Jan Schwinghammer, December 5, 2005
Summary of **N. Ramsey**, *Embedding an Interpreted Language Using Higher-Order Functions and Types*, Proc. Workshop on Interpreters, Virtual Machines, and Emulators (IVME'03), pp. 6–14 (2003)

In his paper *Embedding an Interpreted Language Using Higher-Order Functions and Types*, Ramsey presents an API to interface applications, written in the host language, from within embedded interpreters. While the general principles and advantages of embedding an interpreted language are not new, his contribution is the design of an API suitable for interpreters embedded in an ML-like host language that takes advantage of higher-order functions and static typing. The main novelty of Ramsey's approach is the use of higher-order functions to (semi-) automatically generate much of the "glue code" that mediates between embedded and host language.

The first section reviews the motivation for using embedded, interpreted languages, and contains a discussion of existing work and its shortcomings: *Scripting languages* are a useful tool to customize complex applications; to obtain a *reusable* scripting language its interpreter should be embedded into the host language. This enables applications to take advantage of scripts, for instance by interpreting configuration files. The benefit of the embedding approach is that lexing, parsing and interpretation of scripts need not concern the application developer. The drawback is that *glue code* must be provided to facilitate interaction between application and embedded interpreter, taking into account the different representations of values and different calling conventions.

Previous embedded implementations of various scripting languages were available only for C. The paper addresses this problem by demonstrating how to design an API to embed into a statically typed host language with higher-order functions. The concrete presentation is in terms of Objective Caml and Lua ("Lua-ML"), but Section 2 gives an indication of both the general mechanisms as well as aspects that are peculiar to Lua-ML.

Section 3 contains the key contribution of the paper: A side-effect of replacing C as host language is that Objective Caml's types and higher-order functions can simplify the generation of most glue code. *Embedding-projection pairs* (e-p pairs) are used to translate between the representation of values in host and embedded language. An immediate benefit is that specific programming conventions of either language are explicated in a single, well-defined, point of the program. Moreover, the proposed API provides support to *lift* e-p pairs from base types to data types and higher types, in a type-directed way.

The treatment of functions is where the languages differ most: Caml has curried higher order functions; Lua has uncurried functions, but with an arbitrary number of arguments and results. Moreover, *some* embedded functions need access to the global state of the interpreter. Thus, the major complication is the lifting of e-p pairs from argument and result types to function type. A compositional and elegant solution is provided that adds an additional state parameter only if necessary.

Ramsey concludes by discussing his experience in both implementing and using Lua-ML. It appears that in the majority of cases, glue code for applications can be derived solely from the type of an embedded function. Conversely, the implementation of the Lua libriaries in terms of OCaml libraries sometimes required more complex glue code; explicitly mentioned is the example of the string library where Lua and Caml use differing indexing conventions. Nevertheless, the implementation of the full system is comparable in size to that of implementations in C.

Lacking prior knowledge of both scripting languages and embedded interpreters I found the paper tough going in places. Fortunately, Ramsey succeeded in spelling out the relevant points at the right level of detail to aid understanding, and I enjoyed reading the paper.

Type-indexed embedding-projection pairs are an established concept in the area of denotational semantics, where they can be used to interpret types. It was nice to see that these mathematical ideas are also of practical importance. The only criticism I have is the slightly ambiguous title of the paper; while the functions are higher-order, the types are not.

Advanced Functional Programming

An application with an embedded interpreter for a scripting language becomes programmable by the user. This powerful architecture requires the interpreter to be extended with application-specific functionality such that the interpreter (and therefore the user) can control the application. In *Embedding an Interpreted Language Using Higher-Order Functions and Types* Norman Ramsey discusses the API for extending an embedded interpreter. He demonstrates for a Lua interpreter written in Objective Caml (OCaml) a simple combinator-style API. The *glue code* that is required to add an application-specific function as a new primitive to the interpreter is reduced to an expression that resembles the function's type.

A Lua value is represented inside the Lua-ML interpreter as the OCaml data type `value`. A `value` can represent any of Lua's six types, including numbers and strings. A primitive function implemented inside the interpreter for, say, string length is thus a function of type *value* → *value*. Since the string length function is ultimately implemented as an OCaml function *string* → *int*, this leads to an impedance mismatch. It must be bridged by glue code: glue code checks that the `value` argument is indeed a string, extracts it, and applies the length function to it. It also takes the `int` result and converts it back into a `value`. With traditional interpreter APIs, like the C APIs of TCL and Lua, writing glue code like this amounts to considerable effort. It is especially worrying that similar kinds of checks and conversions must be implemented for each new primitive.

