FFIGEN Back-end for Chez Scheme Version 5*

Lars Thomas Hansen
lth@cs.uoregon.edu

February 7, 1996

1 Introduction

This document describes the FFIGEN back-end for Chez Scheme version 5. The exposition is provided in the hope that it will make it reasonably clear how back-ends for FFIGEN are constructed; many back-ends will probably have a structure similar to that of the present program.

If you haven't read the *FFIGEN Manifesto and Overview*, now is a good time to do so.

If you haven't read the *FFIGEN User's Manual,* which documents the target-independent code for FFIGEN back-ends and the format for the intermediate code, now is a good time to do so.

This document is organized in the following manner. Section 2 documents the overall structure of the translation for Chez Scheme. Section 3 presents the policy choices made for data types and most of the code implementing the policy; section 4 presents several utility procedures used by the implementation. Section 5 discusses some of the implementation choices, particularly in terms of performance, and section 6 summarizes future work on the Chez Scheme back-end. Section 7 presents the implementation of the support libraries.

This document is also the program: the LaTeX source for the document, the Scheme source code for the program, and the standard library sources have been derived from the same file using the **nuweb** system for literate programming.[1] The document presents the program in an order which I believe is conducive to understanding its workings. Here's how you read it: each snippet of code (called a *macro* or *scrap*) has a title which includes the page number on which the scrap is defined and a letter following the page number which shows the relative place on that page where you can find the scrap (for example, 9a is the first scrap on page 9 and 6c is the third scrap on page 6). Scraps reference other scraps; these references are the titles, so any cross-reference immediately gives you the page number of the referenced code. In addition, an index of global identifiers can be found in the appendix.

2 Overall Structure

The FFIGEN back-end for any FFI consists of two parts: a target-independent part which reads the output of the front end into global in-memory data structures, and a target-dependent part which does the actual translation. The target-dependent part is implemented by two procedures, `select-functions` and `generate-translation`.

2.1 The `select-functions` procedure

The procedure `select-functions` is called during initial processing to mark as selected those procedures which are interesting for the translation. This makes it possible to exclude some functions from the translation. For example, if you are translating the interface to a library whose header file includes `stdio.h`, you may wish to exclude all function prototypes included from the latter file. Since each function record in the

*This work has been supported by ARPA under U.S. Army grant No. DABT63-94-C-0029, "Programming Environments, Compiler Technology and Runtime Systems for Object Oriented Parallel Processing".

[1] `nuweb` was implemented by Preston Briggs of Rice University; you may retrieve it by anonymous ftp from `cs.rice.edu` in the directory `/public/preston`.

intermediate form has a field naming the file the record came from, you can easily exclude functions from `stdio.h`.

The implementation of `select-functions` for Chez Scheme simply marks all functions as selected, by turning on the "referenced" bit in the record.

⟨the select-functions interface procedure 2a⟩ ≡

```
(define (select-functions)
  (do ((functions functions (cdr functions)))
      ((null? functions))
    (referenced! (car functions))))
```

◇

Macro referenced in scrap 2d.

2.2 The `generate-translation` procedure

The procedure `generate-translation` is called when the target-independent part is done processing the data, and it must implement all aspects of the target-dependent translation. It takes no arguments. My version sets up the output files, initializes the files, calls the generating functions, and cleans up.

⟨the generate-translation interface procedure 2b⟩ ≡

```
(define (generate-translation)
  ⟨initialize output files 3b, ... ⟩
  ⟨generate code for all constructs 2c⟩
  ⟨finalize output files 4b⟩
  #t)
```

◇

Macro referenced in scrap 2d.

`generate-translation` calls on specialized procedures to generate code for the various intermediate language records:

⟨generate code for all constructs 2c⟩ ≡

```
(dump-structs)
(dump-unions)
(dump-functions)
(dump-variables)
(dump-enums)
(dump-macros)
```

◇

Macro referenced in scrap 2b.

2.3 Structure of the Scheme source file

The final file is called `chez.sch` and is laid out in the following way:

"`chez.sch`" 2d ≡

```
⟨back-end global variables 3a⟩
⟨the select-functions interface procedure 2a⟩
⟨the generate-translation interface procedure 2b⟩
⟨Chez FFI names for primitive types 4c⟩
⟨dump structs and unions 5, ... ⟩
⟨dump global variable accessors 10b⟩
⟨dump function definitions 11, ... ⟩
⟨dump enum definitions 13a⟩
⟨dump macro definitions 13b, ... ⟩
⟨utility functions 14b, ... ⟩
```

◇

2.4 The standard library

To facilitate the translation, a small standard library has been created ahead of time (see section 7 for the implementation). The library implements primitive dereferencing functions and a memory copy procedure. There is a C file and a Scheme file. The C file must be compiled and loaded into the Scheme system using the FFI facilities for this (which happen to be operating-system dependent in Chez Scheme). The Scheme FFI code for the standard library must be loaded into the Scheme system at run time.

