LFortran Intrinsic functions

Harshita Kalani

November 24, 2025

Contents

1 About Me

1.1 Contact Information

- **University** - Indian Institute of Technology, Jodhpur
- **Email IDs** - kalani.1@iitj.ac.in, harshitakalani02@gmail.com
- **Github** - harshitakalani
- **Timezone** - IST (UTC + 5:30)

1.2 Personal Background

I am a final year undergraduate pursuing BTech in the department of Computer Science and Engineering at Indian Institute of Technology, Jodhpur.

I have completed following relevant courses in my academic curriculum in past years: Computer Organization and Architecture, Computer Networks, Theory of Computation, Digital design, Operating Systems, Software Engineering, Object Oriented Analysis and Design, Data Structures and Algorithms, Design and Analysis of Algorithms.

1.3 Programming Background

I utilize Ubuntu 22.04 as my preferred operating system, coupled with the Visual Studio Code (VS Code) editor. I have strong background in Python 3.x, C++17 and Javascript. I have explored Kotlin Multiplatform, Flutter,

Django, ReactJS, REST-APIs and have some experience with HTML and CSS during my undergraduate studies. A list of my ongoing and completed work is as follows:

- **Compiling SciPy packages with LFortran** - At the beginning of August 2023, LFortran implemented sufficient semantics and lowering so that it could correctly compile and run SciPy's minpack, obtaining bit-for-bit identical results as GFortran. When testing other files at random, many of them also compiled successfully. There were a few features that needed to be implemented. I took this task as my Btech project and with collaborative efforts, LFortran has achieved the capability to successfully compile 9 / 15 (60 %) SciPy packages, encompassing amos, cdflib, specfun, mach (integrate), mach (special), minpack, minpack2, fitpack, and quadpack. Also, a platform independent and robust CMake based build system to integrate LFortran into SciPy build is implemented. Notably, LFortran now passes all the rigorous test suites designed by SciPy developers for these packages. This accomplishment signifies a pivotal milestone, as it empowers SciPy users to potentially transition from GFortran to LFortran for these specific packages, offering a viable alternative for enhanced compatibility and performance in their scientific computing workflows.

- **Kotlin Multiplatform (android and iOS)** - This is my intern project done during the summer of 2023 at Warner Bros Discovery where I successfully onboarded the Playback Info Resolver(PIR) component of the Player SDK into Kotlin Multiplatform and implemented the shared library for both Android and iOS Platforms which contains the Kotlin source code that can be transpiled into Swift, JavaScript and C++, demonstrating the practicality of cross-platform development. Here, PIR is the business logic to initiate the playback session for a requested asset by an user on a specific device. I then demonstrated a compelling proof of concept of Kotlin Multiplatform which showcased the journey of developing, integrating and testing the PIR in both Android and iOs platforms.

- **Covid-19 Detection using chest X-Ray** - The outbreak of COVID-19 has led to a global health crisis, making early detection and diagnosis crucial for disease control and management. Medical imaging,

especially chest X-rays, has emerged as a potential tool for the early detection of COVID-19. In this study, we proposed a deep learning-based approach to detect COVID-19 using chest X-rays. Specifically, we employed VGG-19, EfficientNet B3, ResNET-50 models, and performed in-depth analysis to detect COVID-19 infection using chest X-rays after lung segmentation using U-Net and dimensionality reduction using PCA. Our approach achieved an accuracy of 96.4% on a pre-trained U-Net model after lung segmentation.

- **Optical Character Recognition Application** - During this project, I successfully led a team of developers in creating a pipeline that leverages Optical Character Recognition (OCR) technology to extract text from images. This pipeline can then convert the extracted text into various formats like .txt, .mp3, and .csv. We seamlessly integrated this pipeline into a mobile application, streamlining the process for users to extract text from images using their smartphones. The app features a simplified login system through Google authentication and allows users to upload images from their gallery or camera effortlessly. Additionally, to enhance accessibility, we included a text-to-speech feature that converts the extracted text into audio, providing users with the option to listen to the content instead of reading it manually.

More details about the same can be found on my GitHub profile: harshitakalani. In all of the above projects I have used `git` as my version control system.

1.4 Previous Contributions to LFortran

Following is the list of Merge Requests that I have made in the past to the Github repository of LFortran, PS: I have listed contributions from latest to oldest, list order does not determine the difficulty of the PR.

