Application-specific Foreign-interface Generation

John Reppy
University of Chicago
jhr@cs.uchicago.edu

Chunyan Song
University of Chicago
cysong@cs.uchicago.edu

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]: Program transformation; D.2.12 [*Interoperability*]: Data mapping

General Terms Languages

Keywords Foreign-interface generation, Term rewriting

Abstract

A *foreign interface* (FI) mechanism to support interoperability with libraries written in other languages (especially C) is an important feature in most high-level language implementations. Such FI mechanisms provide a *Foreign Function Interface* (FFI) for the high-level language to call C functions and marshaling and unmarshaling mechanisms to support conversion between the high-level and C data representations. Often, systems provide tools to automate the generation of FIs, but these tools typically lock the user into a specific model of interoperability. It is our belief that the *policy* used to craft the mapping between the high-level language and C should be distinct from the underlying *mechanism* used to implement the mapping.

In this paper, we describe a FI generation tool, called FIG(for *Foreign Interface Generator*) that embodies a new approach to the problem of generating foreign interfaces for high-level languages. FIG takes as input raw C header files plus a declarative script that specifies the generation of the foreign interface from the header file. The script sets the policy for the translation, which allows the user to tailor the resulting FI to his or her application. We call this approach *application-specific* foreign-interface generation. The scripting language uses rewriting strategies as its execution model. The other major feature of the scripting language is a novel notion of composable *typemaps* that describe the mapping between high-level and low-level types.

1. Introduction

An important part of most high-level programming language implementations is the *foreign interface* (FI) mechanism, which allows high-level code to access code and data defined in low-level languages (usually C). At a minimum, a FI mechanism allows foreign functions to be called and provides a mechanism for translating between high-level and low-level data representations (*e.g.*, boxed vs. unboxed floats). FI mechanisms may also provide support for callbacks (*i.e.*, calling high-level code from low-level functions) and data-level interoperability (*i.e.*, direct access to low-level data representations from high-level code) [FPR00].

On top of this mechanism, one defines *glue* code that handles the mapping between the low-level and high-level types and functions. For example, a C `struct` might be mapped to a high-level record type, or to an abstract type with accessor functions. Because the C type system does not distinguish between different possible uses of pointer types (*e.g.*, arrays, reference parameters, result parameters), user intervention is if often required to specify the mapping from C types to high-level types. These choice between different mappings are *policy* decisions, which should be left to the designer of the foreign interface. The choice of policy has a major impact on how well the foreign interface fits into the high-level programming model and on the efficiency of the interface.

While glue code can be written by hand, many language implementations provide tools to automate its implementation. One approach has been to use existing *Interface Description Languages* (IDLs) to specify foreign interfaces [FLMP99]. Another recent approach are tools that embed the C type system into polymorphic high-level language using phantom types [Blu01, FPR01]. In both of these examples, the FI policy is fixed by the tool and there is little or no room for customization of the interface.

We believe that the lack of flexibility in specifying the FI policy is a serious weakness in existing FI generation tools. For example, the IDL-based approach of HASKELLDIRECT [FLMP99] provides an effective way to handle simple C functions, but does not provide support for efficient access to aggregate data structures. On the other hand, tools like NLFFIGEN [Blu01] and CHARON [FPR01] provide efficient access to C data representations, but produce large, low-level interfaces that are often ill suited to the high-level progamming model (one is essentially writing C code using a high-level language's syntax). We believe that foreign interfaces should be tuned to the application in question, balancing the goals of aesthetics and performance. In this paper, we present FIG, a tool that supports *application-specific* foreign-interface generation for the MOBY system [FR99]. Our tool takes as input a raw C header file and an application-specific script written in a declarative specification language. It uses the script to guide a rewriting-based transformation of the C header file into a high-level interface. FIG has a "pay as you go" programming model: at their simplest scripts, can use default policies to translate C interfaces in a way that is similar to that of CHARON and NLFFIGEN, but more complicated scripts can be defined to customize the translation. In addition to default policies, FIG also provides a library of common FI idioms, mechanisms that make it easy to factor the specification.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

The main focus of this paper is on the approach that we use to support application-specific foreign-interface generation in FIG. At its core, FIG uses a term-rewriting engine based on the approach of rewriting strategies. FIG scripts are compiled into rewriting rules, which are then applied to a term representation of a C header file. FIG uses the notion of a *typemap* as the building block for constructing FI glue code. Typemaps are terms that are manipulated by the rewriting engine. We present a formal description of typemaps along with a combinator system for defining marshaling and unmarshaling conversions.

The remainder of the paper has the following structure. In the next section, we give review FI mechanisms and give a brief description of the FI mechanism provided by the MOBY system, which serves as the target for FIG. In Section 3, we motivate the need for application-specific foreign-interface generation with some examples taken from the OpenGL API. We then turn to the implementation of FIG in Section 4. There are two important concepts in the implementation of FIG: the first is the use of *typemaps* to represent the translation between the two languages, and the second is the use of rewriting strategies to implement the translation. We describe these in Section 6 and Section 5. Section 7 gives a user's-level view of how FIG is used to generate application-specific foreign interfaces. Finally, we survey related work and conclude.

2. FI mechanisms and policies

It is useful to distinguish between FI mechanisms, which we define to be the way that an implementation connects together low-level and high-level code, and FI *policies*, which we define to be the way that low-level interfaces are mapped into the high-level language. It is often possible to support multiple policies on top of a given mechanism, although some policies may require specific mechanisms.

One common FI mechanism is glue code written in C that provides the connective tissue. The glue code is aware of the high-level language's runtime conventions and is able to convert between representations. This approach has been used by the Java native interface [Lia99], OCaml [FF05], SML/NJ, and Swig [Bea04], among others. An alternative approach is to extend the high-level language with primitives that support the writing of glue code, such as has been done in SML/NJ [Blu01] and Haskell [CFH+03].

The motive of developing FIG is to provide users with mechanisms to support their various selection of policies according to their needs. The following table compares FIG with a number of existing FI tools, which serve as exemplars for the different approaches to the FI generation problem.

	Feature		
System	Scriptable?	C headers as input?	Data-level Interoperability?
FIG	yes	yes	yes
CHARON/NLFFIGEN	no	yes	yes
GREENCARD	yes	no	yes
SWIG	yes	yes	no
HASKELLDIRECT	yes[1]	no	no

The columns in this table correspond to significant characteristics in the design space. The first is whether the tool allows users to control the generation of the FI using some form of specification or script. The second column is whether the tool uses C header files to generate the interface, and the last column is whether the tool (and target system) support *data-level* interoperability.

[1] HASKELLDIRECT uses IDL specifications as input, which are a form of annotated header file, but also allows the annotations to be specified in a separate file.

