Poster for the *Poster Session* of C++Online'24, 29.02.–02.03.2024, https://cpponline.uk/

Towards Scriptable C++ Refactorings with Coccinelle

Michele MARTONE[⋆], **Julia LAWALL**[†]

[⋆]michele.martone@lrz.de, Leibniz Supercomputing Centre, Garching near Munich, Germany

[†]julia.lawall@inria.fr, Inria, Paris, France

Context: Problems with large scientific HPC C/C++ Codebases

In our experience and in our environment, the problems of **large** HPC codebases are not so unlike those of non-HPC codebases:

- Keeping up to date to dependencies (e.g. libraries or parallelism APIs)
- Uncluttering old constructs (e.g. once new standards appear)
- Making code future proof (e.g. amenable to parallelism)
- Refactoring for performance (e.g. data layout change)
- Ensuring code properties (e.g. thread safety or against *code smells*)

A difference is perhaps in longer lifespans and smaller teams. HPC development tends to happen in bursts, e.g., when an idea is being developed and usually associated to a publication. Typically, *future proofing* and correctness efforts are tackled by the original author, who is not a computer scientist. At times, access to staff with specific expertise in Software Engineering or HPC is granted as a service.

Background Work

- Joint expertises:
  - Performance-oriented refactoring of large HPC codes (Martone)
  - Formal methods for program rewriting (Lawall)
- Successful development of a *programmable*, *replayable* refactoring for a **200 kLoC** C codebase (see [C3PO])
- Using *semantic patching* technology of Coccinelle

What this Poster is About

- Preliminary work for a submitted Franco-German project proposal
- In form of `code rewriting rules` and resulting `diff` output
- Refactorings recently enabled in Coccinelle:
  - HPC-affine introduction of `mdspan` (modern C++)
  - A sample rule to replace loops with *STL algorithms*
  - Another sample rule to enforce a programming guideline

Coccinelle

About:

- Design started in 2004 (open source in 2008)
- Targeted *"collateral evolutions"* (recurring changes motivated by changes in API interfaces), in Linux device driver code (C)
- Contributed to around 9000 commits in the Linux kernel
- Available in various Linux distros

Distinguishing features:

- Self-contained project
- Unique `diff`-like (`+ ..`/`- ..`) patch specification language

What *novel* uses do we foresee:

- Large-scale refactorings
- Data-layout changes
- Advanced expression manipulations
- Modern C/C++ standards and GPU-specific language extensions

Managing the Unfeasible

Our collaboration (see [C3PO]) was initiated from the need to obtain a programmable code refactoring that not only could be performed in little time (a matter of seconds), but could also be maintained by the code owners, who are astrophysicists, not C++ or parsing experts.

Manipulate Expressions, Globally

In [C3PO] we had to change all expressions involving a few *array of structures* accesses into corresponding *structure of arrays* accesses. This change impacted nearly the entire code (tens of thousands of *change sites*), and yielded an estimated 2 to 5× speedup on representative runs. We deem *global expression manipulation* to be **the most important application** of our approach.
We hope that the general C++ community can find uses of our otherwise HPC-oriented collaboration.

Low-Cost Performance Experiments

Rules can be instructed to act on selected quantities only, thus enabling refactorings that are *partial*. This opens the door to low-effort performance experiments, including also #pragmas, loops, and function manipulations like in the following subsections.

Modern C++ Transformation Example

Introduction of C++23's multi-index transformations is now possible (e.g. for `std::mdspan`, see [MDSPAN]). The following *semantic patch* with two rules:

```
#spatch --c++=23
@tomultiindex@
symbol a;
expression i,j,k;
@@
- a[i][j][k]
+ a[i, j, k]

@@
symbol b;
@@
- b[...]
+ b[0]
```

leads to the following code patch (context deliberately expanded):

```
@@ -1,8 +1,8 @@
 int main()
 {
   int a[1][1][1];
   int b[1][1][1];
   int i=0,j=0,k=0;
-  a[i][j][k]++;
-  b[i][j][k]++;
+  a[i, j, k]++;
+  b[0][j][k]++;
 }
```

Notice that lines 5–6 of the semantic patch (rule `@tomultiindex@`) modify expressions of **arbitrarily complicated** statements.

