SLIRP Reference Manual, Version 1.9.8

Michael S. Noble, mnoble@space.mit.edu Sep 23, 2009

Preface

SLIRP is the (Sl)ang (I)nte(r)face (P)ackage.

This software was partially developed at the MIT Center for Space
Research, under contract SV1-61010 from the Smithsonian Institution.

Contents

Chapter 1

Introduction

SLIRP, the (SL)ang (I)nte(R)face (P)ackage, is a vectorizing code generator aimed primarily at simplifying the process of creating *modules* for the S-Lang scripting language. It can dramatically reduce the time and effort required to make C, C++, or Fortran code callable directly from the S-Lang interpreter, automatically vectorize functions to take advantage of the powerful numerical and array capabilities native to S-Lang, generate parallelizable wrappers for OpenMP-aware compilers, and even generate `Makefiles` to automate the build process. SLIRP may also be used to generate pure C bindings for C++ code, or empty (stub) implementations for the interface specified by its input. These features are more general in nature and the emitted code has no dependencies upon S-Lang whatsoever. SLIRP has also been directly or indirectly employed in the publication of a number of scientific papers.

1.1 Brief History

SLIRP grew out of an effort by the author to develop SLgtk, the S-Lang bindings to the Gimp Toolkit (more popularly known as Gtk). To it's credit the Gtk development community has paid close attention to the problem of binding Gtk to other languages, and mechanisms which considerably aid that process are readily available. Oddly enough, many Gtk language bindings do not use SWIG, but instead rely upon so-called .defs files, which are descriptions – in the Scheme language – of the apis of the underlying libraries, apparently generated from the header files and then supplemented with additional semantics.

In typical hacker fashion the SLgtk bindings project began by building upon the work of someone else, which in this case amounted to borrowing the .defs files from PyGtk and Gtk itself. While this proved a useful starting point, over time enough undocumentedness/inconsistencies/omissions were encountered to undermine my confidence, so I opted to utilize the .defs files when they provided information not readily available in header files (e.g., to identify functions which accept `NULL` for one or more arguments), and wrote a minimal generator to fabricate the bulk of the bindings directly from the Gtk headers. After all, even though the source may not tell the complete story it certainly doesn't forget and never lies. Relying too heavily upon .defs files could also limit the potential scope of the generator, making it too specialized to be used on other libraries which do not describe their apis in such fashion.

Given that SWIG currently does not support S-Lang a logical question to ask is why I did not write a S-Lang extension for SWIG and use it to generate SLgtk? Well, part of the answer lies above, namely that if Gtk were "more SWIG-able" then presumably more developers would be using SWIG for Gtk bindings generation. Minimizing external dependencies was another significant motivation; SLIRP is 99% pure S-Lang code, and about 1% C, so if you've got S-Lang and a C compiler installed (e.g., as millions of Linux users do by default) you can use SLIRP immediately, without installing any additional packages. Other factors included having greater control over the generated code, as well as providing the ability to support Fortran, generate vectorized wrappers (optionally parallelized with OpenMP for multiprocessors), and generate Makefiles, none of which were offered by SWIG.

1.2 Installation

The remainder of this manual assumes you have installed SLIRP as outlined in the README at the top of the distribution. A number of sample modules are provided in the `examples` subdirectory.

1.3 The `kitchensink` Module

As our first example, consider a library named `kitchensink` whose public interface `ksink.h` contains

```
 typedef struct { char *name; double value; } KDatum;
typedef struct { double p[3]; unsigned long id; } KParams;

typedef enum { KSINK_GOOD=0, KSINK_BAD, KSINK_UGLY, KSINK_HORRIFIC } KErrorCode;

extern      long            ksink_sum               (long augend, long addend);
extern      double          ksink_mult              (double op1, double op2);

extern      KDatum          *ksink_datum_new        (char *name, double value);
extern      KParams*        ksink_params_new   (unsigned long id, double params[3]);
extern      const char*     ksink_params_str        (KParams *p);
extern      void            ksink_datum_destroy     (KDatum *datum);
extern      void            ksink_set_ref_i         (int *i);

extern      void            ksink_print_array_f     (long nelems, float array [  ]);
extern      char**          ksink_make_array_s      (void);
extern      void            ksink_print_array_s     (long   nelems, char **array);
extern      char**          ksink_make_array_nts    (void);
extern      void            ksink_print_array_nts   (char **array);

extern      void            ksink_print_error       (KErrorCode err);

extern      void            ksink_swap_double       (double *i, double *j);
extern      void            ksink_close             (int fd);
extern      FILE*           ksink_fopen             (char *name, char *mode,
                                                     unsigned long  *size,
                                                     unsigned short *nlinks);
```

Further suppose that one day that you misplaced your wits and decided it was absolutely necessary that you be able to call this library from a S-Lang script. How would you go about such? This is the problem SLIRP helps solve. Let's begin by issuing the command

```
unix% slirp ksink.h
```

which will generate, amongst others, the wrapper

```
static void sl_ksink_mult (void)
{
   double result;
   double arg1;
   double arg2;

   if (SLang_Num_Function_Args != 2 ||
        SLang_pop_double(&arg2) == -1 ||
        SLang_pop_double(&arg1) == -1 )
        {Slirp_usage_err("double = ksink_mult(double,double)"); return;}

   result = ksink_mult(arg1, arg2);
   (void) SLang_push_double ( result);
}
```

SLIRP provides a number of ways of tailoring its operation and output, but in this instance the code is emitted to a file named `ksink_glue.c` in the current working directory. Next we need to make the S-Lang interpreter aware of this wrapper by installing it as a so-called *intrinsic* function. There's more than one way to do this, but the best practice is to create an entry for each wrapper within an *intrinsic function table*, and register all of them at once with a single call to either `SLadd_intrin_fun_table()` or its namespace variant `SLns_add_intrin_fun_table()`. To assist the language binder with this SLIRP also generates an intrinsic function table entry for each each wrapper function that it generates. In our example this entry looks like

```
MAKE_INTRINSIC_0("ksink_mult",sl_ksink_mult,V),
```

and is also emitted to `ksink_glue.c`. Now all that remains is to provide a means of binding the generated code to the S-Lang interpreter at runtime. For this we need another code fragment to initialize the module, something along the lines of

```
#include "slang.h"
#include "ksink.h"

static SLang_Intrin_Fun_Type ksink_Intrin_Funcs[] =
{
  MAKE_INTRINSIC_0("ksink_mult",sl_ksink_mult,V),
  ...
  SLANG_END_INTRIN_FUN_TABLE
};

SLANG_MODULE(ksink);
int init_ksink_module_ns(char *ns_name)
```

```
{
   SLang_NameSpace_Type *ns = NULL;

   if (ns_name != NULL) {
        ns = SLns_create_namespace (ns_name);
        if (ns == NULL)
           return -1;
   }

   if (-1 == SLns_add_intrin_fun_table (ns, ksink_Funcs, "__ksink__"))
        return -1;

   return 0;
}
```

Typically the file generated by SLIRP will be compiled via Makefile rules, to build an *importable module* which may be accessed at runtime via the S-Lang `import()` function. In our case this would instruct the S-Lang runtime to dynamically load a shared object library named `ksink-module.so`, invoke its `init_ksink_module_ns()` initializer, and register the wrapper functions, constants, and variables defined within.

Here's how our `ksink_mult` wrapper might be used within a S-Lang script:

```
import("ksink");

define do_mult(op1, op2)
{
   print("ksink_mult(%S, %S) = %S", op1, op2, ksink_mult(op1,op2));
}

do_mult(333, 3);
do_mult(PI/2, 2);
```

For more details consult the code for this example, which is located in the `examples/kitchensink` directory of the SLIRP distribution and can be built and invoked by typing `make` followed by `make demo` at the command line. The `modules` directory within the S-Lang source distribution also contains a number of useful modules, as does the MIT modules page at http://space.mit.edu/cxc/software/slang/modules/ .

1.4 Expectations

It is important to realize that SLIRP is *no silver bullet*. In the abstract code generators are not meant to free the programmer from the responsibility of thinking *at all*, but rather aim at freeing them to think about *only* that which *requires* substantive thought, leaving the rest of the presumably mundane details to be mechanically churned out by a machine.

For a variety of reasons the developer may prefer to wrap portions of a library manually, instead of relying upon wrappers emitted by a generator. As such most bindings projects will likely include both generated and hand-crafted code, so it is pragmatic to view a code generator as but one element in the toolbox of the language binder.

It should also be noted that SLIRP does not fully preprocess, nor is it a compiler for, the C language. By choosing a S-Lang-based implementation (instead of one, say, in C, generated from a grammar specified in Lex/Yacc) we exchange completeness for size, simplicity of design and distribution, ease of use, and portability. However, as discussed in section 3.5.1 (Pointers), even if SLIRP contained a full C preprocessor or compiler it would still be impossible for it to mechanically map every possible construct from the universe of syntactically valid C libraries to an equivalent construction in S-Lang. For these reasons code generators provide mechanisms to customize their behavior and layer additional semantics on top of the underlying api (such as the .defs mechanisms described earlier, or the interface files respectively employed by SWIG and SLIRP). Nevertheless, SLIRP is known to generate useful bindings for numerous C, C++, and Fortran codes, including SLgtk, `OpenGL`, `MySQL`, `PVM`, `libGlade`, `glibc (gettext)`, `HDF5`, `NetCDF`, `cgilib`, `ASURV`, and `volpack`.

Chapter 2

Code Generation

In the broadest sense SLIRP *consumes* an interface definition and *produces* one or more wrapper products. The input interface is specified via one or more *source files* (C or C++ headers, and/or Fortran source), supplemented with an optional SLIRP *interface file*. The generated output may take the form of:

- S-Lang wrappers for functions, constants, and variables defined in the interface
- stub (empty) implementations for the interface
- C bindings for select C++ codes
- Makefiles
- a stdout dump of the interface

By default SLIRP generates S-Lang wrappers; the other output forms are generated by specifying the `-stubs`, `-cfront`, `-make`, and `-print` options, respectively, at invocation.

2.1 Wrapper Functions

SLIRP is capable of wrapping most routines defined in most C and Fortran 77 codes. Its C++ support is less comprehensive, but broad enough to generate useful bindings for a variety of C++ codes. Although the code generated by SLIRP has been verified compliant with more modern Fortran compilers (such as gfortran or f95), there is no explicit support for extensions to the language (e.g. objects or modules) which appear in newer versions of the Fortran standard.

There are three conditions under which SLIRP will not wrap a given function. The first two occur when the `-noautotype` switch has been specified on the command line and there is no

- typemap entry matching the function return type
- matching typemap entry for one or more parameters within the function parameter list

For example, since SLIRP does not contain typemaps for function pointers any function whose signature contains one will not be wrapped. SLIRP will also ignore a function

- when it has been explicitly instructed to do so

This can be useful when you would prefer to code a wrapper manually rather than accept what SLIRP would generate, or if you'd rather not wrap the function at all. The exact mechanisms for ignoring symbols are discussed in the next section.

2.1.1 Wrapper Function Structure

Each S-Lang wrapper function has the general form

```
Function_Declaration
{
   Variable_Declaration_Block
   Argument_Marshal_Block
   Function_Call_Block
   Return_Block
}
```

Wrappers are declared with static linkage, to avoid clashes with other modules that might define similar symbols. Arguments are passed from S-Lang scope in one of two ways: via the function parameter list (for simple scalar types) or within the argument marshalling block. In the latter case the wrapper is declared with a `void` argument list and each argument is explicitly popped from the S-Lang stack and verified to match in type and number the arguments required by the wrapped function. The function call block invokes the underlying routine, while the return block is responsible for finalization tasks such as deallocating temporary resources and pushing the return value onto the S-Lang stack.

Through the use of annotations it's possible to automatically construct S-Lang wrappers whose signatures do not match that of the underlying function, by either accepting fewer arguments or returning more values.

2.1.2 Usage Statements

By default the argument marshalling block also contains code to emit **Usage:** statements when the function has been invoked with the incorrect number or type of arguments. These serve as useful reminders of the purpose of the function, for both new and veteran users of a library, and can often save a trip to the documentation. See the `-nopop` runtime switch for more details.

2.2 Interface Files

Although many command line options are provided to steer the code generation process, interface files are by far the more powerful means of customizing and extending SLIRP, and their description occupies the bulk of this manual. Interface files are S-Lang scripts, and so may contain any legal collection of S-Lang statements. They are also optional, in the sense that SLIRP does not require one for nominal execution; most bindings projects, however, would likely benefit from their use.

The SLgtk interface file, for example, contains hundreds of type mappings and ignore directives, among other content. At the other end of the scale, the `kitchensink` module could in principle be generated from an interface file containing only

```
slirp_define_opaque("KDatum",NULL,"ksink_datum_destroy");
```

While most SLIRP interface files won't be this brief, generating one for a first cut module for the typical library will usually take only a few minutes. Additional interface files are bundled in the examples subdirectory of the SLIRP distribution.

2.2.1 Loading

By default SLIRP looks for an interface file named `slirprc` in the current working directory; failing that, SLIRP then attempts to load the file specified by the environment variable $SLIRPRC. Specifying the `-rc` switch at invocation time overrides both of these, provided the given file is readable.

2.2.2 Nesting

An interface file may reference other interface files by calling

```
slirp_include_rc(filename)
```

This permits modules with obvious dependencies to share common type definitions and variables, in fashion similar to that of the ANSI C `#include` mechanism. Note that this will likely introduce a link time dependency between the modules, which will have to be reflected in your distribution (e.g. within a Makefile or project file).

The function has a void return type, and will signal an error if problems occur while loading the external file. To avoid redundancy and save space, code will not be generated for types loaded through external interfaces files.

2.3 Preprocessing

As noted earlier, SLIRP does not contain a full C preprocessor, with the chief omissions being that it does not `#include` headers at runtime, nor will it perform substitution upon parameterized (function) macros. This should go unnoticed by most bindings projects, especially those whose interface is defined within a single file. Libraries which define their public interface within multiple header files, such as Gtk, may be wrapped with SLIRP by specifying each public header on the command line at invocation time.

2.3.1 Conditional Compilation

SLIRP evaluates the conditional compilation directives #ifdef, #ifndef, #if, #elif, #else, and #endif, and will avoid wrapping any content within an exclusion block. The `?`, `:`, and `,` operators are not supported during conditional evaluation, and will induce the truth value of the entire expression in which they appear to 0 (false).