2.1 The MOBY FI mechanism

FIG is part of the MOBY system, although its architecture is such that it can support backends for other languages. In the design of MOBY, we have paid careful attention providing an efficient foundation for efficient foreign interfaces. The MOBY system supports foreign interfaces through an open compiler infrastructure. While the design of this infrastructure has been described elsewhere [FPR00, FPR01], we review its basic features here to provide the context for the rest of the paper. There are several aspects of this infrastructure that are key to supporting interoperability:

- The compiler's intermediate representation, called BOL, is expressive enough to describe low-level data representations and manipulations that are not expressible in MOBY itself.
- Primitive MOBY types and operations are defined in terms of BOL types and functions. These definitions are given in external MOBY interface files, called MBI files, and play a rôle similar to that of *native methods* in JAVA [Lia99] in that they allow MOBY interfaces to be implemented by low-level code that cannot be written in MOBY.
- The compiler can import and inline code from MBI files.
- There is a tool for generating MBI files from textual descriptions, called MBX files.

BOL is expressive enough to describe C data-structures and operations on them, which allows MOBY code to directly access C data. Furthermore, with cross-module inlining, the compiler will inline the access functions that are generated to manipulate C data structures, which means that manipulation of C data structures is as efficient as in C itself. We call the ability for a high-level language to efficiently access low-level representations *data-level interoperability* [FPR00] and it is essential for the efficient implementation of FIs that involve aggregate data structures.

The MOBY compiler also provides support for exporting MOBY functions to be called by C code (*e.g.*, for callbacks). This support consists of an annotation on BOL functions that specifies that they should use the C calling convention, plus runtime system support for turning MOBY function closures into C function pointers.[2]

3. Motivation

Most FI generation tools take a "one size fits all" approach to the policy used in mapping between the high-level and low-level languages. In this section, we present a number of examples that demonstrate the need for flexibility in specifying the FI policy. These examples are taken from the OpenGL API, which is a widely supported standard for low-level 3D graphics [Ope04] We use OpenGL (and related libraries, such as GLUT), since it has a number of characteristics that make it an interesting case study:

- The OpenGL API defines a large number of symbolic constants (the values of these constants are fixed by the API). At a minimum, a high-level interface to OpenGL should support symbolic names for the constants, but we can go further. The

[2] We the runtime code generation approach suggested by Huelsbergen [Hue95].

OpenGL API uses a single C type (`GLenum`) for all of the constants, but a given function will only accept a fixed subset of the constants and will signal an error at runtime on unexpected values. A high-level API should reflect this behavior and organize the constants into different types. Another complication is that some constants inhabit multiple types (*e.g.*, `GL_NEAREST` is used to specify texture filters for both magnification and minification).

- One of the limiting factors in the performance of 3D-graphics is the bandwidth between the client application and the graphics library. If a high-level language is to be a viable alternative for programming 3D-graphics applications, then the foreign interface to the graphics library (*e.g.*, OpenGL) must provide competitive performance to the native interface.
- While OpenGL is a standard with a well-specified API, there are differences in the header files from platform to platform. For example, some versions of `gl.h` use a single C enumeration type to define the symbolic constants of the interface, while other versions use C pre-processor definitions.
- Because of the rapid evolution in 3D-graphics hardware, the OpenGL specification is revised on a frequent basis.[3] A foreign-interface generation tool should make it possible to easily track an evolving interface.
- The OpenGL API is very large. On MacOS X (10.3), the `gl.h` header file contains 15 typedefs, 465 function prototypes, and 803 symbolic constants (logically organized into 107 different types). A tool that generates the high-level API from the header file can greatly speed the creation of new interfaces.

In the remainder of this section, we present some examples from the OpenGL API that illustrate the need for application-specific foreign interfaces.

3.1 Type abstraction and constants

As mentioned above, OpenGL defines a large number of symbolic constants. Some of these constants are logically grouped into an enumeration, while others serve as bitmasks. In the former case, we want to map the C constants to a collection of abstract values of that have a distinct type, whereas in the latter case, we may want to use MOBY integers to represent the constants, which allows the use of bit operations. To make this choice more concrete, we examine two OpenGL functions that take integer arguments that are specified using the symbolic constants.

```
void glBegin (GLenum mode);
void glPushAttrib (GLbitfield mask);
```

For the `glBegin` function, there is a fixed set of ten mode constants that are valid arguments to the function. We can reflect this restriction by defining a type `BeginMode` and using the following MOBY specification for the function:

```
val begin : BeginMode -> ()
```

We would then map the mode constants, such as `GL_POINTS`, `GL_LINES`, *etc.*, to abstract values of the `BeginMode` type. Alternatively, we might choose to use a datatype for these constants, which requires additional marshaling overhead, but allows pattern matching on the values. In either case, the MOBY type system guarantees that we will never pass an incorrect mode value to `glBegin`.

[3] There have been seven major revisions of the specification since OpenGL 1.0 was released in 1992.

In the case of `glPushAttrib`, the argument is formed by bitwise or of one or more constants. For this reason, we may want to expose the integer representation of the argument type in the MOBY interface

```
val pushAttrib : Int -> ()
```

and map the bitfield constants to integer values

```
val CurrentBit : Int
val PointBit : Int
...
```

An alternative interface is to again use an abstract type (say `Attrib`) and the interface

```
val pushAttrib : List(Attrib) -> ()
```

These examples illustrate that the grouping of constants is application specific and, furthermore, the choice of typing also depends on the application.

3.2 Specialized wrapper functions

The OpenGL API provides several functions for querying the current state of the rendering pipeline.

```
void glGetBooleanv (GLenum pname, int *p);
void glGetDoublev (GLenum pname, double *p);
void glGetFloatv (GLenum pname, float *p);
void glGetIntegerv (GLenum pname, int *p);
```

The first argument to these functions is a constant that specifies which state value should be returned. The type of the second argument depends on the particular state value being queried. For example, if the constant is `GL_BLEND`, then the `glGetBooleanv` function should be called with a single element as its second argument, while if the constant is `GL_BLEND_COLOR`, then the `glGetFloatv` function should be called with a four-element array as its second argument.

One way to define a strongly-typed interface to these functions is to generate a high-level function for each constant that uses the correct underlying C function and arguments. For example, we might define high-level functions

```
val getBlend : () -> Bool
val getBlendColor : () -> RGBA
```

for the two cases above. An alternative approach is to use a single MOBY function for each of these C functions and to express the typing constraints using phantom types.

This example illustrates that a simple one-to-one mapping from C functions to high-level functions is not always desirable. Some applications benefit from mapping a C function to multiple specialized representations.