1.4.1 Merged

- !3672: enh: Code cleanup for intrinsic functions
- !3668: Implement intrinsic index

- !3660: Implement intrinsic scan
- !3659: Implement intrinsic lgt, llt, lle and lge
- !3656: Handle verify check in intrinsic_function pass
- !3647: Remove old implementation of ibits, count and sqrt
- !3621: Implement intrinsic verify
- !3600: Implement intrinsic selected_char_kind
- !3578: Implement intrinsic char
- !3556: Implement lfortran_ToLowerCase
- !3541: Implement intrinsic ibclr, ibset and btest and remove old implementation of the same
- !3534: Implement intrinsic not and remove old implementation of not, nint, floor and ceiling
- !3533: Implement intrinsic iand, ior and ieor and remove their old implementation
- !3522: Implement intrinsic blt, ble and bge
- !3474: Throw error if the first arg of intrinsic cmplx is absent
- !3463: Throw error for comparison of different types of operands
- !3411: Implement intrinsic precision and remove old implementations of max, min and aimag
- !3393: Removed pure/lfortran_intrinsic_trig.f90
- !3386: Implement intrinsic rshift

1.4.2 Open

- !3674: Implement len_trim and remove dead code
- !3703: Implement modulo and handle flip_sign pass
- !3635: Implement achar and remove old implementation of char

I currently have 108 merge requests at LFortran out of which 82 are merged, 16 closed and 10 opened. A comprehensive list of all my merge requests can be found at: lfortran/lfortran/pulls

2 The Project

This project delves into the intricate landscape of intrinsic functions within LFortran, evaluating their current status and proposing strategic enhancements. To enhance efficiency, certain functions require migration from the old runtime and compile-time libraries to become intrinsic functions, while others need to be developed anew. Additionally, there are intrinsic functions that fall outside the scope of this project; they will be flagged with an error message indicating their non-implementation status. The methodology to implement an intrinsic function from scratch is also elucidated, outlining a structured approach to integrate new functionalities. Ideas related to design improvements and enhancing performance has also been proposed.

2.1 Current compilation status

With the latest commit, LFortran supports a wide range of intrinsic functions that are completely implemented and tested rigrously. The issue #492 is created to keep the track of all the intrinsic functions that are implemented and the ones that need to be implemented to match the current GFortran standards of intrinsic procedures. The variety of intrinsic functions cover math, string, logical, complex, trigonometric, inverse trigonometric, array intrinsics, intrinsic subroutines and many more. Additionally, certain functions are implemented separately using the runtime library and APIs, while others are implemented with the assistance of nodes.

2.2 What are intrinsic functions?

Intrinsic functions in Fortran are fundamental tools that perform specific operations, each with its unique name and purpose. For instance, the ABS() function calculates the absolute value of a number, while SQRT() computes the square root. Understanding these functions involves knowing their names, the number and types of arguments they accept, as well as the range of valid argument values. For example, SQRT() requires a non-negative REAL argument to compute the square root accurately. Using these intrinsic functions correctly enhances code clarity and functionality, ensuring robust and error-free calculations.

Let's delve into the concept of intrinsic functions by exploring an example and understanding what LFortran manages during the implementation of such functions.

2.2.1 Implementation of intrinsic Mod

The MOD intrinsic function in Fortran is a mathematical operation that calculates the remainder when one integer is divided by another. The syntax of the MOD function is *MOD(I, J)*, where I and J are both integer or real operands. The function returns the remainder of the division I/J. This operation is particularly useful in scenarios where the precise remainder is required, such as in cyclic or periodic computations. The MOD function is crucial in modular arithmetic and applications involving repetitive patterns or periodicity. Its utilization enhances code clarity and conciseness, especially in mathematical and scientific computing, by providing a straightforward means to obtain the remainder of integer divisions within the Fortran programming language.

The intrinsic MOD operation is represented in the Abstract Syntax Tree (ASR) using the *IntrinsicElementalFunction*. Subsequently, within the *intrinsic_function* pass, a namespace denoted as MOD is established. This namespace incorporates a function named *verify_args* designed for validating the correct input argument types. Furthermore, an *eval_Mod* function is implemented to facilitate compile-time evaluation of *mod(i, j)* under the condition that both i and j are compile-time values.