2.3.2 Simple `#define` Macros

SLIRP wraps *integral-valued* and *real-valued* `#define` macros as `Integer_Type` and `Double_Type` constants, respectively. Likewise, *string-valued* macros (i.e. text enclosed within quotes) are wrapped as variables of `String_Type`. SLIRP adopts a conservative stance for these wrappings, preferring macros of form

```
#define          identifier      value
```

or combinations thereof which, after substitution, are equivalent to such. Source file macros can be redefined in an interface file by using either of

```
slirp_define_macro(name [, value]);
#define    name      [value]
```

Both parameters to the functional form are strings. To define a string constant using the functional form the value must include appropriately delimited quotes. Following C, when `value` is omitted the macro is defined to be the empty string. Source macros may be ignored by specifying an `#undef` directive

```
#undef     name
```

within an interface file, or by using the methods described in section 2.5 (Ignoring). Both the `#define` and `#undef` directives are described further within the chapters on annotations.

2.3.3 Parameterized `#define` Macros

Macros whose LHS contains parentheses, and possibly commas, are referred to as parameterized, or function, macros. Although the preprocessor will not substitute such macros when parsing source files, it is possible to wrap them as functions callable from S-Lang scope with

```
slirp_map_macro(macro_name , "return_type ( argument_list )");
```

This indicates to SLIRP that the named macro should be interpreted as if it were a function prototyped as

```
return_type   strlow(macro_name)  (argument_list);
```

2.4 Enumerations

While processing enumerations SLIRP will wrap each enumerant as a S-Lang `Integer_Type` constant. This allows scripts to utilize the enumeration in the same manner as would C code, and in many instances the two will be indistinguishable from one another. For example, the following S-Lang snippet

```
variable value = KSINK_BAD;
ksink_print_error(value);
```

is equivalent to the C code

```
int value = KSINK_BAD;
ksink_print_error(value);
```

and in fact both could be reduced to the single line

```
ksink_print_error(KSINK_BAD);
```

and used interchangeably.

2.5 Ignoring Symbols

Sometimes it's desirable to have SLIRP avoid wrapping one or more functions, macros, or variables. Prior to the introduction of annotations this was done by adding the name of each individual symbol to an *ignore array*, as described in the following subsections. Although this is still supported, the preferred method is to use an annotation such as `#ignore` or `#undef`, as they are cleaner and more general. In either case, recall that because C symbol names are case-sensitive the contents of ignore lists are as well. Groups of functions may also be tagged en masse for ignoring, by matching their argument signatures to an `#argmap(ignore)` annotation as described in section 5.4 (IgnoreMap).

2.5.1 Functions

Prior to the introduction of the `#ignore` annotation a function could be explicitly ignored by adding its name to the list of *ignored functions* within your interface file. This list is declared by SLIRP as a global array of `String_Type`

```
variable ignored_functions = String_Type[0];
```

that is empty by default. The idiom for extending the list in this manner is

```
ignored_functions = [
"name_of_first_function_to_ignore",
"name_of_second_function_to_ignore",
...
"name_of_nth_function_to_ignore",
ignored_functions ];
```

This approach is deprecated, though, in favor of the `#ignore` or `#argmap(ignore)` annotations.

2.5.2 Macros and Variables

If you'd like SLIRP to ignore a particular #define macro, add its name to the *ignored_macros* list. This is another `String_Type` array, initially declared as

```
variable ignored_macros = String_Type[0];
```

Rather than explicitly adding new elements to this array, though, your interface file should utilize either the `#ignore` or `#undef` directives. Finally, the array

```
variable ignored_variables = String_Type[0];
```

records the names of those variables defined in the input interface which SLIRP should not make visible within S-Lang scope. Again, instead of explicitly manipulating this variable your interface file should employ the `#ignore` directive.

2.6 `NULL` and Omitted Arguments

Unlike in other languages, where `null` is viewed as a `value` that may be assigned to pointers of any type, a `null` in S-Lang is both a value (`NULL`) and a type (`Null_Type`). The `Null_Type` is useful, but can require additional care on the part of language binders. To see why, consider the prototype

```
void SomeFunc(char *name, double value);
```

All C compilers should permit this function to be invoked as follows:

```
SomeFunc(NULL, 5);
```

(whether the function behaves well in such cases is a different question). A straightforward mapping of this function to S-Lang would be registered with a function table entry like

```
MAKE_INTRINSIC_2("SomeFunc", SomeFunc, SLANG_VOID_TYPE,
                                  SLANG_STRING_TYPE, SLANG_DOUBLE_TYPE)
```

But this would prohibit the function being called from S-Lang scope as

```
SomeFunc(NULL, 5);
```

because the interpreter would complain that the `Null_Type` of the first argument cannot be cast to a `String_Type`. Fortunately there is a way around this, namely to register the wrapper as though it accepts zero arguments

```
MAKE_INTRINSIC_0("SomeFunc", SomeFunc, SLANG_VOID_TYPE)
```

and then pop the char* and double values explicitly within the body of the wrapper, at which point their types may be checked and the `NULL` values handled accordingly.

While this technique works, it would seem to require additional – and more important, manual – coding on the part of the language binder. To automate the generation of such code SLIRP provides the associative array

```
variable accepts_null_args = Assoc_Type[Any_Type,NULL];
```

The elements of this array are themselves 1D arrays, whose elements represent those arguments in the function signature for which `NULL` is a valid value. In our case a statement like

```
accepts_null_args["SomeFunc"] = [1];
```

would indicate to SLIRP that when it's asked to generate a wrapper for *SomeFunc* it should consider `NULL` a legal value for the first argument, whatever its type may be. This feature also allows functions to be called with arguments omitted, since the two statements

```
SomeFunc( , 5);
SomeFunc(NULL, 5);
```

are equivalent in S-Lang.

2.7 C++

When wrapping a C++ library SLIRP will attempt to wrap all functions, class methods, constants, enumerated types, and simple #define macros specified in its public interface. Get/set methods are automatically generated to wrap simple public fields of a class. Overloaded functions and methods will be wrapped, and class inheritance relationships will be honored. Default values are also supported, although you should be mindful that the only sensible default value which may be assigned to pointer arguments is `NULL`.

While this enables the module developer to wrap a good fraction of mainstream C++ code, by no means should it be considered a full treatment of the language. For example, get/set wrappers for public class fields are generated only when those fields are of basic/primitive type (in the parlance of Kernighan and Ritchie, 1988), and a number of more semantically deep features of C++ (such as templates, operators, and friend classes) are completely unsupported.

The `-cfront` option may be specified on the commandline to generate standalone C wrappers for a C++ library. The code generated in this case will make no reference to the S-Lang C api, and is thus suitable for calling the C++ library directly from C or Fortran. Examples of both usages are given in the `examples/cpp` directory.

2.8 Fortran

If a `Fortran` compiler is detected on the system during configuration SLIRP will also be able to generate bindings for many Fortran codes. Simply specify the Fortran source (where files are assumed to end with either a .f or .F suffix) to SLIRP in the same manner one would normally specify C/C++ headers. SLIRP allows Fortran and C/C++ to be processed during the same invocation, which simplifies the creation of mixed-language modules. Several sample modules containing wrapped Fortran code are given in the `examples/fortran` and `examples/vec` directories.

Implicit typing is supported, as are complex-valued functions and subroutines with complex arguments. In most common cases necessary conversions like transposition and string array unrolling are performed transparently, allowing multidimensional and CHARACTER arrays to be passed back and forth between S-Lang & Fortran without regard to issues like string length or whether arrays are internally laid out in row-major or column-major formats. COMMON blocks will also be wrapped, and can be accessed from S-Lang scope using the functions

```
get_commblock_list()                      gives names of all common blocks in module
get_commblock(blkname)                    retrieves a CommBlock handle to a given block
get_commblock_value(block,varname)        retrieves value of named variable in block
set_commblock_value(block,varname,value)  assigns value to named variable in block
```

as well as using struct.field notation. Here are some examples:

```
linux%  cd examples/fortran
linux%  slsh

slsh>   slsh_append_semicolon(1);
slsh>
slsh>   import("fsubs")
This module interfaces to 6 Fortran common blocks.
You may need to invoke the Fortran routine(s) which
define(s) them prior to using their values in S-Lang.
You may use struct.field notation to access or modify
the values of individual common block variables.
Type "help commblock" for more information.

slsh>   initcomm("blah")

slsh>   com1 = get_commblock("com1")

slsh>   com1
Fortran common block with 2 variables:
        icom1 Integer_Type
        rcom1 Float_Type

slsh>   com1.icom1
2

slsh>   com1.icom1 = 99

slsh>   com1.icom1
99

slsh>   print(get_commblock_list)
"com_block2_"
"com4"
"_BLNK_"
"com3"
"com5"
"com1"

slsh>   com4 = get_commblock("com4")
slsh>   com4
Fortran common block with 4 variables:
        comp1 Complex_Type
        comp2 Complex_Type[2]
        comp3 Complex_Type
```

```
        comp4 Complex_Type[2]

slsh>   print(com4.comp4)
(40 + 50i)
(60 + 70i)

slsh>   com4.comp4[0] = -9 - 9i
slsh>   print(com4.comp4)
(-9 - 9i)
(60 + 70i)
```

The main limitation of the COMMON block support is that individual elements of string arrays & single-precision complex arrays cannot be modified:

```
slsh>   com4.comp2[0] = -6i
Complex_Type Array is read-only
```

This is because S-Lang does not provide, as of version 2.1.3, a single precision complex type, so COMMON block variables of this type must be converted to/from double precision. Similarly,

```
slsh>   com5 = get_commblock("com5")
slsh>   com5
Fortran common block with 2 variables:
        str String_Type (length 13)
        strarr String_Type[3] (length 13)

slsh>   com5.strarr[0] = "foo"
String_Type Array is read-only
```

string arrays are implemented differently in Fortran and S-Lang/C, so passing them back and forth between the languages also requires conversion. These conversions are, however, transparently performed for scalar instances of string and single-precision complex types:

```
slsh>   com4.comp1
(3 + 2i)
slsh>   com4.comp1 = 99 - 99i
slsh>   com4.comp1
(99 - 99i)
```

Recall that Fortran call semantics differ from C, in that Fortran passes *all* variables *by reference* (i.e., as pointers), even scalars, while C passes only by value. The implications of this are that to pass a scalar variable from C to Fortran one normally needs to use the address operator, as in:

```
int c_variable = 5;
some_fortran_function( ..., &c_variable, ...);
```

As a convenience SLIRP hides this detail from the S-Lang programmer, enabling Fortran entry points to be invoked from S-Lang with the same syntax used to invoke C functions. However, as demonstrated in the sample Fortran module, when writing your own annotations your prototypes will need to use pointer syntax. Another convenience afforded by SLIRP is that Fortran functions

and subroutines are invoked from S-Lang scope using exactly the same name as would be employed in Fortran scope. That is, the S-Lang programmer needs no knowledge of how the Fortran compiler mangles entry point names (e.g. upper case, or underscore suffix, etc) in order to call a function or subroutine from S-Lang scope.

In order to maximize portability SLIRP wraps all Fortran function calls within a subroutine whose first argument corresponds to the return value of the function. In general this tactic will not be visible at the S-Lang layer, although the generated wrappers will themselves need to be reflected as a build dependency within the module Makefile. For example, the SLIRP Makefile generation facility would automatically add `sfwrap_foo.f` as a dependency for a module generated from `foo.f`.

Finally, routines and functions whose signatures contain function pointer arguments (actual arguments which are names, e.g., of external procedures) will not be automatically wrapped.

2.9 Makefiles

As an additional convenience SLIRP can also generate "starter" `make` files to automate compilation and linking of your module. For example, given a header file `cos.h` containing

```
double cos(double x);
```

one might generate the module, compile it, and invocation-test the result in just two simple steps:

```
unix% slirp -lm cos.h
Starter make file generated to 'Makefile'

unix% make test

cc  -fPIC -I. -c cos_glue.c
gcc -shared -o cos-module.so cos_glue.o  -lm -lslang -ldl -lm

slsh cos-test.sl
Success!
```

(see the examples/makef directory for a demonstration). This process can be initiated or tuned by the `-make`, `-ldflags`, `-I`, `-L`, and `-l` flags, which are described in section 9 (Command_Line_Options) and should be reasonably familiar to anyone conversant with compilation.

2.10 Template Modules

SLIRP can be used to generate empty (template) modules, by either specifying an empty header file, such as

```
unix%  touch foo.h
unix%  slirp foo.h
unix%  ls -l foo_glue.c
-rw-rw-r--  1 mnoble asc 3393 Nov 10 17:39 foo_glue.c
```

or by specifying /dev/null as input:

```
unix%  slirp -m template_example /dev/null
unix%  ls -l template_example_glue.c
-rw-rw-r--  1 mnoble asc  3470 Nov 10 17:41 template_example_glue.c
```

2.11 Stubs

When the `-stubs` option is specified SLIRP will generate one callable function with an empty body, or stub, for each entry point in the input interface. This can be a surprisingly useful testing feature, since it allows a module to be generated and nominally exercised without the need to compile or link in the underlying library or any of its (potentially numerous) dependencies.

The SLIRP distribution bundles two stub modules, in the `examples/stubs` and `examples/opengl` subdirectories. In the OpenGL module, for example, stubs for the functions prototyped as

```
void glVertex2d( GLdouble x, GLdouble y );
GLboolean glAreTexturesResident( GLsizei n, const GLuint *textures, GLboolean *residences);
```

resemble

```
void  glVertex2d (GLdouble arg1, GLdouble arg2) {  }

GLboolean  glAreTexturesResident (GLsizei arg1, const GLuint* arg2, GLboolean* arg3)
{
    return (GLboolean) 0;
}
```

Since stub functions are pure C they exhibit no dependency upon S-Lang, and may thus be employed in wider contexts.

2.12 Debugging With slirp_debug_pause()

Debugging modules can be difficult, e.g. because breakpoints cannot be set within a module until after it's been loaded. Further, knowing exactly where (in the source code) and when (during process execution) this occurs requires in-depth knowledge of S-Lang library internals, and is virtually impossible to isolate when the S-Lang library has not been compiled for debugging. To address these issues SLIRP offers the -d option, which will embed within your module the **slirp_debug_pause()** stub.

