The Green Card User's Guide

The Green Card Team

This document describes Green Card, a foreign language interface preprocessor for Haskell, version 2.0. It is based on the Green Card paper presented at the '97 Haskell Workshop by Simon Peyton Jones `simonpj@dcs.gla.ac.uk` and Thomas Nordin `nordin@cse.ogi.edu`.

1. Introduction

A foreign-language interface provides a way for software written in a one language to interact with code written in another. Programming languages that lack foreign-language interfaces die a lingering death.

All Haskell implementations known to us do provide facilities for interfacing to external functionality. For instance, the Glasgow Haskell compiler (GHC) provides a `ccall` construct for mixing calls to C functions with standard Haskell code performing I/O. However, the mechanism is low-level and it often requires the programmer to write lots of tedious boilerplate code to convert Haskell values into the representations expected by the external function.

We present here Green Card, a preprocessor for Haskell that tries to simplify the task of defining an interface/binding to foreign functionality.

2. Goals and non-goals

Our goals are limited. We do not set out to solve the problem of language interoperation and foreign-language interfaces in general; rather, we intend to profit from the work of others' in this area. Specifically, we aim to provide a convenient way to call C procedures from Haskell.

The ability to call C from Haskell is an essential foundation. Through it we can access operating system services and mountains of other software libraries. It also provides a basis on which to tackle the problem of writing and accessing *software components* from Haskell.

In the other direction, should we be able to write a Haskell library that a C program can use? In principle this makes sense, but in practice there is little demand for it. The exception is the ability to support some sort of Haskell callbacks, but that is a very limited form of C calling Haskell.

Should we support languages other than C? The trite answer is that pretty much everything available as a library is available as a C library. For other languages the right thing to do is to interface to a language-independent software component architecture, rather than to a raft of specific languages. There's a couple of such architectures to choose from; COM and CORBA. This document, however, describes only the C interface mechanism.

3. Foreign language interfaces are harder than they look

Even after the scope is restricted to designing a foreign-language interface from Haskell to C, the task remains surprisingly tricky. At first, one might think that one could take the C header file describing a C procedure, and generate suitable interface code to make the procedure callable from Haskell.

Alas, there are numerous tiresome details that are simply not expressed by the C procedure prototype in the header file. For example, consider calling a C procedure that opens a file, passing a character string as argument. The C prototype might look like this:

```
int open( char *filename );
```

Our goal is to generate code that implements a Haskell procedure with type

```
open :: String -> IO FileDescriptor
```

- First there is the question of data representation. One has to decide either to alter the Haskell language implementation, so that is string representation is identical to that of C, or to translate the string from one representation to another at run time. This translation is conventionally called *marshalling*.

  Since Haskell is lazy, the second approach is required. (In general, it is tremendously constraining to try to keep common representations between two languages. For example, precisely how are structures laid out C?)

- Next come questions of allocation and lifetime. Where should we put the translated string? In a static piece of storage? (But how large a block should we allocate? Is it safe to re-use the same block on the next call?) Or in Haskell's heap? (But what if the called procedure does something that triggers garbage collection, and the transformed string is moved? Can the called procedure hold on to the string after it returns?) Or in C's `malloc`'ed heap? (But how will it get deallocated? And `malloc` is expensive too.)
- C procedures often accept pointer parameters (such as strings) that can be `NULL`. How is that to be reflected on the host-language side of the interface? For example, if the documentation for `open` told us that it would do something sensible when called with a `NULL` string, we might like the Haskell type for `open` to be

```
open :: Maybe String -> IO FileDescriptor
```

so that we can model `NULL` by `Nothing`.

- The desired return type, `FileDescriptor`, will presumably have a Haskell definition such as this:

  ```
  newtype FileDescriptor = FD Int
  ```

  The file descriptor returned by `open` is just an integer, but Haskell programmers often use `newtype` declarations create new distinct types isomorphic to existing ones. Now the type system will prevent, say, an attempt to add one to a `FileDescriptor`.

  Needless to say, the Haskell result type is not going to be described in the C header file.

- The file-open procedure might fail; sometimes details of the failure are stored in some global variable, `errno`. Somehow this failure and the details of what went wrong must be reflected into Haskell's `IO` monad.
- The `open` procedure causes a side effect, so it is appropriate for its type to be in Haskell's `IO` monad. Some C functions really are functions (that is, they have no side effects), and in this case it makes sense to give them a "pure" Haskell type. For example, the C function `sin` should appear to the Haskell programmer as a function with type

  ```
  sin :: Float -> Float
  ```

- C function prototypes are not explicit about the *mode* of their parameters. Which parameters are `in` parameters, which `out` and which are `in out` - that is, in what direction do data pass via a parameter?

None of these details are mentioned in the C header file. Instead, many of them are in the manual page for the procedure, while others (such as parameter lifetimes) may not even be written down at all.

4. Overview of Green Card

The previous section bodes ill for an automatic system that attempts to take C header files and automatically generate the "right" Haskell functions; C header files simply do not contain enough information.

The rest of this document describes how we approach the problem. The general idea is to start from the *Haskell* type definition for the foreign function, rather than the *C* prototype. The Haskell type contains quite a bit more information; indeed, it is often enough to generate correct interface code. Sometimes, however, it is not, in which case we provide a way for the programmer to express more details of the interface. All of this is embodied in a program called "Green Card".

Green Card is a Haskell pre-processor. It takes a Haskell module as input, and scans it for Green Card directives (which are lines prefixed by "`%`"). It produces a new Haskell module as output, and sometimes a C module as well,

```
+- - - - - +        +- - - - +          +- - - - -+
| Haskell' |        | Green  |          | Haskell |
|  module  + - - ->| Card   +- - - ->| module  |
+- - - - - +        +- -+- - +          +- - - +- +
                        |                       |
                        |                       /
                        |                   +- - - - - +
                        +- - - - - - ->|  C file  |
                                            +- - - - - +
```

Green Card's output depends on the particular Haskell implementation that is going to compile it. For the Glasgow Haskell Compiler (GHC), Green Card generates Haskell code that uses GHC's primitive `ccall/casm` construct to call C. All of the argument marshalling is done in Haskell. For Hugs, Green Card generates a C module to do most of the argument marshalling, while the generated Haskell code uses Hugs' `primitive` construct to access the generated C stubs. The output for nhc13 is somewhere in between, marshalling is in part done by the generated C stubs and in part by the generated Haskell code.

