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 solutions while avoiding any 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.

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 a constrained environment (e.g., low-powered robot); involves OpenMP, OpenCL (i.e., GPGPU) or AVX512 instructions; benefits from profitable but 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 avoided 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 avoids any 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 addv, 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 ICaml 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.)

Variables	x, y, z
Base Types	$b ::= \mathsf{int} \mid \mathsf{bool}$
Types	$t ::= \mathsf{unit} \mid b \mid t\ \mathsf{ref}$
Constant Types	$s ::= t \mid t \longrightarrow s$
Expressions	$e ::= x \mid c_0 \mid c_1\ e \mid c_2\ e\ e \mid e; e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$

Figure 1. Calculus ICaml. For constants c_i see Fig. 2.

$1, 2, 3, \ldots$	: int	ref	: $t \to t$ ref
$true, false$	: bool	!	: t ref $\to t$
$+$	: int $\to$ int $\to$ int	:=	: t ref $\to t \to$ unit

Figure 2. The constants c_i of ICaml and their types. Their arity i is the number of arrows in their type. For instance, ! is a 1-arity constant c_1, and := is c_2. Only constants have arrow types. We may silently add other similar constants.

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 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 is shown in full (in tagless-final style) in http://okmij.org/ftp/tagless-final/refcalculi.ml.

Variables	x, y, z
Base Types	$b ::=$ int | bool
Types	$t ::=$ unit | b
Constant Types	$s ::= t \mid t \to s$
Expressions	$e ::= x \mid c_0 \mid c_1\, e \mid c_2\, e\, e \mid e; e$
	$t\ x = e; e \mid x := e$

Figure 3. Calculus CoreC. Its constants c_0 and the 2-arity constant + are the same as those in Fig. 2.

The calculus permits expressions like $(\text{int}\, x = 1 + 2; x + 3) + 4$, which is invalid C. However, with the simple post-processing step of lifting variable declarations (always possible if the names are unique, which is assumed), it becomes int x; (x=1+2,x+3)+4, which is proper C.

The calculi ICaml and CoreC are quite alike; the main difference is in mutation. To emphasize the distinction, we show the (natural deduction) typing derivation of integer assignment in the two calculi.

$$\frac{x : \text{int ref} \quad e : \text{int} \quad := : \text{int ref} \to \text{int} \to \text{unit}}{x := e\ : \text{unit}} \qquad \frac{x : \text{int} \quad e : \text{int}}{x := e\ : \text{unit}}$$

The calculi ICaml and CoreC resemble the corresponding calculi in [1], but are much, much simpler – and free from the severe restriction on initializers being constants.

3.2 Extant Offshoring Translation

We can now state the translation $\lceil\cdot\rceil$ from a (type-annotated) expression of ICaml to a type-annotated expression of CoreC:

$$\lceil x : t \rceil = x : \lceil t \rceil \tag{1}$$
$$\lceil !(x : t) \rceil = x : \lceil t \rceil \tag{2}$$
$$\lceil (x : t) := e \rceil = x : \lceil t \rceil := \lceil e \rceil \tag{3}$$
$$\lceil \text{let}\ x : t = \text{ref}\ e\ \text{in}\ e' \rceil = \lceil t \rceil\ x = \lceil e \rceil; \lceil e' \rceil \tag{4}$$
$$\lceil \text{let}\ x : t = e\ \text{in}\ e' \rceil = \lceil t \rceil\ x = \lceil e \rceil; \lceil e' \rceil \tag{5}$$

with the rest being homomorphism. Here $\lceil t\ \text{ref} \rceil = t$ and $\lceil t \rceil = t$ 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

$$\text{int}\ x = 0; x := x + 1$$

The translation clearly expresses the idea that values of reference types in OCaml bound to variables 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 more constants like incr (with arguments of reference types) the translation has to be amended.

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

$$\text{let}\ x = \text{ref}\ 0\ \text{in let}\ y = x\ \text{in}\ y := 41; !x + 1 \tag{6}$$

would give

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

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 … in ICaml and $t\ y = x$; 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 … allocates no new reference cell; it mere 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 = ref e in e' for creating references). The offshoring translation thus becomes:

$$\lceil x : b \rceil = x : b \tag{7}$$
$$\lceil !(x : b\ \text{ref}) \rceil = x : b \tag{8}$$
$$\lceil (x : b\ \text{ref}) := e \rceil = x : b := \lceil e \rceil \tag{9}$$
$$\lceil \text{let}\ x : b\ \text{ref} = \text{ref}\ e\ \text{in}\ e' \rceil = b\ x = \lceil e \rceil; \lceil e' \rceil \tag{10}$$
$$\lceil \text{let}\ x : b = e\ \text{in}\ e' \rceil = b\ x = \lceil e \rceil; \lceil e' \rceil \tag{11}$$

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.