To activate the stub set the SLIRP_DEBUG_PAUSE environment variable before importing your module. This will cause the parent process to wait before exiting module initialization, during which time you may set breakpoints within the codebase of the module (since its symbol table will have been loaded at that point) or its dependencies. This removes the need for module developers to learn the internals of dynamic loading in S-Lang, and permits the module to be debugged even when the S-Lang library itself has not been compiled for such.

The debugging stub will do nothing if SLIRP_DEBUG_PAUSE is unset in the environment. When the variable is set, however, if the result of `atoi(getenv(SLIRP_DEBUG_PAUSE))` evaluates to a negative integer N `slirp_debug_pause()` will sleep for abs(N) seconds, otherwise the stub will pause indefinitely, awaiting a keypress in the terminal from which the parent process was launched.

Chapter 3

Type Mappings

It is important to understand that in order for a generator to emit useful wrappers it needs to know how to map the types it encounters, while parsing function signatures, to the target language. The chief means through which SLIRP does this for S-Lang is by consulting a *type mapping table*. One element of this table, for example, indicates how a S-Lang variable of `Integer_Type` should be passed to a C function expecting an `int` argument. At startup this table contains entries for all of the basic C types (in the parlance of Kernighan and Ritchie, 1988), as well as arrays of and pointers (and C++ references) to them. It may be extended at runtime either

- Automatically, for the selected types described in the next section, or
- Explicitly within your interface file, via the `slirp_map_*` family of functions described below.

SLIRP will map most types transparently, so even though a wide range of type mapping functions are provided in practice there should be little need for your interface file to explicitly use them. Explicit mappings are most appropriate for wrapping large libraries which define many types (e.g. Gtk and its dependencies), and with autotyping disabled, so as to yield the greatest control over what will be wrapped. Another common use is to register initializers or finalizers for opaque types, to automate memory management details as variables enter or leave scope.

3.1 Automatically Generated Type Mappings

As noted earlier, all basic C types (`int`, `double`, etc) are automatically mapped to the analogous built-in S-Lang type. SLIRP also attempts to map all type definitions it encounters to the most appropriate S-Lang type, and will even create new S-Lang types when necessary. For example, each enumerant within a typed enumeration will be mapped to `Integer_Type`, while mappings for "simple" definitions of the form

```
typedef         existing_type           new_type;
```

are simply clones of the type already mapped to `existing_type`.

Unless otherwise instructed, SLIRP maps almost everything else to opaque types. This is described in more detail in the following sections, but in general opaque mappings take one of two forms: types with known definitions and types with unknown definitions. For example, the type

```
typedef   struct { double p[3]; unsigned long id; }     KParams;
```

defined in the beginning of `ksink.h` would be considered known when the prototypes

```
KParams*   ksink_params_new     (unsigned long id, double params[3]);
void       ksink_params_print   (KParams *p);
```

are encountered later during the wrapping of `kitchensink`. A unique opaque `KParams` S-Lang type would be created to wrap `KParams*` instances from C scope, and both functions would be wrapped accordingly.

In contrast, suppose at the outset of a bindings project you elect to wrap only a fraction of a library whose public interface is specified in multiple header files, say `pub1.h, pub2.h, ... pubN.h`; the SLgtk module, in fact, was developed in exactly this manner. In this case you would invoke SLIRP on a subset of those headers, say `pub3.h` and `pub5.h`, making it possible for function signatures within each to refer to types defined only in, say, `pub1.h` or `pub2.h`. Since the formal definitions of those types would be unknown to SLIRP when parsing the prototypes of said functions, the best it can do is map those unknown types to `void_ptr`. This tactic permits larger portions of libraries to be wrapped automatically, with less interface file writing. The `-noautotype` option offers a stricter alternative, by instructing SLIRP to entirely avoid wrapping any function whose signature refers to an unknown type.

3.2 Basic Mapping Functions

The functions given in this section can be used to establish mappings to types defined as equivalent to the built-in types provided by C. For example, a C type definition of

```
typedef  long   glong;
```

would be mapped as follows:

```
slirp_map_long("glong");
```

As noted in the preceding section, SLIRP will automatically establish type mappings for such typedefs, so in practice the basic type mapping functions should rarely be used. It does no harm for you to include such within your interface file, however, and doing so can actually be a convenient way to override the default mapping (especially for types which specifically require a fixed number of bytes).

All of the mapping functions have a void return type, and in each the the `ctype` parameter is a string denoting the name of the type definition to map. Note that in the following list the first two functions map to single-byte types, while the last maps to `char*` strings.

```
slirp_map_char(ctype)          slirp_map_uchar(ctype)
```

```
slirp_map_short(ctype)          slirp_map_ushort(ctype)

slirp_map_int(ctype)            slirp_map_uint(ctype)

slirp_map_long(ctype)           slirp_map_ulong(ctype)

slirp_map_float(ctype)          slirp_map_double(ctype)

slirp_map_long_long(ctype)      slirp_map_ulong_long(ctype)

slirp_map_string(ctype)
```

3.3 Guaranteed-Size Mappings

Variants of these which enforce 16-, 32-, or 64-, or 96-bit size guarantees are as follows:

```
slirp_map_int16(ctype)          slirp_map_uint16(ctype)

slirp_map_int32(ctype)          slirp_map_uint32(ctype)

slirp_map_int64(ctype)          slirp_map_uint64(ctype)

slirp_map_float32(ctype)        slirp_map_float64(ctype)

slirp_map_float96(ctype)
```

3.4 References

C passes all arguments by value, so if you want a function call to change the value of a given variable you must pass it the value of a pointer, or *reference*, to that variable. Reference typemaps can be added with

```
slirp_map_ref(ctype)
```

Note that SLIRP implicitly creates a reference mapping, such as

```
slirp_map_ref("int*");
```

for all basic C types, as well as types mapped to basic C types. This allows the `kitchensink` function

```
void ksink_set_ref_i (int *i) { *i = -9191; }
```

to be automatically wrapped for use in S-Lang as

```
variable i = 111;
ksink_set_ref_i(&i);
vmessage("After ksink_set_ref the new value of i is: %d",i);
```

3.4.1 Using S-Lang References Properly

It is important to understand that the S-Lang C api only allows references to pass information in one direction: *from* a function wrapper *to* a S-Lang variable. A S-Lang reference cannot be used to pass the value of a S-Lang variable *into* a C function. For example, the default SLIRP wrapper for the `kitchensink` function

```
void ksink_swap_double(double *i, double *j)
{
   double tmp;
   if (i == NULL || j == NULL) return;
   tmp = *i;
   *i = *j;
   *j = tmp;
}
```

is effectively unusable, since S-Lang calls such as

```
variable i = 3, j = 4;
ksink_swap_double(&i, &j);
```

will yield undefined results. Using the `-refscalars` switch at generation time would permit usages such as

```
ksink_swap_double(i, &j);
```

(after which both `i` and `j` equal 3), but this addresses only part of the problem since calls such as

```
ksink_swap_double(i, j);
```

would result in neither value being swapped. Therefore it is preferable in these situations to either use annotations or craft the wrapper entirely by hand.

3.5 Mappings For Aggregate Types

3.5.1 Pointers

In this section we expand upon a comment made in section 1.4 (Expectations), regarding the inability of any code generator, *by default*, to properly map *all* constructs from C to their "most natural" form in some target scripting language. To see why, consider a function with signature

```
char** account_get_users(Account *a);
```

Here Account is an opaque structure, whose layout and content are implementation details hidden from the programmer. By virtue of autotyping — or, when it's turned off, through the use of an opaque mapping (described below) such as `slirp_define_opaque("Account")` — SLIRP can easily emit code to create Account* instances in S-Lang scope and pass them back to C routines.

However, the `char**` return value is another matter. The idiom is understood to convey that `account_get_users()` returns an array of strings, but of what size? The number of elements pointed to by the return value is unspecified in the declaration, which renders SLIRP unable to map the return value to a proper S-Lang array of `String_Type`, since to create an array in S-Lang one must indicate its size. Moreover, in principle the return value might not denote an array at all, but rather a pointer to a `char*` whose value the C caller is free to modify. Again, the absence of disambiguating clues in the prototype prevents SLIRP from adopting a definitive interpretation. The same would hold true for return values of type `int*`, `float*`, and so forth.

Left to its own devices the best SLIRP can do when it encounters a return value denoting an array of indeterminate size is pass it to S-Lang scope as a *pointer*, in this case of `string_ptr` type. This enables the C `char**` to exist as a S-Lang variable and be passed back to other C routines which expect an `char**`, but — since pointers are opaque — not directly manipulated in S-Lang scope. For example, attempting to dereference the pointer with the `@` operator or determine the value of the ith element using $[i]$ would both signal an error.

Pointer Type Hierarchy

The complete hierarchy of pointer types provided by SLIRP is as follows:

```
void_ptr
   |
   +-string_ptr
   |
   +-uchar_ptr
   |
   +-short_ptr
   |
   +-ushort_ptr
   |
   +-int_ptr
   |
   +-uint_ptr
   |
   +-float_ptr
   |
   +-long_ptr
   |
   +-ulong_ptr
   |
   +-double_ptr
   |
   +-file_ptr
   |
   +-opaque_ptr
```

The base `void_ptr` type corresponds to generic pointers, such as `void*`. The next 11 wrap pointers to the familiar C types, while the last marks pointers to user-defined opaque types (see next section). The derived pointer types may be cast to and from the `void_ptr` type, but not each other.

To Code or Not To Code?

Another option for the developer would be to forego the pointer types automatically generated by SLIRP in favor of manually coding a wrapper. For example, suppose the documentation for `account_get_users()` noted that its return value is, in fact, an array of strings, and that its final element is guaranteed to be `NULL` A hand-crafted wrapper would then be able to loop over the array until the `NULL` is encountered, use the final loop index value to fabricate an appropriately sized `String_Type` array, and pass the result back to S-Lang scope.

3.5.2 Structures

SLIRP can be instructed to map structures from C scope directly to proper S-Lang structures. This is done for the GdkColor and GdkRectangle types within the SLgtk package, for example, by using

```
slirp_map_struct("GdkColor*");
slirp_map_struct("GdkRectangle*");
```

Structure mappings provide an advantage over opaque mappings (see next section) in that they allow structure internals to be inspected or modified at runtime, but there are presently two tradeoffs with this approach:

1. In the present implementation additional manual coding is required to define the structure layout used by the `SLang_push_cstruct()` and `SLang_pop_cstruct()` routines.
2. It is more suitable for flatter structures, i.e. those which do not deeply nest other structures as fields.

3.5.3 File Handles

S-Lang provides two kinds of handles through which the user may manipulate files: the `FD_Type` returned by the `open` intrinsic, and the `File_Type` returned by the `fopen` and `popen` intrinsics. These types are wrappers around the integer file descriptor defined by POSIX and the `FILE*` pointer defined by the C `stdio` library, respectively.

SLIRP can generate code which permits `FD_Type` variables to be passed to wrapped functions in place of integer file descriptors. As described in section 5.1.1 (Builtin_Input_Mappings), however, these transformations are not implicit, but rather require that you add an extra line or two to your interface file.

In contrast, SLIRP will always permit a `File_Type` variable to be passed to a wrapped function in place of a `FILE*` pointer. Keep in mind, though, that files opened by the S-Lang `fopen()` or `popen()` intrinsic functions should only be closed by the corresponding S-Lang `fclose()` or `pclose()` intrinsics. This stems from the fact that S-Lang maintains additional state within a `File_Type` variable, and this state will not properly reflect that a file has been closed when non-intrinsic wrapper functions are used. S-Lang may then attempt to manipulate the file again (e.g. to close it when the `File_Type` variable goes out of scope), resulting in undefined behavior.

Finally, as noted above SLIRP also provides an opaque `file_ptr`, instances of which will be created when wrapped functions return a `FILE*`. The resulting `file_ptr` variable may then be passed to

any other wrapped function in place of `FILE*`, but may not be passed to S-Lang intrinsic functions in place of `File_Type`.

3.6 Opaque Types

SLIRP provides one additional mechanism for using C aggregate types in S-Lang, namely through the *opaque* mapping functions:

```
slirp_define_opaque(slang_type_name [, parent [, finalizer [, initializer]]])
slirp_map_opaque(ctype [,slang_type_name])
```

Opaque types are so named because, while one can create, pass around, and even destroy variables instantiated from such a type, one cannot modify, or even inspect, their internals. In fact the only information which may be gleaned at runtime from an opaque variable instance is type metadata (e.g. the type name and class id). The expert S-Lang programmer will instantly recognize the equivalence between the notion of SLIRP opaque types and the SLang_MMT_Type defined by the S-Lang C interface. The new terminology is used within SLIRP only because it is felt that *opaque* is a more familiar term than *mmt*, and so will convey more information more quickly to more users.

Opaque variables in S-Lang scope are usually just wrappers around pointers to structures or objects defined and instantiated in C scope. As a case in point again consider the structure type

```
typedef struct { double p[3]; unsigned long id; } KParams;
```

defined by `kitchensink`. When a S-Lang script calls `ksink_params_new` the result would be returned to S-Lang wrapped as an opaque type, and later unwrapped when specified as an argument to the wrapper of a function expecting a `KParams*` (such as `ksink_params_print`).

Normally, for each structured type definition T it encounters SLIRP automatically issues a `slirp_define_opaque(`T`, NULL, "free")` call so that pointers to T may be used in S-Lang scope as uniquely typed opaque variables. This default mapping also specifies that T is a descendant of no other type (parent=NULL) and that when instances of T go out of S-Lang scope that `free` be used to deallocate the memory consumed.

You can instruct SLIRP to not perform this default mapping by either turning autotyping off or by explicitly specifying a mapping for T within your interface file. For example, consider the following sequence of calls taken from the SLgtk interface file:

```
slirp_define_opaque("GtkOpaque");
slirp_map_opaque("gpointer");
slirp_map_opaque("gconstpointer");
```

The first line causes SLIRP to fabricate a definition for a unique S-Lang type named `GtkOpaque`, and emit code to install that type when the SLgtk module is imported. The second and third parameters are omitted in the call, which means that the type will have no parent, no initializer, and no finalizer.

