Generating C: Heterogeneous Metaprogramming System Description

Oleg Kiselyov

[a] *Tohoku University, Sendai, Japan*

Abstract

Heterogeneous metaprogramming systems use a higher-level host language to generate code in a lower-level object language. Their appeal is taking advantage of the module system, higher-order functions, data types, type system and verification tools of the host language to quicker produce high-performant lower-level code with some correctness guarantees.

We present two heterogeneous metaprogramming systems whose host language is OCaml and object language is C. The first relies on *offshoring*: treating a subset of (MetaOCaml-generated) OCaml as a different notation for (a subset of) C. The second embeds C in OCaml in tagless-final style. The systems have been used in several projects, including the generation of C supersets OpenCL and OpenMP.

Generating C with some correctness guarantees is far less trivial than it may appear, with pitfalls abound: e.g., local variables may only be introduced in statement context; mutable variables are not first-class. Maintenance has challenges of its own, e.g., extensibility. On many examples, we expound the pitfalls we have come across in our experience, and describe the design and implementation to address them.

Keywords: heterogeneous metaprogramming, code generation, tagless-final

1. Introduction

Generating C is an odd problem: at first glance, it is trivial and nothing to write about. Beyond simple applications, however, complexity, traps, hazards snowball. One may as well write code directly in C to start with, or use a compiler.

Neither of these two choices may be palatable, especially in high-performance computing (HPC). For one, high-performant code is often voluminous and ob-

Email address: `oleg@okmij.org` (Oleg Kiselyov)

scure – and hence hard to write directly by hand.[1] It also has to be re-adjusted for each new architecture and processor configuration. Hopes that an optimizing compiler save us such trouble were dashed two decades ago: see Cohen et al. [1] for exposition and many references. Furthermore, compiler-generated C code typically can only be executed within the specific run-time environment and cannot be freely linked with other C code.

The alternative to writing performant low-level code by hand or to relying on a general-purpose compiler is code generation. Being domain-specific, a generator may employ very profitable but not widely-applicable expert knowledge and optimizations. Such code generation is being increasingly used in HPC, becoming dominant in some areas: ATLAS [2] for BLAS (Basic Linear Algebra Subroutines), FFTW [3] and SPIRAL [4] for FFT and related transforms, Halide [5] for image filtering, Firedrake [6] for partial differential equations using finite element method.

The present paper describes two orthogonal approaches for generating C using OCaml, with some guarantees and convenience.[2] We take OCaml as a representative high-level language for writing generators, and C as a representative low-level language. The approaches extend to other languages (e.g., [8] for LLVM IR; we have also tried WASM). The systems presented in this system description paper are programming language systems, akin to a domain-specific compiler: The user feeds an OCaml source code file and obtains an executable, which, when run, produces C code. The result can be used as regular C library code; it may also be compiled and dynamically linked into the generator, thus effecting run-time specialization. One may also enter (the fragments of) the generator code at the interactive OCaml top-level, evaluate them, see the generated code and test it. We extensively use this interactive mode in the exposition.

The first approach is offshoring, initially proposed in Eckhardt et al. [9]: generating simple OCaml code with certain correctness guarantees (namely, well-formed, well-typed, well-scoped) and then translating it to C, preserving the guarantees. The second approach is embedding C in OCaml as typed combinators, in tagless-final style [10, 11]. Both ensure the generated C code compiles without errors: see §3.2 and §5.2 for the justification of these assurances. Both explore the metaphor of a subset of OCaml as a notation for C.

[1] For an example, see the manually written high-performance BLAS code in `https://www.openblas.net/`, e.g., dot-product `https://github.com/xianyi/OpenBLAS/blob/develop/kernel/x86_64/ddot.c` and its kernel for a particular, already obsolete processor architecture `https://github.com/xianyi/OpenBLAS/blob/develop/kernel/x86_64/ddot_microk_sandy-2.c`

[2] The present paper is loosely based on [7], published in the Proceedings of FLOPS 2022. Not only the present paper is more than twice as long, the implementation has significantly changed (prompted, in part, by writing and presenting the FLOPS paper) – although the user interface remained the same. Particularly notable is a new approach to generating mutable variables, removing all earlier restrictions (see §4.6), and the normalization of let-bindings (see §4.2). We now generate C99, mixing variable declarations and statements. Tagless-final code generation, which was hardly explained before, is covered in full.

Offshoring is a particularly attractive idea – and has been implemented twice before, see §6. These earlier implementations were initial proofs-of-concept, with many limitations, pitfalls and challenges, which may explain why they have become unmaintainable and are no longer available. In contrast, the current implementation of offshoring in BER MetaOCaml is fully developed and mature. It is done from scratch, in complete re-design and re-thinking of the earlier implementations to clearly expose and address the challenges and ensure maintainability. To stress, not only the current implementation has no code in common with the earlier prototypes, but the very design and the underlying algorithms are also different. The implementation has been publicly available since 2018 (but privately, quite earlier) and used in several Master and Bachelor projects, among others, for generating performant OpenCL (GPGPU) [12] and OpenMP [13] code and for robot control code. The present paper is the first complete presentation of the fully developed offshoring.

1.1. Contributions

1. Explicate the challenges in generating low-level (C) code, many of which have only become clear from our experience;
2. Present the implementations of the two approaches, which are freely available and have been used in practice, see §7;
3. Describe how the two systems are designed to address or mitigate the challenges. The design and many algorithms (dealing with local §4.2 and mutable §4.6, §5.3 variables, extensibility §4.3, etc.) are novel and presented here for the first time.

In short, this paper is the first presentation of offshoring that has come of age, and of the matching tagless-final approach.

1.2. Challenges

There are two sorts of challenges we have encountered in designing and using assured C code generation: technical and engineering.

Technically, the metaphor 'simple OCaml as a notation for C' does not actually hold: see, for example, control operators such as `break`, `continue` and `goto`, as well as the general `for` and `do-while` loops, which have no analogue in OCaml (§4.4). Whereas local variables in OCaml may be introduced in any context, C does not permit variable declarations in expressions. The problem is particularly acute for expressions in conditional branches or loop conditions: see §4.2. More subtly, and hence insidiously, is the difference between variables in C and variables of reference types in OCaml – which can easily lead to the generation of type-correct but ill-behaving code (§4.6). These problems make the embedding of C difficult, and all but doom offshoring – it may seem.

Eventually, the problems have been overcome. One of the key ideas reverberating throughout the paper is to keep the eyes on the goal: expressing an algorithm in an efficient-to-execute way – which can be done in a *subset* of C. For an example, see the manually written high-performance C code in OpenBLAS,[3] which is syntactically spartan. After all, C, as many languages, is redundant, with many ways of expressing the same algorithm. Some expressions may be more elegant or idiomatic – but we only care about efficiency. Covering all of C and generating every its construct is hence explicitly not the goal. A subset suffices, which is easier to put in correspondence with a subset of OCaml.

It goes without saying that the covered subset of C must still be useful. Ours has proven useful, and used in several projects (§7). It is also the least restrictive offshoring implementation (see §6).

Among the engineering challenges is the unanticipated need for type inference (§4.1), and, mainly, extensibility, §4.3. If a system is not easy to extend, it falls into disuse. We should be able to support architecture-specific types of C and its extensions (SIMD, CUDA, OpenMP, etc.) and generate code that interacts with external libraries.

The challenges, first encountered in offshoring, also arise in tagless-final. Fortunately, the lessons learned in offshoring carry forward and help: see §5.3.

1.3. Structure of the Paper

The next section reminds the straightforward C generation, and the reasons one may quickly move beyond it. §3 describes the offshoring and §4 the challenges and how they have been addressed. The tagless-final embedding of C is presented in §5. Related work is discussed in §6. Section 7 briefly evaluates the usefulness and adequacy of the approaches.

The obvious question is how the two approaches presented in the paper compare; when to use one over the other. It is a rather nuanced question to answer, although some aspects are clear: e.g., the tagless-final approach can be realized in OCaml as is, without any extensions, and is overall more portable; quasi-quotes in MetaOCaml are syntactically more pleasing. We revisit the comparison question in §7.1, having expounded the approaches.

MetaOCaml (which includes offshoring) is available from Opam,[4] among other sources (the current version is N114). The complete code for all examples (including extra examples) and the tagless-final embedding are available at `http://okmij.org/ftp/meta-programming/tutorial/genc.html`

2. Prelude: Direct C Generation

What springs to mind when talking about code generation is directly emitting the code as strings. It also quickly becomes apparent why some abstractions and guarantees are desirable – as this section illustrates.

A notable example of directly emitting C code as strings is ATLAS [2]: a generator of automatically tuned linear algebra routines, which "is often the first or even only optimized BLAS implementation available on new systems

[3]`https://www.openblas.net/`
[4]`https://opam.ocaml.org/`

```
for (j=0; j < nu; j++) {
  for (i=0; i < mu; i++) {
    if (Asg1stC && !k)
      fprintf(fpout, "%s␣␣%s%d_%d␣=␣%s%d␣*␣%s%d;\n", spc, rC, i, j, rA, i, rB, j);
    else fprintf(fpout, "%s␣␣%s%d_%d␣+=␣%s%d␣*␣%s%d;\n", spc, rC, i, j, rA, i, rB, j);
    opfetch(fpout, spc, nfetch, rA, rB, pA, pB, mu, nu, offA, offB,
                       lda, ldb, mulA, mulB, rowA, rowB, &ia, &ib);
  } }
```

Figure 1: A snippet of ATLAS: generation of the inner loop body for matrix-matrix multiplication

and is a large improvement over the generic BLAS".[5] Fig. 1 shows a typical snippet of ATLAS code, itself written in C and using fprintf to generate C code. The variable spc is whitespace for indentation, and the variables rC, i, and j are combined to name identifiers declared elsewhere. Nothing guarantees that these identifiers are indeed all declared and the declarations are in scope – nor that the result is syntactically well-formed. Even if a fprintf output is syntactically correct, the presence of loops and branching in the generator makes it hard to see that the overall generated code will be too. It is also not at all obvious that we are indeed generating matrix multiplication code.

Incidentally, the implementor of ATLAS himself is quite frustrated:

> "As you have seen, this note and the protocols it describes have plenty of room for improvement. Now, as the end-user of this function, you may have a naturally strong and negative reaction to these crude mechanisms, tempting you to send messages decrying my lack of humanity, decency, and legal parentage to the atlas or developer mailing lists. . . . So, the proper bitch format involves
>
> - *First,* thanking me for spending time in hell getting things to their present crude state
> - *Then,* supplying your constructive ideas"
>
> (R. Clint Whaley: User contribution to ATLAS. Conclusion. 2012-07-10. math-atlas.sourceforge.net/devel/atlas_contrib/)

In his retrospective [14], Sheard summarized such lessons as: "The lack of internal structure [corresponding to the target language structure] is so serious that my advice to programmers is unequivocal: No serious meta-programmer should ever consider representing programs as strings."[14, §5].

What comes to mind next is a sort of an abstract syntax tree for C (several of which are available just in OCaml, see §6 for details). The generator then produces the tree data structure, which is pretty-printed into C code at the end. The pretty-printing ensures the result syntactically well-formed – and nothing

[5]https://en.wikipedia.org/wiki/Automatically_Tuned_Linear_Algebra_Software

more than that. One would like, however, further guarantees: at the very least, that the generated code compiles without errors and contains no problematic expressions, like a[i++] = b[i++].

The importance of having the generated code compile without errors may be hard to see. Ofenbeck et al. [15] sound a warning:

> While we were able to resolve the performance issues, we introduced new bugs ... [that] would manifest in errors such as:[6]
>
> ```
> forward reference extends over definition of value x1620
>
> [error] val x1343 = x1232(x1123, x1124, x1180, x1181,
> x1223, x1224, x1223, x1229, x1216, x1120, x1122, x1121)
> ```
>
> Note that variables are indexed in ascending order starting at zero, meaning that a large piece of code is processed before we hit this error. The root cause of bugs such as this one often proved to be very simple but heavily obfuscated in the code it manifested in.

We strive to avoid such problems altogether: if the generator successfully finished, the produced code must be free from scoping and typing errors. This is the subject of the rest of the paper.

3. Offshoring

As we have seen, emitting C code is better done not directly but through a level of abstraction that provides some guarantees. The selection of guarantees is an engineering decision, balancing against the ease of use and the implementation and maintenance effort.

One particularly attractive balance is offshoring [9]: treating a *subset* of OCaml as if it were a (non-canonical) notation for C. For example, consider the OCaml code for vector addition (a typical BLAS operation): the left column of the table below.

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

```
void addv(int n, int∗ vout, int∗ v1, int∗ v2) {
  for(int i=0; i≤n−1; i++)
    vout[i] = v1[i] + v2[i]; }
```

It is rather easy to imagine the C code it corresponds to (the right column). One may even argue [9] that OCaml's addv is C's addv, written in a different but easily relatable way. (As we shall see in §5.1, the similarity of the two addv pieces of code is not accidental.) 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. Offshoring hence turns homogeneous metaprogramming into heterogeneous.

[6]The generated code here is Scala.

The first premise of offshoring is the ability to convert well-typed OCaml code into C code that surely compiles, without errors. Eckhardt et al. [9] formalized the (rather restricted, see §6) OCaml subset-to-C translation and proved it type preserving; meaning preservation was not addressed. Bussone has formally (in Coq) shown the meaning preservation of the offshoring translation, in a yet unpublished work.[7] Kiselyov [16] demonstrated the type- and meaning-preservation of the translation between calculi representing subsets of OCaml and C with mutable variables.

The second premise is the ability to produce OCaml with some correctness guarantees. It is fulfilled by MetaOCaml [17, 18], which generates OCaml code that surely compiles – meaning, i.a., it is well-typed and has no unbound variables. The type soundness is demonstrated in Calcagno et al. [19], and the scope soundness is justified in [17]. We now illustrate how it is all put together, continuing the vector addition example.[8]

Definition 1 (Generating C via offshoring). Generating C via offshoring proceeds as:

1. implement the algorithm in OCaml
2. stage it – add staging annotations – and generate (possibly specialized) OCaml code
   (a) test the generated code
3. convert the generated OCaml code to C, saving it into a file
4. (a) compile and link the generated C code as ordinary C library code
   (b) compile the generated C code and (dynamically) link into an OCaml program, via an FFI such as [20].

Definition 1 presents the overall flow of offshoring. In our example, the first step has already been accomplished, by the OCaml addv above. The next step is to turn it into a generator of OCaml vector addition, with the help of MetaOCaml's *staging annotations* – specifically, so-called brackets .< and >.:

```
let addv_staged =
 .<fun n vout v1 v2 → for i=0 to n−1 do vout.(i) ← v1.(i) + v2.(i) done>.
```

Brackets enclose the code to generate and in this sense are analogous to string quotation marks '"'. However, brackets are not opaque: the enclosed code must be well-formed, moreover, well-typed OCaml code.