No Raw Loops Example

Coccinelle can match several constructs, comprehensive of control flow. The following example replaces a loop on elements with the use of an *STL algorithm*, which is good practice [NRL].

```
#spatch --c++=17
@@ @@
  #include <iostream>
+ #include <algorithm>
+ #include <functional>

@@
type T;
constant k;
identifier elem,result,arrid;
@@
-  bool result = false;
   ...
-  for ( T &elem : arrid )
-  if ( \( elem == k \| k == elem \) )
-    {
-      ...
-      result = true;
-      break;
-    }
+ const bool result =
+   (find(begin(arrid),end(arrid),k) !=
+    end(arrid));
```

```
@@ -1,20 +1,15 @@
 #include <vector>
 #include <iostream>
+#include <algorithm>
+#include <functional>
 int main()
 {
   using namespace std;
   vector v = {1,2,3};
-  bool has_zero = false;

   v[2] = 0;

-  for ( int & a : v )
-    if (0 == a)
-    {
-      cout << "doing things\n";
-
-      has_zero = true;
-      break;
-    }
+  const bool has_zero =
+    (find(begin(v), end(v), 0) != v.end());
   cout << has_zero << endl;
 }
```

Enforcing Coding Guidelines Example

Matching programming guidelines, whether project-specific or idiomatic. Here an example to recognize F.17 from the guidelines collection of Stroustrup and Sutter [F17].

```
#spatch --c++
@r1@
type T;
identifier f;
parameter list pl;
@@

T f(pl) {}

@r2@
typedef A, B;
type heavy_type = {A, B};
type r1.T;
identifier r1.f;
symbol i;
@@

+// Note: heavy copy!
  T f (
  ..., heavy_type i, ...
  ) { ... }
```

```
@@ -1,13 +1,16 @@
 #include <array>
 struct A { std::array<int,999> a; }; // heavy
 struct B { std::array<int,999> a; }; // heavy
 struct C { std::array<int, 16> a; }; // light
 void if1(int i) {}
 void if2(int &arg) {}
 void if3(const int &arg) {}
+// Note: heavy copy!
 void af1(A i) {}
+// Note: heavy copy!
 void bf1(B i) {}
+// Note: heavy copy!
 void bf2(const B i) {}
 void bf3(const B & i) {}
 void cf1(C i) {}
 int main() { }
```

Key Points

- Best if used on codebases with code conventions in place
- Transformations preserve most spacing and comments
- Specificity vs generality is at the user's discretion
- This is preliminary work
- C++ codes are very diverse
- **We're curious to hear about your refactoring patterns**

Current Work

We are expanding Coccinelle's C++ support:

- constructors, destructors
- template declarations and instantiation
- namespaces, misc keywords
- lambda functions
- variadic operators

Unsupported syntax in source code leads to skipping transformations. Once we have enough syntax covered, we will develop use cases.

Acknowledgements

- This work was partially funded by *SiVeGCS*
- Work visits in this collaboration have been supported by the BayFrance'23 scheme (https://www.bayern-france.org/), financed by the *Bavarian Ministry of State for Education, Culture, Science and Art (StMBW)* and the *French Ministery of Europe and Foreign Affairs (MEAE)*

References

- [C3PO] M. Martone, J. Lawall; *"Refactoring for Performance with Semantic Patching: Case Study with Recipes"*; Proceedings of the "Compiler-assisted Correctness Checking and Performance Optimization for HPC" Workshop at ISC'21 (**preprint**: https://hal.inria.fr/hal-03266521; **doi**: https://link.springer.com/chapter/10.1007/978-3-030-90539-2_15)
- [NRL] Sean Parent. *GoingNative 2013 C++ Seasoning* at https://www.youtube.com/watch?v=W2tWOdzgXHA
- [F17] Bjarne Stroustrup and Herb Sutter. *F.17: For "in-out" parameters, pass by reference to non-const* at https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- [MDSPAN] VV.AA. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0009r18.html

michelemartone https://coccinelle.gitlabpages.inria.fr/website/