Do Mutable Variables Have Reference Types?

Oleg Kiselyov  
Tohoku University  
Japan  
oleg@okmij.org

Abstract
Implicit heterogeneous metaprogramming (a.k.a. offshoring) is an attractive approach for generating C with some correctness guarantees: generate OCaml code, where the correctness guarantees are easier to establish, and then map that code to C. The key idea is that simple imperative OCaml code looks like a non-standard notation for C. Regretfully, it is false, when it comes to mutable variables. In the past, the approach was salvaged by imposing strong ad hoc restrictions. The present paper for the first time investigates the problem systematically and discovers general solutions needing no restrictions. In the process we explicate the subtleties of modeling mutable variables by values of reference types and arrive at an intuitively and formally clear correspondence. We also explain C assignment without resorting to L-values.

1 Introduction
Generating C (or other such low-level language) is inevitable if we want the convenience and guarantees of abstractions – and we want the code that runs in a constrained environment (e.g., a low-powered robot); involves OpenMP, OpenCL (i.e., GPGPU) or AVX512 instructions; benefits from profitable but highly domain-specific optimizations typical in HPC. In fact, we have done all of the above, using the offshoring technique first proposed in [1] and re-thought and re-implemented in [2]. The key idea of offshoring, explained below, is the close correspondence between imperative OCaml and C code.

Mutable variables of C is the biggest stumbling block: the straightforward mapping of OCaml variables of reference types to C mutable variables is insidiously wrong, when it comes to aliasing. In the past, pitfalls were steered around of by imposing strong ad hoc restrictions – which made generating C code with mutable variables of pointer types, for example, out of reach.

We propose a better mapping between reference-type and mutable variables that needs no restrictions and hence widens the scope of offshoring.

After introducing offshoring, the paper explains the problem of generating code with mutable variables, and, in §3.2, its current, imperfect resolution. §4 and §5 each introduce new proposals, improving the state of the art. §5’s approach is the most general, intuitive, easier to show correct, and insightful. It applies to any other language which uses values of reference types to model mutable variables, such as F# and SML.

2 Offshoring
Offshoring turns homogeneous metaprogramming – generating OCaml in OCaml – into heterogeneous: generating C. The key idea is that simple imperative subset of OCaml may be regarded as a different notation for C. Taking the running example of vector addition from [2], contrast the OCaml code

let addv = fun n vout v1 v2 →
  for i=0 to n-1 do
    vout.(i) ← v1.(i) + v2.(i) done

and the C code:

void addv(int n, int* vout, int* v1, int* v2) {
  for(int i=0; i<n-1; i++)
    vout[i] = v1[i] + v2[i];
}

The similarity is so striking that one may argue that OCaml’s addv is C’s add, written in a different but easily relatable way. Offshoring is the facility that realizes such correspondence between a subset of OCaml and C (or other low-level language). With offshoring, by generating OCaml we, in effect, generate C.

With MetaOCaml, we may statically ensure the generated OCaml code compiles without errors. If we can map OCaml to C while preserving the guarantees, we in effect obtain the assured C code generator. Needless to say, the mapping ought to preserve the dynamic semantics/behavior.

It should be stressed that the mapping from OCaml to C in not total. We are not aiming to translate all of OCaml to C – only a small imperative subset. That is why we are not concerned with closures, recursion, user-defined datatypes, let alone more complicated features. Therefore, we generate efficient C that does not need any special run-time. We are not aiming to generate every C feature either. After all, like other languages C is redundant: many differently phrased expressions compile to the same machine code. The supported subset of OCaml and C, if small, should still be useful – and it proved to be in our experience, for generating HPC and embedded code.

3 Problem with Mutable Variables
However small the mappable OCaml subset may be, its range should include mutable variables, which are pervasive in C. Offshoring would hardly be useful otherwise. Thus the central problem is what should be the OCaml code that maps to C code with mutable variables. OCaml values of reference types are not a straightforward match of C mutable variables – as one realizes upon close inspection. Hereby we undertake the systematic investigation of the problem.

