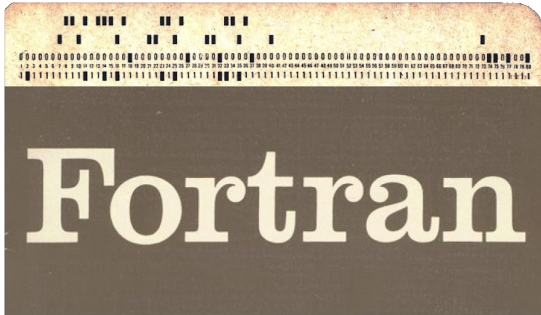
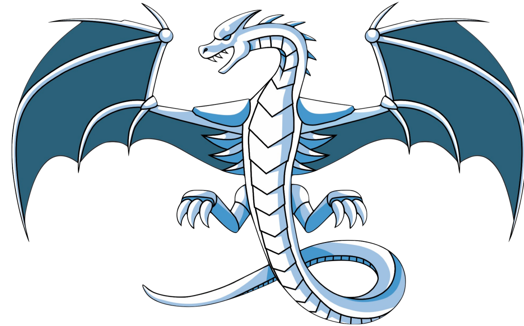
BACHELOR THESIS

LLVM-based tooling for the Fortran programming language

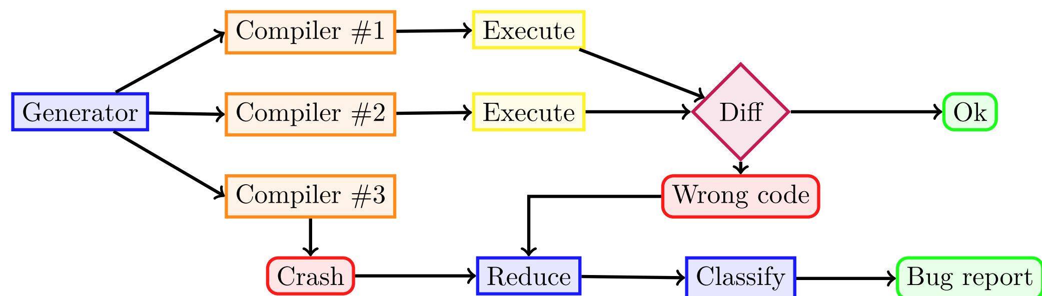
Contact: henri.menke@mpcdf.mpg.de





Created in the 1950s Fortran is still the prevailing language in many high-performance computing applications today. Most of the quantum chemistry codes that form the foundations of modern materials science are written in Fortran and it is not realistically feasible to rewrite these massive and complex computer programs in another language. In light of today's advances in software development Fortran might be viewed as a dinosaur, but the language has a rich history and inspired many of the programming paradigms that we take for granted today. In more recent iterations of Fortran's standardization process features were added to bring the language more on par with the current ways to develop software, such as object oriented programming [1]. One consequence of Fortran having fallen out of fashion for contemporary projects is that the available compilers and associated tooling do not get exercised to the extent of more popular general purpose programming languages such as C or C++. To this end we offer three Bachelor thesis projects that deal with Fortran compiler technologies.

I. RANDOM PROGRAM GENERATION TO DETECT FORTRAN COMPILER BUGS



Due to the limited use there are often subtle bugs present in various Fortran compiler implementations. Usually these are triggered accidentally in an existing large project which can be quite annoying as it will require workarounds or even prohibit the use of certain language features until the bug is fixed upstream.

This issue naturally does not only plague Fortran compilers, but C and C++ compilers are equally affected. To this end an interesting approach was developed, named *Random Program Generation*. A tool generates a random but valid program that is fully standard conforming, i.e., does not invoke undefined or unspecified behavior. In the first stage, the generated program is then compiled to see whether it crashes the compiler under investigation. When that is successful, the program is compiled by another (ideally known good) compiler and the two binaries are executed to understand whether there is any observable difference in runtime behavior. If either of these two stages is unsuccessful, there is a compiler bug that needs to be reported and fixed by the compiler developers. The approach of random program generation is supposed to preempt the accidental discovery of compiler bugs during project development.