3 Policy in the Chez Scheme Back-end

Chez Scheme version 5 has a fairly simple native FFI. It supports most primitive C data types for both parameters and return values, and performs data conversions on call and return. A string is passed as a `char*` to the first character (in Chez Scheme, strings are 0-terminated for C compatibility), but is returned by copying the data into a fresh Scheme string. There is no direct support for the `short` datatype, variadic procedures, call by reference, calling functions through a function pointer, struct/union arguments, or struct/union return values, nor is there support for passing general Scheme objects like pairs, arrays, or functions to foreign functions.

In this (first) version of the translation I have decided not to implement any mechanisms for handling special cases like the `fgets()` example outlined in the *Manifesto*; that will have to come later. Such a mechanism might take the form of a file containing rules for how to override certain function signatures, for example. See section 6.

3.1 The Output

Two output files are produced: a C file and a Scheme file. The Scheme file will hold the FFI code which interfaces to the library and its data types. The C file will hold supporting ("glue") code generated along the way: accessors, mutators, and so on. The need for supporting code will become clear as you read.

`c-output` and `sch-output` are global variables which hold output ports for generated C code and Scheme code, respectively.

⟨back-end global variables 3a⟩ ≡

```
(define c-output #f)
(define sch-output #f)
◇
```

Macro referenced in scrap 2d.

The files are initialized in `generate-translation` by computing file names and assigning opened ports to the global variables; for simplicity I use fixed names for the time being. The use of `delete-file` is necessary in Chez Scheme, lest it complain when opening an existing file for output.

⟨initialize output files 3b⟩ ≡

```
(delete-file "C-OUTPUT")
(delete-file "SCH-OUTPUT")
(set! c-output (open-output-file "C-OUTPUT"))
(set! sch-output (open-output-file "SCH-OUTPUT"))
◇
```

Macro defined by scraps 3b, 4a.
Macro referenced in scrap 2b.

Once the files are opened they are initialized with standard definitions. The C file `#include`s the FFIGEN standard library header and the ANSI C standard library header; there is nothing to be done in the Scheme case. We should probably also emit a line here which includes the original header file, but the name of that file is not currently available in the intermediate format.

⟨initialize output files 4a⟩ ≡

```
(display "#include \"chez-stdlib.h\"" c-output) (newline c-output)
(display "#include \"stdlib.h\"" c-output) (newline c-output)
```

◇

Macro defined by scraps 3b, 4a.
Macro referenced in scrap 2b.

When processing is done, the output files must be closed:

⟨finalize output files 4b⟩ ≡

```
(close-output-port c-output)
(close-output-port sch-output)
```

◇

Macro referenced in scrap 2b.

3.2 Names

Since Scheme is case-insensitive and C is not, names which do not clash in C may do so in Scheme. This can be handled fairly simply by keeping track of all generated Scheme names and warning about conflicts. In this version I have not implemented this, but it will be implemented in the future.

In general, parameter names in generated C procedures are prepended with underscores to prevent name clashes with `typedef`s: names starting with a single underscore are reserved for libraries and will in principle not appear in user code. However, this scheme is not foolproof, considering that FFIGEN is used to translate library code. We might need to come up with something better.

3.3 Primitive Types

The procedure `chez-type` takes a primitive type structure and returns the name of the Chez Scheme parameter or return type. Already we have to make hard choices: what do we do with pointers, in particular character pointers? In the current implementation I let a `char*` be a `char*`; programs which wish to pass strings will have to first copy the string data into an allocated character buffer and then pass the address of the buffer. A function which performs the copy operation is part of the standard library.

Other decisions I have made are to represent pointers as unsigned 32-bit ints (hence all pointers are the same size), and to represent `char`, `signed char`, and `unsigned char` all as the same `char` FFI type. If the type is a `short` variant, a warning is generated.[2] Enums are assumed to be 32-bit integers.

⟨Chez FFI names for primitive types 4c⟩ ≡

```
(define (chez-type type)
  (case (record-tag type)
    ((pointer) 'unsigned-32)
    ((int long enum) 'integer-32)
    ((unsigned unsigned-long) 'unsigned-32)
    ((char unsigned-char signed-char) 'char)
    ((void) 'void)
    ((double) 'double-float)
    ((float) 'single-float)
    ((***invalid***) '***invalid***)
    (else
     (warn "Cannot translate this type: " type)
     (string->symbol (string-append (symbol->string '***invalid:)
                                    (symbol->string (record-tag type))
                                    "***")))))
```

◇

Macro referenced in scrap 2d.

[2] Chez Scheme does not support a `short` FFI type. Since the ANSI C standard (or at least K&R 2nd edition) seems to say that a `short` can be passed to a function with a prototype in scope without being widened to `int`, we can't simply use `integer-32` as the FFI argument type for a `short`. On many architectures using a 32-bit integer would work because the APIs don't support parameters smaller than 32 bits. We could also have used proxy functions; see section 3.4.3.

The special type `***invalid***` and its variant `***invalid:tag***` are used to signal errors detected during translation; by using these types in the output, no file that was generated with errors can actually be used.