Additionally, the functions *create_Mod* and *instantiate_Mod* are introduced. In scenarios where the input types are integers, these functions execute integer division of i by j, succeeded by integer multiplication with j, and concluded with subtraction from i to derive the remainder. This remainder is then designated as the result. Conversely, when the input types are real, a series of operations ensued. Initially, real division is performed, followed by the casting of the result to an integer. Subsequently, the truncated integer is multiplied by j, and the outcome is subtracted from i to yield the remainder. This comprehensive implementation ensures accurate handling of both integer and real types within the context of the MOD operation in the LFortran compiler.
PS: The below given example is just for illustration purposes, syntax, semantics might be incorrect.

```
function modi32i32(a, p) result(d)
    integer(int32), intent(in) :: a, p
    integer(int32) :: q
    q = a/p
    d = a - p*q
end function

function modr32r32(i, j) result(d)
    integer(real32), intent(int) :: i, j
    integer(int32) :: q
    q = (int32) (i / j)
    d = a - (real32)(p) * (real32)(q)
end function
```

Listing 1: mod.f90

2.3 Steps to implement an intrinsic function

LFortran allows us to follow a number of fixed steps in order to implement an intrinsic function.

- The first step in implementing an intrinsic function is different for elemental functions and other intrinsic functions. The step is automated for intrinsic elemental functions while it needs to be done manually for the rest of the functions. In order to create an elemental intrinsic, we define the argument types and the return type in the *intrinsic_func_registry_util_gen.py* file. This file contains a python script

which automatically generates two essential functions for the specified intrinsic function, namely, create_<function_name> and **verify_args**.

The **create_<function_name>** function is responsible for creating the ASR representation of the intrinsic function, while verify_args ensures that the correct number of arguments and their types are provided, ensuring adherence to function specifications.

The file has a number of functionalities added. It allows us to handle optional arguments like "kind" which is available in a wide number of intrinsic functions. Intrinisics with optional arguments other than "kind" are currently handled manually in the ast_common_visitor.h file as done for intrinsic selected_real_kind in #3438.

Currently, the *intrinsic_func_registry_util_gen.py* file doesn't support the automatic creation of verify_args and create_<function_name> for intrinsic array functions, so that needs to be done manually. We create a namespace in the *intrinsic_array_function_registry.h* file and then write the verify_args and create_function for the particular intrinsic.

- The second step involves specifying the name of the intrinsic, along with the minimum and maximum number of arguments it can accept, as well as the keywords for any necessary and optional arguments. This information is added to the name2signature list, which is located in the *ast_common_visitor.h* file. to ensure that the compiler recognizes and processes the intrinsic function correctly, including its argument requirements and usage specifications.

- The third step involves creating a dedicated namespace and defining **eval_<function_name>** and **instantiate_<function_name>** within it. This is done in the *intrinsic_functions.h* file for the elemental intrinsic functions and *intrinsic_array_function_registry.h* file for array intrinsic functions. This structured approach helps organize and encapsulate the implementation logic for each intrinsic function. Within the namespace, we define eval_<function_name> to handle the compile-time evaluation of the intrinsic function, processing arguments and generating the corresponding ASR representation. Similarly, instantiate_<function_name> is defined to manage the instantiation process,

including variable declarations, function calls, and ASR construction.

- The final step involves registering the intrinsic function in the *intrinsic_function_registry.h* file. We mention the name of the intrinsic in a dictionary defined in this file that maps each intrinsic function to its corresponding ID, facilitating efficient lookup and management within the LFortran compiler framework. There's another dictionary defined here which allows the efficient fetching of the intrinsic name in the frontend.

Let's understand the procedure of implementing an intrinsic function in LFortran using an example to illustrate each step effectively.

The SQRT function is an elemental function in Fortran that calculates the square root of a given argument X. It falls under the class of elemental functions and follows the syntax RESULT = SQRT(X). The argument X must have a type of REAL or COMPLEX, and the return value of the function is also of type REAL or COMPLEX, depending on the type of X. Additionally, the kind type parameter of the return value matches that of the argument X.

```
program test_sqrt
  real(8) :: x = 2.0_8
  complex :: z = (1.0, 2.0)
  x = sqrt(x)
  z = sqrt(z)
end program test_sqrt
```

Listing 2: Example of SQRT(X)

- The first step is to declare the following in *intrinsic_func_registry_util_gen.py* file.

```
"Sqrt": [
        {
            "args": [("real",), ("complex",)],
            "ret_type_arg_idx": 0
        },
    ],
```