3.3 Data-structure representations

While the cost of converting between high-level and low-level representations for simple types (*e.g.*, ints, floats, etc.) is typically low, converting structured data, such as arrays or structs, can be quite expensive. With data-level interoperability, this expense can be avoided by allowing the high-level language direct access to the C representation, but in some cases we may prefer paying the marshaling cost for programmer convenience. For example, the function

```
char *gluErrorString (GLenum error);
```

can be used to map an OpenGL error code to an error message. For this function, we clearly want to convert the C string to a

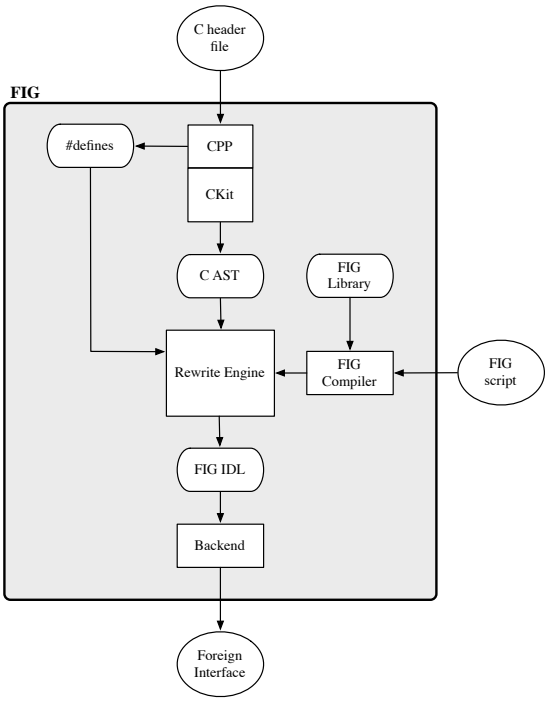

Figure 1. The architecture of FIG

MOBY string. On the other hand, OpenGL also uses arrays to pass large data values, such as textures and vertex buffers, to the library. Imposing a marshaling overhead on these transfers would greatly reduce the rendering performance of applications, so in these cases it is better to use the low-level representation. Thus, we see that policy used to handle data structures depends on the application.

4. Implementation overview

Figure 1 gives an overview of the architecture of FIG. Our front-end consists of a C preprocessor (CPP) library that records information about preprocessor symbols and the CKIT library for parsing and type checking ANSI C code. The front-end produces two objects: a map from preprocessor symbols to definitions and a typed abstract syntax tree representation of the C header file. The core of the FIG implementation is a programmable term-rewriting engine that translates the outputs of the front-end to FIG IDL, which is an intermediate representation of the glue between the HLL and C. The FIG compiler takes the user-supplied script and generates a rewriting program that the engine runs. It is this rewriting phase that fixes the *policies* being used in the foreign interface. The FIG IDL representation contains information about the high-level API that was generated, as well as the typemaps that define the glue code between the high-level and low-level representations. This representation is then translated to a HLL-specific output representation.

We have found rewriting to be particularly well-suited to our problem domain and we explain why in the next section. Following that, we describe FIG's typemaps and give examples of how they can be used to specify various FI policies.

5. Rewriting strategies

The FIG implementation uses a term rewriting engine to implement the translation from C header files to the high-level language API. We chose term rewriting as the evaluation model for several reasons:

- Source-to-source translations are naturally expressed as term rewriting systems.
- Term rewriting is a declarative programming model.
- The failure model of term rewriting supports composition of different policies.

This last point is perhaps the most important and deserves expansion.

Our term-rewriting engine is based on the approach of *rewriting strategies* proposed by Visser and Benaissa [VeAB98]. A *rewriting strategy* is a function from an environment and a term to either $\perp$ (denoting failure) or a new environment and term. The operational semantics of rewriting strategies can be defined using judgments of the form: $e, t \xrightarrow{s} e', t'$, which is read to mean that the strategy s maps the environment e and term t to a new environment e' and new term t'. If s fails when applied to e and t, we write $e, t \xrightarrow{s} \perp$.

The basic operations of pattern matching and term construction can be defined as strategies. If p is a template (*i.e.*, a term with variables), then $??p$ is a strategy for matching the template against a term. Matching is defined by the following two rules:

$$t, e \xrightarrow{??p} t, e' \qquad\qquad t, e \xrightarrow{??p} \perp$$

In the left rule, p matches t and e' is the enrichment of e with the pattern variables in p bound to the corresponding subterms in t.[4] While the right rule covers the case where p does not match the term t. Likewise, we can define term construction where template variables are replaced by their corresponding bindings in the environment:

$$t, e \xrightarrow{!!p} e(p), e$$

If p has variables that are not defined in e, then we get failure.

As we hinted above, what makes term rewriting a good match for our problem domain is the propagation of failure and the composition operators. For our purposes, the two most important composition operators are sequencing and asymmetric choice. If two strategies, s_1 and s_2, are in sequence, then we first apply s_1, and if that succeeds, then we apply s_2 to its result. If either fail, then we get failure. This behavior is formalized in the following rules:

$$\frac{e, t \xrightarrow{s_1} e', t' \quad e', t' \xrightarrow{s_2} e'', t''}{e, t \xrightarrow{s_1;s_2} e'', t''}$$

$$\frac{e, t \xrightarrow{s_1} \perp}{e, t \xrightarrow{s_1;s_2} \perp} \qquad\qquad \frac{e, t \xrightarrow{s_1} \perp \quad e, t \xrightarrow{s_2} \perp}{e, t \xrightarrow{s_1;s_2} \perp}$$

Sequencing can be used to construct conditional rewrites by preceding them by predicate strategies.[5]

The asymmetric choice of two strategies, s_1 and s_2, is a strategy that first attempts s_1 and if that fails it attempts s_2. This behavior is formalized in the following rules:

[4] The original formalization of rewriting strategies defines pattern matching and term building directly as strategies [VeAB98].

[5] A predicate is just a strategy that acts as the identity when true and fails when false.

$$\frac{e,t \xrightarrow{s_1} e',t'}{e,t \xrightarrow{s_1 <+ s_2} e',t'} \qquad \frac{e,t \xrightarrow{s_1} \bot \quad e,t \xrightarrow{s_2} e',t'}{e,t \xrightarrow{s_1 <+ s_2} e',t'}$$

$$\frac{e,t \xrightarrow{s_1} \bot \quad e,t \xrightarrow{s_2} \bot}{e,t \xrightarrow{s_1 <+ s_2} \bot}$$

We use asymmetric choice to combine user-defined rewrites (on the left) with default behaviors (on the right). If the user rewrite does not apply (*i.e.*, it failed), then the default is applied.

5.1 Rewriting combinators in SML

FIG's rewriting engine is implemented using a combinator library written in SML based on the "rewriting strategies" model described above. A strategy is an SML function with the type

```
datatype result = FAIL | OK of (env * term)
type strategy = (env * term) -> result
```

Implementing strategy combinators is quite simple, for example the combinator for sequencing a list of strategies is

```
fun seq [] res = res
  | seq (s::r) arg = (case s arg
       of FAIL => FAIL
        | OK arg => seq r arg)
```

and the asymmetric choice combinator is

```
fun <+ (s1, s2) arg = (case s1 arg
       of FAIL => s2 arg
        | res => res)
```

While these combinators may not be as efficient as the compiled approach used by STRATEGO, they make it easy to integrate rewriting into the rest of the FIG implementation infrastructure. For example, any SML function that matches the `strategy` type can be used as a strategy. Being able to escape to SML has greatly reduced the implementation effort, since many operations can be more directly programmed in SML than by using term rewriting.