3.1 Formalization
We introduce the calculus lCaml to delineate the minimal relevant first-order imperative subset of OCaml. (The subset of OCaml used in offshoring [2] is not much bigger, adding loops and conditionals.) Most of it is self-explanatory. Constants c_i have to be applied to i arguments to be considered expressions. We use the customary infix notation for such applications where appropriate.

The calculus is Church-style: all (sub)expressions are annotated with their types. To avoid clutter however, we mostly elide types, where they can be easily understood. The type system is entirely
We see in particular that whereas \( x \) Assignment in CoreC is a special form; therefore, the same \( x \):

in \([1]\), but are much, much simpler – and free from the severe

deduction) typing derivation of integer assignment in the two

Figure 3. Calculus CoreC. Its constants \( c \) are the same as those in Fig. 2.

The constants \( c \) of ICaml and their types. Their arity \( i \) is the number of arrows in their type. For instance, \( \; \) is a 1-arity constant \( c \), and \( = \) is \( c \). Only constants have arrow types. We may silently add other similar constants.

standard and elided to save space. The dynamic semantics is also standard.

The calculus CoreC, Fig. 3, models the relevant subset of C: the target of the offshoring mapping. It is also entirely standard. The static and dynamic semantics of ICaml and CoreC are shown in full (in tagless-final style) in the file \texttt{refcalculi.ml} accompanying the paper (see also \url{http://okmij.org/ftp/tagless-final/refcalculi.ml}),

Figure 4. Naive offshoring translation

3.2 Extant Offshoring Translation

We now state the translation \( [\cdot] \) from a (type-annotated) expression of ICaml to a type-annotated expression of CoreC, Fig. 4, with the rest being homomorphism. Here \( [t \tau] = t \) and \( [t \tau] = t \) otherwise.

This is basically the translation proposed in \([1]\), adjusted for (many) differences in notation. As an example, the ICaml expression

\[
\text{let } x = \text{ref } 0 \text{ in } x := !x + 1
\]

is translated to

\[
\text{int } x = 0; \; y := x + 1
\]

The translation clearly expresses the idea that an OCaml value of a reference type bound to a variable is a model of mutable variables in C. It is also clear that the translation is partial: ICaml code like \( !(\text{ref } 0) \) or \( (\text{ref } 0) := 1 \) is not translatable. The translation is also non-compositional: variable references are translated differently if they appear as the first argument of \( ! \) or the assignment operation.

If we add constants with arguments of reference types, like \( \) incr\( \), the translation has to be amended.

There is also a subtle and serious problem with the translation as written. Applying it to

\[
\text{let } x = \text{ref } 0 \text{ in } y = x \text{ in } y := 41; !x + 1
\]

would give

\[
\text{int } y = 0; \; \text{int } y = x; \; \text{int } y := 41; x + 1
\]

which has a different meaning. Whereas the ICaml expression evaluates to 42, its CoreC translation returns 1.

The root of the problem is the difference in meaning between let \( y = x \) in \( y \) in ICaml and \( y = x \) in CoreC. In the latter case, a new mutable variable is allocated whose initial contents is the current value of \( x \). Then \( y \) and \( x \) are mutated independently. On the other hand, let \( y = x \) in \( y \) allocates no new reference cell; it merely introduces a new name, \( y \), for the existing reference cell named \( x \). One may informally say that in ICaml, names of mutable cells are first-class.

Although the interpretation of names in ICaml and CoreC differs, as we have just seen, the difference fades in restricted contexts. The offshoring translation can be made meaning-preserving, after all – if we impose restrictions that preclude aliasing. The paper \([1]\) never mentions that fact explicitly. However, if we carefully examine the typing rules in its Appendix A2, we discover the silently imposed restrictions: only base-type references, and only base-type let-bindings (sans the dedicated expression let \( x = \text{ref } e \) for creating references). The fixed offshoring translation is shown in Fig. 5.

