Quicksilver/OCaml: A Poor Man's Type-Safe and Abstraction-Secure Communication Library

(Work in Progress)

Hisatoshi Sutou Eijiro Sumii

Tohoku University

{sutou,sumii}@kb.ecei.tohoku.ac.jp

Abstract

We present Quicksilver/OCaml, an Objective Caml library for communicating data in a possibly hostile network environment without losing type safety or abstraction. It is entirely implemented at user level, i.e., without modifying the compiler or the runtime at all. The key ideas are (1) representing types by (tuples of) functions, (2) abstraction by encryption, and (3) distinction of type representations for input and output. We show by examples that the library is useful for implementing applications where not only *safety*, but also *security* against malicious attackers is required.

1. Introduction

Safety and security are considered important.[1] In programming languages, *type safety* and *abstraction* are two major ways of achieving safety and security. Although opinions vary even among experts,[2] we here take the position that a type represents the set of operations applicable to values of that type. (Dually, an effect would represent the set of operations that a term may perform on the environment.) Thus, a program is type-safe when no unexpected operations are applied to values in the program. Type abstraction is a traditional means of proper user-defined types: it not only defines an abbreviation for a combination of preexisting types, but also restricts operations on values of the new type.

Type safety and abstraction are well guaranteed in friendly environments, where all programs are statically type-checked or dynamically monitored by the runtime environment. However, they do not always extend to hostile environments such as open networks. (In this paper, we use the word "network" in the broad sense of any environment where programs can communicate with each other. Thus, even file systems or databases may be considered as instances of networks.)

[1] They have even become a main political slogan of the Japenese government, "Anzen-Anshin" (meaning "security and safety").

[2] See, for instance, the gigantic thread in the TYPES mailing list, starting at `http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00270.html`. See also [13, Section 1].

For example, in Objective Caml—the implementation of an almost type-safe programming language—the following code

```
Marshal.from_channel inchan ^ "abc"
```

incurs unspecified behavior (such as segmentation fault), when executed in parallel with code like

```
Marshal.to_channel outchan 123 []
```

as clearly warned in the official manual [11, Chapter 20, Module Marshal]. Superficially, this is because `Marshal.from_channel` has the unsafe type `in_channel -> 'a`, but the real reason is of course that (the standard version of) Objective Caml does not support dynamic typing.

A similar observation applies to CORBA [12]. That is, it is neither type-safe, nor meant to be. Although the specification [12] does mention type safety in several places, a simple experiment—like sending a `double` and receiving it as a `long`—reveals that it is not always guaranteed (e.g., when a client and a server use different interface definition files).

Even with dynamic typing, confusion of abstract data types may occur. For example, if one program sends a complex number in Cartesian representation (either by mistake or by malice) to another program that expects a polar representation, then its absolute value can become negative, breaking the invariant and perhaps producing an arbitrary result. Java RMI tries to prevent such problems by considering the location of a class file as part of the class type, but it is too strong in the sense that it breaks the compatibility of identical classes loaded from different locations. HashCaml [1] solves this problem by using the hash value of the source code of a module as its type, but it is not intended for protection against malicious attackers, either passive or active.

In addition, most implementations of type-safe and "abstraction-safe" [10] communication are done at the meta-level—i.e., by modifying the compiler and/or the runtime—raising the hurdle for casual users.

We therefore are developing Quicksilver/OCaml, a user-level library (and preprocessor) in Objective Caml (and Camlp4) for type-safe and abstraction-safe communication. It is a poor man's implementation: it does not cover all the types in Objective Caml, and it requires some annotations by programmers. To be more specific, (i) functions and references cannot be communicated, and (ii) type representations must be provided by hand to marshaling and unmarshaling functions (`to_channel` and `from_channel`). However, it supports many standard data types as well as user-defined ones, and reports a static type error if the annotations are inconsistent with the rest of the program.

The rest of this report proceeds as follows. Section 2 reviews related work. Section 3 introduces the basic idea of our type rep-

resentation, which is standard. Section 4 explains how to automatically generate representations of algebraic data types defined in user programs. Section 5 discusses abstraction by encryption, and Section 6 gives an example. Section 7 extends our framework by refining the type representations to support asymmetric encryption, Section 8 gives another example, and Section 9 concludes, describing the inherant limitations of our approach and the current status of our project.

The main contributions of this work are: (i) the automatic generation of type representations for user-defined algebraic data types, and (ii) abstraction by encryption, with examples of applications that require security in addition to safety.