The second and third lines then map the C types `gpointer` and `gconstpointer` to the new `GtkOpaque` S-Lang type. With this typemap SLIRP will be able to automatically generate wrappers, when later processing Gtk header files, for functions whose signatures contain either C type. Without such a typemap SLIRP would otherwise ignore such functions. Now consider

```
slirp_define_opaque("GtkTreePath","GtkOpaque","gtk_tree_path_free");
```

As above, this call fabricates a new opaque S-Lang type, `GtkTreePath`, and ensures that code will be emitted to install the type when the module is imported. Unlike above, however, SLIRP will consider this new type a descendant of an existing type, namely `GtkOpaque`. Moreover, when instances of the type go out of S-Lang scope the opaque destructor will call `gtk_tree_path_free` to finalize the underlying C instance.

3.6.1 Initializers and Finalizers

The initializers and finalizers mentioned above are optional helper functions which the programmer can direct SLIRP to call when opaque variables come into S-Lang scope (are created) or go out of S-Lang scope (are destroyed), loosely analogous to *constructors* and *destructors* within object-oriented languages such as *C++* and *Java*. Both are pointer types declared as

```
void    (*INITIALIZER)          (void*);
void    (*FINALIZER)            (void*);
```

and usually refer to functions within either the api of the library being wrapped or the hand-crafted portion of the glue layer. Typically a finalizer is called to free resources allocated to the C variable which the opaque wraps; this can be useful, for example, to *free* fields within a wrapped C structure that were *malloc*-ed when it was instantiated, thus avoiding memory leaks. The default finalizer used by SLIRP, when automatically mapping pointers to structured types as opaque S-Lang types, is the C `free` function.

3.6.2 Default Mappings

Note that each call to `slirp_define_opaque()` also creates a typemap between the new S-Lang type and a C type whose name is the concatenation of the S-Lang type name with an asterisk. For example, an invocation such as

```
slirp_define_opaque("GtkObject","GObject","slgtk_object_destroyer");
```

would implicitly issue the call

```
slirp_map_opaque("GtkObject*","GtkObject");
```

This simplifies SLIRP interface files by eliminating the redundancy of having each opaque type definition followed immediately by the obvious C type mapping.

Finally, note that each opaque type defined becomes the *default* S-Lang type for subsequent opaque mappings. SLIRP uses this default to assign a S-Lang type when one is omitted in an opaque mapping, which simplifies the SLIRP interface file by allowing the programmer to specify a type only when it is explicitly necessary. This tactic also increases the performance of the code generator, since the S-Lang interpreter needs to parse fewer tokens. The default remains in effect until the next call to either `slirp_define_opaque()` or `slirp_set_opaque_default()`

3.7 Defining New Type Mapping Functions

For completeness we note that the lowest-level function called by all mapping functions is:

```
slirp_map (ctype,gluetype,mnemonic,typeid,typeclass [,freer])
```

In principle this could be used to write custom type mapping functions, but in practice there should be little need for your interface file to call it explicitly.

Chapter 4

Introduction To Annotations

The previous chapter focused on the use of type maps to extend SLIRP and tailor its behavior. Type mappings are simple to express and comprehend, and in many cases will be all the developer needs to create reasonably capable wrappers for a given library, especially in the initial phases of a bindings project.

We now turn to a series of features, collectively referred to as *annotations*, which are arguably the most powerful SLIRP has to offer. In exchange for learning a bit of new and initially strange syntax, annotations provide a degree of control over the code generation process that goes well beyond what can be achieved with type mappings alone. For instance, consider the problem of morphing a C function of `M` inputs and `N` outputs into a S-Lang function of `M'` inputs and `N'` outputs. Such might arise when wrapping a function prototyped as

```
int print_array_f(float *arr, int len);
```

where the second argument indicates the number of elements in the first. The default wrapper generated for this function would yield S-Lang usages such as

```
variable arr = [1.1, 2.2, 3.3, 4.4, 5.5];
print_array_f(arr, length(arr));
```

However, since S-Lang arrays "know their size" the second parameter in the function call is superfluous and can be dropped, yielding the more natural usage

```
print_array_f(arr);
```

In our nomenclature the C version of this function has `M`=2 inputs and `N`=1 outputs, while the S-Lang version has `M'`=1 input and `N'`=`N` outputs. To achieve this transformation a module writer might hand-craft a wrapper along the lines of

```
static int wrap_print_array_f(void)
{
   SLang_Array_Type *arr;
   int status;
```

```
   if ((-1 == SLang_pop_array_of_type (&arr, SLANG_FLOAT_TYPE))
        return -1;

   status = print_array_f(arr->data, arr->num_elements);
   SLang_free_array(arr);
   return status;
}
```

This approach works, but is laborious and requires in-depth knowledge of the S-Lang C api (e.g. to correctly use the `SLang_Array_Type`). Furthermore, since the pattern captured is a common one it would be better if such transformative wrappers could be generated in a more automated fashion.

SLIRP annotations serve exactly this purpose, by enabling developers to tag function prototypes with semantic hints and source code fragments; with these additional semantics SLIRP is able to make more informed judgements about what prototypes mean; the additional code fosters the creation of custom wrappers, without resorting to writing them entirely by hand.

Some users may recognize a resemblance — in both spirit and syntax — between SLIRP annotations and SWIG `%typemaps`. This is not accidental, and it is only appropriate that we acknowledge the numerous insights that have been gleaned from SWIG through study of its `%typemap` capability.

4.1 A Sample Annotation

By adding to the interface file an annotation such as

```
#argmap(in, which=1) (float *arr, int)
   $2 = ($2_type) $1_dim0;      /* float* argmap */
#end
```

SLIRP would generate a wrapper for `print_array_f` resembling

```
static void sl_print_array_f (void)
{
   int result;
   float* arg1;
   Slirp_Ref arg1_ref = Slirp_ref_init(SLANG_FLOAT_TYPE, sizeof(float), arg1);
   int arg2;

   if (SLang_Num_Function_Args != 1 ||
        pop_array_or_ref( &arg1_ref) == -1 )
        {Slirp_usage_err("int = print_array_f(float_ptr)"); return;}

   arg2 = (int) Slirp_ref_get_size(&arg1_ref,0);     /* float* argmap */
   result = print_array_f(arg1, arg2);
   (void) Slirp_ref_finalize(&arg1_ref);
   (void) SLang_push_int ( result);
}
```

The desired effect has been achieved: the generated code expects only an array argument to be passed in from S-Lang scope. Fewer lines of hand-written code (3 lines in an interface file, versus 10+ lines in

C) were needed to wrap the function, while bookkeeping work — such as registering wrappers within an intrinsic function table, merging manually-crafted code fragments with automatically-generated code, and writing usage statements — is virtually eliminated. The fact that the annotation does not reference the `print_array_f` function by name allows it to be used on any function with a matching prototype, and can sharply reduce the amount of coding required to create custom wrappers.

Annotations may be used for other purposes as well, such as omitting return values, making values returned through a function parameter list in C appear as if they were returned on the stack in S-Lang, or injecting user-defined fragments of C code into the generated wrappers. These and other ideas are explored in the context of numerous examples, many of which can be found in interface files bundled within the `examples` directory tree of the SLIRP distribution.

4.2 Dissecting an `#argmap`

The syntax of an `#argmap` may seem peculiar to the uninitiated, and you would be justified in wondering at this point *How does it actually work?* This question actually has two parts: the first concerns how annotations are read by the S-Lang interpreter, and is relevant because their semantics are expressed in a syntax not defined within the S-Lang grammar; the second concerns how SLIRP selects which annotations to apply while generating wrappers.

4.2.1 Custom File Loader

The first part of the question is by far the easier to answer: SLIRP contains a custom file loader, installed via a *load file hook*, which S-Lang will call when evaluating scripts. This loader scans the input interface file for directives (which happen to look like preprocessor tokens) marking an annotation block. Each annotation directive utilizes a unique callback function which

- validates the syntax of the directive (at block open)
- accumulates each line within the body of the block into a text buffer
- processes the resulting buffer (at block close)

4.2.2 Matching Rules

The second key to using annotations is understanding the pattern matching rules employed by SLIRP to decide whether a given annotation should be applied to some function. For example, given

```
#argmap(in, which=2) (long nelems, char **array)
   $1 = ($1_type) $2_dim0;              /* char** argmap #1 */
#end

#argmap (in) char **
   /* char** argmap #2 : just a single comment */
#end
```

and a function prototyped as

```
extern  void print_array_s      (long   nelems, char **array);
```

only the first mapping would be applied, since it is a stronger match. Now if the mapping

```
#argmap(in, which=2) (long, char **)
   $1 = ($1_type) $2_dim0;              /* char** argmap #3 */
#end
```

were also present which would SLIRP select? Still the first, since not only does it match the prototype in the *quantity* and *type* of its parameters, but also in their *names*. However, if the prototype instead looked like

```
extern  void print_array_s      (long, char **);
```

then the first `#argmap` would be rejected (because parameters named within an annotation do not match unnamed parameters within a prototype) and the third would be used (since it provides a longer match than the second). Finally, if the first and third mappings were removed then the second would be applied and yield broken runtime behavior, since the array size parameter would remain uninitialized in the wrapper (the argmap body contains only a comment). The rules governing the matching of annotations with function prototypes may thus be summarized as:

- function names are not relevant
- each parameter within a function prototype will match at most one annotation
- longer parameter sequences within parameter lists have higher precedence than shorter ones
- named annotation parameters never match unnamed prototype parameters
- unnamed annotation parameters will match *either* unnamed prototype parameters *or*, provided all named annotation parameters have already matched
- multiple annotations may be applied to the wrapper generated for a given function

Matching can also be tuned with the `#prototype` and `#copy` directives, as discussed later.

4.3 Parameter Substitutions

At this point it should be clear that parameters specified within the parameter list of an annotation are referenced elsewhere within the annotation by prefixing them with a dollar sign. That is, the first parameter is referred to as `$1`, the second as `$2`, et cetera. This allows the module writer to craft code fragments which explicitly refer to function arguments, irrespective of the names SLIRP later generates for their corresponding local variables.

When the body of the annotation is injected into the wrapper SLIRP substitutes each `$`-delimited reference with the name of the respective local variable. While at first glance this may seem trivial, you should recall that because multiple annotations can be applied to a function it is by no means clear what names SLIRP will assign to each variable declared within its wrapper.

4.3.1 Metadata

SLIRP also provides lightweight introspective capabilities, loosely analogous to reflection in Java, which enable annotations to discern metadata about a parameter, such as its type, size, or argument number. These substitutions are:

- $argnum

  Yields the ordinal position that the *first* argmap parameter occupies within the argument list of the function prototype to which it has been applied. For example, the annotation

  ```
  #argmap(in) LongInt (tmp)
     ...
     tmp = $argnum;
     ...
  #endif
  ```

  would be applied twice to a function prototyped as

  ```
  void diff(LongInt x1,  LongInt x2);
  ```

  causing the lines

  ```
     tmp1 = 1;
     ...
     tmp2 = 2;
  ```

  to appear within its generated wrapper.

- $cleanup

  If your annotation does error checking which can result in the wrapper exiting prematurely, use this substitution within its body to ensure that whatever resources were allocated while marshaling the functions arguments from S-Lang will be properly released; this avoids memory leaks within your module.

- $funcname

  Yields the name of the function to which the annotation has been applied.

- $funcnargs

  Yields the number of arguments expected by the wrapper of the function to which the annotation has been applied.

- $n_dim*I* Yields the size of the I-th dimension (numbered from 1) of argument n when it is an array, otherwise 1. When I is greater than the number of the dimensions in the array the size returned will be 0. When I is 0 the size returned will be the number of elements in the entire array.

- $n_length Equivalent to $n_dim*0*.

- $n_ndims Yields the number of dimensions of argument n when it is an array, otherwise 1.

- $\$n$_nullify Yields code which will set argument n to NULL. For non-opaque arguments this is trivial (and can be done through other means), but for opaquely typed arguments the wrapped object will be set to NULL. This might be used in conjunction with the final method (described below) to avoid manual coding in cases where one wants to prevent an opaquely typed variable from pointing to freed memory.

- $return

  When used within an output argmap, this yields the proper code to push the given typed argument onto the S-Lang stack.

- $\$n$_type Yields the C type of the local variable corresponding to argument n. In the above example this substitution is used to cast the value yielded by $\$n$_dim0, so as to avoid potential compile warnings.

4.4 Variable Substitutions

Another form of substitution is performed when an annotation includes variable declarations, as in

```
#argmap (in) char ** (int size)
   {
        char **copy = $1;
        size = 0;
        while (*copy++)
           size++;
   }
   printf("\nNull terminated string array size: %d elements\n",size);
#end
```

This annotation declares an integer size variable, and if applied to a function prototyped as

```
extern void print_array_nts (char **array);
```

(where the final element of array is expected to be NULL) would yield a wrapper resembling

```
static void sl_print_array_nts (void)
{
   int size1;
   char** arg1;

   ...

   {
        char **copy = arg1;
        size1 = 0;
        while (*copy++)
           size1++;
   }
   printf("\nNull terminated string array size: %d elements\n",size1);
```

```
    ...
}
```

Here the `size` declaration maps to the `size1` automatic variable. A numeric suffix is used to uniquely identify the instance of the declared variable since, as noted above, a single annotation might match multiple parameters within a prototype, causing its code fragment to be injected into the generated wrapper multiple times. Notice that locally-scoped variables, such as `copy`, may also be declared and used within inner blocks. Unlike wrapper-global variables, however, these do not require disambiguation.

Chapter 5

#argmap Annotations

Formally, an `#argmap` specifies how a sequence of `N >= 1` function arguments in C map to a sequence of `M <= N` arguments specified to a S-Lang function call. In the common case `N = M = 1`, resulting in a one-to-one correspondence between C functions and their S-Lang wrappers. The case where `N > 1` is referred to as a *multi-argument* map and, as shown in the examples, is commonly used to collapse a sequence of C arguments into a shorter sequence of of S-Lang arguments. The argmap grammar is given by

> `#argmap (` *method [,qualifier_list]* `)` *parameter_list [(* *variable_declaration_list* `)]`
>
> *code_fragment*
>
> `#end`

More details are given in the grammar reference, but in the context of the first argmap from the previous chapter

```
#argmap(in, which=1) (float *arr, int)
   $2 = ($2_type) $1_dim0;      /* float* argmap */
#end
```

- The method is `in`, meaning that the arguments specified within its parameter list should be interpreted as inputs to the function.
- One qualifier has been specified, `which=1`, indicating that only the first parameter within the parameter list should be passed when the function is called from S-Lang scope and, consequently, that the second argument should be omitted. That is, `N = 2` and `M = 1`.
- The parameter list is `(float *arr, int)`.
- There are no local variables declared.
- The single line `$2 = ($2_type) $1_dim0;` represents the body of the annotation, and initializes the second parameter to the (appropriately casted) number of elements in the first.

