Interface Chicken Scheme $\leftrightarrow$ C

v.0.0002 (rascunho inicial)

Jerônimo C. Pellegrini

1 FFI

É muito comum que compiladores e interpretadores ofereçam ao usuário algum mecanismo que permita integrar em um mesmo programa rotinas escritas em outras linguagens. Desta forma é possível construir programas em Scheme, por exemplo, que usem funções C ou C++ (e como normalmente também se pode realizar chamadas de C para outras linguagens, fica clara a possibilidade de usar diversas linguagens de programação em um só programa.

Na discussão que segue o cenário é de um programa Scheme usando funções C, mas deve-se entender que os conceitos são válidos para outras linguagens também.

1.1 Callbacks

Quando um programa Scheme chama uma função C, e esta função por sua vez chama outra função Scheme, esta última chamada tem o nome de *callback*[1].

O problema com *callbacks* é que as diferentes linguagens podem tratar a pilha de maneiras muito diferentes, e isso deve ser levado em consideração por quem implementa o mecanismo de FFI.

2 FFI no Chicken Scheme

O compilador do Chicken Scheme traduz programas Scheme para C, e em seguida chama um compilador C comum para gerar código binário. É natural então que haja maneiras de incluir, dentro de código Scheme, chamadas a funções C (e mesmo trechos de código C).

O mecanismo básico do Chicken Scheme para interface com C só funciona, portanto, em código compilado (e não no interpretador). Para usar a FFI a partir do interpretador há o módulo `lazy-ffi`, descrito na seção 5.

1 Há uma analogia a uma prática para realização de chamadas telefônicas, quando uma pessoa liga para outra e pede que a outra "ligue de volta" – o nome dado em Inglês a esta prática é *callback*, e em certa época havia empresas que ofereciam um serviço especializado de *callback* para baratear ligações (como uma ligação do país A para o país B é mais cara que do país B para o país A, a empresa oferecia a possibilidade do cidadão do país A pedir um callback do país B, mas de forma transparente para seu interlocutor.

A maneira mais simples de chamar funções C em programas Chicken Scheme é usando a macro `foreign-code`:

```
(foreign-code "puts (\"Ola'!\");"
              "puts (\"Bom dia!\");")
```

Note as aspas escapadas dentro da string e o ponto-e-vírgula após o comando C.

3 Chamando funções C

Embora `foreign-code` seja útil para executar código C simples, é inconveniente quando se quer chamar uma função C como se fosse um procedimento Scheme. Para situações como esta, `foreign-lambda` define um procedimento Scheme tão similar quanto possível a uma dada função C:

```
(define seno (foreign-lambda double sin double))
(display (seno 3.1415926536))
```

Os argumentos para `foreign-lambda` são o tipo do valor retornado pela função C seguido dos tipos dos parâmetros.

É claro que se a função será chamada uma única vez, pode-se dispensar o `define`:

```
(display ((foreign-lambda double sin double) 3.1415926536))
```

No entanto, `foreign-lambda` só funciona para funções C que não fazem *callback*.

A chamada de uma única função C nem sempre é conveniente; algumas vezes é necessário inserir no programa Scheme um pequeno trecho de código. Isso pode ser feito com a macro `foreign-lambda*`:

```
(define distancia
  (foreign-lambda* ()
                   ((double x)
                    (double y))
                   "double res;"
                   "res = sqrt(x*x+y*y);"
                   "C_return(res);"))
```

Note que o trecho em C retorna um valor, mas usa `C_return` (e não o `return` comum de C). Isso é necessário por causa da maneira como o Chicken Scheme compila programas (a função `C_return` precisa fazer uma "limpeza administrativa" antes de retornar o valor).

3.1 Variáveis externas

Há diferença entre *definir* uma variável (a definição determina quando, como, e que tipo de variável é criada, e só pode existir em um único lugar) e *declarar* uma variável (a declaração apenas informa ao compilador ou interpretador que a variável existe – e possivelmente informa também seu tipo).