In this thesis we will implement random program generator for the Fortran programming language informed by the prior work on Csmith [2] and YARPGen [3] for the C and C++ programming languages. This tool will allow us to more comprehensively test Fortran compilers for bugs that we normally encounter during our in-house development of electronic structure and plasma physics codes. We offer an interesting hands-on project that deals with the fundamentals of a well-established programming language and potentially the inner workings of several

industry-standard compilers and thereby teaches important transferable skills. A collaboration with Advanced Micro Devices, Inc. (AMD) is conceivable. The ideal candidate has a good knowledge of compiler construction and a strong background in concepts of programming languages, ideally with good knowledge of C++ as well as some basic familiarity with Fortran.

II. FORTRAN PROGRAM REDUCER FOR DELTA DEBUGGING

```
nfft_plan *cths = (nfft_plan*) ths;
for (j = 0, f = cths->f; j < cths->M_total; j++)
  *f++ = ((R) 0.0);
for (k = 0; k < cths->n_total; k++)
  cths->g1[k] = ((R) 0.0);
for (k = -cths->N_total / 2, g1 = cths->g1 + cths->n_total
    - cths->N_total / 2, f_hat = cths->f_hat; k < 0; k++)
  (*g1++) = cpow(-((R) 6.2831853071795864769252867665590057683943387987502) * (__extension__ 1.0iF)
cths->g1[0] = cths->f_hat[cths->N_total / 2];
for (k = 1, g1 = cths->g1 + 1, f_hat = cths->f_hat + cths->N_total / 2 + 1;
    k < cths->N_total / 2; k++)
  (*g1++) = cpow(-((R) 6.2831853071795864769252867665590057683943387987502) * (__extension__ 1.0iF)
```

Reduce

```
typedef double a;
_Complex double *b;
void c() { *b /= 1.0iF * (a)0; }
```

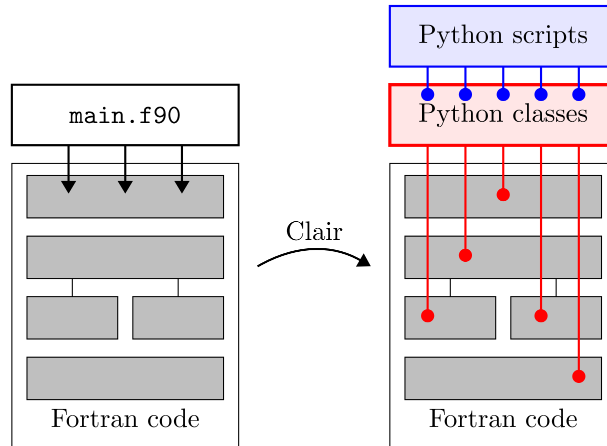
When compiler bugs are accidentally triggered during development of an existing large project, it is often difficult to clearly isolate the offending statements into a minimal exmaple. Justifiably, compiler maintainers demand minimal examples, because it is not a good use of time to sift through the masses of abstract syntax trees, intermediate representations and generated machine code instructions that are generated by a large project.

To this end a technique commonly referred to as *test case reduction* was developed. At the start a so-called "interestingness test" must be formulated which indicates whether its input does reproduce the desired error, i.e. it is interesting, and should be refined further. In the naïvest form the refinement randomly discards lines, words, or characters and from the input file and applies the interestingness test repeatedly. This leads to poor convergence behavior towards a minimal example, because randomly deleting elements from the input file, generally results in invalid programs. A much more promising approach is to delete elements in a way that is informed by the semantic analysis of the input program to efficiently remove dead code and resolve dependencies. Such an approach was previously realised for C/C++ under the name C-Reduce [4] and its successor CVise.

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies [5]. The C-Reduce and CVise program reducers for C/C++ internally rely on a helper program called `clang_delta` based on the LLVM Clang C/C++ compiler to perform the guided reduction of programs mentioned above. LLVM also ships with a Fortran compiler named Flang [6], which is currently under active development, but is already being used in production as the basis for the AMD Optimizing C/C++ and Fortran Compilers (AOCC). At the moment there does not exist a `flang_delta` program.