Listing 3: Define return and arg types for sqrt

This on building generates a namespace sqrt with the verify_args and the create_Sqrt function in the *intrinsic_function_registry_util.h* file.

```
namespace Sqrt {

    static inline void verify_args(const ASR::
    IntrinsicElementalFunction_t& x, diag::Diagnostics&
    diagnostics) {
        if (x.n_args == 1)  {
            ASRUtils::require_impl(x.m_overload_id == 0,
    "Overload Id for Sqrt expected to be 0, found " + std
    ::to_string(x.m_overload_id), x.base.base.loc,
    diagnostics);
            ASR::ttype_t *arg_type0 = ASRUtils::
    type_get_past_const(ASRUtils::expr_type(x.m_args[0]));
            ASRUtils::require_impl((is_real(*arg_type0))
    || (is_complex(*arg_type0)), "Unexpected args, Sqrt
    expects (real) or (complex) as arguments", x.base.base
    .loc, diagnostics);
        }
        else {
            ASRUtils::require_impl(false, "Unexpected
    number of args, Sqrt takes 1 arguments, found " + std
    ::to_string(x.n_args), x.base.base.loc, diagnostics);
        }
    }

    static inline ASR::asr_t* create_Sqrt(Allocator& al,
    const Location& loc, Vec<ASR::expr_t*>& args, diag::
    Diagnostics& diag) {
        if (args.size() == 1)  {
            ASR::ttype_t *arg_type0 = ASRUtils::
    type_get_past_const(ASRUtils::expr_type(args[0]));
            if(!((is_real(*arg_type0)) || (is_complex(*
    arg_type0)))) {
                append_error(diag, "Unexpected args, Sqrt
     expects (real) or (complex) as arguments", loc);
                return nullptr;
            }
        }
        else {
            append_error(diag, "Unexpected number of args
    , Sqrt takes 1 arguments, found " + std::to_string(
    args.size()), loc);
            return nullptr;
        }
        ASRUtils::ExprStmtDuplicator expr_duplicator(al);
        expr_duplicator.allow_procedure_calls = true;
```

```
28          ASR::ttype_t* type_ = expr_duplicator.
    duplicate_ttype(expr_type(args[0]));
29          ASR::ttype_t *return_type = type_;
30          ASR::expr_t *m_value = nullptr;
31          Vec<ASR::expr_t*> m_args; m_args.reserve(al, 1);
32          m_args.push_back(al, args[0]);
33          if (all_args_evaluated(m_args)) {
34              Vec<ASR::expr_t*> args_values; args_values.
    reserve(al, 1);
35              args_values.push_back(al, expr_value(m_args
    [0]));
36              m_value = eval_Sqrt(al, loc, return_type,
    args_values, diag);
37          }
38          return ASR::make_IntrinsicElementalFunction_t(al,
     loc, static_cast<int64_t>(IntrinsicElementalFunctions
    ::Sqrt), m_args.p, m_args.n, 0, return_type, m_value);
39      }
40  }
```

Listing 4: Namespace containing verify_args and create_Sqrt function

The verify_args function in LFortran's ASR implementation serves as a key validation mechanism for intrinsic functions. It verifies the correct number of arguments and their types, ensuring adherence to function specifications. For instance, when verifying the sqrt function, it checks that the number of arguments is exactly one and that the argument type is either real or complex. Additionally, for overloaded functions like sqrt, it checks the overload ID to guarantee the correct function variant is called. When discrepancies are detected, detailed error messages are generated using the diagnostics parameter, providing precise feedback on argument count, types, and overload mismatches.

The create_Sqrt function in LFortran's ASR (Abstract Syntax Representation) system is designed to generate the ASR representation for the sqrt intrinsic function. In case of invalid arguments, it appends precise error messages to the diagnostics and returns nullptr. Assuming valid arguments, the function proceeds to construct the ASR representation for the sqrt function, including setting the return type, initializing argument values, and evaluating the function if necessary. Ultimately, the function creates an instance of the appropriate ASR

class representing the sqrt intrinsic function, adhering to LFortran's standards and conventions while ensuring the integrity of the ASR representation.