Preprocessor tokens may not appear within the body of an annotation, which should be viewed as a fragment of C code subject to the substitution rules described earlier. After substitution these fragments are injected directly into the wrapper during code generation. No further validation is

performed upon the generated code, making it entirely possible to write annotations which yield wrappers that will not compile.

The *method* keyword of each argmap governs where in the generated wrapper the argmap code fragments will be placed. For instance, the body of an `in` argmap would appear within the `Argument_Marshalling_Block` of a wrapper, while the code from a `final` argmap would be injected into its `Return_Block`.

5.1 `in` Method

The `#argmap(in)` form allows you to customize the code generated for passing inputs from S-Lang to wrapped functions. By default all arguments expected by a wrapped function must be supplied to its wrapper at call time. The optional `which=`*selection* qualifier may be used to change that, by identifying which parameters of a multi-argument map must be specified in the S-Lang call sequence (and, by implication, which should be omitted). The *selection* value is a S-Lang array index expression, and gives rise to several forms for the qualifier:

- `which=i` selects only the i-th argmap parameter
- `which=i:j` selects the i-th through j-th parameters
- `which=i:j:k` selects the i-th through j-th parameters, in increments of k
- `which=[i` *`[,j,k,...]`*`]` selects the parameters i (and optionally j, k, ...)

Since function parameters are numbered from 1 the `i, j, k, ...` here may only take on positive integral values. A compact way of omitting all parameters from the S-Lang call sequence is to use an `omit` qualifier. For example, imagine you would like to ensure that a function prototyped as

```
arg_dropper(unsigned long int ul);
```

be called from S-Lang only with the value 112233. This could be achieved by forcing the function to be called from S-Lang with zero arguments, via

```
#argmap(in, omit) unsigned long int ul
   $1 = 112233;
#end
```

and letting the wrapper pass on the desired value.

5.1.1 Built-in `in` Maps

As a convenience SLIRP provides a number of input maps. As discussed here, some of these will be applied automatically by SLIRP, while others will need to be specified manually within your interface file.

File Descriptors

```
#argmap(in, proxy=SLFile_FD_Type) int FD_PROXY
   if (-1 == SLfile_get_fd (proxy, &$1)) {
        SLang_verror(SLEI, "could not assign file descriptor proxy");
        return;
   }
#end
```

The `FD_PROXY` annotation allows S-Lang `FD_Type` file handles, which are opaque, to be passed directly to compiled routines expecting an integer file descriptor. This permits natural usages such as

```
variable fd = open("/some/dir/some_file",O_RDONLY);
ksink_close(fd);
```

where the corresponding `kitchensink` function is prototyped as

```
extern  void  ksink_close (int fd);
```

The `int` parameter in this annotation is explicitly named `FD_PROXY` so that it will not be applied to just any `int` argument within any function signature. That is, because integers are a common type and we cannot reliably expect all libraries to employ the "`int fd`" idiom, your interface file must explicitly assert when it should be applied. For the `ksink_close` function above this is achieved with

```
#copy int FD_PROXY { int fd }
```

Generic Pointers

```
#argmap(in, proxy=SLang_Array_Type) void* ARRAY_2_VOIDP
   $1 = proxy->data;
#end

#argmap(in, proxy=SLang_Any_Type) void* ANY_2_VOIDP
   $1 = proxy;
#end
```

The `ARRAY_2_VOIDP` and `ANY_2_VOIDP` annotations facilitate the passing of arrays or arbitrary objects, respectively, from S-Lang to compiled routines expecting a `void*` pointer. As with the file descriptor mapping above, these mappings are explicitly named as a precautionary measure, so as to prevent them from being applied to just any function with a `void *` argument. As an example consider the HDF5 function

```
herr_t H5LTmake_dataset( hid_t loc_id, const char *dset_name,
                 int rank, const hsize_t *dims, hid_t type_id,
                 const void *buffer );
```

In C this function can be called with an arbitrarily-typed array as the last parameter (e.g. `float*`, `double*`, etc). By default, however, SLIRP will wrap this function as if the last argument were an opaque pointer instance (absent any hints what else can it do?) One way to use the builtin argmap to change the generated wrapper would be to copy it with

```
#copy void* ARRAY_2_VOIDP { const void* }
```

so that any function in the HDF5 api which expects a `const void*` argument would automatically support the C idiom of passing in arbitrarily-typed arrays.

C++ `string` Objects

These annotations allow the use of S-Lang strings in place of C++ `string` objects. They support both scalars and arrays, and will be applied transparently.

```
#argmap(in, proxy="char*") string
   $1 = proxy;
#end

#argmap(in, proxy=SLang_Array_Type) string*
   {
   int i, l;
   char **arr;
   if (proxy->data_type != SLANG_STRING_TYPE) {
        SLang_verror(SL_USAGE_ERROR, (char*)"String array is required here");
        return;
   }
   arr = (char**)proxy->data;
   l = proxy->num_elements;
   $1 = new string[l];
   for (i=0; i<l; i++)
        $1[i] = string(arr[i]);
   }
#end
```

If the C++ interface you're wrapping contains many functions or methods with `string` array arguments, then to avoid duplicative bloat you may wish to move the content of the `string*` annotation into a function (defined, e.g., within an `#inline_c` directive) and override the built-in version in your interface file with an annotation whose body simply calls that function. A similar strategy is employed by the built-in `#retmap` for `NT_STR_ARRAY`.

5.2 `out` Method

The `#argmap(out)` form allows you to customize the code generated for transferring output from wrapped functions back to S-Lang scope. It's most commonly used to drop a single argument (typically a reference) from the function inputs, in favor of having its value returned on the stack instead. This would allow, for example, a function prototyped as

```
void ksink_mult2 (double op1, double op2, double *result);
```

to be called more naturally from S-Lang as

```
result = ksink_mult2(333, 3);
```

instead of

```
variable result;
ksink_mult2(333, 3, &result);
```

as would be required by the default wrapper. The latter form can be cumbersome, especially for interactive use (e.g. within ISIS , which presents a Matlab(tm)-style command line prompt for scientific analysis), but with an output annotation such as

```
#argmap(out) double *result
   $return;
#end
```

SLIRP is able to transform the default generated wrapper

```
static void sl_ksink_mult2 (void)
{
   double arg1;
   double arg2;
   double* arg3;
   Slirp_Ref arg3_ref = Slirp_ref_init(SLANG_DOUBLE_TYPE, sizeof(double), arg3);

   if (SLang_Num_Function_Args != 3 ||
        pop_array_or_ref( &arg3_ref) == -1 ||
        SLang_pop_double(&arg2) == -1 ||
        SLang_pop_double(&arg1) == -1 )
        {Slirp_usage_err("ksink_mult2(double,double,double_ptr)"); return;}

   ksink_mult2(arg1, arg2, arg3);
   (void) Slirp_ref_finalize(&arg3_ref);
}
```

into

```
static void sl_ksink_mult2 (void)
{
   double arg1;
   double arg2;
   double arg3;

   if (SLang_Num_Function_Args != 2 ||
        SLang_pop_double(&arg2) == -1 ||
        SLang_pop_double(&arg1) == -1 )
        {Slirp_usage_err("double = ksink_mult2(double,double)"); return;}

   ksink_mult2(arg1, arg2, &arg3);
   SLang_push_double(arg3);
}
```

and effectively make ksink_mult2 a duplicate of ksink_mult given in the opening example. This also allows using the wrapper result as an inlined argument to another function, such as

```
printf("The result is %d\n", ksink_mult2(333, 3));
```

which is simply not possible in its original form. Note that, as is often the case with SLIRP, the same effect could be achieved in other ways. For example, we could have simply used the #copy directive, as described in section 7.2 (copy_example), to reuse the built-in double *OUTPUT annotation:

```
#copy double *OUTPUT { double *result }
```

Alternatively, our outmap could have explicitly pushed its argument onto the stack

```
#argmap(out) double *result
   SLang_push_double($1);
#end
```

although this approach is not advisable. For one, using the type-specific S-Lang push routine requires more knowledge, on the part of the bindings developer, of the internal S-Lang C api. It is also less general, because its type-specificity prevents the annotation from being reused (again, by #copy) for other types.

Finally, note that the usage=C_string_literal qualifier may be used to override the default Usage: message generated by SLIRP for the mapped argument, and that outmaps support neither variable substitution nor multiple-argument parameter lists.

5.2.1 Built-in out Maps

SLIRP provides a number of built-in output mappings, such as:

```
#argmap(out) short *OUTPUT
   $return;
#end
```

The effect of this outmap is duplicated for other types by using #copy

```
#copy short *OUTPUT { unsigned short *OUTPUT, int *OUTPUT,
        unsigned int *OUTPUT, unsigned *OUTPUT, long *OUTPUT,
        unsigned long *OUTPUT, float *OUTPUT, double *OUTPUT,
        spcomplex *OUTPUT, dpcomplex *OUTPUT}
```

Each of these are, for the convenience of brevity, also copied to

```
#copy short *OUTPUT { short *OUT, unsigned short *OUT, int *OUT,
        unsigned int *OUT, unsigned *OUT, long *OUT, unsigned long *OUT,
        float *OUT, double *OUT, spcomplex *OUT, dpcomplex *OUTPUT}
```

Note that the spcomplex and dpcomplex types refer to internal types defined by SLIRP to correspond to the single- and double-precision complex types offered by Fortran. As yet there is no default support for C99 complex types.

As an example consider that the two annotations

```
#copy long *OUTPUT      { long *result }
#copy long *OUT         { long *result }
```

are equivalent: both copy the default `long*` output mapping to the named argument `long *result`, so that if SLIRP sees the latter whilst parsing function signatures it will know to remove it from the input argument list of the wrapper function and instead treat it as a return value of the wrapper.

As shown in the next chapter, these may be combined with the `#prototype` and `#copy` annotations to simplify or tune the matching of annotations to functions. Note that the `complex` and `doublecomplex` mappings refer to the corresponding Fortran types.

5.3 `final` Method

The `#argmap(final)` form provides a means of explicitly specifying how arguments should be finalized by a wrapper just prior to its return. This can be useful for bookkeeping work, such as freeing memory allocated within the wrapper or ensuring that an object is properly destroyed. As an example of the latter, suppose a C table I/O library contained the routines

```
extern Table*   table_open(char *name);
extern int      table_close(Table *t);
extern void*    table_get_column(Table *t, int column);
```

that were wrapped by SLIRP by mapping `Table*` to an opaque type

```
slirp_define_opaque("Table", NULL, "table_close");
```

and then called from S-Lang in the usual way

```
variable table = table_open("table.dat");
variable col1 = table_get_column(table, 1);
variable col2 = table_get_column(table, 2);
```

If the C `table_close` routine frees its input `Table*` pointer (good policy) then after a call

```
table_close(table);
```

in S-Lang scope the opaque `table` variable would encapsulate freed memory. Subsequent attempts to use it, such as

```
table_close(table);
```

would yield undefined behavior (e.g. SEGV). This can be prevented by executing

```
table = NULL;
```

explicitly after the `table_close` call or, more elegantly, by having the `table_close` wrapper transparently achieve a similar effect by nullifying the `Table*` instance wrapped by the opaque S-Lang `table` variable. Since it is, again, advantageous to avoid writing such a wrapper entirely by hand one might instead craft annotations such as

```
#argmap(final) Table* NULLIFY
   $1_nullify;
#end

#prototype
   int   table_close(Table* NULLIFY);
#end
```

The first annotation performs the required nullification, and uses a uniquely named parameter list so as to limit its application only to the `table_close` function. The second annotation redeclares the prototype for this function, allowing it to match the `#argmap(final)`.

5.4 ignore Method

The `#argmap(ignore)` form provides an alternative to the `#ignore` directive. Rather than ignoring a function based upon its name, this annotation supports ignoring functions by matching on their argument signatures. It provides a potentially powerful means of ignoring entire groups of functions with a single, simple annotation. For example, when generating pure C bindings for C++ code (`-cfront` mode) SLIRP provides the builtin annotation

```
#argmap(ignore) string*
#end
```

which ensures that functions containing string array arguments will not be reflected in the bindings. Note that because ignore maps cause code NOT to be generated, any code fragment within the body of the mapping is ignored, and in fact there is no need to explicitly end the argmap block with `#end`. That is, the single line

```
#argmap(ignore) string*
```

achieves the same end as above.

5.5 setup Method

The `#argmap(setup)` form supports the injection of code fragments into a wrapper just prior to argument marshaling, allowing wrappers to be customized to take preliminary action based upon the number of arguments passed, or their types, et cetera.