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 of the problematic (6) is not fixed: merely outlawed.

Each of the two following sections propose a new translation, overcoming the drawbacks of the state of the art.

4 C without Mutable Variables?

Paper [2] briefly mentions, merely on two examples and without details or formalization, an alternative translation: avoiding mutable variables altogether. In terms of the present paper, the calculus CoreC becomes unnecessary: ICaml as is gets mapped to the surface syntax of C. One may wonder how this is possible and if C without mutable variables is of any use. Not only it turns out possible; the resulting code also compiles to the same machine code as the conventional C. We now present the translation formally and systematically, noting its advantages and disadvantages. The disadvantages motivate the proposal in §5.

The key idea is that C already has the analogue of ICaml values of reference type: arrays. Expressions of reference types of ICaml are C pointer expressions. Assuming the lifting transformation mentioned in §3.1, the mapping $\lfloor\cdot\rfloor$ from ICaml to C is as follows:

$$\begin{array}{rcll}
\lfloor \mathsf{ref}\ (e : t) \rfloor & = & \mathsf{t\ z[1] = \{}\lfloor e \rfloor\mathsf{\};\ z} & \text{(z is fresh)} \\
\lfloor !e \rfloor & = & *\lfloor e \rfloor & \\
\lfloor e := e' \rfloor & = & *\lfloor e \rfloor := \lfloor e' \rfloor & \\
\lfloor \mathsf{let}\ x : t\ \mathsf{ref} = e\ \mathsf{in}\ e' \rfloor & = & \mathsf{t * const\ x} = \lfloor e \rfloor;\ \lfloor e' \rfloor & \\
\lfloor \mathsf{let}\ x : b = e\ \mathsf{in}\ e' \rfloor & = & \mathsf{b\ x} = \lfloor e \rfloor;\ \lfloor e' \rfloor &
\end{array}$$

One is reminded of Algol68, in which .int x := 1 is the abbreviation for .ref.int x = .local.int := 1, where .local.int is a stack allocator of an integer (similar to alloca in C).

The earlier example

$$\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ x := !x + 1$$

now looks like

```
int * const x = (int z[1]=0; z);  *x = *x + 1;
```

or, after variable lifting

```
int z[1] = 0;  int * const x = z;  *x = *x + 1;
```

The problematic (6), that is,

$$\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{let}\ y = x\ \mathsf{in}\ y := 41; !x + 1$$

becomes

```
int z[1] = 0; int * const x = z;
int * const y = x; *y = 41; *x + 1
```

and returns the same result as the ICaml code.

There are no longer any restrictions to base types; adding constants like incr is easy. Arrays, conditionals, loops are straightforward as well.

Other extensions are more problematic, however. If we extend ICaml with composite data structures (e.g., to express linked lists), functions returning values of reference types, or global variables or arguments of higher-reference types – we have to worry about the lifetime of reference cells allocated by ref e. (These extensions are not common in numeric computing however.) They have to be allocated on heap, and managed somehow (e.g., via reference counting).

The translation results in highly unidiomatic C code, which, from personal experience, provokes negative reaction and strong doubts about correctness. A reviewer suggested a slight adjustment, which makes the result look a bit more familiar.

$$\lfloor \mathsf{ref}\ (e : t) \rfloor = \quad \mathsf{alloca}(\lfloor e \rfloor)$$

where alloca could be avoided by allocating a fresh variable in scope. Thus the running example

$$\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ x := !x + 1$$

becomes

```
int z = 0; int * const x = &z; *x = *x + 1;
```

which looks a bit more like conventional C.

One of the advantages of the present translation is that

$$\lfloor t\ \mathsf{ref} \rfloor = \lfloor t \rfloor * \mathsf{const}$$

That is, reference types of ICaml map directly to pointer types in C. On the downside, we do not represent mutable variables of C or CoreC directly in ICaml. The disadvantage has a practical side: as seen from the translation examples, each reference-type variable of ICaml is translated to *two* variables in C: one holds the content and is mutated, and the other is the pointer to the former. The C code hence needs twice as many variables – and twice as much stack storage for them. In simple code, a C compiler can notice variables that are not mutated and effectively inline them, removing the need to store them. However, we are aiming to generate very complicated code. Take, for example matrix-matrix multiplication from our past work: applying standard techniques to make it fast results in thousands of lines of C code. There, C compiler may not see that some variables are redundant.