- Declare the name of the intrinsic function along with the minimum and maximum number of arguments that it can take and the keywords for the same in the name2signature list, which is located in the *ast_common_visitor.h* file as follows:

```
{"sqrt", {IntrinsicSignature({"X"}, 1, 1)}},
```

Listing 5: SQRT node in AST

The following is the AST for the example we saw above for the sqrt intrinsic.

```
(Assignment
    0
    x
    (FuncCallOrArray
        sqrt
        []
        [(()
        x
        ()
        0)]
        []
        []
        []
    )
    ()
)
```

Listing 6: FuncCallOrArray node for intrinsic SQRT in the AST

The FuncCallOrArray node shown in the AST is visited when we call any intrinsic function, sqrt in this case. This function then calls intrinsic_as_node which checks if the function being called is an intrinsic function and accordingly calls the handle_intrinsic_node_args which fetches the signature of the intrinsic function defined in the step 2 and checks if the number of arguments provided matches the signature of the function and fills the optional arguments with default values and finally call the create_sqrt.
The following ASR is obtained accordingly and on looking at it we can see that it is represented using IntrinsicElementalFunction node now.

```
(Assignment
                        (Var 2 x)
                        (IntrinsicElementalFunction
                            Sqrt
                            [(Var 2 x)]
                            0
                            (Real 8)
                            ()
                        )
                        ()
                    )
```

Listing 7: IntrinsicElementalFunnction node for intrinsic SQRT in the ASR

- Then we create a namespace sqrt and define **eval_Sqrt** and **instantiate_Sqrt** in the *intrinsic_function_registry.h* file.

```
namespace Sqrt {

    static ASR::expr_t *eval_Sqrt(Allocator &al, const Location &loc,
            ASR::ttype_t* arg_type, Vec<ASR::expr_t*> & args, diag::Diagnostics& /*diag*/) {
        ASRUtils::ASRBuilder b(al, loc);
        if (is_real(*arg_type)) {
            double val = ASR::down_cast<ASR::RealConstant_t>(expr_value(args[0]))->m_r;
            return b.f(std::sqrt(val), arg_type);
        } else {
            std::complex<double> crv;
            if( ASRUtils::extract_value(args[0], crv) ) {
                std::complex<double> val = std::sqrt(crv);
                return ASRUtils::EXPR(ASR::make_ComplexConstant_t(
                    al, loc, val.real(), val.imag(), arg_type));
            } else {
                return nullptr;
            }
        }
    }

    static inline ASR::expr_t* instantiate_Sqrt(Allocator &al, const Location &loc,
```

```
22              SymbolTable *scope, Vec<ASR::ttype_t*>&
    arg_types, ASR::ttype_t *return_type,
23              Vec<ASR::call_arg_t>& new_args, int64_t
    overload_id) {
24          ASR::ttype_t* arg_type = arg_types[0];
25          if (is_real(*arg_type)) {
26              return EXPR(ASR::make_RealSqrt_t(al, loc,
27                  new_args[0].m_value, return_type, nullptr
    ));
28          } else {
29              return UnaryIntrinsicFunction::
    instantiate_functions(al, loc, scope,
30                  "sqrt", arg_type, return_type, new_args,
    overload_id);
31          }
32      }
33
34  }  // namespace Sqrt
```

Listing 8: Namespace containing eval_Sqrt and instantiate_Sqrt function

The eval_Sqrt function in LFortran's ASR (Abstract Syntax Representation) system handles the compile-time evaluation of a sqrt. It calculates the square root of a real or complex argument and creates an ASR instance with the computed result, showcasing the function's role in producing valid ASR representations of intrinsic function evaluations.

The instantiate_Sqrt function in the namespace Sqrt handles the instantiation of the sqrt intrinsic function within LFortran's ASR (Abstract Syntax Representation) system. This function takes into account the argument types, return type, and overload ID to generate the appropriate ASR representation for the sqrt function. Specifically, if the argument type is real, it creates an ASR instance of RealSqrt_t with the corresponding arguments. However, if the argument type is not real, it delegates the instantiation process to the UnaryIntrinsicFunction::instantiate_functions method, which handles the instantiation for unary intrinsic functions like sqrt. This structured approach ensures the correct instantiation of the sqrt function within the LFortran environment, maintaining consistency and adherence to language specifications.

- Finally, register the function inside intrinsic_function_registry.h file in order to call it to the frontend. Adding tests would be a plus and it helps checking if the implementation works perfectly and gives the expected output.

3 Deliverables

This section offers an in-depth overview of the intrinsic functions and outlines the approach I'll take to address them during the project.

3.1 Implementing intrinsic functions from scratch

I'll focus on implementing the pending intrinsic functions listed in the tracker: #492. The initial step involves fully implementing all non-coarray and non-atomic functions specified in the tracker. This includes functions like associated, spread eoshift, spacing, merge_bits and other intrinsics like the bessel functions. Following this, I'll proceed to add comprehensive tests for these implementations to ensure correctness and efficiency.