This submission intentionally falls far short of the 12-page limit of full research papers, following the spirit of "functional pearls" (see [2] for example) and that of work-in-progress reports (as described in the call for papers).

2. Related Work

In general, implementing marshaling and unmarshaling—also known as serialization and deserialization, or pickling and unpickling—is an easy exercise. However, safely implementing them *inside* a statically typed language (i.e., not using unsafe operations, not modifying the language runtime, and not resorting to runtime reflection) is harder.

Kennedy [8] and Elsman [6] presented pickler combinators or type-specialized serialization, which are similar to our type representations (as well as Danvy and Yang's type encoding [5, 16], which was inspired by Filinski's implementation of type-directed partial evaluation [4] in ML). On one hand, our current implementation does not support sharing of data structures [8, Section 4] [6, Section 3.7]. On the other hand, we support automatic generation of type representations for algebraic data types defined by users, which [6, 8] do not. We also support abstraction by encryption (Section 5 and 7), which is essential for security in some applications (see Section 6 and 8). As discussed in the examples, the combination of marshaling and encryption is also essential there.

Marshaling is an example of generic programming. Although generic programming is not the goal of our work, one can nevertheless compare our approach with marshaling implemented by means of generic programming. To our knowledge, existing methods of generic programming either (i) require an extension to the language itself, as in Generic Haskell [7] and GHC [15, Section 7.4.11] (so that instances of the type classes `Data` and `Typeable` [9] can be derived automatically), and/or (ii) are not so automatic for user-defined algebraic data types.

Sumii and Pierce [14] briefly mentioned the idea of using encryption for abstraction. Their work was mainly theoretical and implementation was not discussed at all.

3. Type Representation

By definition, dynamic typing requires runtime type information. However, the naive approach—i.e., defining functions like

```
to_channel : rep -> out_channel -> 'a -> unit
```

and

```
from_channel : rep -> in_channel -> 'a
```

along with the type `rep` of type representations—does not work in the standard type system of ML, which lacks dependent types. (Type classes are not available, either, in ML.)

Thus, we adopt the idea of type encoding by Danvy and Yang [5, 16] and define functions

```
to_channel : 'a rep -> out_channel -> 'a -> unit
```

and

```
from_channel : 'a rep -> in_channel -> 'a
```

where τ `rep` is the representation of type τ. Concretely, the value of type τ `rep` is a triple of

- the *tag* (name in string) for values of type τ,
- the *putter* (writer) function for values of type τ, and
- the *getter* (reader) function for values of type τ.

Even more concretely, the definition of type constructor `rep` is:

```
type 'a rep =
  string                          (* tag *)
* (Buffer.t -> 'a -> unit)        (* putter *)
* (string -> int -> int -> 'a)    (* getter *)
```

The "getter" function takes two integers so that it can read the value from the middle of a string, for the sake of efficiency.

For example, the representation of type `float` can be defined like:

```
let float : float rep =
  "Pervasives.float",
  (fun buf x -> Printf.bprintf buf "%.20f" x),
  (fun s pos len ->
     float_of_string (String.sub s pos len))
```

(Of course, we could just as well use more efficient binary representation of floating point numbers.) Representations of other primitive types are defined one by one in similar ways.

Then, the input and output functions `from_channel` and `to_channel` have only to take a type representation, project its elements (which are putter and getter functions themselves), and use them appropriately.

4. Automatic Representation Generation for Algebraic Data Types

Although representations of predefined types can be given a priori by the library, ML-like languages also have user-defined algebraic data types like:

```
type 'a tree =
  | Leaf of 'a
  | Node of 'a tree * 'a tree
```

We automatically generate their representations in a type-directed manner by using Camlp4 (a pre-processor-pretty-printer for Objective Caml). We consider the following aspects of algebraic data types.

Enumeration. For simple enumeration types such as

```
type day = Sun | Mon | Tue | ... | Sat
```

we assign a 1-byte value to each data constructor, like

```
let day : day rep =
  "(M.day = <M.Sun, ..., M.Sat>)",
  (fun buf x -> match x with
    | Sun -> Buffer.add_char buf '\001'
      ...
    | Sat -> Buffer.add_char buf '\007'),
  (fun s pos len -> match s.[pos] with
    | '\001' -> Sun | ... | '\007' -> Sat
    | _ -> raise DynamicTypeError)
```

where `M` is the absolute path of the current module. (This only supports up to 255 data constructors for each type, but a similar

restriction already exists in Objective Caml itself if any of the constructors have an argument.[3])