Can mutable variables in (Core)C be represented directly in ICaml? Can the translation hence map those ICaml variables directly, one-to-one, to CoreC mutable variables, without allocating pointers to them? Can we intuitively and formally be confident in the translation, even for arbitrarily complex reference types? The following section shows the answer.

5 Mutable Variables and Reference Types

We have just seen the translation from ICaml to C that maps ICaml variables of reference types to constant-pointer–type variables of C. We now present the translation that relates reference type variables of ICaml and mutable variables of C. It requires no restrictions, 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 mentioned earlier. We notice that $\mathsf{let}\ x = \mathsf{ref}\ e\ \mathsf{in}\ e'$ 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 constant function application. Therefore, other pointer-taking functions like incr can be added at will. We also add constant (binder) types, to

Variables	x, y, z
Base Types	$b ::= \mathsf{int} \mid \mathsf{bool}$
Types	$t ::= \mathsf{unit} \mid b \mid t\ \mathsf{ptr}$
Binder Types	$u ::= t \mid t\ \mathsf{const}$
Constant Types	$s ::= t \mid t \to s$
Expressions	$e ::= x \mid c_0 \mid c_1\ e \mid c_2\ e\ e \mid e; e$
	$u\ x = e; e \mid \&x$

Additional Constants

$$* \ : t\ \mathsf{ptr} \to t \qquad\qquad \leftarrow\ : t\ \mathsf{ptr} \to t \to \mathsf{unit}$$

Figure 4. Calculus CoreCE. Constants c_0 and + are same as those in Fig. 2.

indicate that some variables are immutable. In surface C, $e \leftarrow e'$ is to be rendered as *e = e'. Also, in surface C we abbreviate *&x to just x. (We may also leave *&x as is: it is valid.) The type system and dynamic semantics are fairly standard: see refcalculi.ml.

The offshoring translation $\lceil \cdot \rceil_L$ is parameterized by the set L of mutable variables in scope:

$$\begin{array}{rcl}
\lceil x : t \rceil_L & = & x : \lceil t \rceil \quad x \notin L \\
\lceil x : t \rceil_L & = & \&x : \lceil t \rceil \quad x \in L \\
\lceil ! \rceil & = & * \\
\lceil := \rceil & = & \leftarrow \\
\lceil \mathsf{let}\ x : t\ \mathsf{ref} = \mathsf{ref}\ e\ \mathsf{in}\ e' \rceil_L & = & \lceil t \rceil\ x = \lceil e \rceil_L; \lceil e' \rceil_{L \cup \{x\}} \\
\lceil \mathsf{let}\ x : t = e\ \mathsf{in}\ e' \rceil_L & = & \lceil t \rceil\ \mathsf{const}\ x = \lceil e \rceil_L; \lceil e' \rceil_L
\end{array}$$

where the translation of types is $\lceil b \rceil = b$ and $\lceil t\ \mathsf{ref} \rceil = \lceil t \rceil\ \mathsf{ptr}$. The constant ref outside a let-binding can be translated as alloca or malloc. Compared to the extant translation in §3.2, there are no longer any restrictions on types. Adding constants like incr is easy.

The running example

$$\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ x := !x + 1$$

now looks like

$$\mathsf{int}\ x = 0;\ \&x \leftarrow *\&x + 1$$

or, after rendering in C:

```
int x = 0; x = x + 1;
```

The (extended) problematic example

$$\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{let}\ y = x\ \mathsf{in}\ y := 41; x := !x + 1$$

translates to CoreCE as

$$\mathsf{int}\ x = 0;\ \mathsf{int}\ \mathsf{ptr}\ \mathsf{const}\ y = \&x;\ y \leftarrow 41;\ \&x \leftarrow *\&x + 1$$

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: $\mathsf{let}\ x : t\ \mathsf{ref} = \mathsf{ref}\ e\ \mathsf{in}\ e'$, 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 $[[\cdot]]_{ICaml}$ and $[[\cdot]]_{CoreCE}$, 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, $[[\, e\,]]_{ICaml} = [[\, \lceil e \rceil\,]]_{CoreCE}$. 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.

Implementing the new offshoring translation, §5, in BER MetaOCaml is the subject of the current work. MetaOCaml has an extensive test suite, which includes offshoring; it will help gauge the improvement due to the new translation.

References

[1] Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar N. Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming. *New Generation Computing*, 25(3):305–336, 2007. doi: 10.1007/s00354-007-0020-x.

[2] Oleg Kiselyov. Generating C. In Michael Hanus and Atsushi Igarashi, editors, *Functional and Logic Programming*, volume 13215 of *Lecture Notes in Computer Science*, pages 75–93. Springer International Publishing, 2022. doi: 10.1007/978-3-030-99461-7_5.