Ao misturar Scheme e C, é importante manter em mente que cada variável deve ser definida em um único ponto, que pode ser na parte Scheme ou na parte C do programa

(mas não em ambas) – mas quando definida em um dos lados, deve ser declarada no outro para que seja visível ali.

```
(define-foreign-variable  NAME TYPE [STRING])
```

A macro `define-foreign-variable` vincula o símbolo scheme `NAME` à expressão C `STRING` de tipo `TYPE`. Se a expressão C for um identificador, será possível usar `set!` em `NAME` para atribuir valor à variável C.

Note que o seguinte programa:

```
(define-foreign-variable x double "var_x")
(print x)
```

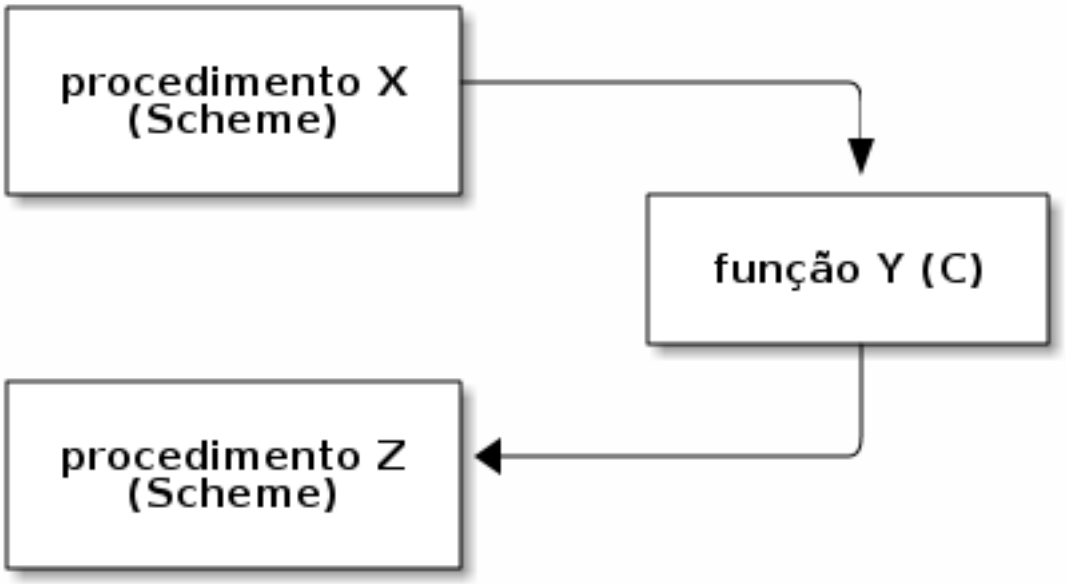
não funciona, porque `define-foreign-variable` apenas determina que o símbolo Scheme `x` deve ser traduzido para a variável C `var_x` – e *não* faz a definição de `var_x`.

3.2 Callbacks

Esta seção trata da situação em que procedimentos Scheme chamam uma função C que chama novamente um procedimento *callback* de Scheme.

3.2.1 Scheme → C → Scheme

Pode ser necessário escrever um procedmiento Scheme chama uma função C que por sua vez volta a chamar procedimentos Scheme:



O primeiro procedimento Scheme (X na figura) deve ser feito com `foreign-safe-lambda` e `foreign-safe-lambda*` ao invés de `foreign-lambda`. O "`safe`" significa que este procedimento fará uma "arrumação" na pilha antes de retornar (isto é necessário neste caso).

3.2.2 Definindo os procedimentos de callback

Para definir procedimentos Scheme visíveis a partir de C, usa-se a macro `define-external`:

```
(define-external  [QUALIFIERS]
        (NAME (ARGUMENTTYPE1 VARIABLE1) ...) RETURNTYPE
     BODY ...)
```

Por exemplo:

```
(define-external (foo (c-string x)) int (string-length x))
```

3.2.3 Acessando variáveis Scheme a partir de C

```
(define-external x double 0.5)
(print x)
((foreign-lambda* int ()
  "C_return(x);"))           ==> 0.5
```

porque `define-external` definirá uma variável `x`, e também a declarará na parte C do programa. Sua forma geral é:

```
(define-external  NAME TYPE [INIT])
```

4 Embarcando Scheme em programas C

Esta seção trata de quando se quer usar procedimentos Scheme em um programa C (cujo ponto de entrada é em C, e portanto iniciou o *runtime* de C, e não o do Chicken Scheme).