Constructor Arguments. For constructors with arguments, like `ISome` below,

```
type int_option = INone | ISome of int
```

we use the type representations of the arguments as in

```
let int_option : int_option rep =
  ("(M.int_option = " ^
      "<M.INone, M.ISome : Pervasives.int>)"),
  (fun buf x -> match x with
    | INone -> Buffer.add_char buf '\001'
    | ISome y1 ->
        Buffer.add_char buf '\002';
        put int buf y1),
  (fun s pos len -> match s.[pos] with
    | '\001' -> INone
    | '\002' ->
        ISome(get int s (pos + 1) (len - 1))
    | _ -> raise DynamicTypeError)
```

where `put` and `get` are functions to extract the second or third element from the triple representing a type.

Type Operators. Parametrized types such as

```
type 'a option = None | Some of 'a
```

are naturally represented by functions from type representations to type representations:

```
let option (_a : 'a rep) : 'a option rep =
  (let a = tag _a in
   Printf.sprintf
     "(%s M.option = <M.None, M.Some : %s)"
     a a),
  (fun buf x -> match x with
    | None -> Buffer.add_char buf '\001'
    | Some y1 ->
        Buffer.add_char buf '\002';
        put _a buf y1),
  (fun s pos len -> match s.[pos] with
    | '\001' -> None
    | '\002' ->
        Some(get _a s (pos + 1) (len - 1))
    | _ -> raise DynamicTypeError)
```

Here, `tag` is a function to retrieve the third element of a type representation triple.

Recursive Types. Last, recusrive types like `'a tree` (see above) are represented by using recursive putter and getter functions:

```
let rec put_tree _a =
  fun buf x -> match x with
    | Leaf y1 ->
        Buffer.add_char buf '\001';
        put _a buf y1
    | Node(y1, y2) ->
        Buffer.add_char buf '\002';
        put_tree _a buf y1;
        put_tree _a buf y2
let rec get_tree _a =
  fun s pos len -> match s.[pos] with
    | '\001' ->
        Leaf(get _a s (pos + 1) (len - 1))
    | '\002' ->
        let len1, len2 = ... in (* see below *)
        Node(get_tree _a s (pos + 1) len1,
             get_tree _a s (pos + 1 + len1) len2)
    | _ -> raise DynamicTypeError
let tree (_a : 'a rep) : 'a tree rep =
  (let a = tag _a in
   Printf.sprintf
     ("(%s M.tree = " ^
         "<M.Leaf : %s, " ^
          "M.Node : %s M.tree * %s M.tree>)")
     a a a a),
  put_tree _a,
  get_tree _a
```

Mutually recursive types are supported by the implementation but omitted in this presentation.

In addition, we actually prepend length information in front of the value representation of every constructor argument. This design is chosen in favor of uniformity over efficiency. We could alternatively have the getter functions return the length of the string that they consumed, except for a few necessary cases such as strings and arrays.

Appendix A shows generic rules for generating type representations.

5. Abstraction by Encryption (and Encryption by Abstraction)

The previous type representations suffice for simple and parametric types, but did not support abstract types. To implement abstraction in possibly hostile environments, we actually use encryption.[4]

Specifically, values of abstract types are encrypted when sent, and decrypted when received, using a temporary buffer for storing the plain texts. For this purpose, we define a higher-order function `abs` of type `key -> 'a rep -> 'a rep`, defined roughly like:

```
let abs k r =
  encrypt k (tag r),
  (fun buf x ->
    let tmp = Buffer.create 16 in
    put r tmp x;
    let str = Buffer.contents tmp in
    put string buf (encrypt k str)),
  (fun str pos len ->
    let (str', pos', len') =
      decrypt k str pos len in
    get r str' pos' len')
```

Like the getter functions, the decryption function also handles substrings for efficiency. (For concreteness, we are using Xavier Leroy's Cryptokit library, available at `http://pauillac.inria.fr/~xleroy/software.html#cryptokit`, in our implementation.)

[3] "The current implementation limits each variant type to have at most 246 non-constant constructors." (`http://caml.inria.fr/pub/docs/manual-ocaml/manual010.html#htoc58`)

[4] One might think that encryption is too expensive an operation, but modern cipher such as AES is rather fast [3], in particular when compared to network (or file) input and output. Indeed, AES was originally targeted for smart cards with 8-bit processors, so efficiency was the main point [3]. (According to the README file, the throughput of 128-bit AES raw encryption in CryptoKit is about 0.3 Gbits/s on a 2 GHz Pentium 4 processor.) Even asymmetric encryption can often be as fast: one usually encrypts only a shared key, and uses symmetric encryption in fact. In addition, encryption is usually an $O(n)$ operation anyway, while checking the invariant of an abstract data type may take more time (and space).