In addition, we have to consider how to dereference pointers to primitive types (pointers to structured types are handled later). For example, if I have an `int*` and I want to access the integer it points to, how do I do this? For this purpose, dereferencing functions are provided in the standard library. Each function takes a pointer `p` and an offset `k` and fetches the element `p[k]`. There are also functions which store values through pointers to primitive types. See section 7 for the signatures and the actual implementation.

By treating pointers to pointers as generic pointer types, we handle the general case of recursive dereferencing. Since pointers and integers are the same size and type checking is lax, we can use the unsigned integer access functions to follow pointer chains. For example, if we have an `int** p` and want the integer, the code to access it would be (in C) `_ref_int(_ref_uint(p,0),0)`.

3.4 Structured Types

The translations for structs and unions are similar. I'll talk about structures, but the discussion pertains to unions as well. The procedures `dump-structs` and `dump-unions` are almost identical, and call the same function to do their work:

⟨dump structs and unions 5⟩ ≡

```
(define (dump-structs)
  (dump-struct/union structs struct-names "struct"))

(define (dump-unions)
  (dump-struct/union unions union-names "union"))
◇
```

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

`struct-names` and `union-names` are procedures which return a list of all `typedef`s which directly name a struct or union; see the *FFIGEN User's Manual* for a discussion.

In the following, we will use this structure as an example:

```
struct X {
  int i;
  char c[5];
  union {
    char *s;
    int *q;
  } u;
};
```

Each structure in the intermediate form has a tag (`X` in the example), although some of these tags were generated by the compiler. Whether the tag was programmer-defined or not is important for how code is generated (see below), and the procedure `dump-struct/union` takes it into account when generating FFI code for each referenced structure.

Only structures which have programmer-defined tags have operations generated for them with the name of the tag. Structures which are referred to by a `typedef` must be handled specially. There are two cases. If the `typedef`'d structure also has a programmer-defined tag, then Scheme definitions are emitted which define operations using the `typedef` name to be the same as the operations using the structure tag. If the type does not have a programmer-defined tag, then operations are defined on the structure using the `typedef` name only; you get names like `_get_X_i`.

⟨dump structs and unions 6a⟩ ≡

```
(define (dump-struct/union records typedef-name-getter qualifier)
  (for-each
   (lambda (structure)
     (if (referenced? structure)
         (begin
           (if (user-defined-tag? (tag structure))
               (dump-struct/union-def structure qualifier (tag structure)))
           (for-each (lambda (n)
                       (if (user-defined-tag? (tag structure))
                           (generate-reference-to-structure structure n qualifier)
                           (dump-struct/union-def structure "" n)))
                     (typedef-name-getter structure)))))
   records))
```

◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

The procedure **generate-reference-to-structure** takes a structure which has a cached list of defined constructor, destructor, accessor, and mutator function, a **typedef**-name, and a qualifier (**struct**, **union**, or the empty string) and generates Scheme definitions which use the **typedef**-name but which refer to the already-defined functions using the qualified name and structure tag. For example, if there is already a foreign function called **_get_struct_X_F** and the **typedef** name is **Y**, then a variable called **_get_Y_F** will be bound to the value of the existing foreign function.

⟨dump structs and unions 6b⟩ ≡

```
(define (generate-reference-to-structure structure typedef-name qualifier)
  (for-each (lambda (n)
              (let ((newname (compute-newname n typedef-name (tag structure) qualifier)))
                (display `(define ,newname ,n) sch-output)
                (newline sch-output)))
            (cached-names structure)))
```

◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

The procedure **compute-newname** takes an already emitted name and generates a name which uses the **typedef** name.

⟨dump structs and unions 6c⟩ ≡

```
(define (compute-newname oldname typedef-name tag qualifier)
  (let ((q (string-append qualifier "_" tag)))
    (let ((get   (string-append "_get_" q))
          (set   (string-append "_set_" q))
          (alloc (string-append "_alloc_" q))
          (free  (string-append "_free_" q)))
      (cond ((string-prefix=? oldname get)
             (string-append "_get_" typedef-name (substring oldname (string-length get)
                                                            (string-length oldname))))
            ((string-prefix=? oldname set)
             (string-append "_set_" typedef-name (substring oldname (string-length set)
                                                            (string-length oldname))))
            ((string-prefix=? oldname alloc) (string-append "_alloc_" typedef-name))
            ((string-prefix=? oldname free) (string-append "_free_" typedef-name))
            (else (error "compute-newname: can't handle: " oldname))))))
```

◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

The procedure **dump-struct/union-def** takes a structure type, a qualifier (a string, either **struct**, **union**, or the empty string) and the C name of the structure (its tag or **typedef** name), and generates constructors, destructors, accessors, and mutators for the structure.