The result of evaluating addv_staged is

```
val addv_staged : (int → int array → int array → int array → unit) code =
 .<fun n_1 vout_2 v1_3 v2_4 →
   for i_5 = 0 to n_1 − 1 do
     Array.set vout_2 i_5 (Array.get v1_3 i_5 + Array.get v2_4 i_5)
   done>.
```

[7] Grègoire Bussone, private communication

[8] The complete code with tests and further examples is in `offshore_simple.ml` of the accompanying code.

It is a so-called code value: a value of the type t code that represents the generated OCaml expression, where t is its type. Code values can be printed. In our case, the printout shows the original addv after desugaring and renaming of all identifiers. (The generated code therefore has no variable name shadowing.) Such code value is the outcome of Step 2 of offshoring. Passing addv_staged to the function offshore_to_c, to be explained in detail later, accomplishes Step 3 and produces a .c file with the code[9]

```
void addv(int const n_1,int * const vout_2,int * const v1_3,int * const v2_4){
  for (int i_5 = 0; i_5 < n_1; i_5 += 1)
    (vout_2[i_5]) = (v1_3[i_5]) + (v2_4[i_5]);
}
```

which is the addv C code shown earlier. It is an ordinary C code and can be linked with any C or other program that needs vector addition. It can also be called from OCaml, via an FFI such as [20].

3.1. Real-life Example: Generating Optimized BLAS

Real-life use of offshoring is more interesting – because of the ability to specialize or otherwise optimize the generated code before offshoring. For example, suppose that the size of vectors to add is known in advance, and is small enough to unroll the loop in addv. The generator of unrolled code cannot be obtained from addv merely by placing brackets; quite a few other modifications are required. It helps to generalize addv first:[10]

```
let addvg n vout v1 v2 = iota n ▷ List.iter (fun i → vout.(i) ← v1.(i) + v2.(i))
```

where iota n generates a list of integers 0 through n−1 and List.iter performs a given action on each element of the list. We may stage it similarly to addv_staged, by enclosing everything in brackets:

```
let addvg_staged_full =
  .<fun n vout v1 v2 → iota n ▷ List.iter (fun i → vout.(i) ← v1.(i) + v2.(i))>.
```

Passing this code value to offshore_to_c, however, results in an exception: this code is outside of domain of offshoring. It is not hard to see why: the argument of List.iter is a first-class function, which cannot be simply represented in C. If the array size n is known at the generation time, a different placement of brackets becomes possible:

```
let addvg_staged n =
  .<fun vout v1 v2 → .~(iota n ▷ iter_seq (fun i → .<vout.(i) ← v1.(i) + v2.(i)>.))>.
```

where

```
let iter_seq f = List.map f ▷ seqs
let seqs : unit code list → unit code = fun l → reduce (fun x y → .<.~x; .~y>.) l
```

[9]We take care to put const modifiers: to clarify our intentions, to follow the current recommended C practices, and for reasons made clear in §4.6.

[10]The left-associative infix operator ▷ of low precedence is the inverse application. The related right-associative infix operator @@ of low precedence, appearing later, is application: x + 1 ▷ f is the same as f @@ x + 1 and is the same as f (x + 1) but avoids the parentheses.

Here, besides brackets we used the other staging annotation, .~ (pronounced 'escape') that denotes a hole in the code template (bracketed expression), to be filled by the code produced by the escaped expression. The function seqs builds code from a sequence of code values. Evaluating addvg_staged 4 gives the fully unrolled vector addition:

```
val addvg_staged4 : (int array → int array → int array → unit) code =
.<fun vout_11 v1_12 v2_13 →
   Array.set vout_11 0 ((Array.get v1_12 0) + (Array.get v2_13 0));
   Array.set vout_11 1 ((Array.get v1_12 1) + (Array.get v2_13 1));
   Array.set vout_11 2 ((Array.get v1_12 2) + (Array.get v2_13 2));
   Array.set vout_11 3 ((Array.get v1_12 3) + (Array.get v2_13 3))>.
```

MetaOCaml offers the facility to compile and run this code, so we can test it on sample 4-element arrays and compare the result with addv or addvg. Testing (see Step 2(a) of Defn. 1) is easier to do on the OCaml version of the code, before converting to C: the MetaOCaml run facility does not require a separately run compilation step and can be scripted in OCaml. Also, an out-of-bound indexing into an array results in an exception in OCaml rather than undefined behavior.

Once we are satisfied that the generated OCaml code works, we pass it to offshore_to_c obtaining:

```
void addv4(int ∗ const vout_11,int ∗ const v1_12,int ∗ const v2_13){
  (vout_11[0]) = (v1_12[0]) + (v2_13[0]);
  (vout_11[1]) = (v1_12[1]) + (v2_13[1]);
  (vout_11[2]) = (v1_12[2]) + (v2_13[2]);
  (vout_11[3]) = (v1_12[3]) + (v2_13[3]);
}
```

It is unsettling that addv can be easily staged and offshored to produce a for loop but not an unrolled loop; addvg is the other way around. By generalizing addvg a bit more:[11]

```
let addv_abs vout v1 v2 = zip_with add v1 v2 ▷ iter_assign vout
```

we not only express vector addition clearly but also, by appropriately instantiating iter_assign, zip_with and add combinators, may generate from the same expression a variety of programs: unrolled, not unrolled, or partially unrolled for-loop – and apply strip mining and scalar promotion optimizations. Let us illustrate it. (The complete code is in the file `addv.ml` accompanying the paper; for comparison, `tf_addv.ml` re-implements the same example using the tagless-final approach, §5, re-using the infrastructure.)

First, we adopt an abstract view of vectors – so-called 'pull vectors' pioneered by APL [22] – as a function from a finite domain of indices to the set of elements:

```
type ι lmad = {lwb: ι option; upe: ι; step: int}
type (ι,α) vec = Vec of ι lmad ∗ (ι → α)
```

The type ι is the type of the index, and α is the type of the elements. Here ι lmad is the description of the index domain: so-called linear memory access

[11]See Sections 4 and 5 of [21] for detailed explanation.

descriptor [23]. The field lwb is the (inclusive) lower bound (None means the default, 0), upe is the *exclusive* upper bound, step is the stride.

The operation zip_with f v1 v2 – applying a binary operation f to vectors v1 and v2 elementwise – may be generically defined as

```
let zip_with : (α → β → γ) → (ι,α) vec → (ι,β) vec → (ι,γ) vec =
  fun tf (Vec (n1,f1)) (Vec (n2,f2)) →
    assert( n1 == n2 );
    Vec (n1, fun i → tf (f1 i) (f2 i))
```

The lmad descriptors of the two vectors must be identical: the assert on the second line checks that, at the *generation* time.

We treat assignment also abstractly: a value of type α may be assigned to a location of type β if there exists a function of the type

```
type (β,α,ω) assign = β → α → ω
```

where ω represents the assignment (action), required to be a monoid: actions are composable. For example, a location to accept an assignment of the value of type α may be represented as α ref or as $\alpha \to$ unit. The assignment function ($\alpha\to$unit,α,unit) assign in the latter case is the function application (@@). A particular case of assignment is the assignment of vectors:

```
type (ι,β,α,ω) iter_assign = (ι, β) vec → (ι, α) vec → ω
```

With thus developed infrastructure, the general vector addition is expressed as mentioned above:

```
let addv_abs (add:α→α→α) (iter_assign : (ι,β,α,ω) iter_assign) :
    (ι,β) vec → (ι,α) vec → (ι,α) vec → ω =
  fun vout v1 v2 → zip_with add v1 v2 ▷ iter_assign vout
```

This general addv can be instantiated to add vectors, or to generate vector addition code. We are interested in the latter, and so chose the value domain to be float code, which represents expressions evaluating to a floating-point number. The addition operation

```
let addfc : float code → float code → float code = fun x y → .< .~x +. .~y >.
```

constructs the summation code. A particular instantiation of addv for code generation (assuming the length of arrays is not statically known) is

```
let addv_gen (iter_assign : (int code,float code → unit code,float code,unit code) iter_assign) :
  (int → float array → float array → float array → unit) code  =
  .<fun n vout v1 v2 →
    .~(let lmad = {lwb=None;upe= .<n>.;step=1} in
       addv_abs add:addfc iter_assign
      (Vec (lmad, vec_set .<vout>.)) (Vec (lmad, vec_get .<v1>.)) (Vec (lmad, vec_get .<v2>.)))
    >.
```

where

```
let vec_get : α array code → int code → α code = fun v i → .< Array.get .~v .~i >.
let vec_set : α array code → int code → α code → unit code = fun v i x → .< Array.set .~v .~i .~x >.
```

(One may easily design a more general instantiation of addv accommodating the possible static knowledge of the array length, or even of some arrays: see [21] for details.)

If the index ι is int – that is, describes values known at the generation time – ι lmad may be enumerated: converted to a list of the denoted indices:[12]

```
let iota : int lmad → int list = fun {lwb; upe; step} →
  let rec loop i = if i ≥ upe then [] else i :: loop (i+step) in
  loop (Option.value lwb ~default:0)
```

On the other hand, int code lmad cannot be converted to the list of statically (i.e., generation-time) known values. It still may be enumerated, but at the time of running the generated code:

```
let iota_dyn : int code lmad → (int code → unit code) → unit code =
  fun {lwb; upe; step} body →
    .<OffshoringIR.forloop .~(Option.value lwb ~default:.<0>.)
      ~step ~upe:.~upe @@ fun i → .~(body .<i>.)>.
```

Here OffshoringIR.forloop is a for-loop with a possibly non-unit stride (and the exclusive upper-bound) provided by the offshoring library, and explained in more detail in §4.4. Corresponding to the two lmad-enumerators are two combinators that lift element assignment to vector assignment:

```
let iter_assign_sta : (β,α,unit code) assign → (int, β, α, unit code) iter_assign =
  fun assign vout v →
  zip_with assign vout v ▷ function Vec (lmad,f) → iota lmad ▷ List.map f ▷ seqs

let iter_assign_dyn : (β,α,unit code) assign → (int code, β, α, unit code) iter_assign =
  fun assign vout v →
  zip_with assign vout v ▷ function Vec (lmad,f) → iota_dyn lmad f
```

Using the latter, the expression addv_gen (iter_assign_dyn (@@)) generates:

```
  val addv_c : (int → float array → float array → float array → unit) code = .<
  fun n_1 vout_2 v1_3 v2_4 →
        (OffshoringIR.forloop 0 ~upe:n_1 ~step:1)
          (fun i_5 → Array.set vout_2 i_5 ((Array.get v1_3 i_5) +. (Array.get v2_4 i_5)))>.
```

Applying Offshoring.offshore_to_c ~name:"addv" to the above OCaml code produces

```
void addv(int const n_1,double ∗ const vout_2,double ∗ const v1_3, double ∗ const v2_4){
  for (int i_5 = 0; i_5 < n_1; i_5 += 1)
    (vout_2[i_5]) = (v1_3[i_5]) + (v2_4[i_5]);
}
```

This is the addv we started §3 with and saw several times already.

There are other implementations of iter_assign. One is based on strip-mining: turning an $N \times 1$ column vector into a matrix of N/s rows and s columns, where s is the strip size. (N does not have to be evenly divisible by s, so we may have a remainder of a vector.)

```
val iter_assign_strip (strip:int) (assign: (β,α,unit code) assign)
   (iter_assign_inner: (int, β,α, unit code) iter_assign) : (int code, β, α, unit code) iter_assign
```

[12] ~default:l is the so-called named-argument: the feature of OCaml letting us name arguments, for clarity. Named arguments can be given in any order. All in all, Option.value lwb ~default:0 or Option.value ~default:0 lwb means the value of lwb, or 0 if it is None.

(See the complete code in addv.ml for the implementation.) It behaves like iter_assign_dyn assign for the remainder, and like iter_assign_dyn iter_assign_inner to assign rows of the matrix. The index type for the inner assignment is int rather than int code: the length of the rows, which is strip, is statically known. Using iter_assign_sta for the row assigner:

```
let addv_c1 = addv_gen (iter_assign_strip 4 (@@) (iter_assign_sta (@@)))
```

produces

```
val addv_c1 : (int → float array → float array → float array → unit) code = .<
  fun n_6 vout_7 v1_8 v2_9 →
    let t_11 = n_6 land (−4) in
    ();
    (OffshoringIR.forloop 0 ~upe:t_11 ~step:4)
      (fun i_13 →
         Array.set vout_7 (i_13 + 0)
           ((Array.get v1_8 (i_13 + 0)) +. (Array.get v2_9 (i_13 + 0)));
         Array.set vout_7 (i_13 + 1)
           ((Array.get v1_8 (i_13 + 1)) +. (Array.get v2_9 (i_13 + 1)));
         Array.set vout_7 (i_13 + 2)
           ((Array.get v1_8 (i_13 + 2)) +. (Array.get v2_9 (i_13 + 2)));
         Array.set vout_7 (i_13 + 3)
           ((Array.get v1_8 (i_13 + 3)) +. (Array.get v2_9 (i_13 + 3))));
    (OffshoringIR.forloop t_11 ~upe:n_6 ~step:1)
      (fun i_12 → Array.set vout_7 i_12 ((Array.get v1_8 i_12) +. (Array.get v2_9 i_12)))>.
```

which we can test and then offshore, using the same Offshoring.offshore_to_c ~name:"addv" operation:

```
void addv(int const n_6,double * const vout_7,double * const v1_8, double * const v2_9){
  int const t_11 = n_6 & −4;
  for (int i_13 = 0; i_13 < t_11; i_13 += 4){
    (vout_7[i_13 + 0]) = (v1_8[i_13 + 0]) + (v2_9[i_13 + 0]);
    (vout_7[i_13 + 1]) = (v1_8[i_13 + 1]) + (v2_9[i_13 + 1]);
    (vout_7[i_13 + 2]) = (v1_8[i_13 + 2]) + (v2_9[i_13 + 2]);
    (vout_7[i_13 + 3]) = (v1_8[i_13 + 3]) + (v2_9[i_13 + 3]);
  }
  for (int i_12 = t_11; i_12 < n_6; i_12 += 1)
    (vout_7[i_12]) = (v1_8[i_12]) + (v2_9[i_12]);
}
```

Compared to the simple addv earlier, the main loop is split in two (not counting the epilogue loop over the remainder): the outer one over strips, and the inner over the elements within a strip. The inner loop is then fully unrolled – which is how strip mining is often explained [1].