6. Typemaps

The core task of a foreign-interface generator is to implement a mapping between high-level types and low-level representations. In FIG, we use the concept of a *typemap* to model this mapping.[6]

DEFINITION 1. *A* typemap θ *is a 4-tuple* $\langle mt, ct, \mathbf{m}, \mathbf{u} \rangle$*, where*

- mt *is a* high-level *representation type,*
- ct *is a low-level (*i.e.*,* C*) representation type,*
- $\mathbf{m} : mt \rightarrow ct$ *is a* marshalling *function that converts high-level values to their low-level representation, and*
- $\mathbf{u} : ct \rightarrow mt$ *is an* unmarshalling *function that converts low-level values to their high-level representation.*

It is also useful to consider *partial* typemaps, where either the $\mathbf{m}$ or $\mathbf{u}$ conversions are absent. For example, recall the `glGet` functions from Section 3.2 where the parameter-name arguments are constant. We use the following partial typemap

$$\langle \mathbf{void}, \mathbf{int}, \mathbf{val}(v), \cdot \rangle$$

[6] The term "typemap" was coined by Beazley for the SWIG system [Bea96, Bea04], but there are some significant differences between our typemaps and those of SWIG, which are discussed in Section 8.

c	::=	$\mathbf{void}$	discards value
	|	$\mathbf{val}(v)$	constant value
	|	$\mathbf{id}$	identity
	|	$\mathbf{wrap}$	wrap unboxed representation
	|	$\mathbf{unwrap}$	unwrap boxed representation
	|	$\#i$	select ith field of tuple
	|	$\mathbf{alloc}_n$	allocate n-tuple
	|	$\mathbf{stackalloc}_n\{c\}$	stack allocate temporary storage
	|	$\mathbf{prim}_{op}$	primitive operation
	|	$c_1; c_n$	sequencing
	|	$[c_1, \ldots, c_n]$	congruence
	|	$\Pi_m(i_1, \ldots, i_n)$	permute
	|	$c_1 \,\&\, \cdots \,\&\, c_n$	branching
	|	$\mathbf{entry}_{m \rightarrow n}\{c\}$	MOBY-callable function
	|	$\mathbf{centry}_n\{c\}$	C-callable function
	|	$\mathbf{call}_{m \rightarrow n}$	MOBY function call
	|	$\mathbf{ccall}_n$	C function call

Figure 2. Conversion combinators

where v is the constant value, to represent such constant parameters. FIG provides combinators for combining typemaps (described in Section 6.3), so we also need tuples of typemaps and typemaps that have tuples of types for either the low-level or high-level representation.

FIG uses typemaps as the basic building block for defining FI policies. We start with a collection of standard typemaps for basic types. For example, the typemap

$$\langle \texttt{wrap(double)}, \texttt{double}, \mathbf{unwrap}, \mathbf{wrap} \rangle$$

defines the conversion between high-level double-precision floating-point values, which have a wrapped representation, and C's `double` type. Using FIG's typemap combinators, these basic typemaps can be combined to to constructmore complicated typemaps that represent a wide range of FI policies. For example, a typemap for converting a C function like

```
double sqrt (double);
```

to a MOBY function

```
val sqrt : Double -> Double
```

can be constructed from the typemap for doubles shown above.

6.1 Conversions

The marshaling and unmarshaling conversions in a typemap are representations of the glue code needed to connect the high and low-level languages. We represent these functions using a simple combinator language that is inspired by the STRATEGO rewriting language. A simplified version of the combinators are given in Figure 2. A conversion works on one or more values and produces one or more results. In the implementation of FIG, conversions are just terms that specify marshaling and unmarshaling code. The backend translates these terms to glue code; this translation is described in Section 6.4.

We say that a conversion is *well-formed* if the arities of consecutive operators match up. We formalize the notion of well-formedness using a simple type system that tracks the arity of the combinators. Types in this system are defined as follows:

$$\begin{array}{rcll} \xi & ::= & \bullet & \\ & | & \langle \xi_1, \ldots, \xi_n \rangle & n \geq 0 \end{array}$$

where $\bullet$ represents a BOL type[7] and $\langle \xi_1, \ldots, \xi_n \rangle$ is an n-tuple of types. Note that while the nested-tuple structure of types is needed for bookkeeping purposes, these tuples have no runtime effect. We equate the types $\langle \xi \rangle$ and ξ and use the notation $\langle \bullet^n \rangle$ to denote a tuple of n BOL types. The typing judgement on conversions is

$$\vdash c : \xi \to \xi'$$

In the remainder of this section, we give the typing rules for the conversion combinators and informally describe their semantics.

The **void** combinator discards its argument, which is reflected in its type

$$\vdash \mathbf{void} : \bullet \to \langle\rangle$$

while the **val** combinator injects a new constant value

$$\vdash \mathbf{val}(v) : \langle\rangle \to \bullet$$

We also have an identity conversion (**id**), a conversion for wrapping unboxed values (**wrap**), a conversion for unwrapping boxed values (**unwrap**), and one for selecting a field from a heap-allocated tuple ($\#i$). These conversions all have the type $\bullet \to \bullet$. For heap allocation, we have $\vdash \mathbf{alloc}_n : \langle \bullet^n \rangle \to \bullet$. To support C return parameters, we provide a stack allocation combinator:

$$\frac{\vdash c : \langle \bullet, \xi \rangle \to \xi'}{\vdash \mathbf{stackalloc}_n\{c\} : \xi \to \xi'}$$

This combinator passes the address of n bytes of stack allocated memory as the first argument to the conversion c.

Access to the BOL primitive operations (including address arithmetic and load and store operations) is provided by the $\mathbf{prim}_{op}$ conversion, where op is a BOL primitive operation.

$$\frac{op \text{ takes } m \text{ arguments and produces } n \text{ results}}{\vdash \mathbf{prim}_{op} : \langle \bullet^m \rangle \to \langle \bullet^n \rangle}$$

The rule for sequencing is where types are forced to match

$$\frac{\vdash c_1 : \xi \to \xi' \quad \vdash c_2 : \xi' \to \xi''}{\vdash c_1; c_2 : \xi \to \xi''}$$

The congruence operator lifts a tuple of conversions to be a conversion of a tuple of values

$$\frac{\vdash c_i : \xi_i \to \xi_i' \quad \textit{for } 1 \le i \le n}{\vdash [c_1, \ldots, c_n] : \langle \xi_1, \ldots, \xi_n \rangle \to \langle \xi_1', \ldots, \xi_n' \rangle}$$

The permutation operator can be used to rearrange a tuple of values, including duplicating and dropping elements.

$$\frac{i_1, \ldots, i_n \in \{i \mid 1 \le i \le m\}}{\vdash \Pi_m(i_1, \ldots, i_n) : \langle \xi_1, \ldots, \xi_m \rangle \to \langle \xi_{i_1}, \ldots, \xi_{i_n} \rangle}$$

The branch operator maps a single value to a tuple

$$\frac{\vdash c_i : \xi \to \xi_i' \quad \textit{for } 1 \le i \le n}{\vdash c_1 \,\&\, \cdots \,\&\, c_n : \xi \to \langle \xi_1', \ldots, \xi_n' \rangle}$$

A conversion can be lifted to be a conversion from C functions to MOBY functions by using the **entry** combinator.

$$\frac{\vdash c : \langle \bullet, \langle \bullet^m \rangle \rangle \to \bullet^n}{\vdash \mathbf{entry}_{m \to n}\{c\} : \bullet \to \bullet}$$

Here, m is the number of arguments to the function and n is the number of results. The conversion c is applied to the pair of a C function and a tuple of the function's arguments. We also have a conversion combinator for converting from MOBY functions to C functions.

$$\frac{\vdash c : \langle \bullet, \langle \bullet^n \rangle \rangle \to \bullet}{\vdash \mathbf{centry}_n\{c\} : \bullet \to \bullet}$$

[7] Note that in practice, we track the actual BOL types of the values being translated, but we have not yet formalized this system.