I will keep the tracker up-to-date with the intrinsic procedures, incorporating new ones as they become relevant and marking the implemented intrinsics accordingly. This tracker will be a comprehensive record of the GFortran supported intrinsic procedures, ensuring that the project stays aligned with the latest standards and functionalities.

There are some functions which are not present in the list of standard intrinsic procedures but are helpful for a compiler. One of the examples of such a function is ToLowerCase implemented in LFortran which is kind of a support function in order to do a case insensitive comparison of strings. We can lowercase all the strings and then compare them in order to ignore the case. This function has been implemented with the PR #3556. I'll add such functions into the list and will implement them too.

3.2 Implementing Bessel functions

For the bessel functions, we currently use the code from external sources. The implementation of most of them follows the same code, so we can create

a helper function that facilitates these implementation, register it in ASR-Builder and use that in the intrinsic_functions.h file. I'm also planning to implement the complete function using ASR.

3.3 Handling coarray and atomic functions

For coarray and atomic functions mentioned in the tracker #492, I'll prioritize providing clear and informative error messages indicating their lack of implementation within the project scope. If feasible and time permits, I may explore the possibility of implementing these functions, although they are considered out of scope for the current project objectives. I'll ensure to update the tracker regularly to reflect the progress and maintain transparency throughout the implementation phase.

3.4 Cleanup of runtime library

We currently rely on BindC for system interfacing, but there's a need to introduce dedicated ASR nodes for tasks like managing time, handling file I/O, and executing intrinsic math functions. This shift aims to make the runtime library purely Fortran-based, eliminating the need for C calls. Additionally, we may integrate "LFortran builtin" functions to access specialized features, ensuring compatibility across different compilers. With analysis of the LFortran runtime library, we can categorize the files into distinct groups to better understand their content and purpose within the project:

- **Impure**
  - **lfortran_intrinsic_sin.f90**: This file contains a pure Fortran implementation of the sine function (sin), but it's currently not utilized.
  - **lfortran_intrinsic_math.f90**: Most functions in this file have been ported to use the IntrinsicElementalFunction. My plan is to finish porting all functions and then remove this file.
  - **lfortran_intrinsic_bit.f90**: This file includes various bit manipulation functions, which have already been implemented using the intrinsic_function_registry mechanism. As a result, this file has been removed.

- **Pure**
  - **lfortran_intrinsic_iso_c_binding.f90**: Implements ISO_C_BINDING module functions such as c_associated, c_loc, c_ptr, which are currently in use.
  - **lfortran_intrinsic_iso_fortran_env.f90**: Defines constants and functions related to the Fortran environment provided by the ISO_FORTRAN_ENV intrinsic module.
  - **lfortran_intrinsic_ieee_arithmetic.f90**: This file contains stubs for IEEE arithmetic functions, which I plan to fully implement as part of the project.
  - **lfortran_intrinsic_string.f90**: Nearly completed cleaning up this file with only a few intrinsics left. A pull request (#3674) is opened, and I'll finalize it soon.
  - **lfortran_intrinsic_math2.f90**: Only one function, modulo, remains in this file. A PR (#3703) has been opened, and I'll complete it shortly.
  - **lfortran_intrinsic_kind.f90**: This file contained functions like selected_real_kind, selected_int_kind, selected_char_kind which have been implemented and hence this file has been removed.
  - **lfortran_intrinsic_trig.f90**: All the functions inside this file have been shifted to the intrinsic functions and this file has been removed too.
  - **lfortran_intrinsic_math3.f90**: All the functions inside this file have been ported to ASR and this file is removed.
- **Builtin**
  - **lfortran_intrinsic_builtin.f90**: Includes various interfaces, some of which are already implemented as intrinsics. I'll work on implementing the remaining interfaces and remove this file afterward.
  - **lfortran_intrinsic_optimization.f90**: Contains implementations of custom "special functions" used for optimization. I'll transfer these functions directly to ASR-¿ASR and eliminate this file entirely.
- **Custom**

- **lfortran_intrinsic_custom.f90**: Contains the newunit function, which also needs to be removed as part of the cleanup process.