⟨dump structs and unions 7a⟩ ≡

```
(define (dump-struct/union-def structure qualifier name)
  (let* ((funcname (if (string=? qualifier "")
                       name
                       (string-append qualifier "_" name)))
         (cast     (if (string=? qualifier "")
                       name
                       (string-append qualifier " " name))))
    (generate-constructor-and-destructor structure funcname cast)
    (generate-accessors-and-mutators structure funcname cast "")))
```

◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

3.4.1 Constructors and destructors

The procedure **generate-constructor-and-destructor** generates constructor and destructor procedures for its argument. The constructor procedure is called **_alloc_struct_X**. This procedure takes no parameters and allocates memory for the given type, returning a pointer (cast to an unsigned integer) or the literal integer 0 (which may be different from the null pointer).

The destructor procedure is called **_free_struct_X**; it takes a value returned from the constructor and frees the allocated memory. Generating a constructor may be excessive; a single interface to **free()** is probably enough. The destructors may go away in the future.

The choice of value for the null pointer warrants discussion. I use the integer 0 to make it possible to test for a null pointer by comparing with 0 on the Scheme side; it is a decision which probably simplifies the Scheme code. Using an unadulterated null pointer would make comparison with 0 nonportable, since a null pointer is not guaranteed to be all-bits-zero.

An alternative would be to generate foreign procedures which return null pointers, one procedure for each pointer type in the program, and these return values could be used for pointer comparisons.[3]

⟨dump structs and unions 7b⟩ ≡

```
(define (generate-constructor-and-destructor structure funcname cast)
  (function-pair constructor-template
                 (vector funcname cast)
                 (string-append "_alloc_" funcname)
                 '((void ()))
                 `(pointer ,(struct/union-ref structure)))
  (function-pair destructor-template
                 (vector funcname cast)
                 (string-append "_free_" funcname)
                 `((pointer ,(struct/union-ref structure)))
                 '(void ()))
  (cache-name structure (string-append "_alloc_" funcname))
  (cache-name structure (string-append "_free_" funcname)))
```

◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

In the templates for constructors and destructors, **@0** is the name of the structure as we want it to be generated: **struct_X**, **union_X**, or **X** (for a **typedef**); and **@1** is the cast expression for the type: **struct X**, **union X**, or **X**.

[3] There's one null pointer for every pointer type.

⟨dump structs and unions 8a⟩ ≡

```
(define constructor-template
  "unsigned _alloc_@0(void) {
     @1 *_p = (@1 *)malloc(sizeof(@1)); return (_p == 0 ? 0 : (unsigned)_p);
   }")

(define destructor-template
  "void _free_@0(unsigned _p) { if (_p == 0) abort(); free((@1 *)_p); }")
◇
```

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

3.4.2 Accessors and Mutators

For each field `F` of basic or array type there will be an accessor function `_get_struct_X_F`, and for each field of basic type there will be a mutator function `_set_struct_X_F`. The accessor takes a pointer and returns the field value; the mutator takes a pointer and a value and updates the field.

If the structure has nested structures, functions are generated which will access fields in the nested structures in the expected way; the naming scheme is to replace the "." in the C syntax with an underscore "_". The generated C code for the accessors and mutators will look like this:

```
int  _get_struct_X_i(unsigned _p) { return ((struct X*)_p)->i; }
void _set_struct_X_i(unsigned _p, int _v) { ((struct X*)_p)->i = _v; }
unsigned _get_struct_X_u_s(unsigned _p) { return (unsigned)((struct X*)_p)->u.s; }
void _set_struct_X_u_s(unsigned _p, unsigned _v) { ((struct X*)_p)->u.s = (char*)_v; }
```

Procedures are not generated which directly dereference pointer fields; the program must first fetch the pointer field and then dereference it. Arrays in structures are handled like pointer fields, so `_get_struct_X_c` will be generated and will return a pointer to the first element of `c`, which can then be dereferenced.

The procedure `generate-accessors-and-mutators` takes a structure type, a name to be used for the function (initially just `struct_X`, `union_X`, or `X` for a typedef name), a cast expression (`struct X`, `union X`, or just `X`), and a field selector expression (initially the empty string). As the program descends into the structure the function name and the selector expression will be updated to reflect the field which is being accessed. Appended to the function name will be an underscore and the field name, so we'll see `struct_X_i` and `struct_X_u_s`, for example. The selector will be the corresponding access expression for C, so `i` and `u.s`.

When generating accessors and mutators, there are three cases: fields of some basic type (primitive types and pointers), fields of array type, and fields of structured type.

⟨dump structs and unions 8b⟩ ≡

```
(define (generate-accessors-and-mutators structure funcname cast selector)
  (for-each
   (lambda (field)
     (let ((funcname (string-append funcname "_" (canonical-name (name field))))
           (selector (string-append selector (if (string=? selector "") "" ".") (name field))))
       (cond ((basic-type? (type field))
              (getset-basic-type structure funcname cast selector field))
             ((array-type? (type field))
              (getset-array-type structure funcname cast selector field))
             ((structured-type? (type field))
              (getset-structured-type structure funcname cast selector field))
             (else (error 'generate-accessors-and-mutators "Unknown: " field)))))
   (fields structure)))
◇
```

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

The use of `canonical-name` is important. This procedure transforms an identifier from its C syntax into a syntax acceptable to Scheme by transforming the letters to the canonical case of the representation. (For example, if the symbol `a` prints as `A` then the canonical case of the implementation is upper case.)

