Control-flow aware MLIR tracing

Gaëtan Lounes, Robin Gerzaguet, Matthieu Gautier

Control-flow aware MLIR tracing

Gaëtan LOUNES
Univ Rennes, CNRS, IRISA
France

Robin GERZAGUET
Univ Rennes, CNRS, IRISA
France

Matthieu GAUTIER
Univ Rennes, CNRS, IRISA
France

Abstract

MLIR (Multi-Level Intermediate Representation) has proven to be a scalable framework for building modern compilers. A key challenge, however, lies in effectively provide these compilers with input programs. Several methods have been explored: static methods, such as Domain-Specific Language, generic language front-end that transform an abstract syntax tree using tools like Polygeist, or leveraging compiler intermediate representation with Brutus. Dynamic approaches, such as tracing with JAX or Reactant, offer another solution. Each class of methods presents trade-offs: static approaches are easily integrated through compiler passes but lack runtime information, while dynamic tracing captures runtime behavior but cannot handle compile-time information. In this paper, we propose enhancing the Reactant tracing system by integrating the Julia compiler static analysis. This hybrid approach improves the expressiveness and precision of tracing. The effectiveness of our approach is demonstrated on the Polybench benchmark suite.

Keywords: JIT, MLIR, Tracing, IR to IR, Julia

ACM Reference Format: Gaëtan LOUNES,Robin GERZAGUET, and Matthieu GAUTIER. 2025 Control-flow aware MLIR tracing . In *36th International Workshop on Rapid System Prototyping (RSP '25), October 02, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 7 pages.

1 Introduction

The rapid evolution of heterogeneous computing and the growing complexity of modern software systems have intensified the needs for compiler infrastructures capable of bridging high-level programming abstractions to modular and generic representation. LLVM [14] has unified compiler back-end by providing a hardware agnostic Intermediate Representation (IR), but this form is not adapted to higher abstraction IR. The Multi-Level Intermediate Representation (MLIR) [13] framework has emerged as a powerful solution, enabling the construction of modular and extensible compilers [27] which can target a wide range of domains and architectures. However, a question remains: how to provide these compilers with an expressive language.

Julia [3], a high-level, high-performance programming language, has become a popular choice for scientific computing and rapid prototyping due to its compiler, dynamic type system, multiple dispatch and ecosystem especially for heterogeneous hardware. Because of it expressiveness, Julia appears as a great fit for an MLIR front-end. To that aim, two classes of MLIR front-ends have been explored. Firstly, static approaches use compiler information to achieve IR-to-IR conversion: from a compiler front-end [20] or middle-end [16, 18, 26] representation. Yet, they are inherently limited in capturing dynamic program behaviors and runtime information, which are increasingly important in domains like machine learning and scientific computing. On the other hand, dynamic tracing systems such as JAX [24] and Reactant [23] can capture the rich runtime information by recording program execution traces. While the latter excel at handling dynamic constructs and enabling features like automatic differentiation, for instance, they cannot automatically exploit static program structure, leading to losses of abstraction and reduced expressiveness.

This paper addresses these limitations by introducing a hybrid approach that integrates static analysis within the tracing system of Reactant inside the Julia compiler, making the tracing process aware of control-flow constructs and leveraging static type inference. By combining the strengths of both static and dynamic methodologies, our approach enables more precise and expressive MLIR code generation. We demonstrate the benefits of this approach for improving the quality and expressiveness of tracing systems with several examples from a standard benchmark.

2 Background

2.1 MLIR framework

MLIR is a compiler framework for building modular and scalable compilation toolchains using different levels of abstraction within the same IR. An MLIR toolchain is composed of several *dialects* defining, at different levels of abstraction, concepts such as *types*, *attributes*, and *operations*. Each IR is structured using *regions* and *blocks*, several interactions are showed in Figure 1. *Passes* define transformations which are applied on a dialect or between several dialects, and *translation* is a transformation which converts an MLIR IR to an external representation, such as LLVM IR or C.