In this thesis we will implement a `flang_delta` program based on LLVM Flang. This tool will allow us to more effectively reduce programs that trigger compiler bugs that we find during our in-house development of electronic structure codes. We offer an interesting hands-on project that deals with the inner workings of one of the most advanced industry-standard compilers and thereby teaches important transferable skills. A collaboration with Advanced Micro Devices, Inc. (AMD) is conceivable. The ideal candidate has a good knowledge of compiler construction and a strong background in concepts of programming languages, ideally with excellent knowledge of C++, exprience with Python and Perl, as well as some basic familiarity with Fortran.

III. AUTOMATIC PYTHON INTERFACE GENERATION FOR FORTRAN LIBRARIES



Probably the most used programming language in contemporary computational science is Python. Its expressive, highly readable syntax combined with dynamic typing and a rich ecosystem for numerical methods are key enablers for its popularity. However, Python itself being an interpreted language is slow and not well-suited for high-performance computing by itself. This is why many libraries such as NumPy or SciPy are implemented in C/C++/Fortran with an interface leveraging the Python/C API to expose the desired functionality to the Python scripting language.

Two main approaches exist to interface low-level code with Python. The first is to directly write Python/C API calls into the library or do so via a wrapper such as pybind11 or nanobind. While this offers the greatest amount of flexibility, it also comes with a high development cost, due to interfaces having to be constructed manually. The other approach is to parse the source code of the library and then programmatically generate the required interface code. This is the approach taken by NumPy and SciPy which both use their own tool F2PY [7] to perform the wrapping. However, F2PY is fairly limited in what Fortran language features it supports and is also not really intended for use outside of NumPy/SciPy. A similar tool for the C/C++ programming languages is SWIG [8] which also supports a large varieties of other programming and scripting languages to generate interfaces for.

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies [5]. As an alternative to SWIG there exists a tool called Clair (Clang Introspection and Reflection tools) [9], developed by the Flatiron Institute in New York, USA, that also parses the C++ code and generates interfaces, but in contrast to SWIG leverages LLVM for the parsing. LLVM also ships with a Fortran compiler named Flang [6], which is currently under active development, but is already being used in production as the basis for the AMD Optimizing C/C++ and Fortran Compilers (AOCC).

In this thesis we will extent Clair to generate Python interfaces for Fortran code based on LLVM Flang. This tool will allow us to easily generate Python interfaces for a number of high-performance applications with focus on the Octopus electronic structure code. We offer an interesting hands-on project that deals with the inner workings of one of the most advanced industry-standard compilers and thereby teaches important transferable skills. A collaboration with the Center for Computational Quantum Physics (CCQ) of the Flatiron Institute in New York, USA is conceivable. The ideal candidate has a good knowledge of compiler construction and a strong background in concepts of programming languages, ideally with excellent knowledge of C++ and some basic familiarity with Fortran.

---

[1] J. Reid, The new features of Fortran 2018, ACM SIGPLAN Fortran Forum **37**, 5 (2018).
[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, Finding and understanding bugs in C compilers, ACM SIGPLAN Notices **46**, 283–294 (2011).
[3] V. Livinskii, D. Babokin, and J. Regehr, Random testing for C and C++ compilers with YARPGen, Proceedings of the ACM on Programming Languages **4**, 1–25 (2020).
[4] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, Test-case reduction for C compiler bugs, ACM SIGPLAN Notices **47**, 335–346 (2012).
[5] C. Lattner and V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (IEEE).
[6] `https://flang.llvm.org/`.
[7] P. Peterson, F2PY: a tool for connecting Fortran and Python programs, International Journal of Computational Science and Engineering **4**, 296–305 (2009).

[8] D. M. Beazley *et al.*, SWIG : An easy to use tool for integrating scripting languages with C and C++, in *Tcl/Tk Workshop*, Vol. 43 (1996) p. 74.

[9] `https://github.com/flatironinstitute/clair`.