Here is an issue to keep the track for the same: #3034

3.5 Enhancing Intrinsic Function Automation

With the Automation of intrinsic_function_* lists using intrinsic_func_registry_util_gen.py, we have achieved significant automation in managing intrinsic function lists using intrinsic_func_registry_util_gen.py. This automation has significantly reduced manual work during the implementation of intrinsic functions. While the automation is comprehensive, there are still areas for improvement to enhance robustness. For example, adding verify conditions such as kind1==kind2 for intrinsics that support arguments only of the same kind: #3574 can further strengthen the automation process. Currently, the automation covers most aspects, except for specific lists like INTRINSIC_NAME_CASE(Conjg), intrinsic_function_by_id_db, intrinsic_function_id_to_name, and intrinsic_function_by_name_db specified in #3383 The goal is to automate the generation of these lists as well through intrinsic_func_registry_util_gen.py, contributing to a more streamlined and efficient workflow in managing intrinsic functions within the project.

3.6 Refactor ASRBuilder

I plan to conduct a refactoring of the ASR Builder to enhance its efficiency and readability. One key aspect of this refactoring involves replacing numerous macros with callable functions. For instance, instead of having separate macros for converting integers of different sizes like i8(x), i16(x), and i32(x), we can introduce a single function i_t(x, t) where t denotes the kind of integer, allowing for a more streamlined and consistent interface.

Additionally, we can improve the overall interface of the ASR Builder to make it simpler and more concise. This includes identifying and eliminating repetitive code patterns, enhancing code reusability, and ensuring that the builder's functionality is intuitive and easy to use.

3.7 Support intrinsic functions in fortran backend

One of the aims of this project is to add support for all the intrinsics in the fortran backend. An issue #3416 is opened for it. I'll create a tracker for all the intrinsics and will try to completely register them in the fortran backend.

3.8 Intrinsic function design improvement

Currently, the intrinsic functions are categorised into two categories: elemental and non-elemental. However, the standard splits all intrinsics into the a number of classes such as atomic subroutine, collective subroutine, elemental function, elemental subroutine, inquiry function, pure subroutine , impure subroutine and transformational function.

This is explained in detail in the issue number #1658. In order to cover all these classes, I'll focus on the plan described in this issue about creating four major categories of intrinsic functions and they are: **IntrinsicElementalFunction**, **IntrinsicArrayFunction**, **IntrinsicImpureSubroutine** and **IntrinsicImpureFunction**, where the IntrinsicImpureSubroutine and IntrinsicImpureFunction are all intrinsics that do not fit into one of the first three categories.

For this, the first task would be to get rid of all frontend intrinsic modules which I'll do as part of the cleanup of runtime library and ensure none of those call into lfortran_intrinsics.c. Then will rename lfortran_intrinsics.c into lcompilers_runtime.c and focus on cleaning up lcompilers_runtime.c by moving/lifting things into ASR passes.

3.9 Performance options for intrinsic functions

In the realm of Fortran, performance in numerical computing is paramount. We categorize performance metrics into two key approaches: "Accuracy First, Performance Second" and "Performance First, Accuracy Second." The former prioritizes achieving highly accurate results for every single or double precision number, while the latter emphasizes obtaining optimal performance.

One notable example of performance enhancement is the utilization of the

Modulo operation with the flip sign pass. This involves employing a pass named "flip_sign," which generates an Abstract Syntax Tree Representation (ASR) that replaces flip sign operations with more efficient bit shifts.

Specifically, the transformation converts constructs like "if (modulo(number, 2) == 1) x = -x" into "x = xor(shiftl(int(number), 31), x)" for 32-bit numbers, or 63 for 64-bit numbers. The detection of flip signs occurs by examining a specific subtree within the ASR tree. This subtree should feature an If node as the parent, with the Compare attribute containing a call to the modulo intrinsic function (with the second argument as 2) and an IntegerConstant of 1. Additionally, the Statement attribute of the If node should consist of a single Assignment statement, where the right-hand side is a UnaryOp expression with the operand being the left-hand side.

To facilitate this detection process, the FlipSignVisitor employs attributes that are set to true only when the specified conditions are met in sequence.

Following successful detection, the subsequent phase involves replacing the flip sign subtree with a call to a generic procedure. This placeholder call is backend-agnostic, with the actual implementation generated based on the specified backend. Moreover, by utilizing the –fast flag, which optimizes functions for performance, the code is compiled with optimized settings, leading to improved overall performance.

Here is a PR which implements modulo function and also handles its optimization with the flip sign pass: #3703. I've planned to implement performance enhancements for all possible intrinsic functions within LFortran, aiming to significantly boost its computational efficiency.

3.10 Miscellaneous Goals

Some additional goals can be to include enhancing test coverage for intrinsic functions and optimizing the performance of passes related to intrinsic functions like "fast," "flip_sign," "fma," etc., to improve their robustness and efficiency. Also, if time permits, I'm planning to look for the asr to c, cpp and julia backends too.