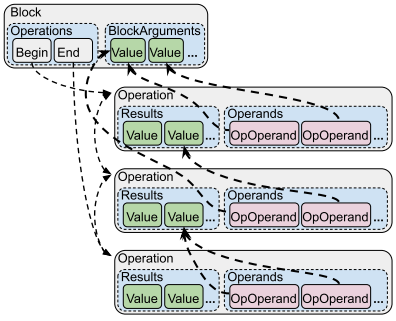


Figure 1: Interaction between *operation* and *value* in *block*.

For instance, high-level abstraction dialects such as `Linalg` or `TOSA` operate on immutable tensor types, abstracting away memory management. In contrast, lower-level dialects, like `MemRef`, explicitly model mutable memory buffers. A bufferization pass is the process of transforming operations on immutable tensors into equivalent operations on mutable buffers, thereby bridging the gap between high-level functional semantics and low-level memory-aware representations.

2.2 Julia Language

Julia [4] is a dynamically typed language using gradual typing backed by a JIT compiler and a simple but expressive type system [7]. Multiple dispatch allows several specializations for the same method thanks to the type system. The main interest is to provide a precise static dispatch and delegate to JIT when type information is missing. The Julia compiler middle-end is presented in the following sections.

2.2.1 A complete compiler

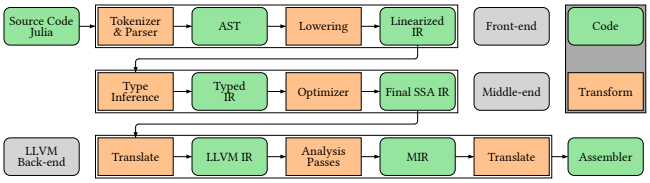


Figure 2: Julia compiler pipeline.

Julia provides a complete compiler following standard practice as shown in Figure 2 and integrating these paradigms: the source code is first tokenized and parsed into an Abstract Syntax Tree (AST). This form is then lowered and flattened to a `CodeInfo` IR. The middle-end starts by using this IR for type inference. The typed `CodeInfo` is then converted to `IRCode` IR for further analysis, such as inlining in the optimizer. The new IR is then translated to LLVM-IR for assembly emission in the back-end.

```
f(x::Union{Int, Float32}) =
begin
  p = if x isa Int
    x
  else
    x + 1
  end
  p
end
```

Listing 2: Example program

```
1 ─      Core.NewvarNode(:(p))
│   %2  = x isa Main.Int
└       goto #3 if not %2
2 ─ %4  = x
└       @_4 = %4
└       goto #4
3 ─      @_4 = x + 1
4 ┄ %8  = @_4
└       p = %8
└       %10 = p
└       return %10
```

Listing 3: Linearized IR of Listing 2

```
1 ─ %1 = (_2 isa Main.Int)::Bool
└       goto #3 if not %1
2 ─ %3 = π (_2, Int64)
└       goto #4
3 ─ %5 = π (_2, Float32)
└   %6 = Base.add_float(%5, 1.0)::Float32
4 ┄ %7 = φ (#2 => %3, #3 => %6)::Union{Float32, Int64}
└       return %7
=> Union{Float32, Int64}
```

Listing 4: Final SSA IR of Listing 2

An example of source code and associated lowered `CodeInfo` and optimized `IRCode`.

2.2.2 Middle-end entry point

The `CodeInfo` IR as shown on Listing 3 has explicit and unstructured control flow using basic blocks, which are linked by a terminator: function return (`return`), implicit branch, branch (`goto`), or conditional branch (`goto if not`). Moreover, each block follows Static Single Assignment (SSA) except for variables, which are handled by slots. In practice during lowering, SSA values are defined and used inside the same block, and slots are used to define inter-block interactions.

2.2.3 Type inference

Unlike ML-based [12] languages using the Hindley-Milner [19] type system, which needs a complete static typing, Julia, with its dynamically typed nature, has different trade-offs. The JIT and Julia type system offer a coarser approach by using only incomplete type information during compile time and resolving ambiguities during runtime. The type inference algorithm used is based on a forward dataflow analysis, which stores every slot type information inside a context and propagates it using abstract interpretation following the control flow until the algorithm converges. The type system uses a partial order relation with sub-typing to refine the type inference via a type lattice [15] . Next, inferred code enables the use of the optimizer.