For example, if a group of programs implements complex numbers by polar representation and wishes to hide their implementation, then they can (and should) be communicated by using `abs k t` as type representation, where `t` is the representation of the concrete type and `k` is a secret key shared by the program group.

The fact that the use of `abs`—as well as the proper management of the secret key `k`—is manual (rather than automatic) can actually be a strength, not weakness, as it gives more flexibility: by "exploiting" the encryption operator `abs`, one can conveniently implement security protocols, as shown in the following sections.

6. Example I: Rock, Paper, Scissors

As a sample for the utility of Quicksilver/OCaml, we implemented a client-server system for the rock-paper-scissors game (`http://en.wikipedia.org/wiki/Rock,_Paper,_Scissors`) among multiple parties. For simplicity, assume that each client i a priori shares a secret key k_i with the server. Then, after connecting to the server, each client i generates a fresh secret key (nonce) k_i', encrypts its throw t_i under k_i', and sends the ciphertext to the server. Here, a throw is a value of the enumeration type defined as:

```
type throw = Rock | Paper | Scissors
```

After receiving the throws of all clients, the server sends an acknowledgement `()` to the clients. The clients sends back k_i' encrypted under k_i. Then, the server decrypts the clients' throws and notifies of the winner(s). In case of a draw, the whole process is repeated.

Thus, the server program (simplified for the two-party case) looks like:

```
(* receives encrypted throws and sends ack *)
let th1' = from_channel string ic1 in
let th2' = from_channel string ic2 in
to_channel unit oc1 ();
to_channel unit oc2 ();
(* receives keys and decrypts the throws *)
let k1' = from_channel (abs k1 key) ic1 in
let k2' = from_channel (abs k2 key) ic2 in
let th1 = from_string (abs k1' throw) th1' in
let th2 = from_string (abs k2' throw) th2' in
... (* judges and notifies of the winner(s) *)
```

(The function `from_string` reads a value from a string instead of a channel.) Correspondingly, the client program looks like:

```
let th = ... (* the client's throw *) in
let k' = generate_fresh_key () in
let th' = to_string (abs k' throw) th in
to_channel string oc th';
from_channel unit ic;
to_channel (abs k key) oc k';
...
```

These are reasonably simpler than possible implementations without Quicksilver/OCaml, which would require manual dynamic checks for the types of keys and throws. A mere combination of dynamic typing and encryption does not help, either, as the encryption and decryption functions themselves would require injection to/projection from type dynamic. Thus, the *integration* of encryption into dynamic typing is crucial here.

7. Distinguishing input and output type representations

Once one begins to use Quicksilver/OCaml to implement security protocols, support for asymmetric encryption is desired almost immediately. However, it does not fit the framework presented so far, because `abs` takes only one key for both encryption and decryption.

Thus, in addition to the developments above, the library (and preprocessor) also provides output- or input-only type representation. To be specific, we define:

```
type 'a orep = (* output-only *)
  string                          (* tag *)
* (Buffer.t -> 'a -> unit)        (* putter *)

type 'a irep = (* input-only *)
  string                          (* tag *)
* (string -> int -> int -> 'a)    (* getter *)
```

The `abs` function is also split into two versions, namely:

```
let oabs (ek : enckey) (or : 'a orep) : 'a orep =
  encrypt ek (tag or),
  (fun buf x ->
    let tmp = Buffer.create 16 in
    put or tmp x;
    let str = Buffer.contents tmp in
    put string buf (encrypt ek str))

let iabs (dk : deckey) (ir : 'a irep) : 'a irep =
  encrypt dk (tag ir),
  (fun str pos len ->
    let (str', pos', len') =
      decrypt dk str pos len in
    get ir str' pos' len')
```

The other type representations, as well as the types of various functions that take type representations as arguments, are changed accordingly. In particular, `to_channel` and `from_channel` are now typed as

```
to_channel : 'a orep -> out_channel -> 'a -> unit
```

and

```
from_channel : 'a irep -> in_channel -> 'a
```

respectively.

This change does not incur any disadvantage against programs that do not need asymmetric encryption, for (1) the modified library can be used together with the original one just under different module names (their *value* representations are compatible except for encrypted ones), and (2) it is easy to convert `'a rep` to the pair of `'a orep` and `'a irep` or vice versa.