The key contribution of the paper is a type-safe combinator-style API. The expression

```
efunc (string **-> return int) String.length
```

has type *value* → *value* and prepares the OCaml function `String.length` (of type *string* → *int*) for inclusion into the interpreter. Here `string` and `int` are values, each of which encapsulates *re-usable* knowledge how to *embed* an OCaml *string* or *int* into a *value*, and *project* it back. Technically `int` is an embedding/projection pair of two functions, one for each direction. While embedding always succeeds, projection may fail at run time: a *value* representing a Lua table cannot be projected to an OCaml *int*. Failure means that a Lua value passed to a primitive function does has not have the right type.

The combinators mirror the type structure of the host language—here OCaml. There are embedding/projection (e/p) pairs for *base types* like `int`, `bool`, `float`, and so on. The embedding/projection pair for a function is built using the binary infix operator `**->`. A *type constructor* like *list* corresponds to a function: `list int` is an e/p pair for an OCaml *int list*, `list string` for a *string list*, and so on. As a consequence, a finite number of combinators can be combined in infinite many ways.

Embedding/projection pairs not only provide conversion between types but also bridge *programming conventions* between host and embedded language. A partially applied Lua function implicitly "adjusts" the missing arguments to nil, whereas a partially applied OCaml function is a function over the remaining arguments. The e/p pair constructor `**->` for functions takes care of this and makes Lua functions used in OCaml curried, and OCaml functions used in Lua uncurried.

Most primitive functions do not depend on the state of the interpreter but some do for reporting errors or to refer to a standard output file. The interpreter state is not passed explicitly to a function being embedded to maintain a clean and functional API. Instead, a function must be registered into the interpreter before it can be used. During the registration the state of the interpreter is in scope and thus can be easily passed to the function.

Embedding an interpreter into an application has been known as a powerful architecture. However, the focus has been on the design of the language whereas the design of the API appeared as an afterthought. This paper rethinks the design of the API and presents type-directed combinators as an elegant solution to a difficult problem. A possible weakness is that all embedding and projection functions are constructed at startup-time of the interpreter where they most often will be known at compile time. Some timing measurements could have been used to quantify this overhead. Also, nothing is said about systematical error handling in embedded functions.

Embedding an Interpreted Language Using Higher-Order Functions and Types by Norman Ramsey was presented at the ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators, June 2003. The paper presents the extension API of Lua-ML, an implementation of a Lua interpreter in Objective Caml (OCAML). The extension API provides glue-code combinators to build functions that let travel an ordinary OCAML value into the Lua interpreter such that it becomes available as a Lua primitive.

The extension API of Tcl, Lua, and many other extension languages typically pass values of the scripting language directly to a function of the implementation (or host) language. It remains the job of the function to convert such complex values into more manageable and natural values of the host language and to detect potential type errors. This so-called glue code amounts for a substantial part of any extension. Ramsey presents with Lua-ML an alternative design that depends on higher-order functions and user-defined infix operators as they are available in OCAML (and other functional languages like SML or Haskell).

A glue-code combinator is a record holding two functions: `embed` and `project`. The `embed` function takes an OCAML value and converts it into a Lua value, the `project` function takes a Lua value and projects to an OCAML value. With such a combinator available, an OCAML value can be exported to Lua, and a Lua value represented more conveniently as an OCAML value.

The idea in Lua-ML is to have such combinators as a library for the basic OCAML types like *int*, *bool*, *string*, and so on. By convention, the combinator that handles values of type *int* is itself named `int`. Such a combinator embodies the knowledge how a particular type is represented in Lua and OCAML. Embedding and projection are not total functions and thus may fail: the `int.project` function will signal an error when asked to convert a Lua string to an OCAML *int* value.

Complex glue-code combinators are built from simpler ones: `list` is a higher-order function that takes any other combinator as argument: `list int` converts integer lists from OCAML to Lua and vice versa. All knowledge about the representation of lists in Lua is built into the combinator `list` and can be reused independently from the values inside a list. Higher-order functions like `list` may create indefinitely many glue-code functions and are the main source of expressiveness.