A row of a stripped matrix is a (int, float code) vec: a vector of known length whose elements are code fragments (float-producing expressions). The next optimization – scalar promotion – is to let-bind those pieces of code, and produce a vector whose elements (still of the type float code) are the names of those let-bound variables. Such a let-binding of vector elements is performed by

```
val scalar_bind : (int, α code) vec → ((int, α code) vec → ω code) → ω code
```

which is a 'lifted' version of (and is expressed in terms of) the let-binding combinator

```
val letl : α code → (α code → ω code) → ω code
```

provided by MetaOCaml. Using the scalar-promoted version of iter_assign_sta (which promotes the input vector before the assignment):

```
let iter_assign_sta_promote : (β,α,unit code) assign → (int, β, α, unit code) iter_assign =
  fun assign vout v → scalar_bind v @@ fun v' → iter_assign_sta assign vout v'
```

as

```
let addv_c2 = addv_gen (iter_assign_strip 4 (@@) (iter_assign_sta_promote (@@)))
```

produces

```
val addv_c2 : (int → float array → float array → float array → unit) code = .<
  fun n_14 vout_15 v1_16 v2_17 →
    let t_19 = n_14 land (−4) in
    ();
    (OffshoringIR.forloop 0 ~upe:t_19 ~step:4)
      (fun i_21 →
         let t_22 = (Array.get v1_16 (i_21 + 0)) +. (Array.get v2_17 (i_21 + 0)) in
         let t_23 = (Array.get v1_16 (i_21 + 1)) +. (Array.get v2_17 (i_21 + 1)) in
         let t_24 = (Array.get v1_16 (i_21 + 2)) +. (Array.get v2_17 (i_21 + 2)) in
         let t_25 = (Array.get v1_16 (i_21 + 3)) +. (Array.get v2_17 (i_21 + 3)) in
         Array.set vout_15 (i_21 + 0) t_22;
         Array.set vout_15 (i_21 + 1) t_23;
         Array.set vout_15 (i_21 + 2) t_24;
         Array.set vout_15 (i_21 + 3) t_25);
    (OffshoringIR.forloop t_19 ~upe:n_14 ~step:1)
      (fun i_20 → Array.set vout_15 i_20 ((Array.get v1_16 i_20) +. (Array.get v2_17 i_20)))>.
```

The let-bindings – the effect of scalar promotion – are clearly visible. Applying the same Offshoring.offshore_to_c ~name:"addv" gives the following C code, with the look and feel of the HPC BLAS:[13]

```
void addv(int const n_14,double * const vout_15,double * const v1_16, double * const v2_17){
  int const t_19 = n_14 & −4;
  for (int i_21 = 0; i_21 < t_19; i_21 += 4){
    double const t_22 = (v1_16[i_21 + 0]) + (v2_17[i_21 + 0]);
    double const t_23 = (v1_16[i_21 + 1]) + (v2_17[i_21 + 1]);
    double const t_24 = (v1_16[i_21 + 2]) + (v2_17[i_21 + 2]);
    double const t_25 = (v1_16[i_21 + 3]) + (v2_17[i_21 + 3]);
    (vout_15[i_21 + 0]) = t_22;
    (vout_15[i_21 + 1]) = t_23;
    (vout_15[i_21 + 2]) = t_24;
    (vout_15[i_21 + 3]) = t_25;
  }
  for (int i_20 = t_19; i_20 < n_14; i_20 += 1)
    (vout_15[i_20]) = (v1_16[i_20]) + (v2_17[i_20]);
}
```

[13] https://raw.githubusercontent.com/xianyi/OpenBLAS/develop/kernel/x86_64/daxpy.c

3.2. Discussion

Thus, generating C is generating OCaml using brackets and escapes, and then passing it to offshore_to_c to produce C code. MetaOCaml statically guarantees that the generated OCaml code is well-typed and well-scoped: see [19, 17], as was already mentioned at the beginning of §3. The translation to C also has been shown type-preserving (and, for a small subset with mutable variables and pointer types, meaning-preserving) [9, 16]. Well-formedness of the resulting C code is assured by first producing C AST and then pretty-printing it.

As we have seen for addvg_staged_full, offshoring applies only to a small imperative subset of OCaml. Therefore, offshoring is much simpler than an OCaml-to-C compiler, which must deal with the full language and support closures, tail recursion, GC, etc. None of this matters in offshoring, which hence produces C code that does not need any special run-time. Albeit simple, the offshorable subset of OCaml has proven to be adequate for numeric and embedded programming (see §7).

Although the offshorable subset is easy to define in theory (see [9]) it is hard to express in types, especially in the extant OCaml type system. Therefore, nothing actually prevents offshore_to_c from being applied to the code outside the supported subset – in which case it throws an exception. It is not a soundness problem: we still guarantee that the produced C code, *if* indeed successfully produced, to be well-formed and well-typed. The problem is that the offshoring exception is raised late, after the code to offshore has all been generated. MetaOCaml (OCaml, actually) supports location information, which could be used to emit detailed error messages (not implemented in the current version however). The best mitigation is to generate OCaml code not via brackets and escapes directly but via further abstraction layers (combinators) such as iter_assign in §3.1 – with the OCaml type system enforcing the abstraction.

4. Challenges

As attractive the metaphor of OCaml as C is, upon close inspection it breaks down, as the present section describes. Fortunately, it can eventually be held together – by workarounds and the design of the library that implements, enforces and steers towards the workarounds. As our refrain goes, the goal is to express an algorithm in efficient C code, not to generate idiomatic code and every C construct. The lessons we have learned in addressing the challenges carry over to the tagless-final approach: §5.3.

4.1. Type Inference

Looking closer at the OCaml addv code, §3, and the corresponding C code one notices that the correspondence is not as straightforward as one may have initially thought: the OCaml code mentions no types, whereas in C any declaration, of arguments and local variables, must be accompanied by their types. The need for type inference is an unpleasant surprise.

Fortunately, the MetaOCaml compiler is an extension of the OCaml compiler and hence may use the OCaml type checker to infer types in the code to offshore. The original implementation of offshoring [9] was hence integrated with the OCaml type checker. Since the type checker notably changes in every release of OCaml, the original offshoring almost immediately became unmaintainable and was removed when porting MetaOCaml to OCaml 3.12, which introduced especially many changes to the type checker.

The lesson was learned when resurrecting offshoring in BER MetaOCaml. The new offshoring is disentangled as much as possible from the OCaml type checker. The key ideas are two intermediary languages, rexp (Fig. 2) and OffshoringIR (Fig. 4). The former forms an abstraction barrier between internal OCaml data structures (typedtree, below) and the rest of offshoring. In terms of Defn. 1, Step 3 is hence split into four:

(i) type checking the generated OCaml code and obtaining the typedtree;
(ii) converting the typedtree to rexp;
(iii) converting the result to OffshoringIR;
(iv) and, finally, pretty-printing OffshoringIR as C code.

The latter two steps are to be explained in §4.2.

The function Runcode.typecheck_code of MetaOCaml performs Step (i): type checking the generated code. The result is the internal compiler data structure typedtree: type-annotated abstract syntax tree. Defined in the 800+-line file `typing/typedtree.mli` in OCaml distribution (as of OCaml 4.14.1), it is a huge, partly mutable and abstract data structure. Some parts have to be accessed via an API, which constantly changes. The type representation, in a separate file (`typing/types.mli`, 700+ lines), is also a large, mutable and partly abstract data structure, to be accessed via an unstable internal API. One has to be aware of many complexities of the OCaml type system, such as potentially recursive type aliases/abbreviations. There are several notions of type equality, and one has to know when to use which. Checking a type being int is not a simple pattern-match but requires a sequence of obscure and undocumented internal compiler function calls.

The language rexp, Figure 2, is a stable and easier to work with version of typedtree representing the imperative subset of OCaml that is subject to offshoring. It should be largely self-explanatory. We should point out the submodule OP that enumerates arithmetic, logical, array and reference operations, common to OCaml, C and other languages. Many operations are indexed by types: it generally does not matter for C, which uses overloading, but does matter, say, for WASM.

Continuing the example of vector addition from §3, the rexp expression for the body of addv_staged, repeated here

```
val addv_staged : (int → int array → int array → int array → unit) code =
 .<fun n_1 vout_2 v1_3 v2_4 →
   for i_5 = 0 to n_1 − 1 do
     Array.set vout_2 i_5 (Array.get v1_3 i_5 + Array.get v2_4 i_5)
   done>.
```

```
type numtyp = I32 | I64 | F32 | F64         (∗ Numeric types ∗)
type typ = ..                               (∗ Extensible type ∗)
type typ +=
  | TVoid                                   (∗ No values of that type ∗)
  | TNum of numtyp
  | TBool
  | TChar
  | TArray1 of typ                          (∗ Usual array or Bigarray.Array1 ∗)
  | TRef of typ
  | TString

module OP : sig                             (∗ Operations ∗)
  type t =
  | ADD of numtyp  | SUB of numtyp | MUL of numtyp  | DIV of numtyp | MOD of numtyp
  | . . .
  | EQ of numtyp   | NE of numtyp | LT of numtyp   | GT of numtyp | LE of numtyp | GE of numtyp
  | NEG of numtyp  | NOT | BNOT of numtyp
  | ASSIGN of typ | INCR of numtyp | DECR of numtyp
  | CAST of {from: numtyp; onto: numtyp} | Assert
  | DEREF of typ | REF of typ                (∗ typ = type of content  ∗)
  | Array1_get of typ | Array1_set of typ    (∗ typ = array element type ∗)
  | Other of varname
  val name : string → t
end

type varname = private string

type constant_t =                           (∗ no constants of void type! ∗)
  | Const_num of numtyp ∗ string            (∗ appropriately serialized ∗)
  | Const_bool of bool
  | Const_char of char
  | Const_string of string

type rexp =
  | Const of constant_t                   (∗ Constant/literal: int, bool,. . . ∗)
  | Array of rexp list                    (∗ Immediate array ∗)
  | LocalVar of varname ∗ typ             (∗ Locally—bound variable ∗)
  | KnownVar of OP.t                      (∗ defined in other modules/Stdlib ∗)
  | FunCall of OP.t ∗ rexp list           (∗ Calls only to known functions ∗)
  | Let of {id: varname; ty: typ; bind: rexp; body: rexp}
  | Cond of rexp ∗ rexp ∗ rexp            (∗ Conditional expression ∗)
  | If of rexp ∗ rexp ∗ rexp option       (∗ branches: unit type ∗)
  | Seq of rexp ∗ rexp
  | For of {id: varname; ty:typ; lwb: rexp; upe: rexp; step: rexp; body: rexp}
  | While of rexp ∗ rexp
  | Unit                                  (∗ empty statement ∗)
```

Figure 2: The intermediate language rexp: an easier to work with and stable version of typedtree

is[14]

```
For {id = "i_5"; ty = TNum I32;
  lwb = Const (Const_num (I32,"0")); upe = LocalVar ("n_1", TNum I32);
  step = Const (Const_num (I32,"1"));
  body =
   FunCall (OP.Array1_set (TNum I32),
    [LocalVar ("vout_2", TArray1 (TNum I32)); LocalVar ("i_5", (TNum I32));
     FunCall (OP.ADD I32,
      [FunCall (OP.Array1_get (TNum I32),
        [LocalVar ("v1_3", TArray1 (TNum I32)); LocalVar ("i_5", (TNum I32))]);
       FunCall (OP.Array1_get (TNum I32),
        [LocalVar ("v2_4", TArray1 (TNum I32)); LocalVar ("i_5", (TNum I32))])])])}
```

We see that in rexp, all identifier references and declarations are type-annotated, which makes it easy to produce C declarations later on. The local identifier names are all unique: courtesy of MetaOCaml. Therefore, no shadowing may occur – and identifier declarations may safely be lifted to a wider scope, which is sometimes necessary when emitting C, as described in §4.2.

The conversion from typedtree to rexp encapsulates all complexities of dealing with internal compiler data structures. As they change, only this conversion needs to be adjusted. The conversion is engineered to be an ordinary library function, outside the OCaml compiler and using only what is exposed in compiler-libs library.

Since rexp represents only an offshorable subset of typedtree, the conversion is partial, raising an exception if the input is outside the supported subset (e.g., contains higher-order functions, local function declarations, etc.) Besides mapping OCaml types to typ, function names to OP.t and typedtree expressions to rexp, the conversion desugars @@ and $\triangleright$ into ordinary function applications, and turns let _ = e1 in e2 and let () = e1 in e2 into sequencing e1; e2. An if-expression of OCaml is converted either into a Cond or If node of rexp depending on the type inferred for it and recorded in typedtree. The result is If for the conditional of type unit.

The for-loop of OCaml describes the iteration range by the smallest and the largest indices. In contrast, the language rexp uses the exclusive upper bound upe instead of the largest index. Since the index stride is always one in for ... to loop, upe is the largest index plus one. The translation makes such adjustment – along with the optimization: if the maximal index in an OCaml for-loop is e−1 (where e is some expression) as often is the case, then upe is set to (the converted) e. The above code exhibits such an adjustment. The language rexp has a provision for loops of non-unit stride, see §4.4.

Finally, the conversion uncurries function applications. As the result, applications like f () are translated to FunCall (OP.Other "f",[]) – that is, thunk invocations passing no arguments; applications like f 1 () are out of scope of offshoring. Although in OCaml the type unit is populated (by the value ()), the conversion maps it to TVoid, with no constants of that type. Therefore,

[14] The upper bound of the OCaml for-loop, n_1-1, is rendered as upe = LocalVar ("n_1", ...) in rexp. This adjustment is explained later in text.

OCaml's () is mapped to a special rexp form Unit, to be treated as an empty C block, as we see next.

4.2. Local Variables

However similar the subsets of OCaml and C may look in simple examples (like addv in §3), there is a profound difference between them, which becomes apparent as soon as we introduce local variables. OCaml is an expression language. The let-form that introduces a local variable is an expression and may appear in any expression context. To put it another way, whatever an expression, one may always introduce a local variable to name its sub-expression. Figure 3 shows a few examples.

```
1  let x = f () in x + 1
2  let y = (let x = f () in x + 1) in y + 2
3  (if test () then let x = f () in x + 1 else 3) + 4
4  while let x = f () in x + 1 > 5 do h () done
```

Figure 3: Examples of variable abstraction (local variables) in OCaml

On the other hand, C is a statement-oriented language, with sharp distinction of expressions and statements. Binding forms are not expressions and may not appear in expression context. (In original C, local variable bindings must be gathered at the beginning of a statement block. C99 permits local variable declarations be interspersed with statements.) Therefore, none of the examples in Figure 3 are straightforward to render in C. The easiest is `let x = f () in x + 1`, which, considered as a complete expression, can be offshored as the following C function body (we elide the function header for clarity):

```
int const x = f(); return (x+1);
```

The x-binding has to be lifted out of the return expression. For Fig. 3(line 2), we would have to write the following C code:

```
int const x = f(); int const y = x+1; return (y + 2);
```

Although the earlier `let x = f () in x + 1` appears as a sub-expression in Fig. 3 line 2, its offshoring is now different. The code on lines 3–4 of Fig. 3 is the most problematic: offshoring cannot leave the variable declarations where they are since they are not allowed in C conditional expressions or test-expressions in while-loops. We cannot lift them out either since their containing expression may be executed zero or multiple times. The best we can do is a mutable-variable conversion:

```
int x; return ((test () ? (x=f (), x + 1) : 3) + 4);
int x; while (x=f (), x + 1 > 5) h ();
```

Offshoring, hence, is more involved than it appeared, requiring a non-trivial handling of variable binding. The language rexp from §4.1 – in essence, a (type-annotated) abstract syntax of an imperative subset of OCaml – is still an expression language and cannot be easily mapped to C.[15] Further transformation – or, normalization – is needed, to lift or convert variable bindings and to distinguish statements from expressions – which is the subject of the present section. The result of the normalization is expressed in the language OffshoringIR: Figure 4. Unlike rexp, it is a statement-oriented language, designed to be translatable to C (or Fortran, LLVM, WASM, etc.) – in *full*, see Prop. 2, and easily.