Para embutir Scheme em C, é necessário:

- Usar o arquivo de cabeçalho `chicken.h`;
- Incluir um arquivo de cabeçalho com as definições das funções Scheme exportadas para C;
- Chamar `CHICKEN_run(C_toplevel)` antes de chamar as funções Scheme.

Por exemplo, uma função `multiplica-em-scheme`, feita em Scheme, pode ser usada a partir de C. Para isto, deve ser declarada no programa Scheme com `define-external`:

```
(import foreign)

(define-external
  (multiplica_em_scheme (int x)
                        (int y)) int
    (* x y))

(return-to-host)
```

A chamada a `return-to-host` é importante: a função `CHICKEN_run` transfere o controle do programa C para o Chicken, e o controle só volta ao programa C quando `return-to-host` é chamado.

Em minhas-funcoes.h:

```
extern int multiplica_em_scheme (int a, int b);
```

No programa C:

```
#include <chicken.h>
#include "minhas-funcoes.h"

int main() {
    C_word result;
    CHICKEN_run(C_toplevel);
    CHICKEN_eval_string("(begin (display 'hello) (newline))",
                        &result);
    printf("%d\n", multiplica_em_scheme(5,4));
}
```

O compilador do Chicken, `csc`, pode ser usado para compilar os dois arquivos:

```
csc -C -I. programa-scheme.scm programa-c.c -e
```

A opção `-e` instrui o compilador a não gerar a função `main`, que já existe no programa C.

Importante: O `csc` gerará um arquivo com o mesmo nome do arquivo Scheme, mas trocando a extensão para `.c` – por isso os nomes dos programas Scheme e *devem* ser diferentes. O seguinte:

```
csc -C -I. prog.scm prog.c -e
```

Não funcionará e você perderá seu programa `prog.c`, que será sobreposto pela tradução de `prog.scm` para C.

Há outras funções para interface C $\leftrightarrow$ Scheme, todas documentadas no manual do Chicken Scheme.

5 FFI em código não compilado: lazy-FFI

As facilidades expostas até agora permitem interface entre C e Scheme para código compilado. Isto é simples porque o compilador precisa apenas incluir os trechos de código e chamadas de função em um arquivo intermediário antes de compilar o executável. Um interpretador não compila, portanto o uso de chamadas externas deve ser feito de maneira diferente.

O módulo `lazy-ffi` permite carregar bibliotecas dinâmicas compartilhadas.

Após carregar o `lazy-ffi` com `(require-extension lazy-ffi)`, use

```
#~"libwhatever.so"
```

Para carregar a biblioteca C `libwhatever`.

Para usar as funções da biblioteca, use `#~SIMBOLO` para usar a função C com o mesmo nome que `SIMBOLO` (o valor de `#~SIMBOLO` em Scheme será um procedimento que chama a função C de mesmo nome).

```
#;1> (use lazy-ffi)
 ...
#;2> #~"libc.so.6"
(libc.so.6)
#;3> (#~puts "Uma string")
Uma string
#;4>
```

Os argumentos são automaticamente convertidos de tipos Scheme para tipos C de acordo com a seguinte tabela:

```
boolean      int (1 or 0)
exact        int
inexact      double
char         char

pointer/
locative     void *

string       char *
symbol       char *
```

O procedimento `#~SIMBOLO`, no entanto, não retorna valor. Para que ele o faça, é necessário informar o tipo do valor usando o token "`return:`":

```
(#~sin 0.1 return: double:)
0.0998334166468282
```

Os possíveis valores de retorno são:

```
   int:   char: float: double:       pointer:
string: symbol:  bool:   void: scheme-object:
```

Procedimentos gerados com `#~` não podem fazer callback para código Scheme, a não ser que lhes seja passado o parâmetro `safe: #t`.