The next and crucial level is the handling of functions: functions in Lua *adjust* to the number of passed values, functions in OCAML are Curried and thus return a function if applied to fewer than the maximum number of values. This impedance mismatch requires substantial effort by the embedding and projecting functions. However, all effort is hidden behind an abstract type and three functions: `func`, `result`, `(**->)`. Thanks to the infix function `**->`, the glue code for a function resembles very much its type: `func (list int ** -> result bool)` converts a function of type *int list* $\rightarrow$ *bool*. In simple cases, writing glue code for a function is as simple as writing down its type.

The handling of a function becomes complicated when the OCAML implementation of the function requires access to the state of the Lua interpreter: passing the state explicitly complicates the design presented so far. The solution is to use a closure: the function is applied to the state once and from there on has again a simple signature that does not need to mention state.

Glue-code combinators are an elegant solution to a difficult problem. An extension implemented against a combinator-based API is is easier to write and shorter than when implemented against a traditional API. Unlike with a traditional API, a combinator factors out the knowledge how a value, even of a user-defined type, passes between the Lua interpreter and the host language. A combinator is an extensible representation of this knowledge.

I find the paper very convincing but would have appreciated the discussion of two more details: an example of a function requiring the interpreter's state for its implementation, and the discussion of error handling. How does a Lua-ML primitive implemented in OCAML signal an error?

Putting a reusable, embedded interpreter in control of an application has significant benefits for both the developer and the user. For the developer, using an embedded interpreter removes the need for parsing command-line arguments and configuration files. The user, on the other hand, is granted a much higher degree of flexibility in working with the application.

The APIs of existing interpreters, e.g. Tcl and Lua, are designed for embedding into C code. In his paper, Norman Ramsay presents a new API for Lua called Lua-ML for Objective Caml, which uses higher-order functions and types, and provides two significant benefits: 1) type safety, and 2) less *glue code*.

The latter is the main contribution of the paper: a novel way to reduce the amount of glue code, which is needed for using application functions in the embedded interpreter. For most functions, no glue code is needed at all, but only a description of the function's type. This is achieved by Lua-ML's ability to create pairs of *conversion functions* for arbitrarily many ML types. These pairs, called *e/p pairs*, embody knowledge for *embedding* (from Caml to Lua) and *projection* (from Lua to Caml), where embedding always succeeds and projection may fail in case of a Lua type error. The pairs are part of a type-indexed family of functions, which is built as follows: 1) for each base type such as `float`, Lua-ML provides a suitable e/p pair, 2) for type constructors such as `list`, Lua-ML provides higher-order functions which map e/p pairs to an e/p pair for the constructed type.

Lua-ML uses these embedding/projection pairs to bridge the gap between the different programming conventions of Caml and Lua. For instance, a string in Lua can represent a floating point number etc. More importantly, Caml and Lua differ in their function calling conventions. In Caml, functions with multiple arguments are conventionally defined in *curried* form, whereas Lua functions are *uncurried*, which becomes apparent when a function is partially applied: in Caml, partial application results in a closure, whereas in Lua, the missing arguments are *adjusted*, i.e., filled in with `nil` values. To convert a Caml function into Lua form, Lua-ML 1) converts the embedding/projection pair for the result into a result type using the operator `result`, 2) adds the argument types using the operator `**->`, implementing *uncurrying* and *adjustment*, and 3) converts the result type back to an embedding/projection pair using the operator `func`. Taken together, the three operators allow to naturally write `func (t **-> u **-> v **-> result w)` for the embedding of an Caml function with type `t -> u -> v -> w`.

Another innovation of the paper is an elegant way to support both 1) embedded Caml functions that do inspect or modify the state of the Lua interpreter, and 2) those that do not. In both cases, the type of a Lua function is `value list -> value list`, without any mention of the state. This guarantees symmetric embedding/projection pairs for functions, and removes the need for bookkeeping, which would be induced by state-passing. The state is hidden in a closure which results from partially applying the function to the state of the interpreter earlier on (upon registration to the interpreter for Caml functions, for Lua functions upon processing their definition).

The paper presents very clever and elegant ideas, and is well written. For my taste, some of the largely irrelevant details (e.g. the comparison between the Lua 2.5 and 4.0 APIs) could have been dropped. In addition, I would have appreciated some more words on Lua fallbacks and exception handling.

Summary by Ralph Debusmann