OffshoringIR, Figure 4, is a typed, imperative language, distinguishing expressions exp and statements stmt and allowing variable bindings be associated only with a statement. A statement with its bindings is called a block. A complete function is represented by proc_t: a block with argument declarations args_t and the return type.

As should be apparent from Fig. 4, expression exp has no bindings. One may think of a statement stmt as a flow-chart element: an expression, branching, or iteration. A statement may also be a sequence of blocks Seq (b1,b2), where b1 is not an Exp statement with empty bindings, and neither b1 nor b2 are the empty (i.e., Unit) block. Each binding in a sequence of bindings represents an association of a newly introduced local variable with its initializing expression exp (to remind, the grammar of exp permits no embedded bindings). In addition, a TVoid expression (an assignment or a function call) is also treated as a binding, called pseudo-binding. Since proper bindings are also statements, a block overall is a sequence of statements.[16]

To present the translation from rexp succinctly, we introduce a more concise notation for OffshoringIR, Figure 5. For clarity, we elide KnownVar (used for global variables), for-loops (analogous to while-loops) and shortcut-evaluation expressions && and || (analogous to conditional expressions).[17] We elide immediate array expressions and limit function applications to binary applications. All variables and bindings are treated as ordinary OCaml or const-annotated C variables. Mutable variables, arising from bindings of the particular form $\underline{\text{let}}$ x = $\underline{\text{ref}}$ e $\underline{\text{in}}$ ... are discussed in §4.6. For now, we assume that such particular let-bindings are absent.

The translation from a rexp r to a block of OffshoringIR is written $\lceil r \rceil$ and presented in Fig. 7. It is clearly structurally recursive: has the form of a *fold* over the (the relevant subset of) rexp. In other words, it has the form of a non-standard evaluation of the rexp – reminding one of normalization-by-evaluation [24]. The translation is partial because the semantic functions $\epsilon(b)$ and $\sigma(b)$ are

[15]Strictly speaking, rexp cannot be fully mapped to C at all, since it includes non-offshorable expressions like (while ... do ... done; 1) + 2

[16]The complicated grammar of OffshoringIR comes from the fact that a void-function call in C may be adjoined to an expression using the comma operator, but an if- or loop- statement may not.

[17]In rexp, as in Typedtree, such expressions are represented as ordinary function applications, but of 'functions' with particular names.

```
type α seq                          (* Abstract sequence *)

type mutble = Mut | Cnst

type exp =
  | Const of constant_t             (* Constant/literal: int, bool,... *)
  | Array of exp list               (* Immediate array *)
  | LocalVar of varname             (* Locally−bound variable *)
  | MutVar of varname               (* Reference to a mutable var *)
  | GlobalVar of varname            (* Global/library function,... *)
  | FunCall of OP.t * exp list
  | Cond of exp * exps * exps       (* Conditional expression *)
  | And of exp * exps               (* shortcut eval: && *)
  | Or of exp * exps                (* shortcut eval: || *)
 and exps = exp list
 and stmt  =
  | Exp of exp
  | If of exp * block * block option
  | While of exps * block
  | For of {id: varname; ty:typ; lwb: exp; upe: exp; step: exp; body: block}
  | Seq of block * block
 and block  =
  | Unit
  | Block of binding seq * stmt
 and binding = bv_desc option * exp
 and bv_desc = {id: varname; ty: typ; mut: mutble}

type args_t = (varname * typ) list
type proc_t = args_t * typ * block        (* Complete procedure *)
val offshore : (module converters) → α code → proc_t    (* For the first argument, see §4.3 *)

type float32 = private float            (* See §4.3 *)
val float32_of_float : float → float32

(* For−loop with a step, like the for−loop in C: see §4.4 *)
val forloop : int → upe:int → step:int → (int → unit) → unit
```

Figure 4: The intermediate language (IR) of offshoring (defined in offshoringIR.mli). Types typ and constants constant_t are the same as those in Fig. 2.

Variables	x, y, z	
Constants	c	
Function names	f	
Expression	$e ::=$	$x \mid c \mid f\ e\ e \mid \text{cond}\ e\ \bar{e}\ \bar{e}$
Statement	$s ::=$	$e \mid \text{if}\ e\ b\ b \mid \text{while}\ \bar{e}\ b \mid b; b$
Block	$b ::=$	$() \mid \bar{v} \triangleright s$
Binding	$v ::=$	$(x, e) \mid e$

Figure 5: A more concise notation for (a representative subset of) OffshoringIR. Here $\bar{u}$ denotes a sequence of u with the empty sequence $\cdot$ and concatenation $+$. In Binding, (x, e) represents a proper binding, and just e pseudo-binding.

$$
\begin{array}{rlrl}
FV(x) &= \{x\} & FV(()) &= \varnothing \\
FV(c) &= \varnothing & FV(\bar{v} \rhd s) &= (FV(s) \backslash BV(\bar{v})) \cup FV(\bar{v}) \\
FV(f\ e_1\ e_2) &= FV(e_1) \cup FV(e_2) & FV(\cdot) &= \varnothing \\
FV(\text{cond}\ e\ \bar{e}_1\ \bar{e}_2) &= FV(e) \cup FV(\bar{e}_1) \cup FV(\bar{e}_2) & FV(\bar{v} + e) &= FV(e) \backslash BV(\bar{v}) \\
FV(\bar{e}) &= \cup_{e \in \bar{e}} FV(e) & FV(\bar{v} + (x, e)) &= FV(e) \backslash BV(\bar{v}) \\
FV(\text{if}\ e\ b_1\ b_2) &= FV(e) \cup FV(b_1) \cup FV(b_2) & BV(\cdot) &= \varnothing \\
FV(\text{while}\ \bar{e}\ b) &= FV(\bar{e}) \cup FV(b) & BV(\bar{v} + (x, e)) &= BV(\bar{v}) \cup \{x\} \\
FV(b_1; b_2) &= FV(b_1) \cup FV(b_2) & BV(\bar{v} + e) &= BV(\bar{v})
\end{array}
$$

Figure 6: Sets of free and bound variables: $FV(-)$ and $BV(-)$

partial. The former checks that a block has the form $\bar{v} \rhd e$ – an expression e with the associated bindings – and returns $(\bar{v}, e)$. The function is not defined for a Unit block or blocks with if- or while statements. The partiality of $\epsilon(b)$ reflects the fact that expressions like (while ... do ... done; 1) + 2 are not offshorable. The semantic function $\sigma(b)$ also verifies its block argument being of the form $\bar{v} \rhd e$, and further, that all bindings in $\bar{v}$ are pseudo-bindings. Such block is then a sequence of expressions, which $\sigma(b)$ returns. This function is hence undefined when the block has proper variable bindings. It is used in the context of a test-expression of a while loop or branches of a conditional expression: contexts where proper bindings cannot be easily lifted out. We do not do the mutable variable conversion at present and hence reject programs where it is needed, since such programs are rare and better re-written. We may re-visit this decision in the future.

The semantic function $\pi(b_1, b_2)$ builds a sequence of blocks respecting the invariant of Seq.

The translation is binding-preserving. To state this more formally, we introduce $BV(-)$ and $FV(-)$ for sets of bound and free variables, defined in Fig. 6. We call an rexp r locally closed if each LocalVar x in it occurs as a sub-expression in the body of the Let expression that binds that x.

Proposition 1. If r is a locally closed rexp with unique variable names then $FV(\lceil r \rceil)$ is empty.

The proof is the simple observation that if Let(x,r,r_b) includes LocalVar x in r_b, then x $\in FV(\lceil r_b \rceil)$ and x $\notin FV(\lceil \text{Let}(\text{x}, r, r_b) \rceil)$. The proposition and its proof imply that after the normalization all variable-use occurrences still refer to the same bindings. All variable names remain unique.

The formalization and proof of the stronger, and more desirable property of meaning preservation is out of scope. It may be ascertained by giving denotational semantics to rexp and OffshoringIR (i.e., writing an interpreter for both languages) and showing the translation meaning-preserving. Here we only remark that overall the translation is the re-association and 'straightening-out' of let-bindings, similar to the transformations in Sabry and Felleisen [25]. The side-conditions of the transformations are satisfied from the fact that all variable names are unique and hence their scopes can be extended without creating conflicts.

The function offshore, Fig. 4, implements the sub-steps (i)-(iii) of §4.1: it takes the closed code value produced by MetaOCaml, invokes the OCaml type checker to infer types, converts the resulting typedtree to rexp and then normalizes to OffshoringIR. (The function's first argument, converters, is explained in §4.3). It raises an exception if the input is outside the supported subset of OCaml.

Continuing the example of vector addition from §3, the invocation offshore (module DefaultConv) addv_staged produces the following IR code:

```
([("n_1", TNum I32); ("vout_2", TArray1 (TNum I32));
  ("v1_3", TArray1 (TNum I32)); ("v2_4", TArray1 (TNum I32))],
 TVoid,
 Block (<empty>,
  For {id = "i_5"; ty = TNum I32;
       lwb = Const (Const_num (I32, "0")); upe = LocalVar "n_1";
       step = Const (Const_num (I32, "1"));
    body =
     Block (<empty>,
      Exp (FunCall (OP.Array1_set (TNum I32),
         [LocalVar "vout_2"; LocalVar "i_5";
          FunCall (OP.ADD I32,
           [FunCall (OP.Array1_get (TNum I32), [LocalVar "v1_3"; LocalVar "i_5"]);
            FunCall (OP.Array1_get (TNum I32), [LocalVar "v2_4"; LocalVar "i_5"])])])))}))
```

Pretty-printing this OffshoringIR expression to C gives the code we have seen in §3. As another example, the code from Fig. 3 (line 2) is transformed to

```
Block (
 [(Some {id="x_1"; ty=TNum I32; mut=Cnst}, FunCall(OP.Other "f",[])),
  (Some {id="y_2"; ty=TNum I32; mut=Cnst},
      FunCall(OP.ADD I32,[LocalVar "x_1"; Const (Const_num (I32,"1"))]))
 ],
Exp (FunCall (OP.ADD I32, [LocalVar "y_2"; Const (Const_num (I32,"2"))])))
```

In either case, pretty-printing to C is straightforward. After all, OffshoringIR has been designed with this goal in mind. The actual C pretty-printing is done via an intermediary stage of mapping OffshoringIR to C AST.

Proposition 2. Each expression of OffshoringIR maps to a well-formed C program fragment. In particular, every proc_t expression maps to a well-formed C procedure.

The proposition can be verified by examining the mapping to C AST in the offshoring implementation. The proposition says nothing about type safety or the absence of unbound variables.

One may just as easily pretty-print the IR to other low-level imperative language, such as Fortran, LLVM IR, or WASM. Extensibility was another design decision behind the IR.

$$
\begin{aligned}
\lceil \mathsf{Const}(c) \rceil &= \cdot \triangleright c \\
\lceil \mathsf{Unit} \rceil &= () \\
\lceil \mathsf{LocalVar}(x) \rceil &= \cdot \triangleright x \\
\lceil \mathsf{FunCall}(f, r_1, r_2) \rceil &= \bar{v}_1 + \bar{v}_2 \triangleright f\ e_1\ e_2 \quad \text{where } \begin{array}{l} (\bar{v}_1, e_1) = \epsilon(\lceil r_1 \rceil) \\ (\bar{v}_2, e_2) = \epsilon(\lceil r_2 \rceil) \end{array} \\
\lceil \mathsf{Cond}(r, r_t, r_f) \rceil &= \bar{v} \triangleright \text{cond}\ e\ \sigma(\lceil r_t \rceil)\ \sigma(\lceil r_f \rceil) \quad \text{where } (\bar{v}, e) = \epsilon(\lceil r \rceil) \\
\lceil \mathsf{If}(r, r_t, r_f) \rceil &= \bar{v} \triangleright \text{if}\ e\ \lceil r_t \rceil\ \lceil r_f \rceil \quad \text{where } (\bar{v}, e) = \epsilon(\lceil r \rceil) \\
\lceil \mathsf{While}(r, r_b) \rceil &= \cdot \triangleright \text{while}\ \sigma(\lceil r \rceil)\ \lceil r_b \rceil \\
\lceil \mathsf{Let}(x, r, r_b) \rceil &= \begin{cases} \bar{v} \triangleright e & \text{if } (\bar{v}, e) = \epsilon(\lceil r \rceil),\ () = \lceil r_b \rceil \\ \bar{v} + (x, e) + \bar{v}_b \triangleright s & \text{if } (\bar{v}, e) = \epsilon(\lceil r \rceil),\ \bar{v}_b \triangleright s = \lceil r_b \rceil \end{cases} \\
\lceil \mathsf{Seq}(r_1, r_2) \rceil &= \pi(\lceil r_1 \rceil, \lceil r_2 \rceil)
\end{aligned}
$$

Semantic functions

$$
\begin{aligned}
\epsilon &: b \to \bar{v} \times e \\
\epsilon(\bar{v} \triangleright s) &= (\bar{v}, e) \quad \text{if } s = e \text{ (i.e., } s \text{ is an expression)} \\
\epsilon(b) &= \bot \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\sigma &: b \to \bar{e} \\
\sigma(\bar{v} \triangleright s) &= \bar{e} + e' \quad \text{if } s = e',\ \bar{v} = \bar{e} \text{ (all } \bar{v} \text{ are pseudo-bindings)} \\
\sigma(b) &= \bot \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\pi &: b \times b \to b \\
\pi((), b) &= b \\
\pi(b, ()) &= b \\
\pi(\bar{v}_1 \triangleright e, \bar{v}_2 \triangleright s) &= \bar{v}_1 + e + \bar{v}_2 \triangleright s \quad \text{all bindings in } \bar{v}_1 \text{ are pseudo-bindings} \\
\pi(b_1, b_2) &= \cdot \triangleright b_1; b_2 \quad \text{otherwise}
\end{aligned}
$$

Figure 7: Normalization algorithm: translation $\lceil - \rceil : \mathsf{rexp} \to b$ from rexp to OffshoringIR. To remind, $+$ is used to add an element to a sequence or to concatenate sequences.