Chapter 6

Vectorization

One of the most powerful features of S-Lang is the transparent manner in which scalars and arrays may be used in the same context. For example,

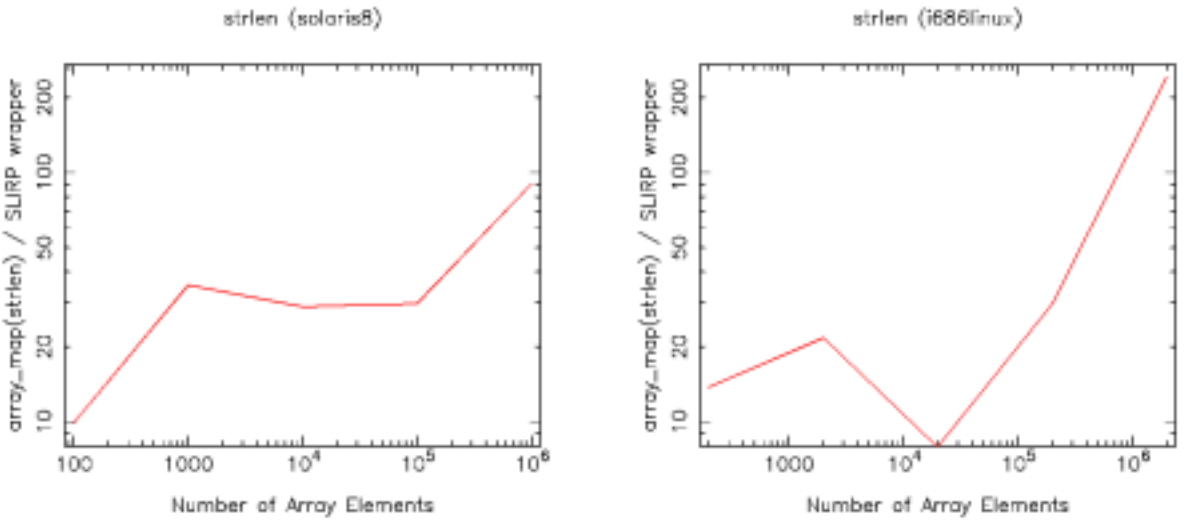
```
y = sin(x)
```

produces a scalar result when `x` is a scalar, or a vector when `x` is a 1D array, and so on. This feature is referred to as *vectorization*, since it obviates the need to explicitly loop over array indices in S-Lang scope; instead, the loop is implied and executes in compiled C scope, yielding significantly greater performance. Vectorization encompasses more than the simple promotion of scalar arguments to arrays, however. More generally, we say a function is vectorized when any of its arguments may be of multiple ranks; no distinction is made between the promotion of a scalar (rank 0) argument to 1D or higher, a 2D array to 3D, and so forth. When a vectorized function is invoked with an argument whose rank exceeds that of its default usage (e.g. as specified in a C function prototype), we say that both the argument and the invocation itself are *vectored*.

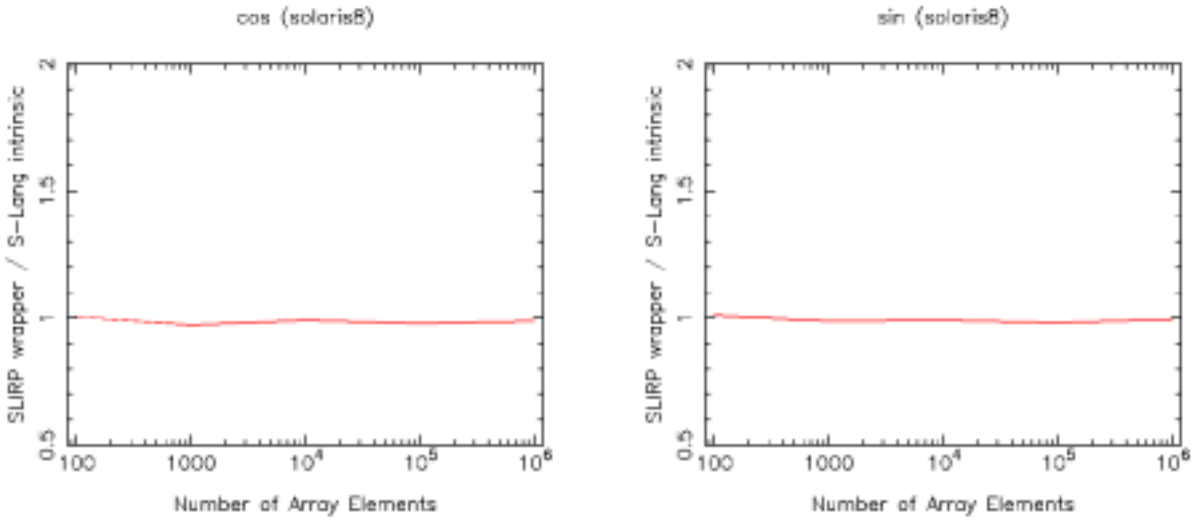
Vectorization naturally complements the strong array handling facilities within S-Lang. It provides powerful multidimensional mathematical capability, nearly analytic in its expressiveness, and with performance on par with – or exceeding – that of compiled C code and commercial analysis packages; together these make S-Lang a very strong platform for numerical analysis in science and engineering. The fact that vectorization is native to S-Lang distinguishes it from other popular scripting languages like Perl, Python, or Tcl, which natively lack high-performance multidimensional numerical capability. Array handling within S-Lang is typically much faster than its counterparts in these languages, as much as an order of magnitude in some cases, as well as more concise because it's built into the very fabric of the language.

SLIRP extends this capability, with commensurate performance and economy of codesize, to a wide range of external C, C++, and Fortran codes; this enables wrappers to be called with either vectored or non-vectored semantics, and with few restrictions on the quantity, datatype, or rank of the arguments to functions that may be automatically vectorized. The `-openmp` option may be used to tune vectorized code so that OpenMP-aware compilers will automatically parallelize the wrappers, boosting performance even further in the presence of multiple cpus (or cores).

As a concrete example of the advantage of vectorization, here are performance ratios of the S-Lang `strlen` intrinsic versus a vectorized wrapper for `strlen` generated by SLIRP :



The vectorized `strlen` is roughly 1 to 2 orders of magnitude faster than the optimal native S-Lang usage (employing `array_map` to iterate over the string array); other S-Lang looping constructs – such as `for`, `while`, or even `loop` – yield an even greater performance advantage for the vectorized `strlen`. The quality of SLIRP vectorization is evident in the following plots



which demonstrate that the generated wrappers for `cos` and `sin` have vectorized performance equivalent to the hand-crafted `cos` and `sin` intrinsics native to S-Lang (see the `examples/vec` directory for examples and more performance details).

6.1 Making and Using a Vectorized Function

Vectorization is a transformation of the *entire* function semantics, in the sense that every argument received by the wrapper is tagged to optionally accept a vectored value. A vectorized function may be called with both vectored and non-vectored arguments at runtime, but at code generation time SLIRP will either vectorize every argument accepted by the wrapper or none at all. When SLIRP

cannot vectorize one or more arguments of a function (see below) it will instead generate a standard, non-vectorized wrapper. There are several ways to request that a function be vectorized:

- use the `-vec` option, which causes SLIRP to attempt to vectorize every function in the interface being wrapped
- add the function name to a `#vectorize` block
- prototype the function within a `#vectorize` block
- use the `-openmp` option, in which case vectorized wrappers may be parallelized with OpenMP

For example, given two functions with C prototypes of

```
void     vmult          (double *x, double *y, double *result, int len);
size_t   strlen         (const char *s);
```

an interface file containing

```
#vectorize
   strlen
   void vmult(double *x, double *y, double *OUT, int DIM1);
#end
```

would yield a module with vectorized wrappers for the `strlen` and `vmult` functions (these and other examples are included within the `examples/vec` subdirectory of SLIRP). While the types in the vectorized prototype for `vmult` match, of course, that of the underlying function being wrapped, the parameter names chosen change how the function will be called from S-Lang scope. First, as discussed in the previous chapter, the `double *OUT` parameter is an output `#argmap`, which transforms the result argument into an output value of the wrapper. Second, DIM1 denotes that the final argument describes the length of each array and that it, too, may be omitted when calling the wrapper. S-Lang calls to `vmult` therefore operate as if it accepts two arrays and returns 1, all of `Double_Type`. If we were to load the module into `slsh`, for example, and invoke `vmult` with no arguments

```
slsh> import("vec")
slsh> vmult
Usage:  double[] = vmult(double[],double[])
        This function has been vectorized.
slsh>
```

(these examples assume a S-Lang 2 shell, with semicolon appending turned on), the usage statement reflects this, as well as the fact that the function has been vectorized. The DIM1 annotation also plays another important role, in that it signifies to SLIRP that before the wrapper calls `vmult` proper it must ensure that the array arguments are all of equal length:

```
slsh> vmult([1,2,3], [3,4])
Array shape or length mismatch

slsh> vmult([1,2,3], 4)
Scalar cannot be used here
```

Without such a precaution the underlying `vmult` function would likely yield undefined results or memory corruption and an application crash. Two examples of calling `vmult` correctly, then, are

```
slsh> print( vmult([1,2,3], [5,5,5]) )
5
10
15

slsh>  Arr = Int_Type[2,3]
slsh>  Arr[0,*] = 5
slsh>  Arr[1,*] = 100
slsh>  print( vmult(Arr, [3, 4, 5]) )
15 20 25
300 400 500
```

Observe that the first call *is not vectored*, because `vmult` is prototyped to expect an array, assumed to be of dimension 1, and that's what was passed in; `vmult` is called only once by the wrapper. The second call *is vectored*, because one of its arguments is a 2D array; now the wrapper calls `vmult` twice, once as

```
vmult([5, 5, 5], [3, 4, 5], retval, 3)
```

and again as

```
vmult([100, 100, 100], [3, 4, 5], retval+3, 3)
```

(where `retval` is a `double*` dynamically allocated to store 6 values) and yields the 2D array

```
| 15     20     25  |
| 300    400    500 |
```

To convince yourself that the latter is actually true, consider:

```
slsh>  vmult(Arr, [3, 4, 5])
Double_Type[2, 3]
```

Note that the vectorization behavior in the second case resembles that of the *array_map()* intrinsic when passed arguments of different lengths: only the first element of shorter arrays will be (re)used in each iteration. Here the vector `[3,4,5]` is treated as the first element of a vacuously 2D array of dimension `1x3`. This flexibility can yield shorter and cleaner code, e.g. by avoiding the creation of temporary container arrays simply to satisfy dimensionality constraints. Finally, it's worth noting that even though `Arr` and `[3,4,5]` are arrays of `Integer_Type`, S-Lang transparently casts them to `Double_Type` when the arguments are transferred to the wrapper.

6.2 Dimensionality Theory

At runtime SLIRP uses a few simple metrics to decide

- whether a wrapper has been called with vectored semantics

- and, if so, how many times to call the wrapped function
- and, with what inputs

Collectively these are referred to as *the parameters of vectorization*, or simply *the vectorization*.

6.2.1 Expected Rank

To begin, each argument passed to a wrapper has an *expected rank*: a non-negative integer representing its dimensionality, or the number of indices required to uniquely identify a single element within the argument. SLIRP infers the expected rank of each function argument directly from its prototype, *at code generation time*. For example, the primary arguments of `vmult` and `strlen` from the preceding section are of ranks 1 and 0: when invoked as prototyped the primary inputs to `vmult` are 1D arrays, while `strlen` expects a scalar (one C-style string), which for generality is considered an array of dimension zero. An N-dimensional C array such as

```
double matrix[3][3][5];
```

has a rank of N (3 in this case), with the proviso that arguments such as

```
int something(double x[3][5])
int something_else(double **x)
```

are not commensurate. The first function will be vectorized with rank 2; the second, however, will be vectorized with rank 1, because `double**` describes a 1D array of pointers, represented as opaque types in S-Lang scope. A pointer argument may be tagged as multidimensional, though, through the use of multiple `DIM` annotations. For example, a C prototype of

```
   void   sub1_2d(int *matrix, int numrows, int numcols);
```

vectorized with

```
#vectorize
   void   sub1_2d(int *matrix, int DIM1, int DIM2);
#end
```

yields a wrapper which accepts 1 argument, an integer array of rank 2. The value of the second and third arguments to the underlying C function will be calculated automatically from the input S-Lang array.

6.2.2 Number of Iterations

The *actual rank* of an argument is its dimensionality *as passed at runtime*. When the actual rank of any argument exceeds its expected rank, SLIRP needs to determine how many times the wrapped function should be called, or the *number of iterations* of the vectorization. In the preceding `vmult` example this value is 2; extended to 3D

```
slsh>  Arr3D = Double_Type[2,2,3]
slsh>  Arr3D[0, *, *] = Arr
slsh>  Arr3D[1, *, *] = 2 * Arr
slsh>  result = vmult(Arr3D, [7,8,9] )
```

`vmult` proper will be called 4 times, and the wrapper will yield the following 2x2x3 array

```
|  | 35   40   45   |  |
|  | 700  800  900  |  |
|                    |
|  | 70   80   90   |  |
|  | 1400 1600 1800 |  |
```

SLIRP determines the number of iterations by selecting a *master array* `M`, simply the input argument of highest rank, and then computing the product of its excess dimensions. Formally, if `A` and `E` respectively represent the actual and expected ranks of `M`, and `D` is a vector of length `A` describing the size of each dimension of `M` (in row-major form), then

$$Num\_Iterations = \prod_{i=1}^{A-E} D[i]$$

with the expected proviso that

$$\prod_{i=J}^{K} D[i] = 1$$

is by definition when `K < J`.

6.2.3 Stride

Finally, SLIRP determines what to pass to each wrapped function invocation by calculating a *stride* for each input argument; this indicates by how much the index into the given argument — viewed as a linear sequence of contiguous elements — should be advanced after each call. SLIRP aims for flexibility by allowing arrays of different shapes to be used within a single vectored call. For instance, in the 3D `vmult` call above the strides of the first and second arguments are 3 and 0, respectively; within the wrapper `Arr3D` and [7, 8, 9] are effectively represented as

```
double  *arg1 = {5, 5, 5, 100, 100, 100, 10, 10, 10, 200, 200, 200};
double  *arg2 = {7, 8, 9};
```

the return value is allocated as

```
double  *retval = malloc( sizeof(double) * 12);
```

and the 4 calls to `vmult` proper are executed as

```
vmult(arg1, arg2, retval, 3);
vmult(arg1+3, arg2+0, retval+3, 3);
vmult(arg1+6, arg2+0, retval+6, 3);
vmult(arg1+9, arg2+0, retval+9, 3);
```

The stride of the master array `M`, and all isomorphic arguments, is thus the number of elements in `M` not covered by its excess dimensions; that is, the product of its expected dimensions (rank)

$$\boldsymbol{Stride} = \prod_{i=A-E+1}^{A} \boldsymbol{D[i]}$$

To see why, recall that the product of all dimensions of `M`, i.e. the total number of elements, is

$$\boldsymbol{Num\_Elements} = \prod_{i=1}^{A} \boldsymbol{D[i]}$$

and must be equal to the product of the excess and expected dimensions

$$\prod_{i=1}^{A-E} \boldsymbol{D[i]} \prod_{i=A-E+1}^{A} \boldsymbol{D[i]}$$

Because the first term here is the number of iterations, the stride may be expressed concisely as

$$Stride = Num\_Elements / Num\_Iterations$$

An error will be signaled if arguments not isomorphic to `M` have number of elements not equal to the stride of M; such arguments will otherwise be assigned a stride of 0.