For example, consider the following Haskell module:

```
module M where

import Prelude hiding (sin)
import StdDIS

%fun sin :: Float -> Float

sin2 :: Float -> Float
sin2 x = sin (sin x)
```

Everything is standard Haskell except the `%fun` line, which asks Green Card to generate an interface to a (pure) C function `sin`. After the GHC-targeted version of Green Card processes the file, it looks like this: [1]

```
module M where

import Prelude hiding (sin)
import StdDIS

sin :: Float -> Float
sin f = unsafePerformIO (
          _casm_ "%r = sin(%0)" f >>= \ f' ->
          return f')
```

```
sin2 :: Float -> Float
sin2 x = sin (sin x)
```

The `%fun` line has been expanded to a blob of gruesome boilerplate, while the rest of the module comes through unchanged.

If Hugs is the target, the Haskell source file remains unchanged, but the Hugs variant of Green Card would generate output that implements `sin` using Hugs' `primitive` mechanisms for calling C. For the nhc13 target, Green Card generates something different again. Much of the Green Card implementation is, however, shared between both variants.

An important point here is that the *Haskell type signature* for `sin` remains the same, regardless of the target that Green Card has been instructed to generate code for. Green Card allows you to abstract away from the low-level details of how foreign function calls are implemented by the individual systems.

5. Green Card directives

Green Card pays attention only to Green Card directives, each of which starts with a "`%`" at the beginning of a line. All other lines are passed through to the output Haskell file unchanged.

The syntax of Green Card directives is as follows (the syntax for *dis* is given in Section 9.1):

```
Program  :  Declaration_1
            ...
            Declaration_n     n >= 1

Declaration : Procedure
            | '%const' [Var] [Var1 .. Varn]
            | '%enum'  [Var] Derivings
                     Type  [Var1 .. Varn]
            | '%dis' [Var1] [Var1 .. Varn] '=' dis
            | '%#include' Filename
            | '%prefix' Var
            | '%C' ccode

Procedure   : [Signature] [call] [ccode] [result]

Signature   : '%fun' Var :: Type

Type        : Var
            | Var Type
            | Type '->' Type
            | '(' Type_1, ... , Type_n ')'    n >= 1

Call        : '%call' Dis_1 ... Dis_n

Result      : '%fail' Cexp Cexp [Result]
```

```
             | '%end' Cexp
             | '%result' Dis

Cexp         : anything up until '}' is encountered.
             | Ccode

Ccode        : '%code' Var
             | '%safecode' Var

Filename     : '<' Var '>'
             | '"' Var '"'
```

Green Card understands the following directives:

- `%fun` begins a *procedure specification*, which describes the interface to a single C procedure, see Section 6.
- `%dis` allows the programmer to describe a new *Data Interface Scheme* (DIS). A DIS describes how to translate, or marshall, data from Haskell to C and back again, see Section 9.
- `%const` makes it easy to generate a collection of new Haskell constants derived from C constants. This can be done with `%fun`, but `%const` is much more concise, see Section 8.1.
- `%enum` is similar to `%const`, but supports the mapping of external constants to a corresponding Haskell data type, see Section 8.2.
- `%prefix` makes it easy to remove standard prefixes from the Haskell function name, those are usually not needed since Haskell allows qualified imports, see Section 8.3.
- Procedure specifications can, as we shall see, contain fragments of C. `%#include` tells Green Card to arrange that a specified C header file will be included with the C code in the procedure specifications when the latter is fed to a C compiler, see Section 11.
- "`%C`" allows you to write fragments of C code which sits *outside* any procedure specification. (We shall later see how to include fragments of C code *within* procedures.) The entire line of text following this directive is simply copied verbatim to the generated C module.

A directive can span more than one line, but the continuation lines must each start with a `%` followed by some whitespace. For example:

```
%fun draw :: Int                -- Length in pixels
%            -> Maybe Int       -- Width in pixels
%            -> IO ()
```

Haskell-style comments are permitted in Green Card directives (except, for obvious reasons, "`%C`".)

A general principle we have followed is to define a single, explicit (and hence long-winded) general mechanism, that should deal with just about anything, and then define convenient abbreviations that save the programmer from writing out the general mechanism in many common cases. We have erred on the conservative side in defining such abbreviations; that is, we have only defined an abbreviation where doing without it seemed unreasonably long-winded, and where there seemed to be a systematic way of defining an abbreviation.

6. Procedure specifications

The most common Green Card directive is a procedure specification. It describes the interface to a C procedure. A procedure specification has four parts:

Type signature `%fun` Section 7.

The `%fun` statement starts a new procedure specification, giving the name and Haskell type of the function.

Parameter marshalling: `%call` Section 7.1.

The `%call` statement tells Green Card how to translate the Haskell parameters into their C representations.

The body: `%code, %end` -- Section 7.2.

The `%code` statement gives the body and it can contain arbitrary C code. Sometimes the body consists of a simple procedure call, but it may also include variable declarations, multiple calls, loops, and so on.

The `%end` statement is used to specify C code that should be performed after having performed the code in `%code`.

Result marshalling: `%result`, `%fail` -- Section 7.3

The result-marshalling statements tell Green Card how to translate the result(s) of the call back into Haskell values.

