paths do not match. Here is the concrete error message given by the Moscow ML compiler:

```
- Window.set_title label "New text";
! Toplevel input:
! Window.set_title label "New text";
!                  ^^^^^
! Type clash: expression of type
!   base label_t widget
! cannot have type
!   'a window_t container_t widget
```

Hence, we have demonstrated that for these concrete examples our encoding is both sound and complete.

5 Process

In constructing the mGTK binding we leverage the foresightedness of the Gtk+ developers. Early on, they recognized that it would be important to have a machine-readable "specification" of the toolkit. The specification would describe the widget classes, the inheritance hierarchy, and methods and functions in the toolkit. This specification was implemented using a lisp-like custom notation in the `gtk.defs` file of the toolkit. One could argue that it is simple enough to extract the same information from the C header files. However, C headers are difficult to parse, whereas the defs format is straightforward to parse.

The bulk of the mGTK binding is constructed automatically from the `gtk.defs` file. The complete binding process is naturally divided into two phases: (1) binding design, where we apply the principles described in Section 4 to a few representative widgets to demonstrate the structure of the binding, and (2) binding construction, where the structure in (1) is applied to the entire toolkit. It is important to note here that the design phase can be carried out for a very small subset of the toolkit, after which the construction phase "mimics" that for the complete toolkit.

This phase separation makes it easier to get the design right, simply because there are fewer issues to deal with. It also makes the work involved in moving the binding to other SML compilers manageable: the compiler writers can provide the equivalent of the small subset for their compiler, and utilize that style during the construction phase. We also hope that the phase separation will help when new releases of Gtk+ are produced. Most of the work in constructing the binding for the new release is over when the design of the small subset has been completed.

Let us return to our running example, and look at some example specifications of widgets, functions/methods, and signals. Figure 4 shows three entries in the `gtk.defs` file. The first entry shows a widget specification indicated by `define-object`. From the entry we see that the `GtkContainer` widget (the name appearing right after `define-object` is a shorthand) inherits from `GtkWidget`, and it belongs in the `Gtk` module. We also see the type assigned to instances of this widget in the *Gtk+ type system* (which is completely unrelated to the SML encoding given above).

```
(define-object Container
  (in-module "Gtk")
  (parent "GtkWidget")
  (c-name "GtkContainer")
  (gtype-id "GTK_TYPE_CONTAINER")
)

(define-method gtk_container_add
  (of-object "GtkContainer")
  (c-name "gtk_container_add")
  (return-type "none")
  (parameters
    '("GtkWidget*" "widget")
  )
)

(define-signal delete-event
  (of-object "GtkWidget")
  (return-type "gboolean")
  (when "last")
  (parameters
    '("GdkEventAny*" "p0")
  )
)
```

Figure 4: `gtk.defs` excerpt.

The next entry shows a method specification for the method `add`. This method takes a `GtkWidget*` (in the C implementation) argument, and returns nothing. Since it is a method, there is an implicit "self" argument of type `GtkContainer*`.

The final entry shows a "signal handler" or ("callback") specification. In this case, we specify the prototype for handlers of delete events on widgets. The signal handler for `delete-event` for widget `GtkWidgets` accepts a parameter of type `GdkEventAny*`, and returns a value of type `gboolean`.

6 The mGTK Binding

The mGTK binding is available at SourceForge `http://mgtk.sf.net/` and is released under the GNU Lesser General Public License (LGPL) [6].

A fundamental difference in producing SML bindings of Gtk+, compared to bindings for other languages, is the existence of a variety of compilers (Section 1). This sets this work apart from bindings to languages such as Python, where there is only one target compiler and runtime system.

The encoding of the Gtk+ class hierarchy in the SML type system in Section 4 is *the* core aspect of the binding. As the encoding stays within the language as defined in the Definition [10], this aspect of the binding remains the same for all SML compilers conforming to the Definition. In other words, the interface exposed to the application programmer is the same across all compilers. One finds SML and Gtk+ implementations on a large variety of platforms. Thus, the GUI porting work in moving application programs from one of these platforms to another is largely eliminated.

The mGTK binding already targets two of the main SML systems, Moscow ML [17] and MLton [16]. The authors are currently looking into constructing bindings for other SML compilers (in particular, the *ML Kit with Regions* [15] and *SML.NET* [18] with Gtk#). As mentioned earlier (Section 5), the issues here mainly involve interfacing to C.

The potential for partial compiler independence sets the present binding apart from other Gtk+ bindings for SML; notably, the `SML-Gtk` binding for the SML of New Jersey compiler [9]. The `SML-Gtk` binding is also based on phantom types. Our binding predates the `SML-Gtk` binding by approximately two years—the `SML-Gtk` User's Manual refers to the mGTK binding. To date no serious attempts has been made to merge these two projects. The reason for this is that, even though the projects seems similar, we have followed rather different strategies for constructing our respective bindings. `SML-Gtk` is partly generated by the `ml-nlffi` foreign function interface, for instance, and does not attempt to automate memory management.