2.2.4 Optimizer

The optimizer starts by converting the typed `CodeInfo` IR to the `IRCode` IR. The slots are transformed into SSA, and phi-nodes are inserted to represent control-flow joins. The `IRCode` IR follows the `CodeInfo` IR structure but contains additional nodes used by the optimizer, such as π nodes to handle more precise type information as shown in Listing 4, or Upsilon nodes to deal with error handling. Optimization passes such as inlining, or dead code elimination are applied, and the resulting IR is used by the LLVM back-end.

2.3 IR to IR compilers

The Julia compiler presented above is modular and can be modified to target different platforms. This section presents how it can be used to generate MLIR programs.

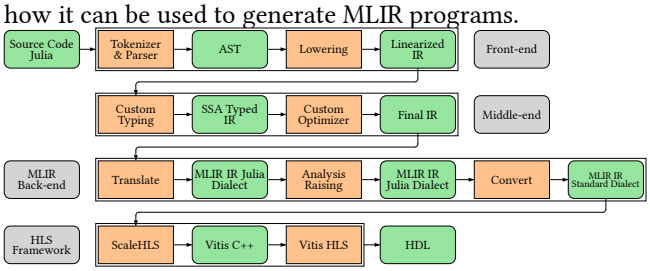


Figure 3: MLIR front-end using static informations provided by the Julia compiler and used by ScaleHLS.

An initial approach developed to generate an MLIR representation with Julia has been explored. It is based on other heterogeneous compilers developed for Julia, such as GPUCompiler [2], which leverages the Julia compiler and supports GPU programming using a custom LLVM back-end. The main idea to target MLIR is to fit Julia semantics inside `IRCode` to MLIR semantics by introducing a Julia dialect and translating it to standard MLIR dialects [8]. Further transformations to retrieve control flow have been explored for High Level Synthesis [16], leveraging both static analysis and ScaleHLS [28] MLIR backend. The pipeline of this compiler is shown in Figure 3. Another approach,

without defining a custom Julia dialect, has been developed [18, 26], targeting higher-level semantics with immutable objects such as tensors.

Either approach requires substantial effort to match MLIR semantics: the first is easier to translate to MLIR but loses abstraction by using low-level information from `JuliaIR`, and the second must ensure adaptation of Julia source code before compilation.

Moreover, both approaches use static type system information, so coarse type information from the compiler and the dynamic nature of Julia limit the scalability of these approaches. Fortunately, tracing systems are a good fit to handle them.

2.4 Tracing system

A tracing system is a tool which monitors program execution and records a trace. For instance, Autograd [17] is a Python tracing system used to generate gradients from arbitrary Python programs. It computes a trace during program execution on which it applies a Jacobian matrix. However, this method has limitations because there are multiple ways to compute a gradient, and the traced one is not always computationally optimal. Non traced approaches can be more efficient: Enzyme [22] integrates auto-differentiation during compilation and outperforms Autograd [21]. Hence, tracing systems enable the use of runtime information but are limited by their trace. A similar method can be used for code generation.

2.5 Jax & Reactant

Jax [11] is a domain-specific tracing JIT compiler that aims to generate, from a Python program, an MLIR representation, using the XLA [1] dialect `StableHLO` [10] and compiler infrastructure, to target heterogeneous platforms such as CPU, GPU, or TPU, and, for instance, enables automatic differentiation with Enzyme. The main goal of this dialect is to have a high-level and expressive semantic. For instance, it contains operations to model neural networks, control flow or functional behaviors. Reactant follows a similar path by providing a tracing system integrated into Julia paradigms of multi-dispatch and metaprogramming. Notably, the tracing system extensively uses both the Julia type system and JIT compiler.

