Lightweight Typed Customizable Unmarshaling

Pascal Cuoq Julien Signoles
CEA, LIST, Software Safety Laboratory
F-91191 Gif-sur-Yvette Cedex, France
Firstname.Lastname@cea.fr

Damien Doligez
INRIA,
F-78153 Le Chesnay, France,
Damien.Doligez@inria.fr

1. Abstract

The contribution of this work is threefold. First, we offer an OCaml unmarshaling algorithm that uses a lightweight type-directed description of the expected structure of data to make consistency checks. The second contribution is the opportunity to specify functions to be systematically applied on values as they are being unmarshaled. Our third contribution is a type-safe layer for these functions and for the unmarshaling algorithm itself.

2. Description

The standard OCaml unmarshaling function is `input_value: in_channel` $\rightarrow \alpha$. The aforementioned first contribution is a function `descr_input_val` with type `in_channel` $\rightarrow$ `descr` $\rightarrow \alpha$. One value of type `descr` that can be passed as second argument is `Abstract`, and then the behavior is exactly identical to `input_value`. The programmer can provide as much information as (s)he wants for the additional consistency checks. For instance, (s)he can pass the value `(t_array Abstract)` to specify that the value being read is an array of undescribed values. Our unmarshaling algorithm maintains a cursor into the structure description, allowing it to know what the current loaded value should look like.

A value of type `descr` can include functions to be applied on the values as they are built. This is necessary for instance when unmarshaling values of hashconsed types: the maximal sharing invariant must be preserved[CD08]. In this case the programmer may use a description containing a function f of type $\tau \rightarrow \tau$ to rehash these values. In order to allow this, our unmarshaling function preserves OCaml's runtime invariants, including for the partially reconstructed values. The historical implementation of marshaling in Ocaml is written in C and does not have this property. The new `unmarshal` is written in low-level unsafe OCaml. It is compatible with the existing format, so that it can be used with the old `marshal` function.

Unfortunately, it is not possible to type the function f above in a safe way. The low-level API requires it to have type `Obj.t` $\rightarrow$ `Obj.t`. The programmer must use unsafe coercions to implement it. We lessen this drawback with a type-safe layer that relies on a dynamic representation of OCaml types[Sig11b, Sig11a]. The type `descr` of descriptors becomes a polymorphic phantom type in which the type variable encodes the static type being described. The type expected for function f above becomes α `descr` $\rightarrow \alpha \rightarrow \alpha$. Thus f can be written without unsafe operations, by providing an extra argument corresponding to the type of the unmarshaled value. Additionally, we also offer a typed unmarshaling function `typed_input_val: in_channel` $\rightarrow \alpha$ `descr` $\rightarrow \alpha$ which constrains the unmarshaled value to be used in a context compatible with the type of the descriptor.

3. Assessment and Availability

The described work has been implemented and has been used for loading analysis projects[Sig09] inside Frama-C[CS09]. The low-level layer was provided in Frama-C version Boron, and the type-safe layer was made available with version Carbon.

A previous implementation used the standard OCaml unmarshaling function. Disadvantages were numerous: the re-hashing of hashconsed values had to be done in a second pass after unmarshaling. Just before this second pass was done, two versions of the loaded project would exist in memory, causing a peak in memory use. It was necessary to remember which subvalues had already been visited, and the corresponding re-hashed versions. Otherwise, rehashing would produce the expected results but would take a very long time to do so, visiting the highly shared DAG representing the analysis project as if it were a tree.

All these problems are solved by the new unmarshaling function. The re-hashing is done value per value as they are created, and the non-shared value that came from disk can be forgotten immediately, streamlining memory usage. A table mapping node ids to corresponding re-hashed values exists during unmarshaling with the new implementation. However, this table is maintained by the unmarshal function, and the programmer does not need to know about it. By contrast, with the previous implementation, the programmer had to maintain such a table for every type, whether unmarshaled or not. While it was possible to factor the implementation of this table a little, the fact that constructors are not first-class value prevented true genericity.

4. Related Work

Kennedy [Ken04] describes a library of marshaling combinators for Haskell. These combinators work at the language level without any magic, but the programmer has to describe the data types in full detail by combining the combinators. The marshaling format is specific to this library. By contrast, our system uses OCaml's low-level marshaler and allows the programmer to give an incomplete description of the type being unmarshaled. Kennedy's work is extended and adapted to Standard ML by Elsman in [Els05], which introduces difficulties due to references and cyclic data structures. In both cases, the programmer can use the `wrap` combinator to get the same effect as our function f. Combinators are a lot more verbose to use than our type `descr`.

