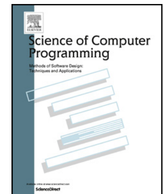
ARTICLE IN PRESS
SCICO:2085

Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



A modular foreign function interface

Jeremy Yallop [a,b,*], David Sheets [a,b], Anil Madhavapeddy [a,b]

[a] *Docker, Inc., United States*
[b] *University of Cambridge, Computer Laboratory, United Kingdom*

ARTICLE INFO

Article history:
Received 24 August 2016
Received in revised form 29 March 2017
Accepted 4 April 2017
Available online xxxx

Keywords:
Foreign functions
Functional programming
Modularity

ABSTRACT

Foreign function interfaces are typically organised monolithically, tying together the *specification* of each foreign function with the *mechanism* used to make the function available in the host language. This leads to inflexible systems, where switching from one binding mechanism to another (say from dynamic binding to static code generation) often requires changing tools and rewriting large portions of code.

We show that ML-style module systems support exactly the kind of abstraction needed to separate these two aspects of a foreign function binding, leading to declarative foreign function bindings that support switching between a wide variety of binding mechanisms — static and dynamic, synchronous and asynchronous, etc. — with no changes to the function specifications.

Note. This is a revised and expanded version of an earlier paper, *Declarative Foreign Function Binding Through Generic Programming* [19]. This paper brings a greater focus on modularity, and adds new sections on error handling, and on the practicality of the approach we describe.

© 2017 Published by Elsevier B.V.

1. Introduction

The need to bind and call functions written in another language arises frequently in programming. For example, an OCaml programmer might call the C function `puts` to display a string to standard output[1]:

```
int puts(const char *);
```

Before calling `puts`, the programmer must write a binding that exposes the C function as an OCaml function. Writing bindings presents many opportunities to introduce subtle errors [10,14,15], although it is a conceptually straightforward task: the programmer must convert the argument of the bound function from an OCaml value to a C value, pass it to `puts`, and convert the result back to an OCaml value.

In fact, bindings for functions such as `puts` can be produced mechanically from their type definitions, and tools that can generate bindings, such as SWIG [2], are widely available. However, using an external tool — i.e. operating *on* rather than *in* the language — can be damaging to program cohesiveness, since there is no connection between the types used within the tool and the types of the resulting code, and since tools introduce types and values into a program that are not apparent in its source code.

* Corresponding author.
E-mail address: jeremy.yallop@cl.cam.ac.uk (J. Yallop).

[1] For the sake of exposition the example is simple, but it captures the issues that arise when writing more realistic bindings.

http://dx.doi.org/10.1016/j.scico.2017.04.002
0167-6423/© 2017 Published by Elsevier B.V.

Please cite this article in press as: J. Yallop et al., A modular foreign function interface, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.04.002

This paper advocates a different approach, in which foreign functions such as `puts` are described using the values and types of the host language. More concretely, each C type constructor (`int`, `*`, `char`, and so on) becomes a value in OCaml, and each value that describes a function type can be interpreted to bind a function of that type. For example, here is a binding to the `puts` function, constructed from its name and a value representing its type:

```
let puts = foreign "puts" (str @→ returning int)
```

(Later sections expound this example in greater detail.)

Describing foreign language types using host language values results in a much closer integration between the two languages than using external tools. For example, the interface to SWIG is a C++ executable that generates OCaml code, and there is no connection between the C++ types used in the implementation of SWIG and the types of the generated OCaml code. In contrast, the type of the `foreign` function (which is expounded in detail in Section 2.2) is directly tied to the type of the OCaml function that `foreign` returns, since calls to `foreign` are part of the same program as the resulting foreign function bindings.

This improved integration has motivated implementations in a number of languages, including Common Lisp,[2] Python[3] and Standard ML [4]. However, although these existing designs enjoy improved integration, they do not significantly improve flexibility, since in each case the mechanism used to bind foreign functions is fixed. For example, Python's `ctypes` module binds C functions using the `libffi` library,[4] which constructs C calls entirely dynamically. The programmer who would like more performance or safety than `libffi` can offer can no longer use `ctypes`, since there is no way to change the binding mechanism.

This paper describes a design that extends the types-as-values approach using modular abstraction to support multiple binding mechanisms, including *(i)* a dynamic approach, backed by `libffi`, *(ii)* a static approach based on code generation, *(iii)* an inverted approach, which exposes host language functions to C, and several more interpretations, including *(iv)* bindings that handle `errno`, and *(v)* bindings with special support for concurrency or cross-process calls. The key is using parameterised modules to abstract the definition of a group of bindings from the interpretation of those bindings, making it possible to supply various interpretations at a later stage. Each binding mechanism (i.e. each interpretation) is then made available as a module implementing three functions: the `@→` and `returning` functions, which construct representations of types, and the `foreign` function, which turns type representations into bindings.

For concreteness this paper focuses on a slightly simplified variant of *ocaml-ctypes* (abbreviated *ctypes*), a widely-used library for calling C functions from OCaml that implements our design. As we shall see, the OCaml module system, with its support for abstracting over groups of bindings, and for higher-kinded polymorphism, provides an ideal setting.

1.1. *Outline*

This paper presents the *ctypes* library as a series of interpretations for a simple binding description, introduced in Section 2. Each interpretation is presented as an implementation of the same signature, FOREIGN, which exposes operations for describing C function types. We gradually refine FOREIGN throughout the paper as new requirements becomes apparent.

Section 3 introduces the simplest implementation of FOREIGN, an interpreter which resolves names and builds calls to foreign functions dynamically.

Section 4 describes a second implementation of FOREIGN that generates OCaml and C code, improving performance and static type checking of foreign function bindings.

Section 5 shows how support for higher-order functions in foreign bindings extends straightforwardly to supporting inverted bindings, using the FOREIGN signature to expose OCaml functions to C.

Section 6 describes some additional interpretations of FOREIGN that support error handling and concurrency.

Section 7 explores a second application of the multiple-interpretation approach, using an abstract signature TYPE to describe C object layout, and giving static and dynamic interpretations of the signature.

Section 8 presents evidence for the practicality of the *ctypes* approach, touching on adoption, performance and some brief case studies.

Finally, Section 9 contextualizes our work in the existing literature.

2. Representing types

C types are divided into three kinds: object types describe the layout of values in memory, function types describe the arguments and return values of functions, and incomplete types give partial information about objects. Bindings descriptions, as for `puts` in the introduction, involve representations of both object types, such as `int`, and function types, such as `int(const char *)`.

[2] CFFI https://common-lisp.net/project/cffi/manual/index.html.
[3] ctypes https://docs.python.org/2/library/ctypes.html.
[4] `libffi` https://sourceware.org/libffi/.

Please cite this article in press as: J. Yallop et al., A modular foreign function interface, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.04.002

Download English Version:

https://daneshyari.com/en/article/6875169

Download Persian Version:

https://daneshyari.com/article/6875169

Daneshyari.com