The **call** conversion applies a MOBY function to its arguments.

$$\vdash \mathbf{call}_{m \to n} : \langle \bullet, \langle \bullet^m \rangle \rangle \to \bullet^n$$

Likewise, a the **ccall** conversion applies a C function to its arguments.

$$\vdash \mathbf{ccall}_n : \langle \bullet, \langle \bullet^n \rangle \rangle \to \bullet$$

6.2 A Conversion example

As an example of how these combinators work, consider the conversion from the C function type

```
double (*)(double, double);
```

to the MOBY function type[8]

```
$(Double, Double) -> Double
```

Applying such a conversion to an appropriately typed C function produces a wrapped function that can be called from MOBY.

The first step is to unbundle the MOBY function's argument, which has the type `$(Double, Double)` and is represented as a pointer to a pair of pointers to doubles. The conversion

$$\#0 \,\&\, \#1$$

will deconstruct the pair. We follow that by a congruence transformation to unwrap the doubles and we have the conversion needed to marshal the C function's argument.

$$\#0 \,\&\, \#1;\ [\mathbf{unwrap}, \mathbf{unwrap}]$$

The next step is to call the C function, using the **ccall** combinator. This combinator expects a pair of the C function and a tuple of the arguments, so we first need to embed the argument conversion in a congruence with the identity and then sequence it with the **ccall**

$$[\mathbf{id}, \#0 \,\&\, \#1;\ [\mathbf{unwrap}, \mathbf{unwrap}]];\ \mathbf{ccall}_2$$

The C call will return an unwrapped double, so we need to wrap its result

$$[\mathbf{id}, \#0 \,\&\, \#1;\ [\mathbf{unwrap}, \mathbf{unwrap}]];\ \mathbf{ccall}_2;\ \mathbf{wrap}$$

The final step is to lift this conversion up to a conversion on functions

$$\begin{array}{l}\mathbf{entry}_{1 \to 1}\{\\ \quad [\mathbf{id}, \#0 \,\&\, \#1;\ [\mathbf{unwrap}, \mathbf{unwrap}]];\ \mathbf{ccall}_2;\ \mathbf{wrap}\\ \}\end{array}$$

which defines the wrapper code needed to map a C function of the above C type to a MOBY function with the above MOBY type.

6.3 Typemap operators

The conversion combinators described in the previous section provide an "assembly code" for data marshalling. In practice, users work at the typemap level when defining FI policies. To facilitate this practice, FIG provides a collection of standard typemaps and typemap operators for combining typemaps.

One common operator is "$", which takes a tuple of typemaps and constructs a typemap that defines a mapping between a high-level heap-allocated tuple and a sequence of C values.

$$\begin{array}{l}\$(\langle mt_1, ct_1, \mathbf{m}_1, \mathbf{u}_1 \rangle, \ldots, \langle mt_n, ct_n, \mathbf{m}_n, \mathbf{u}_n \rangle)\\ \quad = \langle mt, ct, \mathbf{m}, \mathbf{u} \rangle\end{array}$$

[8] The "`$`" is the MOBY type operator for heap-allocated tuples.

where

$$\begin{array}{rcl} mt & = & \mathbf{struct}\{mt_1, \ldots, mt_n\} \\ ct & = & (ct_1, \ldots, ct_n) \\ \mathbf{m} & = & (\#1 \,\&\, \cdots \,\&\, \#n); [\mathbf{m}_1, \ldots, \mathbf{m}_n] \\ \mathbf{u} & = & [\mathbf{u}_1, \ldots, \mathbf{u}_n]; \mathbf{alloc}_n \end{array}$$

Note that the resulting typemap has a sequence of low-level types its low-level representation.

The congruence operator for typemaps, is defined as follows:

$$[m_1, \ldots, m_n] = \langle mt, ct, \mathbf{m}, \mathbf{u} \rangle$$

where

$$\begin{array}{rcl} m_i & = & \langle mt_i, ct_i, \mathbf{m}_i, \mathbf{u}_i \rangle \\ mt & = & (mt_1, \ldots, mt_n) \\ ct & = & (ct_1, \ldots, ct_n) \\ \mathbf{m} & = & [\mathbf{m}_1, \ldots, \mathbf{m}_n] \\ \mathbf{u} & = & [\mathbf{u}_1, \ldots, \mathbf{u}_n] \end{array}$$

Essentially, this operator is lifting the congruence on conversions to the typemap level.

6.4 Translating conversions

After the term-rewriting phase of FIG has selected the FI policies, the residual typemaps are translated to BOL code. In this section, we give a formal description of this translation. Since the combinators define conversions on nested tuples of values, we find it useful to mirror that nested structure in the handling of BOL variables in the translation to BOL code. We use x and y to denote meta variables in the translation and `a` to denote BOL variables. We assume a "gensym" mechanism for generating fresh BOL variables on demand. Variables are organized in a nested-tuple structure, called *patterns*, that follows the structure of conversion types (ξ). The set of patterns (Pat) is defined inductively as

$$\begin{array}{rcll} p, q & ::= & \texttt{a} & \text{BOL variable} \\ & | & x & \text{meta variable} \\ & | & \langle p_1, \ldots, p_n \rangle & n \geq 0 \end{array}$$

We write $\vec{x}$ for $\langle x_1, \ldots, x_n \rangle$ (*i.e.*, a tuple of meta variables). We say that a pattern p *matches* a type ξ (written $p : \xi$) if they have isomorphic structures.

The translation of a conversion c, is specified by the expression $[\![c]\!]\, p\, k$, where p is a pattern matching the inputs to c and k is a "continuation" that consumes the output of the conversion. This translation has the following type:

$$[\![\cdot]\!] \; : \mathrm{Pat} \to (\mathrm{Pat} \to \mathrm{BOL}) \to \mathrm{BOL}$$

where BOL is the type of BOL expression terms produced by FIG. Note that the nested value structure does not represent runtime data structures, but, rather, is just a bookkeeping tool used in the specification. The formal definition of the translation is given in Figure 3. We follow the convention of using `teletype` font for the BOL syntactic terms. As can be seen from this figure, some conversion combinators, such as **void**, **id**, and permutation, do not produce code, but instead just act as plumbing. Most of the conversions that map to actual BOL code are mapped to a let-binding, where the conversion's arguments are used on the right-hand-side of the let and the left-hand-side is passed to the conversion's continuation.