7 Related Work

The list of language bindings for Gtk+ shows a plethora of different languages from which Gtk+ is accessible. In this section we briefly discuss the bindings most related to mGTK.

When considering ML-like languages, there are two major alternatives to the mGTK binding. The `SML-Gtk` binding was discussed earlier. The `lablgtk` binding is a Gtk+ binding for O'Caml. O'Caml is a ML dialect different from SML: among other things, it has object-oriented features. This binding, therefore, can directly utilize the Gtk+ object hierarchy.

Gtk+ has also been bound to other functional languages. For example, `gtk+hs` is a Haskell binding, and `erlgtk` is an Erlang binding. Bindings also exist for other graphical toolkits. For example, `sml_tk` is an SML binding of Tk.

The use of phantom types to express invariants about programs is not new. However, the encoding of a single-inheritance hierarchy as above is original with us. Independent work has established similar results [5]. On the construction side of things, other bindings are also machine generated. For this, some of the bindings use the *Simplified Wrapper and Interface Generator* (SWIG) [1], while others extract appropriate information directly from the C headers files of Gtk+.

From the outset, the necessity of access to libraries has been realized in the functional programming community. Work in this area for SML includes SML/NJ's foreign function interface [3]; for Haskell it includes [4].

8 Conclusions and Future Work

It is our intention to continue this work by utilizing appropriate programming language technology to gradually bind more and more of the GNOME development platform for SML. As was the case above, this entails designing appropriate representations of the platform in the SML world (in particular, preserving the type-safety property mentioned above). It also includes the more practical work of extending the code generator to handle such newly introduced representations.

The long term goal for mGTK is to target most of the GNOME platform. The advantages of bringing GNOME to the SML community in the form of such bindings are twofold. Firstly, it would allow SML programmers access to the vast collection of useful application-level support in GNOME. Secondly, it would allow SML programmers to take part in the development of GNOME components, by allowing them to write such components in SML. The key technical aspect to be solved here is to support type-safe inheritance on the SML side of things. Of course, one will also have to explore exactly how to tie the various languages together. The GNOME community already has experience in this area.

In this paper we have demonstrated that it is theoretically and practically possible to make a type-safe interface from SML to Gtk+. This is interesting for several reasons. First, mGTK was one of the first graphical toolkits available to the SML community. Second, the fact that it is possible to make an SML binding to Gtk+ attests to the claimed "interfaceability" of Gtk+, because SML is so radically different from C in abstraction level and paradigm. Third, by auto-generating the binding, we get a binding of the complete Gtk+ toolkit. Finally, we believe that the particular way we construct the binding can be extended to bind the entire GNOME development platform, using mainly machine generated stub code.

9 Acknowledgments

We are grateful to the lead developer of Moscow ML, Peter Sestoft, who has answered a multitude of questions about interfacing Moscow ML with C libraries. Without Peter's initial technical support, mGTK would never have seen the light of day. Likewise, the MLton developers, in particular Stephen Weeks, has been supportive in answering questions about interfacing MLton to C. They have even made changes to the MLton compiler that were needed for mGTK. Also, we would like to thank the IT University of Copenhagen, were we have been employed while doing much of the work presented in the article. Finally, we must express our deepest gratitude to our FREENIX shepherd on this article, Bart Massey. Without his patience, encouragements, and many suggestions for improvements, this article would have been much less readable.

References

[1] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of 4th Annual USENIX Tcl/Tk Workshop*, pages 129–139. USENIX Association, 1996.

[2] Joshua Bloch. *Effective Java*. The Java Series. Addison-Wesley, 2001.

[3] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C “natively”. *Elec. Notes in Theo. Comp. Sci.*, 59(1), 2001.

[4] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. In *Int. Conf. on Func. Prog. (ICFP’99)*, pages 114–125, 1999.

[5] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *IFIP Theo. Comp. Sci. (TCS’02)*, pages 448–460, 2002.

[6] Free Software Foundation. GNU Lesser General Public License (LGPL), 1999. URL `http://www.gnu.org/licenses/lgpl.html`.

[7] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, pages 115–134. ACM Press, 2003.

[8] Michael R. Hansen and Hans Rischel. *Introduction to Programming using SML*. Addison-Wesley, 1999.

[9] Allen Leung. `SML-Gtk`: Gtk+ bindings for Standard ML of New Jersey, 2003. URL `http://www.cs.nyu.edu/phd_students/leunga/sml-gtk/sml-gtk.html`.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[11] Larry Paulson. *ML for the Working Programmer (2nd edition)*. Cambridge University Press, 1996.

[12] The Gtk+ language bindings webpage, 2004. URL `http://www.gtk.org/bindings.html`.

[13] The Gtk+ reference manual, 2004. URL `http://developer.gnome.org/doc/API/2.0/gtk/index.html`.

[14] The Gtk+ webpage, 2004. URL `http://www.gtk.org/`.

[15] The MLKit web page, 2003. URL `http://www.it-c.dk/research/mlkit/`.

[16] The MLton web page, 2003. URL `http://www.mlton.org/`.

[17] The Moscow ML web page, 2003. URL `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[18] The SML.NET web page, 2003. URL `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`.