Fields of basic types get both accessors and mutators.[4]

⟨dump structs and unions 9a⟩ ≡

```
(define (getset-basic-type struct funcname cast selector field)
  (let* ((typename   (basic-type-name (type field)))
         (fieldtype  (c-cast-expression (type field))))
    (function-pair accessor-template
                   (vector typename funcname cast selector)
                   (string-append "_get_" funcname)
                   `((pointer ,(struct/union-ref struct)))
                   (type field))
    (function-pair mutator-template
                   (vector typename funcname cast selector fieldtype)
                   (string-append "_set_" funcname)
                   `((pointer ,(struct/union-ref struct)) ,(type field))
                   `(void ()))
    (cache-name struct (string-append "_get_" funcname))
    (cache-name struct (string-append "_set_" funcname))))
```
◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

In the accessor and mutator templates, `@0` is the value return or parameter type (pointers are always "unsigned"), `@1` is the function name, `@2` is a cast expression for the structure pointer type, `@3` is the C field selector expression, and `@4` is the cast expression from the parameter type to the field type (used in mutators only).

⟨dump structs and unions 9b⟩ ≡

```
(define accessor-template
  "@0 _get_@1( unsigned _p ) { return (@0)((@2*)_p)->@3; }")

(define mutator-template
  "void _set_@1( unsigned _p, @0 _v ) { ((@2*)_p)->@3 = (@4)_v; }")
```
◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

Array types are just like basic types, but there is no mutator because the array name is `const`. The name of the array denotes a pointer to the first element, so that's what we return.

⟨dump structs and unions 9c⟩ ≡

```
(define (getset-array-type structure funcname cast selector field)
  (function-pair array-accessor-template
                 (vector funcname cast selector)
                 (string-append "_get_" funcname)
                 `((pointer ,(struct/union-ref structure)))
                 '(unsigned))
  (cache-name structure (string-append "_get_" funcname)))

(define array-accessor-template
  "unsigned _get_@0( unsigned _p ) { return (unsigned)(((@1*)_p)->@2); }")
```
◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

[4] Fields which are `const` should not have a mutator; when we get around to implementing general qualifiers on types, this must be fixed.

Structured types are handled by adding the field name to the selector and recurring; we ignore the distinction between struct and union here as it doesn't matter.

⟨dump structs and unions 10a⟩ ≡

```
(define (getset-structured-type structure funcname cast selector field)
  (let (;(selector (string-append selector "." (name field)))
        ;(funcname (string-append funcname "_" (canonical-name (name field))))
        (struct   (if (eq? (record-tag (type field)) 'struct-ref)
                      (lookup (tag (type field)) structs)
                      (lookup (tag (type field)) unions))))
    (generate-accessors-and-mutators struct funcname cast selector)))
```

◇

Macro defined by scraps 5, 6abc, 7ab, 8ab, 9abc, 10a.
Macro referenced in scrap 2d.

3.4.3 Structures as values

As mentioned above, there are no provisions for using structures or unions as values in the program, not even when they have the size of a primitive type. The reason for this is that it is not natural for Scheme to assign objects which are larger than a single location (and special-casing small structures because the are small seems silly). For example, the following function takes a structure value as an argument (by copy, that is, by assignment):

```
void f( struct X x ) { ... }
```

There is no way to call this function directly in the generated FFI for Chez Scheme. We could fudge our way around it by generating a proxy function which acts as a level of indirection:

```
void _proxy_f( struct X *x ) { f( *x ); }
```

but I have not implemented this. If it becomes a problem it needs to be fixed. Libraries which manipulate complex numbers represented as structures may need it.

Another upshot is that there is no way to ask for the field `u` of the structure defined earlier.[5]

3.5 Global Variables

In order to simplify things, global variables are handled by generating a single function for each global variable; the function takes no parameters and returns the address of the global. Technically the following code should be finessed for arrays, since the type of a global array is not a pointer to an array, but an array, or at the very least, a pointer to the base type of the array.

⟨dump global variable accessors 10b⟩ ≡

```
(define (dump-variables)
  (for-each (lambda (v)
              (let ((n (canonical-name (name v))))
                (function-pair global-template
                               (vector n (name v))
                               (string-append "_glob_" n)
                               '((void ()))
                               `(pointer ,(type v)))))
            vars))

(define global-template
  "unsigned _glob_@0( void ) { return (unsigned)&@1; }")
```

◇

Macro referenced in scrap 2d.

[5] There is no way to ask for its address, either, but that's easy to implement.

3.6 Functions

Supporting functions in all their generality is pretty much impossible, and I'm not going to try. The restrictions are:

Variadic functions are not supported. Such support can be implemented but the resulting code is target-specific and probably involves some assembly language. Some C compilers (like Watcom C) provide library functions which makes the job much easier, but most don't.