Any of these parts may be omitted except the type signature. If any part is missing, Green Card will fill in a suitable statement based on the type signature given in the `%fun` statement. For example, consider the `sin` procedure specification again:

```
%fun sin :: Float -> Float
```

Green Card fills in the missing statements like this: [2]

```
%fun sin :: Float -> Float
%call (float arg1)
%code res1 = sin(arg1);
%result (float res1)
```

The rules that guide this automatic fill-in are described in Section 8.

A procedure specification can define a procedure with no input parameter, or even a constant (a "procedure" with no input parameters and no side effects). In the following example, `printBang` is an example of the former, while `grey` is an example of the latter: [3]

```
%fun printBang :: IO ()
%code printf( "!" );

%fun grey :: Colour
%code r = GREY;
%result (colour r)
```

All the C variables bound in the `%call` statement or mentioned in the `%result` statement, are declared by Green Card and in scope throughout the body. In the examples above, Green Card would have declared `arg1`, `res1` and `r`.

7. Type signature

The `%fun` statement starts a new procedure specification.

Green Card supports two sorts of C procedures: ones that may cause side effects (including I/O), and ones that are guaranteed to be pure functions. The two are distinguished by their type signatures. Side-effecting functions have the result type `IO t` for some type `t`. If the programmer specifies any result type other than `IO t`, Green Card takes this as a promise that the C function is indeed pure, and will generate code that calls `unsafePerformIO`.

The procedure specification will expand to the definition of a Haskell function, whose name is that given in the `%fun` statement, with two changes: the longest matching prefix specified with a `%prefix` statement is removed from the name and the first letter of the remaining function name is changed to lower case (Section 8.3 elaborates). Haskell requires all function names to start with a lower-case letter (upper case would indicate a data constructor), but when the C procedure name begins with an upper case letter it is convenient to still be able to make use of Green Card's automatic fill-in facilities. For example:

```
%fun OpenWindow :: Int -> IO Window
```

would expand to a Haskell function `openWindow` that is implemented by calling the C procedure `OpenWindow`.

```
%prefix Win32
%fun Win32OpenWindow :: Int -> IO Window
```

would also expand to a Haskell function `openWindow`, but is implemented by calling the C procedure `Win32OpenWindow`.

7.1. Parameter marshalling

The `%call` statement tells Green Card how to translate the Haskell parameters into C values. Its syntax is designed to look rather like Haskell pattern matching, and consists of a sequence of zero or more Data Interface Schemes (DISs), one for each (curried) argument in the type signature. For example:

```
%fun foo :: Float -> (Int,Int) -> String -> IO ()
%call (float x) (int y, int z) (string s)
...
```

This `%call` statement binds the C variables `x`, `y`, `z`, and `s`, in a similar way that Haskell's pattern-matching binds variables to (parts of) a function's arguments. These bindings are in scope throughout the body and result-marshalling statements.

In the `%call` statement, "`float`", "`int`", and "`string`" are the names of the DISs that are used to translate between Haskell and C. The names of these DISs are deliberately chosen to be the same as the corresponding Haskell types (apart from changing the initial letter to lower case) so that in many cases, including `foo` above, Green Card can generate the `%call` line by itself (Section 8).

In fact there is a fourth DIS hiding in this example, the `(_,_)` pairing DIS. DISs are discussed in detail in Section 9.

7.2. The body

The body consists of arbitrary C code, beginning with `%code`. The reason for allowing arbitrary C is that C procedures sometimes have complicated interfaces. They may return results through parameters passed by address, deposit error codes in global variables, require `#include`'d constants to be passed as parameters, and so on. The body of a Green Card procedure specification allows the programmer to say exactly how to call the procedure, in its native language.

The C code starts a block, and may thus start with declarations that create local variables. For example:

```
%code int x, y;
%     x = foo( &y, GREY );
```

Here, `x` and `y` are declared as local variables. The local C variables declared at the start of the block scope over the rest of the body *and* the result-marshalling statements.

The C code may also mention constants from C header files, such as `GREY` above. Green Card's `%#include` directive tells it which header files to include (Section 11).

> **GHC specific:** The GHC backend makes a distinction between safe and unsafe external calls: If the external call will cause a garbage collection, you have to call it safely. (A good example of where this is likely to occur is when the external call invokes a Haskell callback.) Green Card supports safe calls in a couple of ways:

- The command-line option `--safe-code` will cause the generated code to call all the `%code` snippets safely.
- The `%safecode` declaration is identical to `%code` except that when generating code for GHC, the code snippet will be called safely. Performing a safe call involves saving away all abstract machine state before performing the call, so it is recommended to use `%safecode` over `--safe-code`, since it offers more fine grained control.

7.3. Result marshalling

Functions return their results using a `%result` statement. Side-effecting functions, ones whose result type is `IO t`, can also use `%fail` to specify the failure value.

7.4. Pure functions

The `%result` statement takes a single DIS that describes how to translate one or more C values back into a single Haskell value. For example:

```
%fun sin :: Float -> Float
%call (float x)
%code ans = sin(x);
%result (float ans)
```

As in the case of the `%call` statement, the "`float`" in the `%result` statement is the name of a DIS, chosen as before to coincide with the name of the type. A single DIS, "`float`", is used to denote both the translation from Haskell to C and that from C to Haskell, just as a data constructor can be used both to construct a value and to take one apart (in pattern matching).

All the C variables bound in the `%call` statement, and all those bound in declarations at the start of the body, scope over all the result-marshalling statements (i.e. `%result` and `%fail`).

7.5. Arbitrary C results

In a result-marshalling statement an almost arbitrary C expression, enclosed in braces, can be used in place of a C variable name. The above example could be written more briefly like this: [4]

```
%fun sin :: Float -> Float
%call (float x)
%result (float {sin(x)})
```

The C expression can neither have assignments nor nested braces as that could give rise to syntactic ambiguity.

7.6. Side effecting functions

A side effecting function returns a result of type `IO t` for some type `t`. The `IO` monad supports exceptions, so Green Card allows them to be raised.

