Embedding an Interpreted Language

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 vailable 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 re-used 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* → *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?