Old-style prototypes (that is, declarations of the form `char *malloc()` where the number and types of the arguments are not known) are hard, since the native Chez FFI requires an exact argument count and the type for each argument. My solution is to generate a warning for this and translate the interface as if it had no arguments, which is almost always wrong.

Parameters of function type work fine, but you can't pass Scheme procedures to a C function.

Calling through function pointers is not implemented, since it's hard to express this in the native FFI, which requires the name of the function in order to fetch it from a library. We can get around this by generating a proxy function for each function pointer type in the header file and calling the function through it by passing it the function pointer and the arguments, essentially a trampoline.

`Dump-functions` creates a Scheme-side interface for each referenced function.

⟨dump function definitions 11⟩ ≡

```
(define (dump-functions)
  (for-each (lambda (f) (define-foreign (name f) (type f)))
            functions))
```

◇

Macro defined by scraps 11, 12ab.
Macro referenced in scrap 2d.

`Define-foreign` checks that every argument type and return type is valid and attempts to give a reasonable error message if not; if everything is OK then a Scheme definition for the foreign function is emitted.:

⟨dump function definitions 12a⟩ ≡

```
(define (define-foreign name type)
  (let ((argtypes (arglist type))
        (returntype (rett type)))
    (let loop ((l argtypes))
      (cond ((null? l) #t)
            ((structured-type? (car l))
             (warn "Cannot pass structured value of type"
                   (rational-typename (car l))
                   "to function"
                   name)
             (set-car! l '(***invalid***))
             (loop (cdr l)))
            (else
             (loop (cdr l)))))
    (if (structured-type? returntype)
        (begin (warn "Cannot receive structured value of type"
                     (rational-typename returntype)
                     "from function"
                     name)
               (set! returntype '(***invalid***))))

    (write
     `(define ,(string->symbol (canonical-name name))
        (foreign-function ,name
```

```
                              ,(chez-map-args argtypes name)
                              ,(chez-type returntype)))
          sch-output)
         (newline sch-output)))
◇
```
Macro defined by scraps 11, 12ab.
Macro referenced in scrap 2d.

The argument list must be handled carefully to deal with functions without prototypes and functions with variable-length parameter lists:

⟨dump function definitions 12b⟩ ≡
```
(define (chez-map-args args name)
  (cond ((and (= (length args) 1)
              (eq? (caar args) 'void))
         '())
        ((= (length args) 0)
         (warn "Function without prototype assumed to take no arguments:"
               name)
         '())
        (else
         (map (lambda (x)
                (if (eq? (record-tag x) 'void)
                    (begin (warn "Varargs *cannot* be handled for" name)
                           '***invalid***)
                    (chez-type x)))
              args))))
◇
```
Macro defined by scraps 11, 12ab.
Macro referenced in scrap 2d.

3.7 Enums

Enums are straightforward. There is a definition for each enum-ident:

```
enum { A=100, B, C };
```

becomes

```
(define a 100)
(define b 101)
(define c 102)
```

The enum's tag, should it have one, is ignored. The code is straightforward:

⟨dump enum definitions 13a⟩ ≡
```
(define (dump-enums)
  (for-each (lambda (x)
              (display (instantiate "(define @0 @1)"
                                    (vector (canonical-name (name x))
                                            (number->string (value x))))
                       sch-output)
              (newline sch-output))
            enum-idents))
◇
```
Macro referenced in scrap 2d.

3.8 Macros

Macros are hard in the general case, because the general case is that the right-hand-side names some arbitrarily complex C statement which might even depend on variables in some scope not in effect when the macro is defined. Handling C macros in their full generality is therefore not an attainable goal for most targets. However, we can still do useful things.

If the left-hand-side is a simple identifier (no parameters) and the right-hand-side parses as a number or a string, then we can emit a simple definition which gives the name the value.

Being more ambitious we can take each macro which has parameters and whose right-hand-side is a closed expression–that is, the macro which results from expanding all applicable macros on the right-hand-side of the macro has no free variables–and convert it to a function.

For now, we handle the simple case of an integer right-hand-side only.

⟨dump macro definitions 13b⟩ ≡

```
(define (dump-macros)
    (for-each (lambda (m)
                (if (and (valid-ident? (name m))
                         (valid-number? (value m)))
                    (begin
                      (display `(define ,(canonical-name (name m))
                                   ,(evaluate-number (value m)))
                               sch-output)
                      (newline sch-output))))
              macros))
```

◇

Macro defined by scraps 13b, 14a.
Macro referenced in scrap 2d.

`Valid-ident` and `valid-number` check that their arguments are valid C identifier and numbers, respectively.

⟨dump macro definitions 14a⟩ ≡

```
(define (valid-ident? s)
    (andmap (lambda (c)
              (or (char-upper-case? c)
                  (char-lower-case? c)
                  (char-numeric? c)
                  (char=? c #\_)))
            (string->list s)))

(define (valid-number? s)
  (let ((n (evaluate-number s)))
    n))
```

◇

Macro defined by scraps 13b, 14a.
Macro referenced in scrap 2d.