8. Example II: Chat

To test the asymmetric encryption functionality, we wrote a simple chat server and its client in Quicksilver/OCaml, with the functionality of sending private messages to specific participants only.

Let the server's encryption and decryption keys be ek_s and dk_s, respectively. Let also client i's keys be ek_i and dk_i. The server waits for new connections and messages from existing clients (if any) at the same time.[5] A client connects to the server, registers the user's name, and waits for input from the user as well as replies from the server.

Messages from clients to the server are given the following type:

```
type message =
  Message of string
    (* to all participants *)
```

[5] Since the standard library of Objective Caml does not support the "select" function over high-level channels (`http://caml.inria.fr/mantis/view.php?id=3579`), we used Cash (`http://pauillac.inria.fr/cash/`) instead.

```
| Secret of string * string
    (* to particular participant *)
| Rename of string
    (* change name *)
| Member
    (* request participant list *)
| Quit
    (* log out *)
```

Replies from the server to clients are:

```
type reply =
  Reply of string * string
    (* message from participant *)
| Info of string
    (* information from server *)
| Disconnect
    (* disconnection from server *)
```

So the client's (pseudo-)code for processing user input is roughly

```
let msg = the_user_input in
to_channel (oabs ek_s omessage) outchan msg;
return_to_select
```

and for receiving server replies:

```
let rep = from_channel (iabs dk_i ireply) inchan in
process_rep;
return_to_select
```

(We adopt the convention that type representations for output have prefix `o` and those for input `i`.) The server's code for processing client i's message looks like Figure 1.

Again, the client and server programs are greatly simplified thanks to "encryption by abstraction." In addition, the automatic generation of type representations is essential here, because the types are far more complex than the trivial enumeration in the previous example.

9. Conclusion

To repeat the title of the paper, Quicksilver/OCaml is a poor man's implementation. Although easy to use, it is by no means intended to replace a real distributed language. Specifically,

- We do not support sending or receiving function closures (between different programs, in particular).
- We do not preserve sharing of mutable data structures—such as reference cells, defined as `type 'a ref = {mutable contents : 'a}` in Objective Caml—across multiple programs (though it would be possible to preserve sharing within a single data structure, as in [6, 8]).
- We do not support objects or polymorphic variants, either.
- Tuples are also a difficulty: in ML, we cannot define a function `tuple` such that `tuple n` has type `'a1 -> ... -> 'an -> 'a1 * ... * 'an` for all natural numbers `n`; we therefore make `tuple n` (where `n` is a non-negative integer constant) a special form and preprocess it away.

Nevertheless, we believe that Quicksilver/OCaml will be useful enough in that it is "lightweight" and easy to deploy. Above all, it illustrates the fun of programming with functions.

A prototype of Quicksilver/OCaml has already been implemented and working. We plan to work on polishing and documenting it for public release in the near future.

Acknowledgments

We would like to thank Benjamin Pierce for coining the name Quicksilver as a programming environment with "fluid boundary" (of type abstraction). We also thank Naoki Kobayashi and members of the laboratory for comments on this work.

References

[1] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml. In *Proceedings of the 2006 ACM SIGPLAN Workshop on ML*, pages 20–31, 2006.

[2] Richard Bird. How to write a functional pearl. Invited talk at *the 11th ACM SIGPLAN International Conference on Functional Programming*. Slides available at `http://icfp06.cs.uchicago.edu/bird-talk.pdf`, 2006.

[3] Joan Daemen and Vincent Rijmen. The block cipher Rijndael. In *Smart Card – Research and Applications*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer-Verlag, 2000.

[4] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, 1996.

[5] Olivier Danvy. A simple solution to type specialization. In *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 908–917. Springer-Verlag, 1998.

[6] Martin Elsman. Type-specialized serialization with sharing. In *Sixth Symposium on Trends in Functional Programming*, 2005.

[7] Generic Haskell. `http://www.generic-haskell.org/`.

[8] Andrew Kennedy. Functional pearls: Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[9] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 204–215, 2005.

[10] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 87–98, 2003.

[11] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system – documentation and user's manual. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[12] Object Management Group. Common object request broker architecture: Core specification. Available at `http://www.omg.org/docs/formal/04-03-12.pdf`.

[13] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1–3):169–192, 2007. Extended abstract appeared in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 161–172, 2004.

[15] The GHC Team. The glorious Glasgow Haskell compilation system user's guide. Available at `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

[16] Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, 1998.