4.3. Extensibility

From the very beginning of using offshoring it became clear that extensibility is the must. We should be able to accommodate the ever increasing assortment of integer and floating-point types of C as well as the vector types of various SIMD extensions. The generated C code often needs to interact with external libraries: therefore, we have to generate calls to library functions and deal with their data types. OpenCL/CUDA and OpenMP bring further challenges: generating pragmas, local and global annotations, and their own vector and scalar data types. All these extensions in the target of offshoring have to be representable in its source, i.e., OCaml. Looking back, every single project using offshoring required some sort of extension.

The original offshoring [9] was not extensible at all. Furthermore, it was part of the MetaOCaml *compiler*. Therefore, to make any extension – even to add a new library function – one had to modify a large code file that was part of (Meta)OCaml compiler. A mistake can crash the compiler or make it produce wrong code. After making the change, the entire compiler has to be recompiled. One may try to imagine what programming in C would be like if introducing a new external function required changing and then recompiling gcc.

The current implementation of offshoring is designed for extensibility. Not only it is now an ordinary library, which can be changed and recompiled independently of the MetaOCaml compiler. It is designed so that no recompilation should be needed. We illustrate using the example of offshoring the code to print an array of 32-bit floating-point numbers into a file. The example shows off calls to external library functions (FILE i/o of C) with their own data types (the pointer to the FILE structure), as well as dealing with 32-bit floating-point numbers with no equivalent in OCaml (OCaml float corresponds to double in C). The example is designed to answer some of the most frequently asked questions about offshoring. The complete code accompanies the paper: `offshore_ext.ml`. Fig. 8 shows the salient parts.

The first puzzle is how to generate OCaml code that represents calls to external C functions and uses their data types – the functions that are not generally callable from OCaml as they are. The answer is to define a module to represent that external library: lines 1-10 of Fig. 8. For the purpose of offshoring, all the library data types can be abstract and all functions dummy: we only need their signatures.[18] The type float32 (line 6) is the type of short floats, introduced by the OffshoringIR interface as a private alias to OCaml's float and translated to TNum F32 of OffshoringIR.[19]

With the module File_stub in scope we may generate code. Lines 12-19 show the result: the (well-typed) MetaOCaml code value. Passing it to offshore from Fig. 4 ends in an exception however: "unknown type: file". Indeed, the

[18]If one plans to run the generated code as OCaml as well, e.g., for testing, one needs the working implementation of File_stub. The complete code of the running example contains such an implementation, emulating C FILE i/o using OCaml i/o.

[19]The original OCaml float is translated to TNum F64.

offshoring library knows nothing about this data type. We need to tell it. First, we add to the IR types the new base type TFile (meant to correspond to FILE of C). As OffshoringIR.typ is an extensible data type, its extension is as simple as line 21. Defining the correspondence between the OCaml type File_stub.file and the just introduced TFile is the job of the converters module, whose implementation DefaultConv is provided by default. Lines 23-32 extend this module: line 25 maps File_stub.file to TRef TFile (so that File_stub.fopen matches fopen in the C standard library). Lines 28-29 specify that the names in the File_stub module (namespace) are to be understood as global identifiers. The offshoring library knows about the comparison operators $=$ and $\neq$ – but only when they are applied to numeric types. Our example compares pointers, and we have to tell the library (line 30) how to do that: for example, use NEQ (which can be defined as C macro). With thus set-up module Conv, offshoring succeeds and produces an IR program, which can then be straightforwardly pretty-printed to C (offshore_to_c combines the IR translation and pretty-printing). The result is shown at the end of Fig. 8.

The grouping of C library functions and types as an OCaml module, in the manner of File_stub, has an unexpected benefit: C gets a module system, which it never had.

4.4. Control Structures: Loops and Exits

One place where C and OCaml differ significantly is control structures. Although while loop has the same syntax and meaning in both languages, do-while has no analogue in OCaml. The for loop is present in both, but rather restricted in OCaml: the loop variable must be an integer, stepping only by one, up or down. Since loops with an arbitrary stride are common in HPC, the offshoring library offers a workaround. It defines the function:

```
let forloop : int → upe:int → step:int → (int → unit) → unit =
 fun lwb ~upe ~step body →
   let rec loop i = if i ≥ upe then () else (body i; loop (i+step))
   in loop lwb
```

It has two named arguments: step and upe; the latter is the exclusive upper bound. It is an ordinary OCaml function and can be used as is, in OCaml and in the generated code, e.g.:

```
let sum_ar = .<fun arr n →
  let sum = ref 0 in
  forloop 0 ~upe:n ~step:4 (fun i →
    for j=i to min (i+3) (n−1) do
      sum := !sum + arr.(j) done);
  !sum>.
```

Its applications, however, are translated to OffshoringIR by a special rule, which is better understood by looking at the result of translating the two nested loops in sum_ar:

```
For {id = "i_23"; ty = TNum I32; lwb = Const (Const_num (I32, "0"));
  upe = LocalVar "n_21"; step = Const (Const_num (I32, "4"));
  body = Block (<empty>,
module File_stub : sig
  type file                                  (* abstract *)
  val fnull : file                           (* null pointer FILE *)
  val fopen : string → string → file         (* standard C fopen, fclose *)
  val fclose : file → unit
  val write_f32 : file → float32 → unit      (* not in C standard library, but *)
  val write_delim : file → unit              (* assume as also available, e.g. as a C macro *)
 end = struct
   type file = unit    let fnull = ()   let fopen = fun _ _ → assert false  ...
end

let write_arr =  .<fun arr n fname →
    let fp = fopen fname "w" in
    assert (fp ≠ fnull);
    for i=0 to n−1 do
      if i > 0 then write_delim fp;
      write_f32 fp arr.(i)
    done;
    fclose fp>.

type typ += TFile

module Conv = struct include DefaultConv
  let type_conv : pathname → typ list → typ = fun pn args → match pn with
    | "File_stub.file"      → TRef TFile
    | _                     → DefaultConv.type_conv pn args
  let id_conv : pathname → string → string = fun pn v → match (pn,v) with
    | ("File_stub","fnull") → "NULL"
    | ("File_stub",v) → v
    | ("Stdlib","≠") → "NEQ"    (* non−numeric type comparison *)
    | _  → DefaultConv.id_conv pn v
end

let _ = Offshoring.offshore_to_c ~cnv:(module Conv) ~name:"writearr" write_arr

(* The code printed by offshore_to_c above: *)
void writearr(float * const arr_1,int const n_2,char * const fname_3){
  FILE * const fp_4 = fopen(fname_3,"w");
  assert(NEQ(fp_4,NULL));
  for (int i_5 = 0; i_5 < n_2; i_5 += 1){
    if (i_5 > 0)
      write_delim(fp_4);
    write_f32(fp_4,arr_1[i_5]);
  }
  fclose(fp_4);
}
```

Figure 8: Extensibility example, slightly abbreviated

```
    For {id = "j_24"; ty = TNum I32; lwb = LocalVar "i_23";
      step = Const (Const_num (I32, "1"));
      upe =
       FunCall (OP.ADD I32,
        [Const (Const_num (I32, "1"));
         FunCall (OP.Other "min",
          [FunCall (OP.ADD I32, [LocalVar "i_23"; Const (Const_num (I32, "3"))]);
           FunCall (OP.SUB I32, [LocalVar "n_21"; Const (Const_num (I32, "1"))])])]);
      body = Block (<empty>,
        Exp (FunCall (OP.ASSIGN (TNum I32),
           [MutVar "sum_22";
            FunCall (OP.ADD I32,
             [FunCall (OP.DEREF (TNum I32), [MutVar "sum_22"]);
              FunCall (OP.Array1_get (TNum I32), [LocalVar "arr_20"; LocalVar "j_24"])])])))}}
```

Pretty-printing the result as C for-loops is straightforward:

```
int sum_ar(int * const arr_20,int const n_21){
  int sum_22 = 0;
  for (int i_23 = 0; i_23 < n_21; i_23 += 4)
    for (int j_24 = i_23; j_24 < (1 + min(i_23 + 3,n_21 − 1)); j_24 += 1)
    sum_22 = sum_22 + (arr_20[j_24]);
  return sum_22;
}
```

The do-while can be supported similarly.

In general, complex control structures may be represented in OCaml as call-by-name functions – that is, functions taking explicit thunks as arguments. Offshoring will then map such function applications, unwrapping thunks, to invocations of appropriately defined C macros, taking advantage of the fact that macro applications look like function applications.

C also has break, continue, return and goto. In principle, one may define dummy OCaml 'functions' like break : unit→unit, whose applications are pretty-printed as C in a special way. The code with those functions can only be offshored, not executed as OCaml. Mainly, nothing prevents using such 'functions' outside loop bodies, hence breaking the guarantee that the result of offshoring always compiles. A better idea is to introduce iteration combinators with an early exit, similar to forloop.

4.5. Other Challenges

The code to offshore must be of the form .<fun *args* → *body*>., with no local function declarations. Therefore, recursive OCaml functions are out of scope: the generated code should use loops rather than recursion. That does not mean that we have to burden the end user with transforming recursion into iteration. Instead, we should offer the end user higher-level combinators, something like iter_assign in §3, representing the needed processing patterns. We should take the full advantage of OCaml as the Meta Language – the purpose for which the OCaml predecessor was initially introduced [26].

Offshoring does not cover the whole of C either, in particular, pointer arithmetic. Pointer arithmetic is an example of redundancy in C since it can be represented with array indexing, without any loss of performance. After all, unrestricted pointer arithmetic is not allowed in modern C either: a pointer and that pointer after integer addition/subtraction must point to the *same* array or one element past the array. Otherwise, the behavior is undefined.[20] Furthermore, the C standard itself defines pointer arithmetic in terms of arrays indexing. HPC code typically uses array indexing rather than pointer arithmetic: perhaps an influence of the still widely used Fortran, which has no pointer arithmetic.

4.6. Pointers and References

The metaphor of OCaml as C is strained the most when it comes to mutable variables. Compare the following OCaml and C code

```
let exr1 = fun y → let x = ref 0 in x := 1; incr x; !x + y

int exr1(int const y) {int x = 0; x = 1; x++; return (x+y);}
```

In OCaml, all variables are variables in the sense of lambda calculus variables: they stand for values, they are substitutable with values, they are first-class. Reference-type variables like x stand for values which are memory cells, whose content can be accessed and modified using '!', ':=', incr operations, which are ordinary functions. In contrast, mutable variables in C are not substitutable and not first-class: i.e., they are not variables in the sense used in algebra and logic. Mentioning of a mutable variable in an expression like x+y or passing it to a function like foo(x) passes the content of the mutable cell associated with the variable, but not the mutable cell itself. Explicit dereference is not needed; on the other hand, assignment and increment are not ordinary functions but rather special forms whose semantics is complicated and requires the concept of so-called L-values. What further complicates the picture is that C also has an analogue of OCaml variables (like y in exr1) and pointer types.

Nevertheless, the two exr1 pieces of code show the very close similarity, which suggests that let (x: t ref) = ref e in... is to be offshored as t x = e'; ... and x := e in OCaml as x = e' in C (where e' is the offshored e), and further the OCaml !x as just x in C. This is basically the translation proposed by Eckhardt et al. [9], to be referred to as 'simple offshoring' below. It clearly expresses the idea that an OCaml variable bound to a value of reference type is a model of a mutable C variable.

The model breaks however, upon closer inspection. Consider the OCaml expression !x + 1, where x is a variable of int ref type. The simple offshoring suggests x + 1 as C translation. Indeed, in a bigger context, the following OCaml and C code

```
let exr21 () = let x = ref 0 in !x + 1
int exr21() { int x = 0; return x+1; }
```

clearly correspond. Let us put the same !x + 1 in a different context, however:

```
let exr22 x = !x + 1
```

The corresponding C code is then

[20]ISO/IEC 9899:2011, Sec 6.5.6 "Additive Operators"

```
int exr22(int * const x) { return *x + 1; }
```

The very same !x + 1 is now translated differently.

Here is an even more difficult example for simple offshoring:

```
let exr3 () = let x = ref 0 in let y = x in y := 42; !x + !y
```

The problem is translating let (y:int ref) = x in .... Clearly we cannot translate it as int y = x; as it makes y a fresh mutable variable initialized with the current value of x but mutated independently. In OCaml exr3, however, y is an alias of x, substitutable with x. As mentioned earlier, C has pointer types and the analogue of OCaml variables. Using them we can offshore exr3 as

```
int exr31(){ int temp = 0; int * const x = &temp; int * const y = x;
             *y = 42; return *x + *y; }
```

Although semantically correct, it is not satisfactory since OCaml's x is represented as *two* variables in C, temp and x, and requiring twice as much memory. This translation essentially gives up on representing mutable variables in OCaml.

Simple offshoring can be saved by imposing restrictions that outlaw all OCaml code that simple offshoring has trouble with – which is what Eckhardt et al. [9] do, albeit silently. The paper mentions no restrictions. However, if we carefully examine the typing rules in its Appendix A2, we discover that the restrictions are imposed after all: in a type t ref, t must be a base type; the right-hand side of a let-expression must not be an expression of a reference type, with the sole exception of ref e. Function arguments of reference types are also out of question.

Until very recently, BER MetaOCaml used simple offshoring with its restrictions. Although proved to be more or less adequate for numeric code, the severity of the restrictions (e.g., precluding ref-type function arguments or mutable references of array or pointer types) has been felt at times, necessitating workarounds and fiddling with the generator.

The problem has been systematically investigated in [16],[21] which proposed the translation free from any restrictions on types and occurrences of variables, and maintaining the 'look and feel' of mutable C variables in OCaml. The problem was distilled to the fact that mutable C variables of int type and const variables of int * type are represented in OCaml the same way: as variables of int ref type. Thus translation from C to OCaml is non-injective, and hence the inverse mapping, from OCaml to C (i.e., offshoring) does not exist. The resolution is to annotate OCaml variables of reference types as mutable or constant. The annotation is inferred simply: the variables bound by let-expressions of the form let x = ref e in ... are to be annotated as mutable; the rest are constant. The offshoring translation is hence (see [16] for details, derivation, formal properties and justification):

[21] and further, in https://okmij.org/ftp/meta-programming/mutable-var.html

- let (x : t ref) = ref e in ... in OCaml is offshored as t x = e;... in C, with x marked as mutable;
- Any other let (x:t) = e in ... in OCaml is offshored as t const x = e; ..., with x marked as constant;
- A usage occurrence of x is offshored as &x if x is mutable; or just x otherwise;
- The dereference operator '!' of OCaml is translated to the equivalent '*' operator of C;
- Assignment e1 := e2 of OCaml is translated as *e1' = e2' in C (where e1' and e2' are the translations of e1 and e2, resp.)
- As syntax sugar, we prettier-print *&x as x, although it is valid C and could be left as it is.