4 Utility functions

The procedure `function-pair` generates corresponding definitions in both languages.

⟨utility functions 14b⟩ ≡

```
(define (function-pair c-template template-args scheme-name arglist rett)
  (display (instantiate c-template template-args) c-output)
  (newline c-output)
  (define-foreign scheme-name
    `(function ,arglist ,rett)))
```

◇

Macro defined by scraps 14bc, 15ab, 16ab.
Macro referenced in scrap 2d.

The C type name generated for a given basic type is very much a policy issue; here are our encodings:

⟨utility functions 14c⟩ ≡

```
(define (basic-type-name type)
  (let ((probe (assq (record-tag type)
                     '((char . "char")
                       (signed-char . "signed char")
                       (unsigned-char . "unsigned char")
                       (short . "short")
                       (unsigned-short "unsigned short")
                       (int . "int")
                       (enum . "int")
                       (unsigned . "unsigned")
                       (long . "long")
                       (unsigned-long . "unsigned long")
                       (void . "void")
                       (pointer . "unsigned")
                       (float . "float")
                       (double . "double")
                       ))))
    (if probe
        (cdr probe)
        (begin (warn "Unknown type " type)
               "***invalid***"))))
◇
```

Macro defined by scraps 14bc, 15ab, 16ab.
Macro referenced in scrap 2d.

The procedure `c-cast-expression` takes a type and returns a C-syntax expression which can be used in a type cast for casting a value to the given type. It's a hard problem and the code below does not do a very good job; it needs to be refined considerably. See e.g. K&R 2nd edition section 5.12.

One issue is what happens when we have a structured type which does not have a user-defined tag. There will be exactly one `typedef` which refers to the structure, and that name should be used instead.

⟨utility functions 15a⟩ ≡

```
(define (c-cast-expression type)
  (cond ((primitive-type? type)
         (basic-type-name type))
        ((pointer-type? type)
         (string-append (c-cast-expression (cadr type)) "*"))
        ((eq? (record-tag type) 'enum-ref)
         (basic-type-name '(int ())))
        ((memq (record-tag type) '(struct-ref union-ref))
         (let ((t (tag type)))
           (if (user-defined-tag? t)
               (string-append (if (eq? (record-tag type) 'struct-ref)
                                  "struct "
                                  "union ")
                              t)
               (let ((names (if (eq? (record-tag type) 'struct-ref)
                                (struct-names type)
                                (union-names type))))
                 (if (= (length names) 1)
                     (car names)
                     (error "c-cast-expression: bad: " type))))))
        (else
         (warn "c-cast-expression: Too complicated: " type)
         "unknown")))
◇
```

Macro defined by scraps 14bc, 15ab, 16ab.
Macro referenced in scrap 2d.

String-prefix=? takes a string and a prefix and tests whether the prefix matches the string.

⟨utility functions 15b⟩ ≡

```
(define (string-prefix=? s prefix)
  (let ((limit (string-length prefix)))
    (and (<= limit (string-length s))
         (let loop ((i 0))
           (or (= i limit)
               (and (char=? (string-ref s i) (string-ref prefix i))
                    (loop (+ i 1))))))))
```

◇

Macro defined by scraps 14bc, 15ab, 16ab.
Macro referenced in scrap 2d.

Rational-typename takes a type; if the type is a struct-ref or union-ref and the tag is auto-generated, then a type is returned where the tag is replaced by the unique typedef-name which uses the structure (if one can be found). This procedure is useful for giving meaningful error messages.

⟨utility functions 16a⟩ ≡

```
(define (rational-typename type)
  (case (record-tag type)
    ((struct-ref)
     (if (user-defined-tag? (tag type))
         type
         (let ((t (lookup (tag type) structs)))
           (if (not t)
               type
               (list 'struct-ref (tag t))))))
    ((union-ref)
     (if (user-defined-tag? (tag type))
         type
         (let ((t (lookup (tag type) unions)))
           (if (not t)
               type
               (list 'union-ref (tag t))))))
    (else type)))
```

◇

Macro defined by scraps 14bc, 15ab, 16ab.
Macro referenced in scrap 2d.

Evaluate-number takes a string and attempts to parse it as a C number, returning the value if the parse was successful and #f if not. Currently only integers are handled reliably.

⟨utility functions 16b⟩ ≡

```
(define (evaluate-number s)
  (let ((k (string->list s)))
    (cond ((null? k) #f)
          ((not (char-numeric? (car k))) #f)
          ((char=? (car k) #\0)
           (cond ((null? (cdr k)) 0)
                 ((or (char=? (cadr k) #\x) (char=? (cadr k) #\X))
                  (string->number (list->string (cddr k)) 16))
                 (else
                  (string->number s 8))))
          (else
           (string->number s)))))
```

◇

Macro defined by scraps 14bc, 15ab, 16ab.
Macro referenced in scrap 2d.