A. Generic Rules for Generating Type Representations

Let D the declaration of an algebraic data type

$$\begin{aligned} D \quad = \quad & \texttt{type}\ (\alpha_1, \ldots, \alpha_n)\ t = \\ & \mid C_1\ \texttt{of}\ \tau_{11} \times \ldots \times \tau_{1\ell_1} \end{aligned}$$

```
let msg = from_channel (iabs dk_s imessage) inchan_i in
begin match msg with
   Message(x) ->
     for each i' <> i,
       to_channel (oabs ek_i' oreply) outchan_i' (Reply(names_i, x))
 | Secret(x', x) ->
     for i' such that names_i' = x',
       to_channel (oabs ek_i' oreply) outchan_i' (Reply(names_i, x))
 | Rename x ->
     if no_conflict_of_names then
       for all i',
         to_channel (oabs ek_i' oreply) outchan_i' (Info(names_i ^ " renamed to " ^ x))
     else
       to_channel (oabs ek_i oreply) outchan_i (Info("failed"))
 | Member ->
     to_channel (oabs ek_i oreply) outchan_i (Info(all names))
 | Logout ->
     for each i' <> i,
       to_channel (oabs ek_i' oreply) outchan_i' (Info(names_i ^ " logged out"));
     to_channel (oabs ek_i oreply) outchan_i Disconnect
end;
return_to_select
```

Figure 1. Chat server (pseudo-code excerpt). `names_i` denotes the name of client i as registered to the server.

$$\vdots$$
$$\mid C_m \texttt{ of } \tau_{m1} \times \ldots \times \tau_{m\ell_m}$$

where metavariable τ denotes types, b base types, α type variables, and t type constructors.

$$\tau \quad ::= \quad b \mid \alpha \mid (\tau_1, \ldots, \tau_k)\, t$$

Then, the representation of type $(\alpha_1, \ldots, \alpha_n)\, t$ is given as follows. (We are omitting modules paths here. Also, we abuse notations and write α or τ in term contexts, meaning term variables or function applications. In particular, if $\tau = (\tau_1, \ldots, \tau_k)\, t$, then τ as a term denotes $t\ \tau_1\ \ldots\ \tau_k$.)

$Rep(D) =$
```
  let rec put α1 ... αn buf = function
    PutPat(D)
  and get α1 ... αn str pos len =
    match str.[pos] with
      GetPat(D)
  and t α1 ... αn =
    (Printf.sprintf
      "(%s ... %s t =
        ⟨C1 : τ11 × ... × τ1ℓ1, ..., Cm : τm1 × ... × τmℓm⟩)"
      (tag α1) ... (tag αn)),
    put α1 ... αn,
    get α1 ... αn
```
$PutPat(D) =$
```
  | C1(x11, ..., x1ℓ1) →
    Buffer.add_char buf '\001';
    let tmp = Buffer.create 16 in
    put τ11 tmp x11;
    put int buf (Buffer.length tmp);
    Buffer.add_buffer buf tmp;
      ⋮
    let tmp = Buffer.create 16 in
    put τ1ℓ1 tmp x1ℓ1;
    put int buf (Buffer.length tmp);
    Buffer.add_buffer buf tmp;
  | ...
  | Cm(xm1, ..., xmℓm) →
    Buffer.add_char buf '\00m';
    let tmp = Buffer.create 16 in
    put τm1 tmp xm1;
    put int buf (Buffer.length tmp);
    Buffer.add_buffer buf tmp;
      ⋮
    let tmp = Buffer.create 16 in
    put τmℓm tmp xmℓm;
    put int buf (Buffer.length tmp);
    Buffer.add_buffer buf tmp;
```
$GetPat(D) =$
```
  | '\001' →
    let pos, len = pos - 3, 0 in
    let pos = pos + 4 + len in
    let len = get int str pos 4 in
    let x11 = get τ11 str (pos + 4) len in
      ⋮
    let pos = pos + 4 + len in
    let len = get int str pos 4 in
    let x1ℓ1 = get τ1ℓ1 str (pos + 4) len in
    C1(x11, ..., x1ℓ1)
```

```
| ...
| '\00m' →
  let pos, len = pos - 3, 0 in
  let pos = pos + 4 + len in
  let len = get int str pos 4 in
  let x_m1 = get τ_m1 str (pos + 4) len in
    ⋮
  let pos = pos + 4 + len in
  let len = get int str pos 4 in
  let x_mℓm = get τ_mℓm str (pos + 4) len in
  C_m(x_m1, ..., x_mℓm)
```