For these, traced objects are not gradients anymore but MLIR objects which aim to generate a MLIR program. In practise, `Traced{T} where T` is a parametric type used to store an MLIR `Value` as defined at Figure 1. Using multi-dispatch, Reactant defines methods, which use `Traced` types, generate MLIR code and return `Traced` types too. This way, it can generate MLIR programs by running the trace. For instance, Listing 5 presents a simple program which is executed with arguments `(Traced{Int}(), 10, 10)` where only the first argument is a MLIR variable. A trace is generated in Listing 6 and each of these calls generates MLIR code as seen in Listing 7.

```
f = (x, y, b) ->
  x * y + b
```

Listing 5: Julia program.

```
call f(x::Traced{Int}, 10::Int, 10::Int)
 call Base.(*)(x::Traced{Int}, 10::Int)::Traced{Int}
 call Base.(+)(tmp::Traced{Int}, 10::Int)::Traced{Int}
```

Listing 6: Trace generated.

```
func.func @main(%x: tensor<i64>) -> tensor<i64> {
  %c1 = stablehlo.constant dense<10> : tensor<i64>
  %tmp = stablehlo.multiply %x, %c1 : tensor<i64>
  %c2 = stablehlo.constant dense<10> : tensor<i64>
  %tmp2 = stablehlo.add %tmp, %c2 : tensor<i64>
  return %tmp2 : tensor<i64>}
```

Listing 7: MLIR generated with SSA renamed to fit source program.

As discussed in §2.4, tracing systems operate by recording execution traces, which can lead to suboptimal behavior in certain cases. A similar limitation applies to traced code generation: since these systems rely solely on runtime traces, they lack visibility into static structures such as control flow. However, such static information is crucial for generating high-quality code. As a result, these systems are unable to produce control-flow–aware MLIR.

3 Proposed hybrid solution

The current tradeoffs of both methods limit the expressiveness of a general front-end. The Julia middle-end presented in §2.2 shows the flexibility and power of the compiler. Furthermore, we notice that the tracing approach developed by Reactant is integrated inside the Julia compiler. Hence, static analysis developed in §2.3 can enrich the Reactant tracing system to lift the expressiveness gap.

3.1 Problem statement

Traced control flow refers to branches that are controlled by a `Traced` type variable. Hence, the tracing system in this case is not local to an operation anymore, and the entire basic blocks must be considered by the tracing system because they affect the generated MLIR. In practice, slots used in these basic blocks must be traced. While Autograd only needs the executed line and a context to compute gradients, to support traced control flow, Reactant needs to trace every control flow path to produce a precise MLIR. Therefore, the proposed augmentation consists of integrating this mechanism into the type inference algorithm and modifying the IR to ensure that the tracing system can handle each path.

3.2 Extended type inference

As shown above, traced control-flow requires considering entire code regions inside the tracing system. Thus, type inference must be changed to integrate this new semantic. To achieve this, Julia type inference must be extended to recognize traced control-flow constructs and adjust variable slot types accordingly. However, the Reactant tracing system and Julia type system impose certain constraints on how such extensions can be implemented. The following sections detail the underlying mechanism.

As presented in §2.2.3, the Julia compiler uses a dataflow-based type inference [5]. Listing 8 presents a simplified version of this algorithm, omitting recursion, effects, error handling, and SSA typing. Red lines refer to changed behav-

iors added to support traced control flow and are explained in the next sections.

```
function FunctionTypeInference(interpreter, frame)                    Pseudo
    W = {1} //Basic Block work list
    Γ = [⊥ x length(frame.cfg) x length(frame.slots)] //slot context
1   T = empty_TCFT()
    while !is_empty(W)
      current_block <- pop(W)
      first_stmt <- first(current_block)
      last_stmt <- last(current_block)
      local_Γ <- copy(Γ[current_block])
2     move(T, current_block)
      for stmt_index in first_stmt:last_stmt
        stmt <- frame.code[stmt_index]
        if stmt_index == last_stmt //terminator
          match stmt
            case Goto(bb)
            if update_states(interpreter, frame, bb, local_Γ)
              push(W, bb)
            end if
            case GotoBranch(bb_true, bb_false)
              rt <- abstract_eval_stmt(interpreter, stmt, local_Γ, frame)
              if rt != Bool
                return
              end if
3             add_tcf(frame, rt)
              if update_states(interpreter, frame, bb_false, local_Γ)
                push(W, bb_false)
              end if
              if update_states(interpreter, frame, bb_true, local_Γ)
                push(W, bb_true)
              end if
            end
            ...
          end match
        else
          rt <- abstract_eval_stmt(interpreter, stmt, local_Γ, frame) //
          abstract evaluate the stmt using the current local slot context
4         check_update_slot(T, frame, rt)
5         upgrade_loop(T, frame, rt)
          update_state(intepreter, frame, local_Γ, rt) //apply slot change
          if any to the corresponding slot in local_Γ
        end if
      end for
    end while
end function
```