To illustrate the translation, we consider the translation of a conversion that maps maps a heap-allocated pair of wrapped doubles to a pair of doubles:

$$[\![\#0 \,\&\, \#1; [\mathbf{unwrap}, \mathbf{unwrap}]]\!]\, x\, k$$

This translation evaluates to the following BOL fragment:

```
let t1 = #0(x) in
let t2 = #1(x) in
let t3 = unwrap(t1) in
let t4 = unwrap(t2) in
  k(t3, t4)
```

7. FIG by example

The declarative FIG scripting language allows the users to specify the policies for translating C header files to MOBY interfaces with their supporting glue code. There are two kinds of commands in the scripting language: definitions and rule-applications. A definition command defines typemaps or their components (such as types and conversions), which are usually used by later rule-application commands. A rule-application command usually consists of two parts: matching and building. The matching part provides a pattern embraced by backquotes to match the C term, and the building part usually defines a MOBY term which corresponds to a MOBY type, constant or function. There are *meta variables* carrying information about the C term from the matching part to the building part. The values of meta variables are filled in when this rule applies to a C term successfully.

7.1 Constants

For our first example, we revisit the `glBegin` function that we discussed in Section 3.1. Recall that this function takes an integer argument that can have one of ten legal values; any other value results in an OpenGL runtime error.

The most straightforward way is to translate the ten begin-mode constants to integer constants,[9] and to translate the prototype of `glBegin` to a function that takes an integer argument. This translation results in the following MOBY interface:

```
const GL_POINTS : Int
const GL_LINES  : Int
...
val glBegin : Int -> ()
```

This translation is easy to specify, since it matches FIG's defaults; the script is as follows:

```
MATCH `#define {GL_(POINTS | LINES
    | LINE_STRIP | LINE_LOOP | TRIANGLES
    | TRIANGLE_STRIP | TRIANGLE_FAN | QUADS
    | QUAD_STRIP | POLYGON)}  _`
  => CONST default
MATCH `void glBegin (GLenum mode)`
  => FUNCTION default
```

The first command uses a regular expression to match the names of C constant definitions and defines the corresponding MOBY constants using the default mapping from the FIG library. The second rule uses the default mapping for simple function types.

For a strongly-typed high-level language like MOBY, however, we want better static type checking than provided by this interface. The following MOBY interface illustrates the approach described in Section 3.1:

```
type BeginMode
const POINTS : BeginMode
const LINES : BeginMode
```

[9] The **const** definitions is a MOBY mechanism that allows the programmer to define data constructors and constants independently of datatypes [AR92].

$$
\begin{array}{rcl}
[\![\mathbf{void}]\!]\, x\, k & = & k\,\langle\rangle \\
[\![\mathbf{val}(v)]\!]\,\langle\rangle\, k & = & \mathbf{let}\,\mathtt{t} = v\,\mathbf{in}\,k(\mathtt{t}) \qquad \text{where } \mathtt{t} \text{ is a fresh BOL variable} \\
[\![\mathbf{id}]\!]\, x\, k & = & k(x) \\
[\![\mathbf{wrap}]\!]\, x\, k & = & \mathbf{let}\,\mathtt{t} = \mathbf{wrap}(x)\,\mathbf{in}\,k(\mathtt{t}) \qquad \text{where } \mathtt{t} \text{ is a fresh BOL variable} \\
[\![\mathbf{unwrap}]\!]\, x\, k & = & \mathbf{let}\,\mathtt{t} = \mathbf{unwrap}(x)\,\mathbf{in}\,k(\mathtt{t}) \qquad \text{where } \mathtt{y} \text{ is a fresh BOL variable} \\
[\![\#i]\!]\, x\, k & = & \mathbf{let}\,\mathtt{t} = \mathbf{\#}i(x)\,\mathbf{in}\,k(\mathtt{t}) \qquad \text{where } \mathtt{t} \text{ is a fresh BOL variable} \\
[\![\mathbf{alloc}_n]\!]\,\vec{x}\, k & = & \mathbf{let}\,\mathtt{t} = \mathbf{alloc}(\vec{x})\,\mathbf{in}\,k(\mathtt{t}) \qquad \text{where } \mathtt{t} \text{ is a fresh BOL variable} \\
[\![\mathbf{stackalloc}_n\{c\}]\!]\, p\, k & = & \mathbf{let}\,\mathtt{t} = \mathbf{stackalloc}(n)\,\mathbf{in}\,k\,\langle \mathtt{t}, p\rangle \qquad \text{where } \mathtt{t} \text{ is a fresh BOL variable} \\
[\![\mathbf{prim}_{op}]\!]\,\langle x_1, \ldots, x_m\rangle\, k & = & \mathbf{let}\,(\mathtt{t}_1, \ldots, \mathtt{t}_n) = op(x_1, \ldots, x_m)\,\mathbf{in}\,k\,\langle \mathtt{t}_1, \ldots, \mathtt{t}_n\rangle \\
 & & \qquad \text{where the } \mathtt{t}_i \text{ are fresh BOL variables} \\
[\![c_1; \cdots; c_n]\!]\, p_1\, k & = & [\![c_1]\!]\, p_1\, (\lambda p_2.[\![c_2]\!]\, p_2\, (\ldots (\lambda p_n.[\![c_n]\!]\, p_n\, k)\ldots)) \\
[\![[c_1, \ldots, c_n]]\!]\,\langle p_1, \ldots, p_n\rangle\, k & = & [\![c_1]\!]\, p_1\, (\lambda q_1.[\![c_2]\!]\, p_2\, (\ldots (\lambda q_n.k\langle q_1, \ldots, q_n\rangle)\ldots)) \\
[\![\Pi_m(i_1, \ldots, i_n)]\!]\,\langle p_1, \ldots, p_m\rangle\, k & = & k\langle p_{i_1}, \ldots, p_{i_n}\rangle \\
[\![c_1 \,\&\, \cdots \,\&\, c_n]\!]\, p\, k & = & [\![c_1]\!]\, p\, (\lambda q_1.[\![c_2]\!]\, p\, (\ldots (\lambda q_n.k\langle q_1, \ldots, q_n\rangle)\ldots)) \\
[\![\mathbf{entry}_{m\to n}\{c\}]\!]\, x\, k & = & \mathbf{fun}\,\mathtt{f}\,(\vec{\mathtt{t}}) \;=\; [\![c]\!]\,\langle x, \vec{\mathtt{t}}\rangle\,(\lambda\vec{y}.\mathbf{return}\,(\vec{y}))\,\mathbf{in}\,k(\mathtt{f}) \\
 & & \qquad \text{where } |\vec{\mathtt{t}}| = m \text{ and the } \vec{\mathtt{t}} \text{ are fresh BOL variables, and } |\vec{y}| = n. \\
[\![\mathbf{centry}_n\{c\}]\!]\, x\, k & = & \mathbf{cfun}\,\mathtt{f}\,(\vec{\mathtt{t}}) \;=\; [\![c]\!]\,\langle x, \vec{\mathtt{t}}\rangle\,(\lambda y.\mathbf{return}\,(y))\,\mathbf{in}\,k(\mathtt{f}) \\
 & & \qquad \text{where } |\vec{\mathtt{t}}| = n \text{ and the } \vec{\mathtt{t}} \text{ are fresh BOL variables.} \\
[\![\mathbf{call}_{m\to n}]\!]\,\langle f, \vec{x}\rangle\, k & = & \mathbf{let}\,\vec{\mathtt{t}} = \mathbf{call}\,f(\vec{x})\,\mathbf{in}\,k(\vec{\mathtt{t}}) \\
 & & \qquad \text{where } |\vec{x}| = m, |\vec{\mathtt{t}}| = n, \text{ and the } \vec{\mathtt{t}} \text{ are fresh BOL variables.} \\
[\![\mathbf{ccall}_n]\!]\,\langle f, \vec{x}\rangle\, k & = & \mathbf{let}\,\mathtt{t} = \mathbf{ccall}\,f(\vec{x})\,\mathbf{in}\,k(\mathtt{t}) \qquad \text{where } |\vec{x}| = n \text{ and } \mathtt{t} \text{ is a fresh BOL variable}
\end{array}
$$