4 Timeline

In this section, I outline a tentative plan for incorporating the discussed features from the previous section. This plan includes post-GSoC periods as part of the timeline. It's important to note that the list of intrinsic functions is extensive, and strictly adhering to the proposed timeline may pose challenges. Therefore, there is a possibility of slight deviations from the projected schedule, depending on the extent to which certain features are integrated into the project.

4.1 GSoC Period

According to the official dates, the complete program can be divided into three parts namely, **Community Bonding Period**, **Phase-1** and **Phase-2**. The details of each of these is discussed in the following subsections.

4.1.1 Community Bonding Period

During the period from May 1^{st} to May 26^{th}, 2024, spanning 3.5 weeks, I aim to finalize all intrinsic functions in the frontend, specifically targeting the implementation of all intrinsics in F23, excluding coarrays and atomics (although they will still be recognized, accompanied by a clear error message indicating their lack of implementation). My primary objective during this phase will be to efficiently list and implement these functions as quickly as possible, marking a significant advancement in the project. Additionally, I will actively collaborate with new contributors, maintain regular community interactions, and facilitate discussions on innovative ideas and implementation strategies to enhance collaboration and progress.

4.1.2 Phase 1

This phase starts from 27^{th} May, 2024 and ends at 12^{th} July, 2024, consisting of around 7 weeks. The weekly plan (tentative) for this duration is as follows,

- In the first three weeks, my focus will be on the smooth transition of implementing every intrinsic function using the intrinsic function registry. I will prioritize the cleanup of the Fortran runtime and the C runtime library. This involves removing redundant functions and tailoring

the libraries to be backend-specific. Additionally, I will introduce ASR nodes to represent tasks that the backend must directly implement, further optimizing the compilation process and ensuring backend-specific functionality is appropriately integrated. Given the current state of intrinsic functions in LFortran, there are numerous functions that require implementation and fixing, so will get them done, write tests and clean it up.

- Over the next three weeks, my focus will be on optimizing the performance of intrinsic functions. Each numerical function will have two implementations: one fast but less accurate, and one slower but highly accurate. Our fast implementation will be directly integrated into ASR (Abstract Syntax Representation), providing users with the flexibility to choose between accuracy and speed.

- I have allocated the seventh week as a buffer week in my schedule. This strategic decision allows me to address any missed deadlines without feeling stressed or pressured.

4.1.3 Phase 2

During this phase, which spans from 12^{th} July to 19^{st} August 2024, lasting around 6 weeks, I will first address any missed deadlines from phase 1 and prioritize their resolution.

Over the next three weeks, I'll work upon enhancing the intrinsic function automation, will improve upon already existing python script to generate more robust code and add other functionalities mentioned above in detail. I'll also cover refactoring of the ASR Builder and supporting the intrinsic functions in fortran backend within this period of time.

After this is done, I'll work on further improving the intrinsic function design which focuses on the plan to categorise the intrinsic functions into a number of categories as described in the plan above.

Finally, I will focus on identifying and fixing any issue with the intrinsic functions, utilizing minimal code reproductions of errors and addressing them based on function priority. This phase will also be the time to tackle miscellaneous features discovered during the implementation of intrinsic functions

in LFortran. If all listed intrinsic functions are successfully implemented, I'll proceed with testing all of them with good examples and improving them with nice error messages wherever required.

My goal is to maintain a work schedule equivalent to 40 hours per week, and I am open to adjusting workloads between phases to ensure efficient goal achievement.

4.2 Post-GSoC Period

After GSoC, my plan is to continue enhancing LFortran to ensure it is more robust and efficient for users. One area I will prioritize is compiling the remaining intrinsic functions related to co-arrays that are yet to be implemented and will continue working on the optimization options. This effort will contribute significantly to the overall usability and functionality of LFortran.

5 Acknowledgements

I would like to thank Ondrej Certik for his continuous feedback on my Merge Requests and teaching me how to tackle difficult issues while working for LFortran. I extend my thank to Pranav Goswami, Thirumalai Shaktivel, Ubaid Shaikh and other contributors at LFortran for resolving my queries, reviewing my pull requests and suggesting better approaches.

6 References

- Oracle Docs for Fortran
- Stanford documentation for Fortran
- Pennsylvania State University docs for Fortran
- University of Toronto docs for Fortran
- LLVM Language Reference Manual