The result-marshalling statements for a side-effecting call consists of zero or more `%fail` statements, each of which conditionally raise an exception in the `IO` monad, followed by a single `%result` statement that returns successfully in the `IO` monad.

Just as in Section 7.3, the `%result` statement gives a single DIS that describes how to construct the result Haskell value, following successful completion of a side-effecting operation. For example:

```
%fun windowSize :: Window -> IO (Int,Int)
%call (window w)
%code struct WindowInfo wi;
%     GetWindowInfo( w, &wi );
%result (int {wi.x}, int {wi.y})
```

Here, a pairing DIS is used, with two `int` DISs inside it. The arguments to the `int` DISs are C record selections, enclosed in braces; they extract the relevant information from the `WindowInfo` structure that was filled in by the `GetWindowInfo` call. [5]

The `%fail` statement has two fields, each of which is either a C variable, or a C expression enclosed in braces. The first field is a boolean-valued expression that indicates when the call should fail; the second is a `(char *)` value that indicates what sort of failure occurred. If the boolean is true (i.e. non zero) then the call fails with a `userError` in the `IO` monad containing the specified string.

For example:

```
%fun fopen :: String -> IO FileHandle
%call (string s)
%code f = fopen( s );
%fail {f == NULL} {errstring(errno)}
%result (fileHandle f)
```

The assumption here is that `fopen` puts its error code in the global variable `errno`, and `errstring` converts that error number to a string.

`UserErrors` can be caught with `catch`, but exactly which error occurred must be encoded in the string, and parsed by the error-handling code. This is rather slow, but errors are meant to be exceptional.

8. Automatic fill-in

Any or all of the parameter-marshalling, body, and result-marshalling statements may be omitted. If they are omitted, Green Card will *fill in* plausible statements instead, guided by the function's type signature. The rules by which Green Card does this filling in are as follows:

- A missing `%call` statement is filled in with a DIS for each curried argument. Each DIS is constructed from the corresponding argument type as follows:
  - A tuple argument type generates a tuple DIS, with the same algorithm applied to the components.
  - All other types generate a DIS function application, Section 9.1. The DIS function name is derived from the type of the corresponding argument, except that the first letter of the type is changed to lower case. The DIS function is applied to as many argument variables as required by the arity of the DIS function.
  - The automatically-generated argument variables are named left-to-right as `arg1`, `arg2`, `arg3`, and so on.
- If the body is missing, Green Card fills in a body of the form:

  ```
  %code r = f ( a_1, ... , a_n );
  ```

  where

  - *f* is the function name given in the type signature.
  - *a_1 ... a_n* are the argument names extracted from the `%call` statement.
  - *r* is the variable name for the variable used in the `%result` statement. (There should only be one such variable if the body is automatically filled in).
- A missing `%result` statement is filled in by a `%result` with a DIS constructed from the result type in the same way as for a `%call`. The result variables are named `res1`, `res2`, `res3`, and so on.
- Green Card never fills in `%fail` statements.

8.1. Constants

Some C header files define a large number of constants of a particular type. The `%const` statement provides a convenient abbreviation to allow these constants to be imported into Haskell. For example:

```
%const PosixError [EACCES, ENOENT]
```

This statement is equivalent to the following `%fun` statements:

```
%fun EACCES :: PosixError
%fun ENOENT :: PosixError
```

After the automatic fill-in has taken place we would obtain:

```
  %fun EACCES :: PosixError
  %result (posixError { EACCES })
```

```
  %fun ENOENT :: PosixError
  %result (posixError { ENOENT })
```

Each constant is made available as a Haskell value of the specified type, converted into Haskell by the DIS function for that type. (It is up to the programmer to write a `%dis` definition for the function -- see Section 9.2.)

There's a variant way of declaring constant within the '`%const`' directive: you may specify the Haskell name that the C constant name maps to:

```
%const PosixError [
%   errAccess = {EACCES},
%   errNoEnt  = {ENOENT}
% ]
```

8.2. Enumerations

The `%const` declarations allows you to map external names/constants onto Haskell `Int`s. A more typesafe mapping of constants is provided by the `%enum` declaration, which maps a set of constants to a Haskell type. [6]

For example, here's `PosixError` expressed using `%enum`:

```
%enum PosixError Int [EACCES, ENOENT]
```

This creates the following data type plus marshalling functions:

```
data PosixError = EACCES | ENOENT
marshall_PosixError :: PosixError -> Int
marshall_PosixError = ...
unmarshall_PosixError :: Int -> PosixError
unmarshall_PosixError = ...
```

Additionally, it also implicitly creates a DIS:

```
%dis posixError x = int x
```

In the event you want the Haskell compiler to automatically generate instances for the enumeration type it generates, `%enum` supports this too:

```
%enum PosixError (Eq,Show) Int [EACCES, ENOENT]
```

will generate a data type with the derived instances `Eq` and `Show`:

```
data PosixError = EACCES | ENOENT deriving (Eq, Show)
```

8.3. Prefixes

In C, some libraries give all their exported names the same prefix, thereby minimizing the impact on the shared namespace. In Haskell we use qualified imports to achieve the same result. To simplify the conversion of C style namespace management to Haskell the `%prefix` statement specifies which prefixes to remove from the Haskell function names.

```
module OpenGL where

%prefix OpenGL
%prefix gl

%fun OpenGLInit :: Int -> IO Window
%fun glSphere :: Coord -> Int -> IO Object
```

This would define the two functions `init` and `sphere` which would be implemented by calling `OpenGLInit` and `glSphere` respectively.

8.4. Including arbitrary C code

It is sometimes useful to be able to write arbitrary lines of C code outside any procedure specification, for instance to include a helper C function or define a C structure. The "`%C`" statement is provided for this purpose:

```
%C typedef struct _point  {
%    int x;
%    int y;
%  } point;
```

The C code is added directly to the generated C file.

9. Data Interface Schemes

A *Data Interface Scheme*, or DIS, tells Green Card how to translate from a Haskell data type to a C data type, and vice versa.