Figure 3. Translating conversion combinators to BOL

```
...
val begin : BeginMode -> ()
```

This translation requires more work on the part of the FIG user, since we must specify a new type and the assignment of the constants to the type. We have also stripped the `GL_` prefixes from the constants (and the `gl` prefix from the function), since the MOBY module system will provide the necessary name-space management. The FIG script defining this interface is as follows:

```
MOBYTYPE BeginMode = (int, int, id, id)
MATCH `#define {GL_?name{POINTS | LINES
       | LINE_STRIP | LINE_LOOP | TRIANGLES
       | TRIANGLE_STRIP | TRIANGLE_FAN | QUADS
       | QUAD_STRIP | POLYGON}} ?i`
  => CONST {NAME=!name,
            VALUE=!i,
            TYPE = BeginMode}
MATCH `void {glBegin} _`
  => FUNCTION {NAME = "begin",
               TYPE = BeginMode -> (),
               GLUECODE = default}
```

The first command defines a MOBY type `BeginMode` using a typemap specifying its BOL representation, and conversions from/to the C `int` type. In this way, in the MOBY interface, `BeginMode` appears as an abstract type, and the underlying BOL representation is hidden. In the second command, "`?name`" and "?i" are meta variables in the matching part, and their values are defined in when this rule successfully applies to a C term. The building part of the rule uses the syntax "`!name`" and "`!i`" to refer to the variables' values. The third command defines a MOBY function `begin` that takes an argument of type `BeginMode`. The glue code of the function can be generated automatically by FIG using the `BeginMode` typemap.

7.2 Pattern matching

Libraries often have common naming conventions and type structure. FIG provides pattern-matching mechanisms to allow the FIG user to take advantage of these patterns and factor the FIG script at several levels. For example, the `gl.h` header has ten functions that have prototypes of the form

```
void glOperation3f (GLfloat, GLfloat, GLfloat);
```

By pattern matching the names of the functions, we can define a single FIG rule that will uniformly translate these prototypes to the form

```
val operation : (Float, Float, Float) -> ()
```

This generic translation can be defined by the following script:

```
MATCH `void {gl?name{[a-zA-Z]*}3f}
    (float, float, float)`
  => FUNCTION { NAME = lower (!name),
    TYPE = (Float, Float, Float) -> ()
    GLUECODE = default }
```

The function `lower` converts the letters in a string to lower case. For example, the above script command translates the C function

```
void glColor3f (GLfloat, GLfloat, GLfloat);
```

to

```
val color : (Float, Float, Float) -> ()
```

We can further take advantage of the fact that the OpenGL API follows the convention of encoding the arity and type information of an operation into its name and write a single rule for translating all of the fixed arity operations. FIG patterns involve both regular expressions (useful in the OpenGL example) and the structure of

the terms that represent the C type information, so it is possible to write generic rules even when the source library does not follow strict naming conventions.

7.3 C data structures

While making it easy to generate interfaces to C functions with simple types is important, we also want to support interfaces to libraries that have more complicated C data structures that might not easily match the higher-level language's representations. We consider two such examples in this section.

In many cases, OpenGL uses small fixed-size C arrays to pass data to the library. For example, the following function is used to set a number of lighting properties:

```
void glLightfv (
    GLenum light,
    GLenum pname,
    const GLfloat *params)
```

One approach to translating this function is to embed the C types into the high-level language as is done by the CHARON and NLFFIGEN tools. For MOBY, the resulting type would be

```
val glLightv : (Int, Int, C_Ptr(Float)) -> ()
```

where `C_Ptr(Float)` is an abstract type for accessing the C array representation. While this approach has the advantage that it can handle the whole range of C interfaces (except for *varargs*), it has the disadvantage that it type of the third argument does not specify the array size. On the other hand, we could define the MOBY function as follows:

```
val lightv:
      (Int, Int, $(Float, Float, Float, Float))
        -> ()
```

In this interface, the C array type "`const GLfloat *`" is mapped to a high-level representation — a heap-allocated tuple. This interface has stronger type-safety guarantees and fits MOBY better than the CHARON generated interface.

The script of this translation is a little complicated. An important part is to define a typemap between the C array and MOBY tuple types. The script is as follows:

```
TYPEMAP tymap1 = (int, int, id, id)
TYPE m_float = wrap(float)
CONVERSION moby2c = {
  (#0 & #1 & #2 & #3);
  all(unwrap);
  stackalloc;
  [puta 0, puta 1, puta 2, puta 3];
  ptr}
CONVERSION c2moby = {
  ref;
  (geta 0 & geta 1 & geta 2 & geta 3);
  all(wrap);
  alloc}
TYPEMAP tymap2 = (
  $(m_float, m_float, m_float, m_float),
  ptr(float),
  moby2c,
  c2moby)
TYPEMAP tymap3 = [tymap1, tymap1, tymap2]
MATCH `void glLightv ?args`
  => FUNCTION {
    NAME = "lightv",
    TYPE = (x:Int, y:Int,
        z:$(Float, Float, Float, Float)) -> (),
    GLUECODE = {%toC (a,b,c) = tymap3 (x,y,z),
                %ccall = glLightv (a,b,c)}}
```

In this script, we first define `tymap1` to be the identity typemap on integers. The typemap `tymap2` is defined to map between a C float pointer and a heap-allocated 4-tuple of MOBY `Float` values. This typemap consists of two conversions: `moby2c` and `c2moby`. The first works by decomposing the tuple into four values, then unwrapping the values (the "`all`" operator lifts a conversion to a congruence), then allocating temporary space in the stack, storing the values in the temporary storage, and finally yielding the address of the storage. The conversion operations `geta` and `puta` mean loading and storing array elements. Typemap `tymap3` is the typemap for the arguments of the `glLightfv` function, which we then use in the glue code to convert from MOBY arguments to C.

8. Related work

Term rewriting has been used to implement a number of program-transformation systems [Vis01], but we are not aware of it previously being used to generate foreign interfaces. The main body of related work is the large collection of tools that people have developed for generating foreign interfaces. We compare FIG to a representative sample.