Rossberg *et al.* [RTK07] describe a much more ambitious system: higher-order typed pickles in Alice ML, where they are an important component of the system. Unlike OCaml's marshaled values, they carry type information. In this system, *transform* nodes can be used to get the effect of our f function.

The programming language Acute [SLW+05] and its extension HashCaml [BSSS06] provide safe (un)marshaling by marshaling a type representation ty computed by the compiler for each written value v and by comparing ty to the expected type at the time v is unmarshaled. This approach is based on a dedicated language which provides type representations at runtime. It also requires the marshaling of some additional data. Neither is required in our approach.

Henry [HMC07, Hen11] provides type-safe unmarshaling for OCaml without marshaling types by verifying the compatibility of the value being unmarshaled with the expected type. Its approach nevertheless modifies the OCaml compiler, while our approach is fully compatible with the standard OCaml compiler.

Neither Acute nor Henry's proposal allow to apply custom functions at unmarshal time, which was our primary goal.

The ATerms library [BJKO00] (in C and Java) has both hashconsing and unmarshaling. Not having to be backwards compatible with an existing format, it marshals data bottom-up (the standard OCaml format, with which our function is compatible, is topdown). Bottom-up complicates marshaling a little, but simplifies unmarshaling a lot. Being untyped, in ATerms many types of data can be represented with a few, fixed kinds of nodes chosen in advance.

5. Conclusion

We have presented a solution to mix hashconsing and unmarshaling. This solution is implemented and, at the time of publishing, has been used for months inside the analysis framework Frama-C. We get the impression that the problems we had to solve have been discovered and solved many times over in different contexts, and we hope that, in the context of OCaml, the next person who needs something like this will find this implementation rather than going through the same issues again.

Acknowledgements

Norman Ramsey and Jurgen Vinju suggested many of the comparable, existing systems to look at. All core Frama-C developers had to suffer while unmarshaling and hashconsing issues were being understood and solved, but we would like to apologize in particular to Boris Yakobowski, who bore most of the brunt.

References

[BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[BSSS06] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml. In *Proceedings of the 2006 workshop on ML*, pages 20–31. ACM, 2006.

[CD08] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in OCaml 3.10.2. In *Proceedings of the 2008 Workshop on ML*, pages 13–22. ACM, 2008.

[CS09] Pascal Cuoq and Julien Signoles. Experience Report: OCaml for an Industrial-Strength Static Analysis Framework. In *Proceedings of the 2009 International Conference on Functional Programming*, pages 281–286. ACM, September 2009.

[Els05] Martin Elsman. Type-specialized serialization with sharing. In *Proceedings of the Sixth Symposium on Trends in Functional Programming*, September 2005.

[Hen11] Grégoire Henry. *Typer la désérialisation sans sérialiser les types*. PhD thesis, Université Paris 7, June 2011. In French.

[HMC07] Grégoire Henry, Michel Mauny, and Emmanuel Chailloux. Typer la désérialisation sans sérialiser les types. *Technique et Science Informatiques*, 26(9):1067–1090, 2007. In French.

[Ken04] Andrew Kennedy. Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[RTK07] Andreas Rossberg, Guido Tack, and Leif Kornstaedt. Status report: Hot pickles, and how to serve them. In Claudio Russo and Derek Dreyer, editor, *Proceedings of the 2007 Workshop on ML*, pages 25–36. ACM, 2007.

[Sig09] Julien Signoles. Foncteurs impératifs et composés: la notion de projets dans Frama-C. In Hermann, editor, *Actes des Journées Francophones des Langages Applicatifs*, pages 37–54, January 2009. In French.

[Sig11a] Julien Signoles. An OCaml Library for Dynamic Typing. In *Trends in Functional Programming*, 2011. Submitted for publication.

[Sig11b] Julien Signoles. Une bibliothèque de typage dynamique en OCaml. In Hermann, editor, *Actes des Journées Francophones des Langages Applicatifs*, pages 209–242, January 2011. In French.

[SLW+05] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. In *Proceedings of the tenth International Conference on Functional Programming*, pages 15–26. ACM, 2005.