9.1. Forms of DISs

The syntax of DISs is as follows,

```
Dis : DisFun [ arg_1 ... arg_n ]         Application
    | Cons   [ arg_1 ... arg_n ]         Constructor, (n >= 0)
    |  Cons  '{' field_1 = dis_1 ,
```

```
                 ... ,
                 field_n = dis_n '}'     Record,      (n >= 1)
    | < Var / Var >
             [ arg_1, ... , arg_n]       User defined
                                         marshalling, (n >= 0)
    | 'declare' [cexp Var] 'in' [Dis]
    | Adis

Adis : '(' Dis ')'
     | TypeCast Cexp                Result only
     | TypeCast Var
     | Var                          Bound by '%dis'
     | '(' [ Dis_1, ... , Dis_n] ')'  Tuple n >= 1

Arg  : Adis
     | Cexp
     | Var

DisFun   : Var

TypeCast : Cexp                         C Expression

Var      : Var                          Initial letter lower case
```

It is designed to be similar to the syntax of Haskell patterns. A DIS takes one of the following forms:

The application of a DIS macro to zero or more arguments.

Like Haskell functions, a DIS macro starts with a lower-case letter. DIS macros are described in Section 9.2. Standard DIS functions include `int`, `float`, `double`; the full set is given in Section 10. For example:

```
%fun foo :: This -> Int -> That
%call (this x y) (int z)
%code r = c_foo( x, y, z );
%result (that r)
```

In this example `this` and `that` are DIS functions defined elsewhere.

The application of a Haskell data constructor to zero or more DISs.

For example:

```
newtype Age = Age Int
%fun foo :: (Age,Age) -> Age
%call (Age (int x), Age (int y))
%code r = foo(x,y);
%result (Age (int r))
```

As the `%call` line of this example illustrates, tuples are understood as data constructors, including their special syntax. The labelled fields syntax is also supported, i.e.

```
data Point = Point { px,py::Int }

%fun foo :: Point -> Point
%call (Point { px = int x, py = int y })
...
```

The use of records is also the reason for the restriction that simple C expressions can't contain assignment. Without this restriction examples like this would be ambiguous:

```
%result Foo { a = bar x, b = bar y }
```

Green Card does not attempt to perform type inference; it simply assumes that any DIS starting with an upper case letter is a data constructor, and that the number of argument DISs matches the arity of the constructor.

The application of a user function to one or more DISs

This form allows you to do user defined marshalling, using a pair Haskell functions. Since DISs are used to either pack/marshall a Haskell value into a form that can be passed to C and unpack values that come back, a pair of Haskell functions is required. For example:

```
data Nat = Zero | Succ Nat
fromNat :: Nat -> Int
toNat   :: Int -> Nat

%fun square :: T -> T
%call (< fromNat / toNat > (int x))
%code r = square(x);
%result (< fromNat / toNat > (int r))
```

Here the function `fromNat` is applied to `square`'s argument, converting it to an integer before it crosses the fence into C. Likewise, the result coming back from C is converted back to type `Nat` by the function `toNat`.

The user functions can have any name at all: in fact, the `<../..>` syntax simply encloses two fragments of arbitrary Haskell to be applied to the succeeding arguments. One may specify a partially applied function, or anything else (excluding the use of the / and > symbols - so lambda abstractions are unfortunately not possible.) The user-defined DIS may of course also take more than one parameter. For example:

```
data Point = P Dist Vector
polarToCart :: Polar     -> (Int,Int)
cartToPolar :: (Int,Int) -> Polar

%fun mirror :: Polar -> Polar
%call (< polarToCart / cartToPolar > (int x) (int y))
%code y = -y;
%     x = -x;
```

```
%result (< polarToCart / cartToPolar > (int x) (int y))
```

Notice that all the example marshalling functions have so far been pure functions, e.g., `fromNat` has type `Nat -> Int` rather than `T -> IO Int`.) Sometimes you need to write a marshalling function that is internally stateful. When you do, you'll need to inform Green Card of this, so that it can generate code that invokes the marshalling functions correctly. For example:

```
marshallString   :: String -> IO Addr
unmarshallString :: Addr   -> IO

%fun setWindowTitle :: String -> IO ()
%call (<< marshallString / unmarshallString >> str)
```

i.e., stateful marshalling functions are enclosed by double angle brackets.

A C type cast

Occasionally one wishes to declare and use a C variable at a type which slightly differs form the type produced by a standard DIS, although it shares the same machine representation. The `declare {cexp} var in dis` form can be used to do the necessary type conversion in C. Examples:

```
%fun foo :: Int -> IO ()
%call (declare {unsigned int} x in (int x))

data T = MkT Int
%fun faz :: T -> IO ()
%call (declare {c_t} x in MkT (int x))
```

The application of a base DIS to exactly one variable

This is the primitive form of a DIS -- the way all values actually get passed across the Haskell - C boundary. Base DISs denote a fixed set of primitive types known to both C and Haskell, such as `int` and `Int` respectively, and consist of the Haskell type name prefixed by `%%` (e.g., `%%Int`.) Because the exact set of base DISs may vary slightly between compilers, it is recommended that programmers use the standard DIS macros listed in Section 10 insead. The base form is noted here primarily for completeness.

9.2. DIS macros

It would be unbearably tedious to have to write out complete DISs in every procedure specification, so Green Card supports *DIS functions* in much the same way that Haskell provides functions. (The big difference is that DIS functions can be used in "patterns" -- such as `%call` statements -- whereas Haskell functions cannot.)

DIS macros allow the programmer to define abbreviations for commonly-occurring DISs. For example:

```
newtype This = MkThis Int (Float, Float)
%dis this x y z = MkThis (int x) (float y, float z)
```

Along with the `newtype` declaration the programmer can write a `%dis` function definition that defines the DIS function `this` in the obvious manner.