This translation is implemented in MetaOCaml N114. As an example, it offshores exr21 and exr22 as expected (described earlier); exr3, repeated below

```
let exr3 () = let x = ref 0 in let y = x in y := 42; !x + !y
```

is offshored to

```
int exr32(){
  int x_1 = 0;
  int * const y_2 = &x_1;
  ( *y_2) = 42;
  return (x_1 + ( *y_2));
}
```

What feels like a mutable variable in OCaml's exr3 is represented as the mutable variable x_1 in C without further ado (contrast with exr31). Finally, we can write idiomatic OCaml code and have it mapped to just as idiomatic C code – without any restrictions or hidden overhead.

5. Tagless-final Embedding

A different way to generate C is to embed it in OCaml, in tagless-final style [10, 11]. This embedding is emphatically different from the mere representation of C AST in OCaml (§2, §6): the latter gives no assurances about well-typedness, absence of unbound variables or unexpected shadowing. Tagless-final embedding makes such assurances and hence guarantees that the generated C code compiles without errors. We discuss the guarantees more formally in §5.2.

Tagless-final approach may seem orthogonal to offshoring: whereas the latter is akin to embedding a language using quotes/templates, tagless-final is embedding using combinators [27]. Nevertheless, the two approaches converge in the end, see §5.4. The challenges of offshoring – type inference, local variables, control structures, mutable variables, etc. – detailed in §4 confront tagless-final as well. The lessons learned in addressing those challenges help here, too.

5.1. Tagless-final by Example

In the tagless-final style, an embedded language is represented as a (multi-sorted) algebra whose operations are the syntactic forms of the language. Let's take an example: the same example of vector addition that we used when introducing offshoring in §3, reproduced below for ease of reference.

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

The minimal language to write such code can be described as follows.

```
module type cde = sig
  type α exp                          (* Abstract type of an expression *)

  val int : int → int exp
  val ( + ) : int exp → int exp → int exp
  val ( − ) : int exp → int exp → int exp

  type α stm                          (* A statement *)
  val for_ : int exp → int exp → (int exp → unit stm) → unit stm

  type α arr                          (* An array (variable) *)
  val array_get : α arr → int exp → α exp
  val array_set : α arr → int exp → α exp → unit stm

  type α proc_t                      (* The complete function *)
  val mk3arr :
    (int exp → int arr → int arr → int arr → unit stm) →
    (int → int array → int array → int array → unit) proc_t
end
```

It is a typed first-order statement-oriented DSL embedded in OCaml. OCaml values of the type `t exp` represent DSL expressions of type `t`; OCaml values of the type `t stm` represent statements (which may return the result unless `t` is `unit`). Expressions are built of integer literals and variables (at present, arguments) using addition, subtraction and array dereference operations. Statements are the for-loop and array element assignment. The operation `mk3arr` builds the function header and provides arguments to use in the function body. Granted, it is too specific, but can be generalized, see §5.3. The definitions of these DSL constructors (DSL expression forms) are collected as an OCaml signature.

Using the operations of the signature, the vector addition takes the form:

```
let addv_dsl = mk3arr @@ fun n vout v1 v2 →
  for_ (int 0) (n − int 1) @@ fun i →
    array_set vout i @@ array_get v1 i + array_get v2 i
```

which is quite like the OCaml `addv` we started with, after desugaring of array operators (cf. particularly the output of `add_staged` in §3).

One may implement `cde` in many ways. The simplest is to relate each DSL operation with the corresponding OCaml operation: meta-circular interpreter.

```
module OCde = struct
  type α exp = unit → α
  let int x = fun () → x
  let ( + ) x y = fun () → Stdlib.( + ) (x ()) (y ())
  let ( − ) x y = fun () → Stdlib.( − ) (x ()) (y ())

  type α stm = unit → α
  let for_ lwb upb body = fun () → for i = lwb () to upb () do body (fun () → i) () done

  type α arr = α array                    (* An array (variable) *)
  let array_get a x = fun () → Array.get a (x ())
  let array_set a i v = fun () → Array.set a (i ()) (v ())

  type α proc_t = α                      (* The complete function *)
  let mk3arr :
    (int exp → int arr → int arr → int arr → unit stm) →
    (int → int array → int array → int array → unit) proc_t = fun body →
      fun n vout v1 v2 → body (fun () → n) vout v1 v2 ()
end
```

DSL expressions map to OCaml expressions: to be precise, DSL expressions are represented as OCaml thunk values. Thunks for proper sequencing of imperative updates were first proposed and explained by Landin in 1965 [28]. Since DSL functions are realized as OCaml functions, they can be immediately applied to sample data, which is useful for testing. The significance of OCde to state and prove assurances is explained in §5.2.

Tagless-final permits, even encourages, multiple DSL implementations. For example, one may also implement cde to generate OCaml code, with the help of MetaOCaml brackets and escapes:

```
module MOCde = struct
  type α exp = α code

  let int : int → int exp = fun x → .<x>.
  let ( + ) : int exp → int exp → int exp = fun x y → .<.~x + .~y>.
  let ( − ) : int exp → int exp → int exp = fun x y → .<.~x − .~y>.

  type α stm = α code
  let for_ lwb upb body = .<for i= .~lwb to .~upb do .~(body .<i>.) done>.

  type α arr = α array code
  let array_get a i = .< (.~a).(.~i) >.
  let array_set a i v = .< (.~a).(.~i) ← .~v >.

  type α proc_t = α code                    (* The complete function *)
  let mk3arr body = .<fun n vout v1 v2 → .~(body .<n>. .<vout>. .<v1>. .<v2>.)>.
end
```

With this implementation, addv_dsl gives *exactly* the same result as addv_staged in §3: the OCaml code for vector addition.

A related implementation generates C code:

```
module CCde  = struct
  type α exp = string

  let int = string_of_int
  let ( + ) = Printf.sprintf "(%s␣+␣%s)"
  let ( − ) = Printf.sprintf "(%s␣−␣%s)"
  type α stm = string
  let for_ lwb upb body =
    let i = gensym "i" in
    "for(int␣" ^ i ^ "=" ^ lwb ^ ";␣" ^ i ^ "≤" ^ upb ^ ";␣" ^ i ^ "++){\n" ^
    "␣␣" ^ body i ^ "}"

  type α arr = string
  let array_get = Printf.sprintf "%s[%s]"
  let array_set = Printf.sprintf "%s[%s]=%s;"

  type α proc_t = string                        (* The complete function *)
  let mk3arr body =
    let n   = gensym "n"  in let vout = gensym "vout" in
    let v1  = gensym "v1" in let v2   = gensym "v2" in
    Printf.sprintf
      "void␣addv(int␣const␣%s,␣int*␣const␣%s,␣int*␣const␣%s,␣int*␣const␣%s){\n%s\n}"
      n vout v1 v2 (body n vout v1 v2)
end
```

With this implementation of the DSL, the same addv_dsl produces the string

```
void addv(int const n_1, int* const vout_2, int* const v1_3, int* const v2_4){
for(int i_5=0; i_5≤(n_1 − 1); i_5++){
  vout_2[i_5]=(v1_3[i_5] + v2_4[i_5]);}
}
```

which is *exactly* the C vector addition code, obtained in §3 via offshoring. We started §3 by noticing that the vector addition code in OCaml and C look very similar. Now we see why: they are different interpretations of the same DSL expression. Speaking algebraically, they are the images of the same (initial algebra) term addv_dsl upon different homomorphisms. One may write many more similar implementations of the cde signature, to obtain code in Fortran, WASM, LLVM IR, etc.

In §3.1 we have generalized the simple addv first to unroll the loop, and then to perform strip mining and scalar promotion, generating OpenBLAS-like code. All of this is easily possible in the tagless-final style as well, which is demonstrated in the accompanying code: `tf_addv.ml` is the re-write of `addv.ml` in the tagless-final style. The generated C code is identical.

The tagless-final DSL used in [29] is more extended, with full integer, floating-point and boolean operations. We also added a partial-evaluation layer for online partial-evaluation of any cde implementation, be it OCaml or C or Scala.

5.2. Formalities

We now state the assurances provided by the tagless-final approach more formally, indicating how they can be demonstrated.

The signature cde defined our DSL: its syntactic categories – expression, statement, array, procedure – and the operations to construct them. The DSL is typed: the categories are indexed by types; the signatures of DSL operations express the typing rules. For example,

```
val for_ : int exp → int exp → (int exp → unit stm) → unit stm
```

represents the typing rule (in natural deduction style):

$$\frac{\vdash_{exp} e_1 : \mathsf{int} \qquad \vdash_{exp} e_2 : \mathsf{int} \qquad \begin{array}{c}[\vdash_{exp} i : \mathsf{int}]\\ \vdots \\ \vdash_{stm} b : \mathsf{unit}\end{array}}{\vdash_{stm} \mathsf{for}\ e_1\ e_2\ b : \mathsf{unit}}$$

The DSL is embedded in OCaml: OCaml expressions of the unit stm type stand for DSL statements, for example

```
for_ (int 0) (int 10) (fun x → array_set a x (int 0))
```

(where a is a variable of int arr type in scope) – and similar for t exp, t arr, t proc_t expressions. One has to be aware that not all unit stm OCaml expressions correspond to DSL statements: e.g.,

```
for_ (int 0) (int 10) (fun x → raise Not_found)
```

does not – so-called 'exotic term' (see [14, §13] for detailed discussion).

Let us delineate the subset of OCaml expressions that correspond to DSL terms: call it $\mathcal{D}_\tau$. It is defined as a set of well-typed, closed, normal OCaml expressions of type τ that are built using *only* abstraction, application, int n and the other operations of the cde signature. The type τ is not arbitrary either: see Fig. 9. The exotic term above is not in $\mathcal{D}_\tau$ because raise is not defined in the cde signature. OCaml expressions $\mathcal{D}_\sigma$ represent closed DSL terms (expressions, statements, etc.) whereas $\mathcal{D}_\tau$ expressions where τ is $\iota_1 \to \iota_2 \to \ldots \to \sigma$ stand for open DSL terms, with the variables of type ι_1, ι_2, etc.

DSL base types	b ::=	int
DSL statement types	s ::=	b | unit
DSL proc types	p ::=	s | $b \to p$ | b array $\to p$
Representation types	σ ::=	b exp | s stm | b arr | p proc_t
Variable types	ι ::=	b exp | b arr
Open term types	τ ::=	σ | $\iota \to \tau$

Figure 9: DSL (object) types and representation types

An implementation of the cde signature may then be regarded as a denotational semantics of the DSL: a collection of semantic functions that *compositionally* map DSL expressions to closed OCaml values. This style of denotational semantics is explained in detail in [30]. Here we give a short overview.

To give the meaning to DSL types, we introduce semantic domains: Exp_b (the set of meanings of DSL expressions of type b), Stm_s (meanings of DSL statements), Arr_b (for DSL arrays) and Proc_p for DSL procedures. Particular implementations of the cde signature (particular denotational semantics) define the content of these sets and the mapping from a term in $\mathcal{D}_\tau$ to their element. If the term has free variables, its meaning depends on the meanings assigned to them. This assignment is called valuation. The type system of DSL is called sound (with respect to the semantics) if every well-typed term has a meaning, that is included in the meaning of its type.

Definition 2 (Soundness). For every $d \in \mathcal{D}_\tau$ where $\tau = \iota_1 \to \ldots \to \iota_n \to \sigma$ ($n \geq 0$) and any valuation of its free variables if any, the meaning of d is in Exp_b (if σ is of the form b exp), Stm_s (for $\sigma = s$ stm), Arr_b (for $\sigma = b$ arr) or Proc_p (for $\sigma = p$ proc_t)

The implementation OCde defines the semantic domains as:

Exp_b	: OCaml values of type unit$\to b$
Stm_s	: OCaml values of type unit$\to s$
Arr_b	: OCaml arrays of type b array
Proc_p	: OCaml functions of type p

By inspection of OCde we confirm that Soundness holds for OCde: the type system of the cde DSL is sound. The inspection of OCde is particularly simple: all its semantic functions are patently total, and OCde is well-typed in OCaml.

The implementation MOCde is another way of giving meaning to DSL expressions in terms of (Meta)OCaml. The semantic domains are:

Exp_b	: b code values
Stm_s	: s code values
Arr_b	: b array code values
Proc_p	: p code values

Again, by simple inspection of MOCde we confirm that Soundness holds. In particular, any $\mathcal{D}_\sigma$ term where $\sigma = p$ proc_t maps to a p code value, and, further, to a closed and well-typed OCaml expression of type p. (The second inference relies on the soundness of MetaOCaml: a value of type t code corresponds to a well-typed and well-scoped OCaml expression of type t). In other words, any closed p proc_t DSL term generates a well-typed OCaml function with no unbound variables. One may further show that the OCaml expression generated by MOCde for a p proc_t term has the same meaning as given by OCde for that term – relying on the semantics of the multi-staged calculus underlying OCaml developed by Taha [31].

Consider now the CCDe implementation, whose semantic domains are[22]

Exp_b	: C expressions of type b
Stm_s	: C statements returning the value of type s (unless s is unit)
Arr_b	: C variables of the type b*
Proc_p	: C procedures of the type corresponding to p

We likewise observe that CCde is sound, mapping each $\mathcal{D}_\tau$ term to a C program fragment. As a consequence, a closed p proc_t DSL term is mapped to a closed and well-typed C procedure. The key observation is that all semantic functions are total and that the types α exp, etc. are all abstract, justifying the induction principle. We also check, e.g., that statement generators produce code ending

[22]In cde, only unit stm may be constructed; consequently, there is no provision for return. The extended signature cdemut in §5.3 adds return, and the claim becomes more meaningful.

in a closing brace or a semicolon.[23]

The C code is generated in CCde via printf – just as in ATLAS, Fig. 1. There is a drastic difference however: printf statements in CCde are encapsulated in that module and not available for, or even visible to the end user. The end user only uses the combinators of the cde signature, as in addv_dsl.

All in all, any closed proc_t term in $\mathcal{D}$ corresponds to a well-formed and well-typed procedure code, in OCaml or C, depending on the instance of cde. We are statically assured of it.