5 Discussion

Clearly using a function call to access each field is not very efficient, compared to C. But if the foreign function interface is reasonably efficient it shouldn't be too bad. It's hard to know how efficient it will be in Chez Scheme without some benchmarking, but a call-out to a foreign function does not need to be much more expensive than a call to a procedure in the same language.

6 Future Work on the Chez Back-end

- Handling name clashes, both finding them and working around them (if necessary).
- Dealing with qualifiers on types in general; handling `const` in particular.
- Better generation of C `typedef` casts.
- Proper choice of names for the output files.
- Experiment with policy choices defined in a file: for example, a file could contain a specification of which prototypes to override (like in the `fgets` example), which file names to include or exclude, how to name procedures on the Scheme side, and how to compute the referenced or unreferenced sets. Since we are programming in Scheme here, these decisions could be implemented as code in a file which is loaded at run-time. As an example of what has been thought of but not implemented, read the file `chez-policy.sch` in the current distribution.

7 Support libraries

The file `chez-stdlib.h` defines the prototypes for the C-side standard library procedures.

"chez-stdlib.h" 17a ≡

```
int _ref_int( unsigned _p, int _k );
void _set_int( unsigned _p, int _k, int _v );
unsigned _ref_uint( unsigned _p, int _k );
void _set_uint( unsigned _p, int _k, unsigned _v );
char _ref_char( unsigned _p, int _k );
void _set_char( unsigned _p, int _k, char _v );
double _ref_double( unsigned _p, int _k );
void _set_double( unsigned _p, int _k, double _v );
float _ref_float( unsigned _p, int _k );
void _set_float( unsigned _p, int _k, float _v );
◇
```

The implementation of the library is in `chez-stdlib.c`; it's pretty obvious.

"chez-stdlib.c" 17b ≡

```
#include "chez-stdlib.h"

int _ref_int( unsigned _p, int _k ) { return ((int*)_p)[_k]; }
void _set_int( unsigned _p, int _k, int _v ) { ((int*)_p)[_k] = _v; }

unsigned _ref_uint( unsigned _p, int _k ) { return ((unsigned*)_p)[_k]; }
void _set_uint( unsigned _p, int _k, unsigned _v ) { ((unsigned*)_p)[_k] = _v; }

char _ref_char( unsigned _p, int _k ) { return ((char*)_p)p[_k]; }
void _set_char( unsigned _p, int _k, char _v ) { ((char*)_p)[_k] = _v; }

double _ref_double( unsigned _p, int _k ) { return ((double*)_p)[_k]; }
void _set_double( unsigned _p, int _k, double _v ) { ((double*)_p)[_k] = _v; }
```

```
float _ref_float( unsigned _p, int _k ) { return ((float*)_p)[_k]; }
void _set_float( unsigned _p, int _k, float _v ) { ((float*)_p)[_k] = _v; }
◇
```

Finally, a Scheme file **chez-stdlib.sch** defines the foreign function interfaces to the standard library:

"chez-stdlib.sch" 17c ≡

```
(define _ref_int
  (foreign-function "_ref_int" (unsigned-32 integer-32) integer-32))

(define _set_int
  (foreign-function "_set_int" (unsigned-32 integer-32 integer-32) void))

(define _ref_uint
  (foreign-function "_ref_uint" (unsigned-32 integer-32) integer-32))

(define _set_uint
  (foreign-function "_set_uint" (unsigned-32 integer-32 unsigned-32) void))

(define _ref_char
  (foreign-function "_ref_char" (unsigned-32 integer-32) char))

(define _set_char
  (foreign-function "_set_char" (unsigned-32 integer-32 char) void))

(define _ref_double
  (foreign-function "_ref_double" (unsigned-32 integer-32) double-float))

(define _set_double
  (foreign-function "_set_double" (unsigned-32 integer-32 double-float) void))

(define _ref_float
  (foreign-function "_ref_float" (unsigned-32 integer-32) single-float))

(define _set_float
  (foreign-function "_set_float" (unsigned-32 integer-32 single-float) void))

(define _memcpy
  (let ((memcpy-*-*
          (foreign-function "memcpy" (unsigned-32 unsigned-32 unsigned-32) void))
         (memcpy-string-string
          (foreign-function "memcpy" (string string unsigned-32) void))
         (memcpy-string-*
          (foreign-function "memcpy" (string unsigned-32 unsigned-32) void))
         (memcpy-*-string
          (foreign-function "memcpy" (unsigned-32 string unsigned-32) void)))
    (lambda (a b count)
      (cond ((string? a)
             (cond ((string? b) (memcpy-string-string a b count))
                   ((integer? b) (memcpy-string-* a b count))
                   (else ???)))
            ((integer? a)
             (cond ((string? b) (memcpy-*-string a b count))
                   ((integer? b) (memcpy-*-* a b count))
                   (else ???)))
            (else ???)))))
◇
```

Index of Identifiers

The underlined scrap number shows the defining scrap for the identifier; all other scrap numbers are uses. Also note that many procedures are not defined in this file but in the target-independent part of the back-end, in file `process.sch`.