The FFIGEN [Han96] tool for SCHEME is perhaps the closest in design to FIG. It translates a C header file into a collection of SCHEME data structures for the constants, types, prototypes, etc. User-written SCHEME code is then applied to these data structures to generate the foreign interface. The user code can be specialized to both the target system and the source header file. The main difference in our approach is that we are using a declarative scripting language to specify the transformation and a term-rewriting engine to implement the transformation, instead of a general-purpose programming language. Furthermore, FFIGEN does not have any notion of typemaps.

As we mentioned in Section 6, the notion of typemaps was invented by Beazley in SWIG [Bea96, Bea04], which is a tool that uses a combination of user-supplied scripts and information mined from header files to generate foreign interfaces for a number of high-level languages (most notably Python, but also JAVA, OCAML, Perl, Tcl, and others). While SWIG does allow customization of interfaces, it does not support data-level interoperability (it generates C functions for that have to be called to access C data structures). Typemaps in SWIG have an object-oriented flavor and are more substantial those of FIG. Specifically, they have a pattern matching component that controls when they are applied and they can have a number of "methods" in addition to the basic marshaling/unmarshaling methods (*e.g.*, error checking code, cleanup code, *etc.*). Methods are specified as C code fragments with meta-variables that get expended when SWIG generates the glue code.

One popular approach to FI generation are tools that use *Interface Definition Language* (IDL) specifications to generate glue code [FLMP99]. The advantage of this approach is that IDL is a standard (two, actually) and there are many existing IDL specifications (especially for Microsoft's APIs). Unfortunately, the IDL approach does not provide much room for customizing the API and does not support data-level interoperability for data structures.

GREENCARD [Tea] is a FI preprocessor for HASKELL. Similar to FIG, GREENCARD supports application-specific FIs, but requires that the entire interface be specified. It does not extract any information from C header files. GREENCARD has the mechanism of *data-interface schemes* (DIS) that are similar to typemaps.

Another HASKELL FI generator is `C->Haskell` [Cha99]. This tool analyses C header files to determine the data layout and function prototypes and then processes a HASKELL template that

contains hooks for foreign references. The hooks are expanded by the tool to produce the foreign interface consisting of a HASKELL file and a C file that implements glue code. This tool support customization of the interface, but the set of hooks is fixed so the customization has to be done by writing HASKELL (or C) code.

An alternative to mapping a low-level API into a high-level one, is to embed the low-level API directly in a high-level language. The tools NLFFIGEN [Blu01] and CHARON [FPR01] follow this approach. They take raw C header files and generate an embedding of the C types into the high-level language. While the resulting code is efficient and requires little user effort to generate, it low-level. Essentially, one is writing C code using the high-level language syntax.

9. Conclusion

In this paper, we have argued for foreign-interface generation tools that support the generation of application-specific interfaces. We have presented FIG, which is a tool that we have developed to meet this need. FIG processes the raw C header files together with a declarative FIG script, using the script as the guide of transformation, and produces foreign interface code for MOBY programs. FIG's scripting language is based on a theory of typemaps and term rewriting strategies. FIG's scripting language is designed to have good defaults, which minimize the user effort for most foreign functions, but at the same time, users have almost unlimited power to customize the foreign interface.

The typemapping between high-level types and low-level types is the core of a foreign interface and our composable typemaps make it possible for users to define complicated and even higher-order typemaps from basic building blocks. We have presented a formal description of typemaps, including a formal translation from our conversion combinators to BOL expressions.

We are continuing this work in several ways. First, we are working on a type system for conversions that tracks the BOL types. We believe that this type system will allow us to prove that well-formed typemaps produce type-correct glue code (in the sense of Furr and Foster [FF05]). We are also continuing to refine the FIG scripting language to better cover common idioms. Our current implementation is missing a number of important features that we plan to add. This include support for error handling (*i.e.*, mapping error return values in C functions to exceptions) and handling constant expressions in header files. The architecture of FIG is flexible enough to support other targets and we are currently implementing a version for SML.

References

[AR92] Aitken, W. E. and J. H. Reppy. Abstract value constructors: Symbolic constants for standard ml. *Technical Report TR 92-1290*, Dept. of CS, Cornell University, June 1992. A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

[Bea96] Beazley, D. M. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *TCL'96*, July 1996.

[Bea04] Beazley, D. *SWIG-1.3 Documentation*, December 2004. Available from `www.swig.org`.

[Blu01] Blume, M. No-longer-foreign: Teaching an ML compiler to speak C "natively". In N. Benton and A. Kennedy (eds.), *BABEL'01*, vol. 59 of *ENTCS*, New York, NY, September 2001. Elsevier Science Publishers. Available from `http://www.elsevier.nl/locate/entcs/volume59.html`.

[CFH+03] Chakravarty, M. M. T., S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. Peyton Jones, A. Reid, M. Wallace, and M. Weber. *The Haskell 98 Foreign Function Interface 1.0*, 2003. Available from `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.

[Cha99] Chakravarty, M. M. T. C → Haskell, or yet another interfacing tool. In *IFL'99*, vol. 1868 of *LNCS*, New York, NY, 1868 1999. Springer-Verlag.

[FF05] Furr, M. and J. S. Foster. Checking type safety of foreign function calls. In *PLDI'05*, New York, NY, June 2005. ACM, pp. 62–72.

[FLMP99] Finne, S., D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *ICFP'98*, September 1999, pp. 153–162.

[FPR00] Fisher, K., R. Pucella, and J. Reppy. Data-level interoperability. *Technical report*, Bell Labs, Lucent Technologies, April 2000. Available from `http://moby.cs.uchicago.edu`.

[FPR01] Fisher, K., R. Pucella, and J. Reppy. A framework for interoperability. In N. Benton and A. Kennedy (eds.), *BABEL'01*, vol. 59 of *ENTCS*, New York, NY, September 2001. Elsevier Science Publishers. Available from `http://www.elsevier.nl/locate/entcs/volume59.html`.

[FR99] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, New York, NY, May 1999. ACM, pp. 37–49.

[Han96] Hansen, L. T. *FFIGEN User's Manual*, February 1996. Available from `www.ccs.neu.edu/home/lth/ffigen/`.

[Hue95] Huelsbergen, L. A portable C interface for Standard ML of New Jersey. *Technical report*, Bell Laboratories, November 1995. Available from `http://www.cs.bell-labs.com/who/lorenz/papers/smlnj-c.pdf`.

[Lia99] Liang, S. *The Java Native Interface*. Addison-Wesley, Reading, MA, 1999.

[Ope04] OpenGL Architecture Review Board. *The OpenGL Graphics System: A Specification (Version 1.5)*, 2004.

[Tea] Team, T. G. C. *The Green Card User's Guide*. Available from `www.haskell.org/greencard`.

[VeAB98] Visser, E. and Z. el Abidine Benaissa. A core language for rewriting. In *WRLA'98*, vol. 15 of *ENTCS*, New York, NY, September 1998. Elsevier Science Publishers.

[Vis01] Visser, E. A survey of rewriting strategies in program transformation systems. In *WRS'01*, vol. 57 of *ENTCS*, New York, NY, May 2001. Elsevier Science Publishers.