DIS macros are simply expanded out by Green Card before it generates code. So for example, if we write:

```
%fun f :: This -> This
%call (this p q r)
...
```

Green Card will expand the call to `this`:

```
%fun f :: This -> This
%call (MkThis (int p) (float q, float r))
...
```

(In fact, `int` and `float` are also DIS macros defined in Green Card's standard DIS prelude, so the `%call` line is further expanded to: [7]

```
%fun f :: This -> This
%call (MkThis (I# ({int} p)) (F# ({float} q), F# ({float} r)))
...
```

The fully expanded calls describe the marshalling code in full detail; you can see why it would be inconvenient to write them out literally on each occasion!)

Notice that DIS macros are automatically bidirectional; that is, they can be used to convert Haskell values to C *and vice versa*. For example, we can write:

```
%fun f :: This -> This
%call (MkThis (int p) (float q, float r))
%code int a, b, c;
%     f( p, q, r, &a, &b, &c);
%result (this a b c)
```

The form of DIS macro definitions, given in Section 9.1, is very simple. The formal parameters can only be variables (not patterns), and the right hand side is simply another DIS. Only first-order DIS macros are permitted.

9.3. User-defined DISs

Sometimes Green Card's primitive DISs (data constructors) are insufficiently expressive. For recursive types, such as lists, it is obviously no good to write a single data constructor.

Green Card therefore provides a "trap door" to allow a sufficiently brave programmer to write his or her own marshalling functions. For example:

```
data T = Zero | Succ T

%fun square :: T -> T
%call (t (int x))
%code r = square( x );
%result (t (int r))
```

Use of `t` requires that the programmer define two ordinary Haskell functions, `marshall_t` to convert from Haskell to C, and `unmarshall_t` to convert in the other direction. In this example, these functions would have the types:

```
marshall_t   :: T -> Int
unmarshall_t :: Int -> T
```

The functions must have precisely these names: "`marshall_`" followed by the name of the DIS, and similarly for unmarshalling. Notice that these marshalling functions have pure types (e.g. `marshall_t` has type `T -> Int` rather than `T -> IO Int`). Sometimes one wants to write a marsalling function that is internally stateful. For example, it might pack a `String` into a `ByteArray`, by allocating a `MutableByteArray` and filling it in with the characters one at a time. This can be done using `runST`, or even `unsafePerformIO`. (These are all GHC-centric comments; so far as Green Card is concerned it is simply up to the programmer to supply suitably-typed marshalling functions.)

Green Card distinguishes user-defined DISs from DIS macros by omission: if there is a DIS macro definition for a DIS function `f` then Green Card treats `f` as a macro, otherwise it assumes `f` is a user-defined DIS and generates calls to `marshall_t` and/or `unmarshall_t`.

9.4. Marshalling complex structures

The full power of DIS macros becomes apparent when mapping between a structured Haskell type a C `struct`. For example, to interface the Haskell `ColourPoint` type with the outside world:

```
data ColourPoint = CP Int Int Colour
data Colour = Red | Green | Blue | ... deriving ( Enum )
```

for which we want to map it onto the following C structure:

```
typedef struct CPoint {
   int x;
   int y;
```

```
   enum colour c;
} CPoint;
```

It requires just two DIS macros to capture the mapping between the two:

```
%dis colourPoint cp =
%   declare {CPoint} cp in
%   CP (int {%cp.x}) (int {%cp.y}) (colour {%cp.c})

%dis colour c =
%   declare {enum colour} c in
%   < fromEnum / toEnum > (int c)
```

Using these, it is then very easy to implement the required interfaces to foreign functions that manipulate coloured points:

```
%fun translate :: Int -> Int -> ColourPoint -> IO ColourPoint
%call (int xrel) (int yrel) (colourPoint p)
%code p.x += xrel;
%     p.y += yrel;
%     render(&p);
%result (colourPoint {p})
```

Note that in this example, the return value is actually the same structure as the argument value (destructively updated.) It is for this reason that the `p` on the `%result` line is quoted as a C literal - this prevents the `declare` clause of the DIS macro from generating a second (overlapping) declaration of the variable in C.

9.5. Semantics of DISs

How does Green Card use these DISs to convert between Haskell values and C values? We give an informal algorithm here, although most programmers should hopefully be able to manage without knowing the details.

To convert from Haskell values to C values, guided by a DIS, Green Card does the following:

- First, Green Card rewrites all DIS function applications, replacing left hand side by right hand side.
- Next, Green Card works from outside in, as follows:
  - For a data constructor DIS (in either positional or record form), Green Card generates a Haskell case statement to take the value apart.
  - For a user-defined DIS, Green Card calls the DIS's `marshall` function.
  - For a type-cast-with-variable DIS, Green Card does no translation.

Much the same happens in the other direction, except that Green Card calls the `unmarshall` function in the user-defined DIS case.

10. Standard DISs

The following table gives the DIS functions that Green Card provides as a "standard prelude".

```
DIS            Haskell type  C type           Comments
==============================================================

int x          Int           int
char c         Char          char
float f        Float         float
double d       Double        double
bool b         Bool          int               0 for False, 1 for True
addr a         Addr          (void *)          An (immovable) external address

string s       String        (char *)          Persistence not required
                                               in either direction.

foreign x f    ForeignObj    (void *x),
                             ((void (*)())f)   f is the free routine; it
                                               takes one parameter, namely
                                               x, the thing to be freed.

stable x       any           unsigned int      Makes it possible to pass a
                                               Haskell pointer to C, and
                                               perhaps get it back later,
                                               without breaking the
                                               garbage collector.

maybe dis      Maybe dis     type of dis       Converts to and from
                                               Maybe's, with 0 as 'Nothing'

maybeT he d    Maybe dis     type of dis       Converts to and from 'Maybe's
```

10.1. Haskell extensions

Several of the standard DISs involve types that go beyond standard Haskell:

- `Addr` is a type large enough to contain a machine address. The Haskell garbage collector treats it as a non-pointer, however.
- `ForeignObj` is a type designed to contain a reference to a foreign resource of some kind: a `malloc`'d structure, a file descriptor, an window system graphical context, or some such. The size

of this reference is assumed to be that of a machine address. When the Haskell garbage collector decides that a value of type `ForeignObj` is unreachable, it calls the object's finalisation routine, which was given as an address in the argument of the DIS. The finalisation routine is passed the object reference as its only argument.