6.3 When Functions Will Not be Vectorized

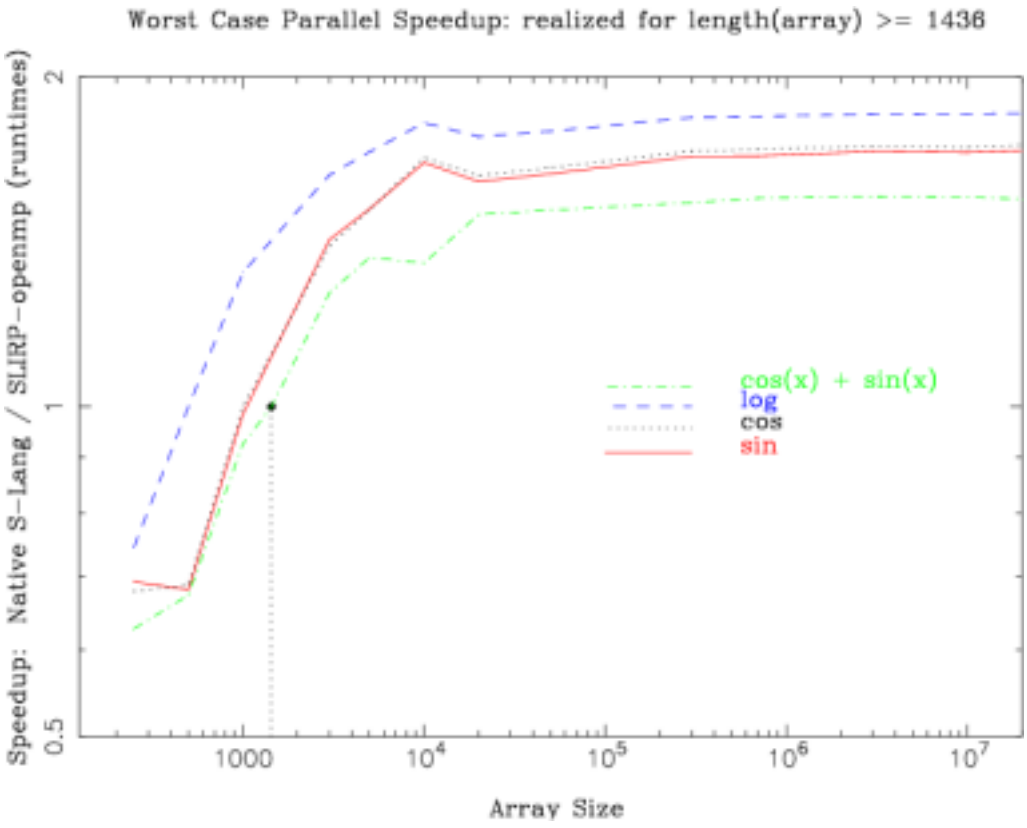
In the majority of cases SLIRP attempts to faithfully vectorize every function requested. However, it will refuse to vectorize functions

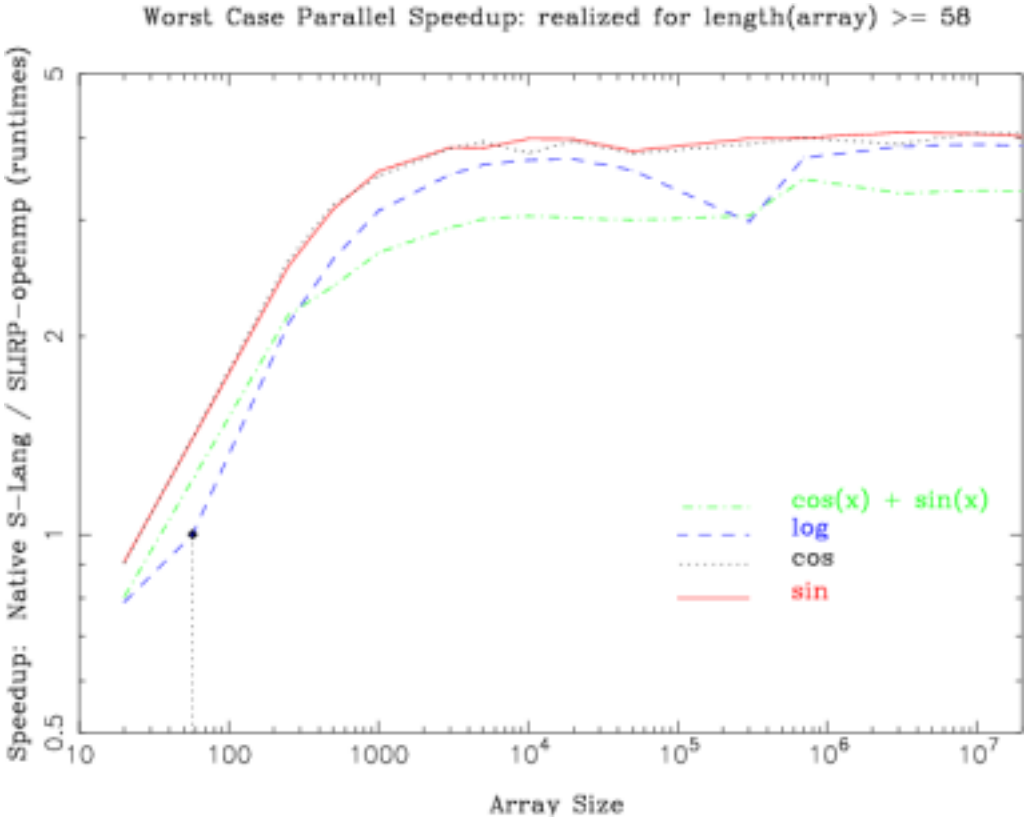
- in which an annotation has already been applied to one or more arguments, because vectorization is itself an annotation.
- with an empty argument list, because an argument would have to be introduced in order to specify the number of iterations of the vectorization.
- with more than 10 input arguments.
- listed within a `#novectorize` block.
- coded in Fortran and accepting an array argument of dimensionality 2 or greater, because of the constraints of transposition.

The constraint on the number of vectorized input arguments may be changed by re-configuring the distribution with `-with-nvec=NEW_VALUE`, although you should consider that functions with too many input arguments may indicate questionable design.

6.4 Parallelization

As noted earlier, the `-openmp` option will tune the emission of vectorized wrappers for parallelization with a suitable OpenMP-aware compiler, making it very easy to gain potentially significant performance increases on multiprocessor (or multicore) machines. Consider the plots below, which indicate the performance of SLIRP OpenMP-aware wrappers for a selection of commonly used ANSI C math functions, over a range of real-valued arrays. The performance numbers for the first plot were generated with 2 1.8 GHz AMD Athlon MP 2200 processors, on Debian Gnu/Linux 3.1 (Sarge) using the 20070221 pre-release version of GCC 4.2; the second set of numbers was generated with 4 750Mhz sparcv9 processors, on Solaris 5.9 with Sun Studio 9. The module and scripts used to generate and plot these performance data, as well as the data themselves, are located in the `examples/openmp` subdirectory of the SLIRP distribution. The plots were generated within the ISIS modeling and analysis tool ( ).


Worst Case Parallel Speedup: realized for length(array) >= 1436
Speedup: Native S-Lang / SLIRP-openmp (runtimes)
0.5
1
2
1000
10^4
10^5
10^6
10^7
Array Size
cos(x) + sin(x)
log
cos
sin


Worst Case Parallel Speedup: realized for length(array) >= 58
Speedup: Native S-Lang / SLIRP-openmp (runtimes)
0.5
1
2
5
10
100
1000
10^4
10^5
10^6
10^7
Array Size
cos(x) + sin(x)
log
cos
sin

6.4.1 Limitations

At present the `-openmp` option cannot be used to wrap Fortran functions. It also reduces the flexibility of striding, in the sense that each array passed to a parallelized wrapper must have a stride of 1.

Chapter 7

Other Annotations

7.1 #clear

Grammar

`#clear` *parameter_list*

Description

This directive deletes all annotations which match the given *parameter_list*.

Example

The directive

```
#clear double *result
```

would delete the annotation applied above to `ksink_mult2`, causing the generated wrapper to assume the default form (expecting 3 input arguments, and returning nothing on the stack). The directive sequence

```
#clear double *result
#copy double *OUTPUT {double *result}
```

also eliminates the annotation, but then replaces it with a copy of the default `double*` outmap.

7.2 #copy

Grammar

`#copy` *parameter_list* { *parameter_list* [, ...] }

Description

This directive provides a convenient way of targeting the application of existing `#argmap` annotations to specific functions, without redeclaring their prototypes. For each destination *parameter_list* pattern a copy is created of every `#argmap` which matches the source *parameter_list*. The *parameter_list* of each duplicate annotation is then changed to the corresponding

destination *parameter_list*. Note that any usage qualifier specified in the original annotation will not be propagated, since such content is usually tailored specifically to the original type.

Example

The `ksink_mult2` transformation given in the previous chapter could have been accomplished more easily by copying the `double *OUTPUT` mapping introduced in section 5.2.1 (Builtin_Out_Maps):

```
#copy double *OUTPUT { double *result }
```

An error will be thrown if any destination *parameter_list* is not isomorphic with the source *parameter_list*. That is, a single-argument copy cannot be made from a multi-argument annotation, and vice-versa.

7.3 #define

Grammar

`#define C_identifier substitution_text`

Description

This directive provides a cleaner and more compact alternative to the `slirp_define_macro()` function. It may be used to define simple numeric and (quoted) string constants, or perform verbatim substitution of non-quoted string literals, but not to define parameterized macros. Attempts to `#define` a macro more than once, without first using `#undef`, will be honored with a warning.

Example

```
#define PLATFORM        "unix"
#define VERSION         5
#define MY_PI           3.1415926535897932384
#define BEGIN_DECLS
```

The first three directives define a string, integer, and double constant, respectively; the last ensures that if BEGIN_DECLS is seen while processing source files it is replaced with the empty string, instead of the definition specified within the source.

7.4 #ignore

Grammar

`#ignore`

symbol_name_list

`#end`

Description

This directive provides an alternative mechanism for instructing SLIRP to bypass the generation of wrapper code for one or more functions, macros, variables, or type definitions. For

example, rather explicitly adding symbol names to the `ignored_functions`, `ignored_macros`, or `ignored_variables` arrays, you would instead place the symbol names within an `#ignore` block.

This approach yields shorter and cleaner interface files, since the names of all ignored symbols may be grouped within a single semantic block (although this is not required, as multiple `#ignore` blocks are permitted), symbol names remain unquoted, and comma delimiters may be avoided.

Example

Several `kitchensink` symbols can be omitted from its corresponding module via:

```
#ignore
ksink_mult      % some functions ...
ksink_print_array_f, ksink_datum_new
KSINK_IGNORABLE_INT     % a variable
ksink_datum_destroy     % another function
#end
```

7.5 #inline_c

Grammar

```
#inline_c /(init)/
  C_translation_unit
#end
```

Description

This directive supports the inclusion of essentially arbitrary C source code within your interface file. Unlike with `#argmap` or `#retmap` fragments, however, inlined source is injected verbatim into the generated code; that is, no parameter or variable substitutions are performed.

Normally inlined code blocks are inserted into the generated source before any wrapper functions are defined, which permits `#argmap` and `#retmap` annotations to reference their content. The `init` qualifier may be used to specify that the code should instead be injected into the module initialization function.

Example

Suppose you were wrapping a C++ library which includes a `ProcessGlobal` class that is intended to be instantiated only once per application. Ordinarily, methods of this (or any) class would be called from S-Lang scope by passing in an instance of the class as its first argument. For example, a `report()` method with a zero-argument C++ signature would be invoked from S-Lang scope as

```
variable pg = ProcessGlobal_new();
ProcessGlobal_report(pg);
```

This is exactly how C++ invokes object methods, the only difference being that in C++ the first argument, *this*, is generated by the compiler and usually hidden from the programmer. You could achieve a similar effect by instantiating a hidden `ProcessGlobal` object within in your wrappers

```
#inline_c
ProcessGlobal *pg = new ProcessGlobal();
#end
```

and then omitting the `ProcessGlobal*` argument from each method

```
#argmap(in, omit) ProcessGlobal*
   $1 = pg;
#end
```

to yield a wrapper (see `examples/cpp` for more details) such as

```
static void sl_ProcessGlobal_report (void)
{
   ProcessGlobal* arg0;

   if (SLang_Num_Function_Args != 0)
        {Slirp_usage_err(13, 13); return;}

   arg0 = pg;
   arg0->report ();
}
```

As another example, consider that an annotation resembling

```
#inline_c(init)
   if (H5open () != 0) return -1;
#end
```

is used to generate the HDF5 module, which ensures that the HDF5 library is itself initialized when the S-Lang interpreter initializes the module.

7.6 #prototype

Grammar

`#prototype`

function_prototype_list

`#end`

Description

This directive supports the declaration of one or more function prototypes directly within a SLIRP interface file. These supersede prototypes declared within header files, and are used to steer the pattern matching and code generation processes by relabeling parameters to match annotations.

Example

A third way to achieve the `ksink_mult2` transformation given in the previous chapter would be to redeclare it as

```
#prototype
   void ksink_mult2 (double op1, double op2, double *OUTPUT);
#end
```

This allows SLIRP to match it against the built-in `double *OUTPUT` argmap, and would yield the same wrapper code as before. Note that, as in C, each function declaration must be terminated with a semicolon.

7.7 #rename

Grammar

`#rename SLang_regular_expression C_identifier`

Description

This directive supports renaming one or more functions in the same manner as does the `-rename` command line option.

Example

```
#rename sin             vsin
#rename strlen          vstrlen
```

The `examples/vec` sample code vectorizes the C `sin` and `strlen` functions, among others. These `#rename` directives are used in the example interface file to ensure that the vectorized wrappers are distinguishable from the corresponding native S-Lang intrinsics.