5.3. Challenges: Type Inference and Mutable Variables

There are also complications, not unlike the ones described for offshoring. The lessons learned there carry forward. Let us illustrate, on the lessons of type inference §4.1 and mutable variables §4.6. Our running example will be summing up a float array. To be able to write it, we extend the DSL – extensibility is the characteristic of tagless-final – with floating-point numbers and their operations, and then mutable variables, sequencing of statements, returning of the result:[24]

```
module type cdemut = sig
  include cde

  val float  : float → float exp
  val ( +. ) : float exp → float exp → float exp

  type α mut                      (* type of mutable variables *)
  val (let*) : α exp → (α mut → ω stm) → ω stm
  val dref : α mut → α exp
  val (:=) : α mut → α exp → unit stm

  val (@.) : unit stm → α stm → α stm (* sequencing *)

  val ret : α exp → α stm

  val mkfun     : ?name:string → α stm → α proc_t
  val arg_base  : ?name:string →  (α exp → β proc_t) → (α → β) proc_t
  val arg_array : ?name:string → (α arr → β proc_t) → (α array → β) proc_t
end
```

We have also introduced the general way to generate C functions of arbitrary number of base- and array-type arguments. The optional argument ?name:string is the hint for the name of the generated function or the argument. As an example, the mk3arr in §5.1 may now be written as

```
let mk3arr body =
  arg_base ~name:"n" @@ fun n →
  arg_array ~name:"vout" @@ fun vout →
  arg_array ~name:"v1" @@ fun v1 →
  arg_array ~name:"v2" @@ fun v2 →
  mkfun ~name:"addv" @@ body n vout v1 v2
```

[23]Our actual implementation uses C AST for C code generation, which ensures well-formedness. C AST does not ensure well-typedness however; but the tagless-final encapsulation does.

[24]let* is a so-called let-operator added in recent OCaml.

With so extended cdemut signature, the array summation code takes the expected form

```
let vsum =
  arg_base ~name:"n" @@ fun n → arg_array ~name:"v" @@ fun v → mkfun ~name:"sumv" @@
  let∗ sum = float 0. in
  begin for_ (int 0) (n − int 1) @@ fun i →
    sum := dref sum +. array_get v i end
  @.
  ret (dref sum)
```

One can see from the type of let* that mutable variables may be of several types, determined from the type of the initializing expression. To build a variable declaration in C we need to know this type: we need type inference. The inference is especially needed for the argument types: cf. §4.1. Since our DSL is so simple, implementing type inference is not complicated – and much easier than using the OCaml typechecker, as we did in offshoring. Here is the gist:

```
module CMutCde : (cdemut with type α proc_t = string) = struct

  type α typ =
    | TInt      : int typ
    | TFloat    : float typ
    | TVoid     : unit typ
    | TVar      : α typ option ref → α typ

  let rec string_of_typ : type a. a typ → string = function
    | TInt      → "int"
    | TFloat    → "double"
    | TVoid     → "void"
    | TVar {contents = None}    → failwith "could␣not␣infer␣type:␣add␣type␣ann"
    | TVar {contents = Some ty} → string_of_typ ty

  let rec unify : type a. a typ → a typ → unit = fun t1 t2 → match (t1,t2) with
   | (t1,t2) when t1 == t2 → ()
   | (TVar t1, TVar t2) when t1 == t2 → ()
   | (TVar {contents = Some t},t') | (t',TVar {contents = Some t}) → unify t t'
   | (TVar ({contents = None} as tr), t) | (t,TVar ({contents = None} as tr)) → tr := Some t
   | _ → assert false

  type α exp = α typ ∗ α CCde.exp

  let int x   = (TInt,   string_of_int x)
  let float x = (TFloat, string_of_float x)

  type α stm = α typ ∗ α CCde.stm

  let for_ (tl,lwb) (tu,upb) body = unify tl TInt; unify tu TInt;
    (TVoid,
     CCde.for_ lwb upb (fun i →
       let (tb,b) = body (TInt,i) in unify TVoid tb; b))

  type α mut = α typ ∗ string
  let (let∗) : α exp → (α mut → ω stm) → ω stm = fun (t,x) body →
    let v = gensym "x" in
    let (tb,b) = body (t,v) in
    (tb,Printf.sprintf "%s␣%s␣=␣%s;\n%s" (string_of_typ t) v x b)
  let dref (t,v) = (t,v)

  let mkfun ?(name="fn") : α stm → α proc_t = fun (tb,b) →
    Printf.sprintf "%s_%s(){\n%s\n}" (string_of_typ tb) name b
  let arg_base ?(name="n") body =
      let n = gensym name and targ = TVar (ref None) in
      let bs = body (targ,n) in . . .
  . . .
end
```

We introduce a run-time (or, more precisely, generation-time) representation of OCaml types, and carry it around, propagating from an expression (the first argument of let*) to the created mutable variable. The functions arg_base and arg_array introduce type variables to bind during the inference. The function string_of_typ uses this type representation to generate a C variable declaration.[25][26] The earlier CCde implementation is reused to actually generate the code. The type representation is a GADT [32], therefore, the OCaml type checker verifies that the type representation agrees with OCaml types (therefore, unify never fails.)

The signature cdemut also incorporates the lessons learned in §4.2 and §4.6. As the type of let* makes it clear, mutable variables may only be introduced in statement context. There is no longer a need in normalization, §4.2, and detection of non-offshorable code. The guarantees stated in §5.2 are maintained: any term written by combining cdemut combinators is convertible to the well-typed and well-formed (OCaml or C) code.

As we learned in §4.6, mutable variables are not first-class. Therefore, cdemut introduces a special type α mut for them, which is different from the expression type α exp. An attempt to write (cf. exr3 in §4.6)

```
let* x = int 0 in let* y = x in (y := int 1) @. ret x
```

will be rejected: the error message points to x in let* y = x and says that "This expression has type int mut but an expression was expected of type 'a exp". That is, x is not an expression and cannot serve as an initializer. The following however is accepted

```
let* x = int 0 in let* y = dref x in (y := int 1) @. ret (dref x)
```

which makes it clear that x and y are two independent mutable variables; the latter is initialized with the current value of the former. The aliasing is prevented by design.[27]

[25] The presence of failwith seems to make the interpretation non-total. One may show however that in interpreting a closed *monomorphic* term t proc_t that failure never occurs.

[26] We can avoid type variables and the unification altogether if we add to arg_base and arg_array an extra argument, a 'type annotation' so to speak. In essence, DSL functions would bear a type signature of sort. The accompanying code tf_addv.ml implements this approach.

[27] If desired, one may introduce reference types and the combinator address_of: α mut → α ref exp. Aliasing becomes possible, but it has to be explicitly marked as such.

5.4. Connection to Offshoring

The implementations CCde and CMutCde used printf to generate C code. Although encapsulated in these modules, it is still ungainly, and still requires the DSL implementors – but not the end users! – to check well-formedness (see the formal reasoning in §5.2 for an example). Generating well-formed C code is a solved problem: we solved it ourselves in offshoring, when pretty-printing OffshoringIR (Fig. 4) to C code (via C AST). It behooves us to re-use that work and not bother with C pretty-printing again (which, albeit mundane, still bothersome, especially the indentation to get the familiar-looking C code.) Furthermore, OffshoringIR was designed as a generic low-level statement-oriented language, to pretty-print not just to C but also FORTRAN, WASM, etc. Therefore, by targeting OffshoringIR in the tagless-final code-generating combinators, we may generate code in these languages as well.

For example, CCde in §5.1 generated addition and for-loop using printf and string operations, as

```
let ( + ) = Printf.sprintf "(%s␣+␣%s)"

let for_ lwb upb body =
  let i = gensym "i" in
  "for(int␣" ^ i ^ "=" ^ lwb ^ ";␣" ^ i ^ "≤" ^ upb ^ ";␣" ^ i ^ "++){\n" ^
  "␣␣" ^ body i ^ "}"
```

With OffshoringIR it becomes:

```
let ( + ) x y = FunCall(OP.ADD I32,[x;y])

let for_ lwb upb body =
  let id = genvarname "i" in
  let upe  = upb + int 1 in
  let step = int 1 in
  let b = body (LocalVar id) in
  Block (Sq.empty, For {id; ty=TNum I32; lwb; upe; step; body=b})
```

We have to stress that to OCaml, OffshoringIR is a data structure. Although convertible to well-formed C code (see Prop. 2), OffshoringIR does not assure per se the well-typedness or the absence of unbound variables. The tagless-final layer, encapsulating OffshoringIR, provides these guarantees, also performing the needed type inference, as discussed in §5.2. Offshoring and tagless-final hence converge on OffshoringIR: both serve as the means of providing well-typedness guarantees, as well as syntax sugar.

One may relate tagless-final with offshoring more directly: by composing, so to speak, MOCde with offshoring. That is, we use the MOCde implementation of the tagless-final signature to generate OCaml code, to be offshored as described in §3. One has to remember however that code templates and their type checking requires compiler support. MetaOCaml thus is a different compiler than OCaml (albeit both source- and binary compatible), requiring its own maintenance. Tagless-final DSL targeting OffshoringIR is implementable in ordinary OCaml, needing only the OffshoringIR pretty printer (which is a small, ordinary OCaml library).

6. Related Work

In his retrospective [14], Sheard lays out the research program on meta-programming, including heterogeneous meta-programming: "A heterogeneous system with a fixed meta-language, in which it is possible to build multiple systems each with a different object-language, or one system with multiple object-languages would be equally useful." [14, §21]. The two systems presented in this paper answer the challenge. The common intermediate language OffshoringIR is designed to be easily rendered in many low-level languages: we have already tried dialects of C such as OpenCL, as well as LLVM IR. We are now experimenting with WASM. Tagless-final approach with its inherent ability of multiple interpretations makes the re-targeting easy. Both systems also demonstrate that heterogeneous metaprogramming is more capable than Sheard thought: "Only in a homogeneous metasystem can a single type system be used to type both the meta-language and the object-language. Only homogeneous meta-systems can support reflection, where there is an operator (run or eval) which translates representations of programs, into the values they represent in a uniform way. This is what makes run-time code generation possible." [14, §2]. First, offshoring has had `run` from the very beginning [9]. After all, running the offshored code is very similar to running native MetaOCaml code: both invoke a compiler followed by dynamic linking. In tagless-final approach, a meta-circular implementation like `OCde` offers another way to run code. As to the type system of the object language, if it is relatively simple as it tends to be for a low-level language, it may be embedded into the meta-language type system – as offshoring and especially tagless-final approaches demonstrate.

Offshoring as a term was first proposed by Eckhardt et al. [9], which we have discussed already. The full potential of offshoring was not yet realized then. It was put forth merely as a way to efficiently execute specialized OCaml code, rather than as a general-purpose C code-generation tool. We must stress that the original offshoring could not support most of the examples of the present paper, because of its severe restrictions: let-expressions are only allowed in statement context and variable declarations in C may only have constants as initializers; loops may only have unit stride; the set of supported operations is not extensible. As discussed in §4.6, the severe restrictions on variables of reference types were not explained or even acknowledged in Eckhardt et al. [9]. Because of the lack of extensibility and tight integration with the OCaml type-checker, the implementation quickly became unmaintainable and is no longer available.

Asuna [33] was an attempt to resurrect offshoring and extend to SIMD extensions, parallelism and LLVM. The paper presented a few applications of offshoring HPC kernels, with few details about the implementation. Curiously the paper does not mention any restrictions on let-bindings in the source language (raising doubts about correctness). The implementation has not been available.

In the hindsight, one can discern offshoring already in Hoare et al. [34], who proposed an approach for formally correct compilation. The machine language, so to speak, is embedded into the source program language. That is, a machine program is the source program of a particular form: it declares all needed variables, after which there is a single loop over the sequence of (i) primitive assignments like `v` := `v1 op v2` where `op` is an arithmetical or logical operation and `v1`, `v2` are variables or constants; (ii) conditionals whose test is a simple boolean expression or a variable. Such program, called normal form, is trivially converted to assembly. Like in offshoring, a low-level program is represented as a particular form of higher-level one. The compilation is then normalization: converting any program to that normal form, by a sequence of transformations (rather than by evaluation).

The model of a low-level code in Hoare et al. [34] is rather crude: for example, it cannot account for idiosyncrasies of register usage (especially common in x86 architectures), or even stack frames. After all, the source language does not have any procedures. A very strong assumption, crucial to the formalism in justifying substitutions is that all expressions are pure. It is not clear if the approach has been implemented as a compilation system and used on realistic examples. The very strong assumptions make one doubt it.

Viewed in the similar light is KreMLin [35] and its language Low*, a subset of F* that is easily mapped to a small subset of C. Because F* is a dependently-typed language, one may state and verify sophisticated correctness properties (including functional correctness). The paper proves that the mapping preserves not just typing but also semantics and side-channel resistance. The system of Protzenko et al. [35] has actually been used in practice, in verifying TLS 1.3. Like in offshoring, the translatable subset of F* is not easy to state in types; therefore, C code extraction ('offshoring') is best effort.

Also related is C code extraction from constructive Coq proofs – although Coq is quite harder to program in; it is also harder to control the form of the produced C code and ensure high performance.

Among other heterogeneous metaprogramming system we should mention MetaHaskell [36], LMS [37] and Terra [38]. Terra, however, does not assure the absence of unbound variables.

FrontC by Hugues Cassé defines the abstract syntax for C, as an OCaml data structure, and includes the parser and the pretty-printer. It is used (with significant modifications) in CIL (C Intermediate Language)[28] [39] and Binary Analysis Platform.[29] Our abstract C syntax (produced from the OffshoringIR) is influenced by Cassé's, but completely re-designed and re-written from scratch. FrontC is developed to represent any existing C program (so to analyze it). We are interested only in C generation, and so chose a small but just as expressive and 'sane' subset of C – quite in the spirit of CIL but with different design choices. The biggest change is the explicit distinction between expressions, simple statements (assignments and function calls) and general statements like loops and branching. This distinction lets us precisely model comma-expressions, and

[28]http://cil-project.github.io/cil/doc/html/cil/cil001.html

[29]https://githubhelp.com/BinaryAnalysisPlatform/bap

support the mixing of variable declarations and statements, allowed since C99. (FrontC does not support C99.) In our subset of C, a declaration introduces only one variable. We regard assignment and increment as statements. Therefore, such C constructions as x=y=0 and --x * y++ are not representable in our abstract syntax and hence never produced.

The idea of using a language with well-developed abstraction facilities as a 'macro' for a low-level object language was forcefully advocated by Kamin et al. [40]. The authors stressed higher-order functions as an abstraction mechanism, contrasting their approach with 'C of Engler et al. [41], which use C as a metalanguage. We also share with Kamin et al. [40] the ideal of composability: object program fragments are represented as values in the meta-language, and these values may be composed – so to build bigger fragments and ultimately whole programs.

However, as Kamin et al. [40] say themselves, they merely "sketch a plausible approach." Furthermore, "A complete and practical solution would need to contend with issues of security and portability...". Indeed, the examples presented in Kamin et al. [40] all represent code as strings, with no abstraction and type annotations. As the paper says, the distinction between object language statements and expressions is not expressed in the metalanguage. The generator hence provides no guarantees: the produced code may easily be ill-formed (let alone ill-typed or with unbound variables). The ability of a metalanguage to reason about the object program is not yet realized. Tagless-final approach presented in this paper solves all these problems. It is hence the completion of the program initiated by Kamin et al. [40].