- The `stable` DIS maps a value of any type onto a C `unsigned int`. The `unsigned int` is actually an index into the *stable pointer table*, which is treated as a source of roots by the garbage collector. Thus the C procedure can effectively get a reference into the Haskell heap. When `stable` is used to map from C to Haskell, the process is reversed.

10.2. Maybe

Almost all DISs work on single-constructor data types. It is much less obvious how to translate values of multi-constructor data types to and from C. Nevertheless, Green Card does deal in an *ad hoc* fashion with the `Maybe` type, because it seems so important. The syntax for the `maybeT` DIS is:

```
maybeT hexp dis
```

where `dis` is any DIS, and `hexp` is a Haskell expression which you want to map to the `Nothing` value.

In the following example, the function `foo` takes an argument of type `Maybe Int`. If the argument value is `Nothing` it will bind `x` to `0`; if it is `Just a` it will bind `x` to the value of `a`. The return value will be `Just r` unless `r == -1` in which case it will be `Nothing`.

```
%fun foo :: Maybe Int -> Maybe Int
%call (maybeT { 0 } (int x))
%code r = foo(x);
%result (maybeT { -1 } (int r))
```

There is also a `maybe` DIS wich just takes the DIS and defaults to `0` as the `Nothing` value.

11. Imports

Green Card "connects" with code in other modules in two ways:

- Green Card reads the source code of any modules directly imported by the Haskell module being processed. It extracts `%dis` function definitions (only) from these modules. This provides an easy mechanism for Green Card to import DIS functions defined elsewhere.
- It is often important to arrange that a C header file is `#include`d when the C code fragments in Green Card directives is compiled. The `%#include` directive performs this delayed `#include`. The syntax is exactly that of a C `#include` apart from the initial `%`.

12. Invoking Green Card

The general syntax for invoking Green Card is:

```
green-card [options] [filename]
```

Green Card reads from standard input if no filename is given. The options can be any of those:

`-?, -h, --help`

Print a usage message listing all available options, then exit successfully.

`-V, --version`

Print the version number, then exit successfully.

`-v, --verbose`

Print more information while processing the input.

`-d, --debug`

Output extra debug information.

`-i <directories>, --include-dir=<directories>`

Search the directories named in the colon (`:`) separated list for imported files. The directories will be searched in a left to right order.

`-s <sufs>, --suffix=<sufs>`

When searching, use colon separated list of source file suffixes (standard: no suffix or .gc)

`-t <target>, --target=<target>`

Generate Haskell code for a particular system. Targets currently supported: `ghc`, `hugs`, `nhc`, and `ffi`. You're encouraged to use the `ffi` target, since the Haskell output it generates can be used with all Haskell systems that support the common FFI (as of June 2001, all of them).

`-S, --safe-code`

Generates code that can use callbacks to Haskell (GHC only.) This makes the generated code slower (but safer!). See Section 7.2 for details.

`-n, --no-inline`

Put C code in a separate file (GHC target only.)

`-4, --haskell1.4`

Generate Haskell 1.4 compatible instead of Haskell 98.

`-m <scheme>, --name-mangling-scheme=<scheme>`

Control how you mangle external names into Haskell ones. Schemes currently supported: `std` and `classic`. The default is `std` mangling, which will just change the casing of the first character:

```
A_STD_EXAMPLE
   ==>  a_STD_EXAMPLE  (variable case)
   ==>  A_STD_EXAMPLE  (datacon  case)
```

The `classic` name mangling scheme removes underscores and upper-cases each word, i.e.:

```
A_CLASSIC_EXAMPLE
   ==>  aClassicExample  (variable case)
   ==>  AClassicExample  (datacon  case)
```

`--stub-debug`

Include debugging code in generated code.

`--dll-name`

Generate "foreign imports" to dynamic library DLL (FFI backend only).

`--callconv`

Use specific calling convention (`stdcall` or `ccall`).

12.1. Green Card output

Green Card will create one or more output files when running. The following options control where the output goes:

`-o <file>, --output=<file>`

where to put generated Haskell file.

`-c <file>, --output-c=<file>`

where to put generated C file (if any header files are generated, they'll go in `(dropSuffix file ++ ".h")`.

`-p <pre>, --output-prefix=<pre>`

Use <prefix> as stem when generating output filenames that haven't been specificed directly, i.e., using `-p Right`, the Haskell and C output will be output in `Right.hs` and `Right.c`, respectively.

If none of the above output options are used, the name of the output files are constructed from the name of the input file. The filename suffix is dropped and the appropriate suffix is appended, e.g., running Green Card on `foo.gc` will create a Haskell file called `foo.hs`.

12.2. Debugging Green Card output

> **ToDo:** *add more debugging tips.*

When working with GHC, a commonly occurring problem for Green Card generated code is that of functions being called unsafely when they should have been called safely. The manifestation of this is as an almighty crash. To help debug this, Green Card supports the option `--stub-debug` which will emit debug code into the generated code. For instance, for the following procedure specification:

```
%fun Foo :: Int -> IO Int
%code Foo(arg1)
```

It will generate something like this:

```
foo :: Int -> IO Int
foo arg1 =
  _casm_ ``do {int arg1;int res1;
                arg1 = %0;
                __current_fun__ = "foo";
                do { Foo(arg1);
                    %r = (int)(res1);} while(0);} while(0);'' arg1
  >>= \  res1  ->
  (return (res1))
```

On top of the normal boilerplate gubbins, the variable `__current_fun__` is assigned (Green Card doesn't define this variable, so you'll have to define it yourself.) Using `__current_fun__`, it is possible to catch unsafe calls that really should have been safe. Here's how a Win32 library catches this condition in the (C wrapper) for its Haskell callback:

```
char* __current_fun__ = NULL;

...
#if defined(DEBUG)
   extern int inCCallGC;
   if (inCCallGC == 0) {
      fprintf(stderr,
              "Warning: unsafe ccall %s invokes Haskell callback\n",
              __current_fun__);
   }
#endif
```

The variable `inCCallGC` is defined by the GHC RTS, and its value is equal to the number of safe calls currently executing. (PS: The above code doesn't catch all erroneous cases.)