This is the (core of the) offshoring translation used in the current MetaOCaml: BER N111. It has been used in all offshoring applications so far, many of which are mentioned in \([2]\), which also details simpler use cases.
Do Mutable Variables Have Reference Types?

![Diagram](image)

Although the translation proved more or less adequate for numeric code, it is clearly severely restrictive: it is impossible, for example, to generate C code with pointer-type function arguments or pointer-type mutable variables – or even represent pointer types to start with. Occasionally we had to fiddle with a generator to make it produce the offshorable OCaml code. Also, the translation to start with. Occasionally we had to fiddle with a generator to make it produce the offshorable OCaml code. Also, the translation

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without

4 C without Mutable Variables?
produces idiomatic C code, and gives insight into the nature of mutable variables.

An easy way to obtain a translation with mutable variables in its range is to start with the straightforward inverse mapping from CoreC to ICaml. Unfortunately, it is very much not surjective (and if we extend CoreC with pointer types, it becomes non-injective). Therefore, inverting it is problematic. Still the CoreC to ICaml mapping gives a hint. Other hints come from looking at the denotational semantics (tagless-final interpreters) of ICaml and CoreC: the file refcalculi.ml in the accompanying code mentioned earlier. We notice that let \( x = \text{ref} \ e \) in \( e' \) has exactly the same denotation (as the function of the denotations of \( e \) and \( e' \) as \( t \ x = e; e' \) in CoreC. Therefore, if the mutable variable \( x \) introduced by \( t x = e; e' \) is not actually mutated in \( e' \), it has the meaning of the let-binding in ICaml.

To formulate the new translation, we extend CoreC with pointer types and corresponding operations, obtaining the calculus CoreCE. Assignment is no longer a special expression form: it is a function (binder) types, to indicate that some variables are immutable. In \( x \)

\[
\text{let } x = \text{ref } 0 \text{ in } y = x : y := 41; x := !x + 1
\]

translates to CoreCE as

\[
\text{int } x = 0; \ \text{int } \text{ptr const } y = \&x; \ y := 41; \ \&x := *\&x + 1
\]

The new translation indeed gives idiomatic C code, which is easier to inspect and build confidence. Unlike the translation of §4, only one CoreCE variable is allocated per ICaml variable, with no extra pointer variables. The extended calculus CoreCE, and the current translation, which tracks mutability, stress the fact that although all variables in C are mutable by default, some are actually not mutated. The latter correspond to ICaml variables. Actually mutable variables of CoreCE correspond to ICaml variables introduced by the bindings of a particular shape: let \( x : \text{ref} \ e = \text{ref} \ e' \), which evoke letref of the original ML.

With full details and formality the translation is presented in the accompanying code refcalculi.ml. As mentioned earlier, the code also states the denotational semantics \([\cdot]_{\text{ICaml}}\) and \([\cdot]_{\text{CoreCE}}\) as compositional mappings from ICaml or CoreCE, resp., to the common metalanguage, which is OCaml. (One could also use Coq with a State monad.) The translation from ICaml to CoreCE is then coded as a functor. Since the (tagless-final) embeddings of ICaml and CoreCE into OCaml are intrinsically typed, the fact that the translation functor is well-typed in OCaml implies the translation is type-preserving. The meaning preservation is expressed by the theorem that for each ICaml expression \( e \), \([ e ]_{\text{ICaml}} = [ [ e ]_{\text{CoreCE}} \])

To show it, we have to check, manually at present, that the theorem holds for each expression form of ICaml, and then appeal to compositionality of the semantics.

In conclusion, we have learned that C variables are quite subtle: one may access a mutable variable via its name or a pointer to it; however, names and pointers are emphatically distinct.

The new offshoring translation, §5, has been implemented in BER MetaOCaml.

Acknowledgments. We thank anonymous reviewers for many, helpful comments and suggestions. This work was partially supported by JSPS KAKENHI Grants Numbers 17K12662, 18H03218, 21K11821 and 22H03563.

References