Generating code using combinators pioneered by Kamin has widely spread: see [10, 42] for extensive bibliography. Lightweight Modular Staging (LMS) [37] (especially Rompf [43]: a rare system description) is also based on combinator-based approach, somewhat inspired by tagless-final. Of more recent work one may mention Ceresa et al. [44] who design an embedded DSL meant for verification (the authors also plan to generate C code). Another work to contrast with is Westphal and Voigtländer [45], who generate (relatively) low-level code and other artifacts from a high-level specification, expressed in (what essentially is) a tagless-final DSL. Unlike us the authors aim not for the most performant but for the most idiomatic code, given their goal is education. Since the naively generated code is hardly idiomatic, the authors are exploring more sophisticated translations. Their approach can thus be called 'optimize from below' the DSL layer. In contrast, the present paper presents 'optimize from above': the tagless-final DSL (or the OCaml subset to offshore) are meant to be straightforward to render as a low-level code. The optimizations happen when we produce DSL expressions from higher-level specifications (such as addv_abs in §3.1.)

The tagless-final approach described in §5 may be traced back to C-code–generating combinators in Cohen et al. [1]. Those combinators were monomorphic, and the explicit passing of the C variable environment made them ungainly: Compare [1, Fig. 18]

```
gen_inst (gen_assign y (gen_add (lvrv x) (gen_int_cst 3))) env
```

with y := x + int 3 in our approach. Ensuring well-scoping of the C code was also the responsibility of a programmer, to pass the environment env in the disciplined way.

An example demonstrating the benefit of tagless-final for generating both reliable *and* performant code is Masuda and Kameyama [46]. The subject is cryptography based on modular arithmetic, which involves a very expensive mod operation. The authors designed a tagless-final DSL and used it to both generate C code and to generate interval analysis verification conditions. Their analysis turns out rather precise and allowed them to prove the absence of integer overflows even if they apply mod less frequently – which notably improved performance. This is a rare example of verification and optimization going hand-in-hand. Another example is Kiselyov and Imai [47]: generating communication code alongside session-type–checking. The paper also demonstrates using tagless-final with staging, similar to the MOCde interpreter in §5.1.

7. Evaluation and Conclusions

We have presented two implemented approaches for generating high-performance C code that compiles without errors or warnings and can be freely linked with other C libraries. Offering correctness guarantees requires generator abstractions, which is a challenge to design and maintain. We have described the notable problems we experienced and the ways we addressed them. One of the main problems turns out maintainability. The current systems are explicitly designed to be extensible and to last.

Offshoring was used in [13] to generate OpenMP matrix-matrix multiplication code that is faster, sometimes 2×, than the state of the art BLAS code generated by ATLAS; tuning was also faster. In [12] offshoring has generated GPGPU (OpenCL) matrix-matrix multiplication and k-means clustering code. The tagless-final approach is used in the highest-performance streaming library [48, 29] to generate OCaml, C and Scala code. One of its application is generating C code for the software-defined FM radio [49]. Both approaches thus proved adequate for their intended tasks.

7.1. Comparison of Offshoring and Tagless-Final

The two approaches share the metaphor of OCaml as C: representing a small imperative language (a subset of C) in the form of OCaml expressions. Their comparison is nuanced and depends on many factors, some of which are listed below.

Offshoring embeds a language using quasi-quotes/templates, whereas the tagless-final approach uses code combinators. Quasi-quotes may be taken as a syntax sugar over the combinators. However, the desugaring is not that obvious or trivial [27, 50, 17, 51].

Quasi-quotation, or templates, are generally regarded as more pleasant to program with, offering a better syntax for loops, control structures, and pattern-matching. MetaOCaml also offers let-insertion.

On the other hand, typed quasi-quotation (code templates) requires compiler support. Offshoring described in this paper depends on MetaOCaml, which is a separately maintained dialect of OCaml (albeit fully source- and binary compatible). In contrast, tagless-final can be implemented in bare OCaml. Therefore, tagless-final is easier to port: it can be, and has been, implemented using modules, type-classes, traits, implicits and objects. In general, some form of higher-rank types (even in the form of module system) is helpful, although one can get by without it if the DSL is simple enough so that it can be monomorphized.

A tagless-final DSL is easier to tailor, to provide just enough combinators needed for the task at hand.

With tagless-final DSL we can ensure that any well-typed DSL expression results in code (which is well-formed and well-typed, by construction). On the other hand, offshoring is inherently partial and best effort.

Acknowledgments. I am very grateful to the anonymous reviewers of SCP, FLOPS 2022, OCaml 2022 and ML 2022 for many, helpful comments and suggestions. This work was partially supported by JSPS KAKENHI Grants Numbers 17K12662, 18H03218, 21K11821 and 22H03563.

References

[1] A. Cohen, S. Donadio, M. J. Garzarán, C. A. Herrmann, O. Kiselyov, D. A. Padua, In search of a program generator to implement generic transformations for high-performance computing, Science of Computer Programming 62 (2006) 25–46.

[2] R. C. Whaley, A. Petitet, Minimizing development and maintenance costs in supporting persistently optimized BLAS, Software—Practice and Experience 35 (2005) 101–121.

[3] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE 93 (2005) 216–231.

[4] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo, SPIRAL: Code generation for DSP transforms, Proceedings of the IEEE 93 (2005) 232–275.

[5] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. P. Amarasinghe, Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, in: H.-J. Boehm, C. Flanagan (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, ACM, 2013, pp. 519–530.

[6] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, P. H. J. Kelly, Firedrake: Automating the finite element method by composing abstractions, ACM Trans. Math.

Softw. 43 (2016). URL: https://www.firedrakeproject.org/. doi:10.1145/2998441.

[7] O. Kiselyov, Generating C, in: M. Hanus, A. Igarashi (Eds.), Functional and Logic Programming, volume 13215 of *Lecture Notes in Computer Science*, Springer International Publishing, 2022, pp. 75–93. doi:10.1007/978-3-030-99461-7_5.

[8] N. Takashima, O. Kiselyov, Y. Kameyama, MetaOCaml as a high-level LLVM macro, Japan Society for Software Science and Technology (JSSST), 31st annual meeting, September 2014 (in Japanese), 2014.

[9] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. N. Swadi, W. Taha, Implicitly heterogeneous multi-stage programming, New Generation Computing 25 (2007) 305–336. doi:10.1007/s00354-007-0020-x.

[10] J. Carette, O. Kiselyov, C.-c. Shan, Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages, Journal of Functional Programming 19 (2009) 509–543.

[11] O. Kiselyov, Typed tagless final interpreters, in: Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming, SSGIP'10, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 130–174. doi:10.1007/978-3-642-32202-0_3.

[12] K. Hirohara, Generating GPU kernels from high-level specifications using MetaOCaml, 2019. Tohoku University, Master Thesis, in Japanese.

[13] G. Bussone, Generating OpenMP code from high-level specifications, 2020. Internship report to ENS.

[14] T. Sheard, Accomplishments and research challenges in meta-programming, in: W. Taha (Ed.), Proceedings of SAIG 2001: 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation, number 2196 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001, pp. 2–44.

[15] G. Ofenbeck, T. Rompf, M. Püschel, RandIR: differential testing for embedded compilers, in: A. Biboudis, M. Jonnalagedda, S. Stucki, V. Ureche (Eds.), Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, ACM, 2016, pp. 21–30. doi:10.1145/2998392.

[16] O. Kiselyov, Do mutable variables have reference types?, 2022. URL: http://arxiv.org/abs/2211.04107. doi:10.48550/arXiv.2211.04107, ML Family Workshop 2022.

[17] O. Kiselyov, The design and implementation of BER MetaOCaml - system description, in: FLOPS, number 8475 in Lecture Notes in Computer Science, Springer, 2014, pp. 86–102. doi:10.1007/978-3-319-07151-0_6.

[18] O. Kiselyov, BER MetaOCaml N114, https://okmij.org/ftp/ML/MetaOCaml.html, 2023.

[19] C. Calcagno, W. Taha, L. Huang, X. Leroy, Implementing multi-stage languages using ASTs, gensym, and reflection, in: GPCE, number 2830 in Lecture Notes in Computer Science, 2003, pp. 57–76. doi:10.1007/978-3-540-39815-8_4.

[20] J. Yallop, D. Sheets, A. Madhavapeddy, A modular foreign function interface, Science of Computer Programming 164 (2018) 82–97. doi:10.1016/j.scico.2017.04.002.

[21] O. Kiselyov, Reconciling Abstraction with High Performance: A MetaOCaml approach, Foundations and Trends in Programming Languages, Now Publishers, 2018. doi:10.1561/2500000038.

[22] P. S. Abrams, An APL machine, Ph.D. thesis, Stanford Linear Accelerator Center, Stanford University, Stanford, CA, USA, 1970. SLAC-114 UC-32 (MISC).

[23] J. Zhu, J. Hoeflinger, D. Padua, Compiling for a hybrid programming model using the LMAD representation, in: Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, 2003, pp. 321–335. doi:10.1007/3-540-35767-x_21.

[24] P. Dybjer, A. Filinski, Normalization and partial evaluation, in: G. Barthe, P. Dybjer, L. Pinto, J. Saraiva (Eds.), APPSEM 2000: International Summer School on Applied Semantics, Advanced Lectures, number 2395 in Lecture Notes in Computer Science, Springer, 2002, pp. 137–192.

[25] A. Sabry, M. Felleisen, Reasoning about programs in continuation-passing style, Lisp and Symbolic Computation 6 (1993) 289–360.

[26] M. Gordon, R. Milner, L. Morris, M. Newey, C. Wadsworth, A metalanguage for interactive proof in LCF, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, Tucson, Arizona, 1978, pp. 119–130. URL: http://www-public.tem-tsp.eu/~gibson/Teaching/CSC4504/ReadingMaterial/GordonMMNW78.pdf.

[27] C. Chen, H. Xi, Meta-programming through typeful code representation, Journal of Functional Programming 15 (2005) 797–835. doi:10.1017/S0956796805005617.

[28] P. J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation: Part I, Communications of the ACM 8 (1965) 89–101. doi:10.1145/363744.363749.

[29] strymonas, Strymonas streams: stream fusion, to completeness, 2022. URL: https://strymonas.github.io.

[30] O. Kiselyov, K. Sivaramakrishnan, Eff directly in OCaml, Electronic proceedings in theoretical computer science 285 (2018) 23–58. doi:10.4204/EPTCS.285.2.

[31] W. Taha, Multi-Stage Programming: Its Theory and Applications, Ph.D. thesis, Oregon Graduate Institute of Science and Technology, 1999.

[32] H. Xi, C. Chen, G. Chen, Guarded recursive datatype constructors, in: POPL, 2003, pp. 224–235. doi:10.1145/640128.604150.

[33] N. Takashima, H. Sakamoto, Y. Kameyama, Generate and offshore: type-safe and modular code generation for low-level optimization, in: Proc. ACM SIGPLAN Workshop on Functional High-Performance Computing, FHPC@ICFP 2015, Vancouver, BC, Canada, September 3, 2015, ACM, 2015, pp. 45–53. doi:10.1145/2808091.

[34] C. A. R. Hoare, H. Jifeng, A. Sampaio, Normal form approach to compiler design, Acta Informatica 30 (1993) 701–739. doi:10.1007/BF01191809.

[35] J. Protzenko, J. K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Z. Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, N. Swamy, Verified low-level programming embedded in F*, Proc. ACM Program. Lang 1 (2017) 17:1–17:29. doi:10.1145/3110261.

[36] G. Mainland, Explicitly heterogeneous metaprogramming with MetaHaskell, in: ICFP, ACM Press, New York, 2012, pp. 311–322. doi:10.1145/2398856.2364572.

[37] T. Rompf, M. Odersky, Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs, Commun. ACM 55 (2012) 121–130. doi:10.1145/2184319.2184345.

[38] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, J. Vitek, Terra: a multi-stage language for high-performance computing, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, ACM, 2013, pp. 105–116. doi:10.1145/2499370.2462166.

[39] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer, CIL: Intermediate language and tools for analysis and transformation of C programs, in: Proceedings of Conference on Compilier Construction, volume 2304 of *Lecture Notes in Computer Science*, 2002, p. 213.

[40] S. Kamin, M. Callahan, L. Clausen, Lightweight and generative components I: Source-level components, in: Proc. GCSE'99, volume 1799 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 49–62.

[41] D. R. Engler, W. C. Hsieh, M. F. Kaashoek, 'C: A language for high-level, efficient, and machine-independent dynamic code generation, in: In Proceedings of the ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, 1996, pp. 131–144.

[42] Y. Kameyama, O. Kiselyov, C.-c. Shan, Combinators for impure yet hygienic code generation, Science of Computer Programming 112 (part 2) (2015) 120–144. doi:10.1016/j.scico.2015.08.007.

[43] T. Rompf, Reflections on LMS: exploring front-end alternatives, in: Proc. 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH, ACM, 2016, pp. 41–50. doi:10.1145/2998392.2998399.

[44] M. Ceresa, F. Gorostiaga, C. Sánchez, Declarative stream runtime verification (hLola), in: Programming Languages and Systems, Springer International Publishing, Cham, 2020, pp. 25–43. doi:10.1007/978-3-030-64437-6_2.

[45] O. Westphal, J. Voigtländer, Implementing, and keeping in check, a DSL used in E-Learning, in: Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, volume 12073 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 179–197. doi:10.1007/978-3-030-59025-3_11.

[46] M. Masuda, Y. Kameyama, Unified program generation and verification: A case study on Number-Theoretic Transform, in: Functional and Logic Programming, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2022, pp. 133–151. doi:10.1007/978-3-030-99461-7_8.

[47] O. Kiselyov, K. Imai, Session types without sophistry, in: Functional and Logic Programming, volume 12073 of *Lecture Notes in Computer Science*, Springer International Publishing, 2020, pp. 66–87. doi:10.1007/978-3-030-59025-3_5.

[48] O. Kiselyov, T. Kobayashi, A. Biboudis, N. Palladinos, Highest-performance stream processing, 2022. doi:10.48550/arXiv.2211.13461, OCAML Workshop 2022.

[49] T. Kobayashi, O. Kiselyov, Complete stream fusion for software-defined radio, 2022. URL: http://arxiv.org/abs/2208.08732. doi:10.48550/arXiv.2208.08732.

[50] Y. Kameyama, O. Kiselyov, C.-c. Shan, Closing the stage: From staged code to typed closures, in: PEPM, ACM Press, New York, 2008, pp. 147–157. doi:10.1145/1328408.1328430.

[51] O. Kiselyov, Generating code with polymorphic let: A ballad of value restriction, copying and sharing, EPTCS 241 (2017) 1–22. doi:10.4204/EPTCS.241.1.