13. Related Work

A Portable C Interface for Standard ML of New Jersey

> by Lorenz Huelsbergen. Describes the implementation of a general interface to C for SML/NJ, including the ability to expose ML closures as C function pointers.

Simplified Wrapper and Interface Generator (SWIG)

> SWIG generate interfaces from (extended) ANSI C/C++ function and variable declarations. It can generate output for Tcl/Tk, Python, Perl5, Perl4 and Guile-iii. More information available via the SWIG home page (http://www.cs.utah.edu/~beazley/SWIG/).

Foreign Function Interface GENerator (FFIGEN)

> FFIGEN is a tool that parses C header files and presents an intermediate data representation suitable for writing backends. More information available via the FFIGEN home page. (http://www.ccs.neu.edu/home/lth/ffigen/)

Header2Scheme

> Header2Scheme is a program which reads C++ header files and compiles them into C++ code. This code implements the back end for a Scheme interface to the classes defined by these header files. See . the Header2Scheme home page (http://www-white.media.mit.edu/~kbrussel/Header2Scheme/) for more information.

14. Alternative design choices and avenues for improvement

Here we summarise aspects of Green Card that are less than ideal, and indicate possible improvements.

Automatic DIS generation.

> Pretty much every `newtype` or single-constructor declaration that is involved in a foreign language call needs a corresponding `%dis` definition. Maybe this `%dis` definition should be automated. On the other hand, there are many fewer data types than procedures, so perhaps it isn't too big a burden to define a `%dis` for each.
>
> Alternatively, go the other way and have Green Card generate a Haskell type from a `%dis` definition, along the lines of what `%enum` already does.

Error handling.

The error handling provided by `%fail` is fairly rudimentary. It isn't obvious how to improve it in a systematic manner though.

15. Using Green Card

So, you probably wonder, how can I actually make use of GreenCard and its output with the Haskell system you work with? This section outlines the series of steps you need to follow, concentrating for the moment on the use of GreenCard together with either Hugs or GHC.

15.1. Running Green Card

If you downloaded GreenCard as part of a binary distribution, you should have a binary that you can invoke like other command-line tools, i.e., assuming you have written the following spec:

```
module GCTest where

import StdDIS

%fun my_sin :: Double -> Double
%code res1=sin(arg1);
```

you can generate Hugs-compatible Haskell stubs as follows:

```
  green-card -c GCTest.gc -o GCTest.hs --target=hugs \
     -i/directory/where/you/put/StdDIS
```

and GHC-specific ones with

```
  green-card -c GCTest.gc -o GCTest.hs --target=ghc \
     -i/directory/where/you/put/StdDIS
```

In the case you're working on a Win32 platform and downloaded the GreenCard installer, the `-i...` option isn't required.

15.2. Compiling Green Card stubs

Having generated some Haskell stubs, and in the case of Hugs, some C code too, compiling these is the natural next step. If you're using GHC, the following should do:

```
 ghc -c GCTest.hs -o GCTest.o -fglasgow-exts
     -i/directory/where/you/put/StdDIS
```

Upon completion, it should have produced the object file, GCTest.o, which can then be used and linked into your application.

The situation is a little bit different on the Hugs side. Here, you'll need to build a dynamically loadable extension module, which performs the calls to the external library you've written GreenCard stubs for. When Hugs loads in the Green Card generated Haskell stubs, it also looks for and hoist in the extension module at the same time.

The C code that goes into the extension module is naturally generated by Green Card, `GCTest.c` in our example. The details of how to compile this into an extension module differs between OSes; consult your Hugs documentation for exact guidelines (if any...). Here's how to compile it under an ELF-based OS (e.g., modern versions of Linux):

```
gcc -fPIC -shared -o GCTest.so GCTest.c -I/usr/local/hugs/src
```

Notice that the command line needs to include the path to where you got your local copy of the `GreenCard.h` header file. It comes with both Hugs source and binary distributions (in the `src` subdirectory).

On the Win32 side, here's how to compile up the extension module using Visual C++,

```
  cl /nologo /LD /MD -o GCTest.dll GCTest.c -I/hugs/src
```

or, using the `mingw32` backend of `gcc` that's supplied by the Cygwin toolchain distribution:

```
gcc -mno-cygwin -o GCTest.o GCTest.c -I/hugs/src
dllwrap -mno-cygwin --target=i386-mingw32 --export-all \
  --def HugsExt.def GCTest.o -o GCTest.dll
```

`HugsExt.def` is the module definition file, and contains the following

```
EXPORTS
initModule
```

Of course, if your stubs interface to some external library, you'll no doubt need to augment with at least extra `-lwhatever` options to satisfy the demands of the linker.

Notes

1. Only GHC aficionados will understand this code. The whole point of Green Card is that Joe Programmer should not have to learn how to write this stuff.
2. The details of the filled-in statements will make more sense after reading the rest of Section 6
3. When there are no parameters, the `%call` line can be omitted. The second example can also be shortened by writing a C expression in the `%result` statement; see Section 7.3.
4. It can be written more briefly still by using automatic fill-in, Section 8.

5. This example also shows one way to interface to C procedures that manipulate structures.
6. The support for `%enum` is closely based on a design suggested by Sven Panne.
7. The expansion shown here assumes the GHC version of the standard DIS' are used.