7.8 #retmap

Grammar

`#retmap /(omit)/ C_type_expression`

code_fragment

`#end`

Description

By default SLIRP generates code to pass the return value from a wrapped function back to S-Lang scope, typically by having the last statement within the wrapper push an object of the appropriate type onto the stack. The `#retmap` directive can be used to alter this scheme, allowing a wrapper to return an object of different type, more than 1 object, or even — if the `omit` qualifier is specified — no objects at all.

Example

Suppose you were wrapping a library in which some functions were prototyped to return an integer status code, such as

```
typedef enum {
        STATUS_NULL_POINTER_REF=-2,
        STATUS_OUT_OF_MEMORY=-1,
        STATUS_GOOD=0
        ...
} StatusCode;

int do_something(int i, int j);
int do_something_else(int i, int j);
```

To ensure robustness the values returned from these functions must be checked, regardless of how unlikely the functions are to fail. This yields S-Lang code such as

```
variable i = 100, j = 1, k = 200;

if (do_something(i, j) != STATUS_GOOD)
   error("Could not do_something");

if (do_something_else(i, j) != STATUS_GOOD)
   error("Could not do_something_else");

if (do_something_else(k, j) != STATUS_GOOD)
   error("Could not do_something_else");

...
```

If failure is a relatively rare event then this (necessary) strategy results in code that is not only longer and slower, but is also harder to read. However, by using the `omit` qualifier to generate a wrapper which swallows the return status code, such as

```
#retmap(omit) int
   if (result != STATUS_GOOD)
        SLang_verror(SL_INTRINSIC_ERROR,"Wrapper error: %d",result);
#end
```

we retain the same level of robustness (an error will still be signaled whenever a library function returns something other than STATUS_GOOD) while eliminating boilerplate safety code in S-Lang scope:

```
variable i = 100, j = 1, k = 200;

do_something(i, j);
do_something_else(i, j);
do_something_else(k, j);

...
```

7.8.1 Built-in `#retmap` Annotations

As a convenience SLIRP provides the following return maps:

```
#retmap  NT_STR_ARRAY
```

```
        push_null_term_str_array($1, $funcname, 0);
#end

#retmap  NT_STR_ARRAY_FREE
        push_null_term_str_array($1, $funcname, 1);
#end
```

which can be used to automate the generation of wrappers for functions which return string arrays of known length. To see how these might be used let's return to the `Account*` example given earlier in section 3.5.1 (Pointers)

```
char** account_get_users(Account *a);
```

only now suppose that the documentation for this hypothetical function states that the final element of its return value is `NULL`. With this knowledge we can determine the length of the array by traversal, and thereby return a more useful S-Lang array of `String_Type`, rather than merely an opaque string pointer. To apply the appropriate return map simply redeclare the function within your interface file, such as

```
#prototype
        NT_STR_ARRAY account_get_users(Account *a);
#end
```

or, if you'd like the wrapper to free the C array and all of its elements prior to returning,

```
#prototype
        NT_STR_ARRAY_FREE account_get_users(Account *a);
#end
```

These annotations work because, as described below, `NT_STR_ARRAY` and `NT_STR_ARRAY_FREE` are built-in SLIRP type definitions.

7.9 #typedef

Grammar

`#typedef C_identifier C_identifier /;/`

Description

This directive supports the definition of simple types, as described in section 3.1 (TypeDefinitions), directly within a SLIRP interface file.

Example

Consider applying the finalizer

```
#argmap(final) KInt (int copy)
   copy = $argnum;
   printf("Finalizing arg %d of %s (with local copy)\n",copy,$funcname);
#end
```

to the `kitchensink` module. Because input header files (in this case `ksink.h`) are not read until *after* SLIRP processes the interface file no type map entry (associating `KInt` with `int`) will be defined when this annotation is read. In such cases (in fact, whenever an unmapped type is encountered) SLIRP will either fabricate a mapping (e.g. to `void_ptr`) for it and proceed on its merry way (the default), or signal an "unmapped type" error (when the `-noautotype` switch has been specified). A `#typedef` annotation such as

```
#typedef int KInt;
```

can be used to steer the mapping if neither of these results are desired. Types defined in this manner are only hints to SLIRP, and so are not replicated within the generated code.

7.9.1 Built-in Type Definitions

SLIRP provides the following built-in type definitions

```
#typedef char** NT_STR_ARRAY
#typedef char** NT_STR_ARRAY_FREE
```

which, as described above, are used to return `NULL` terminated arrays of C strings as S-Lang arrays of `String_Type`.

7.10 #undef

Grammar

```
#undef C_identifier
```

Description

This directive is the companion to `#define`; it causes the named macro to be ignored by removing it from the internal list of macro definitions. This prevents the macro from being wrapped as a constant (as described section 2.3.2 (GeneratedConstants)), and causes preprocessing conditionals such as #ifdef to evaluate to zero. Attempts to `#undef` an undefined macro will be silently ignored, however specifying a malformed identifier will signal an error.

Example

```
#define BLAH  1
#undef BLAH             % this S-Lang comment is ignored
```

Chapter 8

Annotation Grammar

The style used in this chapter generally follows Kernighan and Ritchie, 1988. Non-terminal productions are given in *italics*, while terminals and literals are given in `typewriter` style. Productions prefixed with `C_` should be understood as referring to corresponding elements of the C grammar. Ellipses connote comma-delimited sequences of one or more elements. Empty lines and whitespace between whole syntactic elements are ignored. Content within *[italicized brackets]* is optional, while `[typewriter-style brackets]` denote regular expressions. For brevity all *qualifier* and *method* literals are lumped into single production rules, even though each is not supported by every annotation. Only annotations whose grammars explicitly enumerate methods or qualifiers support such.

annotation:

- *argmap_annotation*
- *clear_annotation*
- *copy_annotation*
- *define_annotation*
- *ignore_annotation*
- *inlinec_annotation*
- *prototype_annotation*
- *rename_annotation*
- *retmap_annotation*
- *typedef_annotation*
- *undef_annotation*
- *vectorize_annotation*
- *novectorize_annotation*

argmap_annotation:

`#argmap` `(` *method* *[* `,` *qualifier_list]* `)` *parameter_list* *[* `(` *variable_declaration_list* `)]*

code_fragment

`#end`

clear_annotation:

`#clear` *parameter_list*

copy_annotation:

`#copy` *parameter_list* { *parameter_list* *[*`,` *...]* }

define_annotation:

`#define` `C_identifier` `substitution_text`

ignore_annotation:

`#ignore`

symbol_name_list

`#end`

inlinec_annotation:

`#inline_c` *[*`(init)`*]*

`C_translation_unit`

`#end`

prototype_annotation:

`#prototype`

function_prototype_list

`#end`

rename_annotation:

`#rename` `SLang_regular_expression` `C_identifier`

retmap_annotation:

`#retmap` *[*`(omit)`*]*

code_fragment

`#end`

typedef_annotation:

`#typedef C_identifier C_identifier` [`;`]

undef_annotation:

`#undef C_identifier`

vectorize_annotation:

`#vectorize`

vectorized_function_list

`#end`

novectorize_annotation:

`#novectorize`

symbol_name_list

`#end`

code_fragment:

statement_list

{ *statement_list* }

{ *code_fragment* }

statement_list:

statement

statement_list statement

statement:

`C_statement`

`C_statement_with_SLIRP_substitutions`

parameter_list:

parameter_expression

(*parameter_expression* , ...)

qualifier_list:

qualifier_expression

qualifier_list , ...

qualifier_expression:

qualifier

qualifier = *qualifier_value*

variable_declaration_list:

`C_type_expression` `C_identifier`

variable_declaration_list , ...

parameter_expression:

`C_type_expression`

`C_type_expression` `C_identifier`

function_prototype_list:

function_prototype

function_prototype `SLang_comment`

function_prototype function_prototype_list

function_prototype:

`C_identifier` (*parameter_expression* [, ...]) ;

vectorized_function_list:

vectorized_function_specifier

vectorized_function_specifier vectorized_function_list

vectorized_function_specifier:

function_prototype

`C_identifier`

symbol_name_list:

`C_identifier`

`C_identifier SLang_comment`

`C_identifier ,` *symbol_name_list*

`C_identifier` *symbol_name_list*

method: one of

`in out final ignore setup`

qualifier_value: one of

`[1-9]+`

`SLang_array_index_expression`

`C_string_literal`

qualifier: one of

`which omit usage`

Chapter 9

Command Line Reference

`-c++`

Mandate that input headers be interpreted as C++, which can be helpful to coerce the interpretation of files that either lack a .hh suffix or do not contain class definitions or other explicit C++ syntax.

`-cfront`

Generate standalone C wrappers (.cc and .h files) from the headers of a C++ library, enabling it to be called from C scope but not S-Lang proper.

`-const_strings`

Directs SLIRP to interpret functions with 'char*' return values as though they were actually 'const char*' instead. Without the aid of hints like 'const' SLIRP has no way of ascertaining if the function being wrapped returns a string which must be freed by the caller to avoid a memory leak. Since newer libraries that are carefully written tend to make judicious use of the 'const' qualifier, SLIRP takes the position that if 'const' is missing then a call to free() is required after pushing char* instances onto the stack. The `-const_strings` switch turns that behavior off, but may result in memory leaks within your module.

`-d`

Include slirp_debug_pause() debugging stub within module.

`-fprefix` *`function_prefix`*

Directs SLIRP to generate wrapper functions *only* for those functions with names that begin with an exact match of *function_prefix*. This can be useful in situations when SLIRP either generates too many functions or two few, perhaps because it cannot distinguish between private and public functions in the api being wrapped. By default SLIRP attempts to generate code for any function whose name matches the regular expression "^[a-zA-Z]+".

`-g`

Directs SLIRP to print debugging information during code generation.

`-ignore` *option*

Tune the emission of ignored symbol messages: use `notrunc` to append to existing ignored symbol list (default: the ignored symbol list is truncated at startup), or specify an alternate output file name (default: ./ignored.txt)

`-I` *directory*

Specify directory to search for headers during code generation. Each `-I` opton is automatically propagated to make files generated by `-make`, as is any path prefix included in the specification of any input header file.

`-L` *directory*

Specify directory to search, within emitted make file, at link time. Use of this option implicitly turns on `-make`.

`-l` *lib*

Specify the short-form name (consult your linker documentation) of a library to pull in, within emitted make file, at link time. Use of this option implicitly turns on `-make`.

`-ldflags` *flags*

Specify additional flags to pass to the linker within emitted make file. May also be used as an alternative to `-L` or `-l`. To specify multiple linker flags with a single `-ldflags` option, separate them with whitespace and enclose within single- or double-quotes. Use of this option implicitly turns on `-make`.

`-m` *module_name*

Emit generated code to files whose names begin with *module_name*, instead of the default (which uses the stem of the first input file that is processed).

`-make`

Emit a make file that can be used to automate compilation of the module. The content of the generated file can be tuned by use of the `-I`, `-L`, `-l`, and `-ldflags` options.

`-mapnames` *regular_expression replacement*

Deprecated synonym for `-rename`

`-noautotype`

By default SLIRP will automatically map unknown types (i.e. types which have no entry in the SLIRP typemap table) to `void_ptr`. This tactic permits larger portions of libraries to be wrapped automatically, and with less interface file writing. This switch should be employed when this behavior is not desired, with the effect that SLIRP will not emit wrapper code for any function whose signature contains unknown types.

`-noinit`

Do not generate a module initialization fragment.

`-nopop`

Whenever possible, do not generate code to explicitly pop arguments from the S-Lang stack into local variables within the wrapper. For instance, using `-nopop` during the generation of the `kitchensink` module above would yield a `ksink_mult` wrapper which looks like

```
static void sl_ksink_mult (double* arg1, double* arg2)
{
   double result;
   result = ksink_mult(*arg1, *arg2);
   (void) SLang_push_double ( result);
}
```

Using this option will thus shrink the size of generated code and may boost runtime performance slightly, but at the expense of other features like the automatic generation of **Usage:** statements when an incorrect number of arguments are passed to the function. Note even when -nopop is specified there may be instances when SLIRP has no choice but to explicitly pop arguments from the stack; these conditions are enumerated within the internal source code documentation.

`-otree`

Emit a hierarchical list of all opaque types defined by the current invocation, then exit. The former name of this option was `-l`.

`-openmp`

Emit OpenMP #pragmas to parallelize vectorized code. This option implies -vec, i.e. it turns on vectorization.

`-print`

Print the interface for code which would be generated, but don't actually generate it.

`-rc` *file*

Load SLIRP customizations from *file*, rather than from the first instance found within $PWD/slirprc or the $SLIRPRC environment variable.

`-refscalars`

Permit S-Lang scalars to be passed to C functions expecting arrays, or Fortran functions (which passes all arguments by reference), by popping them off the stack as 1-element arrays. By default this behavior is on for bindings to Fortran codes, and off for bindings to C codes. Note that this does not provide a way of modifying scalar variable values absent the use of the S-Lang reference operator (&), as the original scalar is effectively read-only in the glue layer.

`-rename` *regular_expression replacement*

Directs SLIRP to map C function names matching the given *regular_expression* to S-Lang function names beginning with *replacement*. Suppose, for example, that

```
slirp -mapnames ksink_ yada_ ksink.h
```

were specified during the generation of the `kitchensink` module in the opening chapter. Then the `do_mult()` sample function would need to reflect such

```
define do_mult(op1, op2)
{
   () = printf("ksink_mult(%S,%S) = %S\n",
                          op1, op2, yada_mult(op1, op2));
}
```

This option may be specified multiple times at invocation, using any regular expression suitable for the S-Lang `string_match()` function. Consult the *Regular Expressions* chapter within the S-Lang language guide for more details.

`-stdout`

Cause generated code to be emitted to stdout instead of a named file.

`-stubs`

Generate source code for the interface specified as input to SLIRP, as a set of empty (stub) functions. This supports exercising the module interface without the need to link in the underlying library or any of its (potentially numerous) dependencies.

`-tmapout` *file*

Save a copy of the type mappings table (generated during the first pass SLIRP makes over its input header files) to the given file, as a series of SLIRP commands that can executed by the S-Lang interpreter to recreate the typemap table.

`-tmapin` *file*

Load, using `evalfile()`, additional typemaps from the given file, which was presumably created by a `-tmapout` switch from a previous invocation of SLIRP.}

`-v`

Show verbose messages when loading S-Lang scripts

`-vec`

Attempt to vectorize every function within input interface

`-version`

Output the SLIRP version.