Listing 8: Simplified type inference dataflow algorithm used by the Julia compiler.

3.2.1 Traced control-flow lifting

Raising passes [6] are a class of transformations used to retrieve implicit abstractions in an IR. Control-flow lifting [25] is one such transformation and is used to recover, from an unstructured IR, the control flow structure such as `for` or `if`. The main idea is to analyze the Control Flow Graph (CFG): each structure is composed of a header block, terminal block, body blocks, and a latch block for a `for` loop, or true and false branch blocks for an `if`. Thus, from the header block, an `if` can be detected by the presence in the CFG of two successors and by exploiting post-dominance analysis to detect the terminal block, which must post-dominate the branch blocks. Similarly, loops can be detected by checking for the presence of cycles in the CFG. However, it is to note that not all CFG structures are relevant: only the portions corresponding to traced control flow need to be lifted. Consequently, the analysis must be directly integrated into the type inference process. In practice, it is performed when inferring the terminator of a `goto if not` branch, and the results are recorded in a Traced Control Flow Tree (TCFT). The control flow context is updated following the exploration of the dataflow analysis and is either used to add an element or to navigate inside the TCFT. The context is created by Listing 8 (1), moved by (2) and the tree is filled by (3). Hence, the control flow context can be used during slot type inference.

3.2.2 Slot promotion

```
1 ─       Core.NewvarNode(:(p))                          JuliaIR CodeInfo
│   %2  = x isa Main.Int
└──       goto #3 if not %2
2 ─ %4  = x
│         @_4 = %4
└──       goto #4
3 ─       @_4 = x + 1
4 ┄ %8  = @_4
│         p = %8
│   %10 = p
└──       return %10
```

Figure 4: Highlighting of slot usage between basic block of Listing 2.

As visible in Figure 4, `CodeInfo` IR uses slots to represent variable storage, which, unlike SSA values, are not scoped to individual basic blocks. Therefore, accounting for traced control flow involves adapting the slot types according to the specific control-flow structure they are part of. An important point is that these changes cannot be integrated into the type lattice as inference rules because there is no subtype relationship between `T` and `Traced{T}`. Adding this rule will break the dataflow type inference algorithm. A way to integrate these new pseudo-inference rules is to directly modify `CodeInfo` by inserting a promotion call, which takes an object `T` and returns a `Traced{T}`.

Now, we define rules to apply this type inference change: for `if` and `for` structures, each slot written inside bodies or branches must be upgraded if its liveness exceeds the structure scope. The idea behind this is to ensure that TCFT slots cannot be evaluated as Julia variables because they now have a dependency on the traced object of the TCF. Two additional rules must be addressed: `if` legalization: this structure condition cannot be `Traced{Bool}` because that type does not pass type inference; as a result, the condition must remain in the original, untraced form. The `for` structure uses CFG cycles and variables named accumulators, which are changed each cycle. Upgrading a slot in the bodies is thus not enough to get precise type information: their slot write definition must be upgraded to get exact type information. For written slots, there are three cases of definition upgrade:

- local to `for` bodies: local definitions are not upgraded because they do not affect the rest of the program.
- local to TCFT branch: inside a tracing context, the slot definition just needs to be upgraded.
- global: an upgraded slot is inserted at the root header of the TCFT so that the upgrade impact is minimal on the rest of the program.

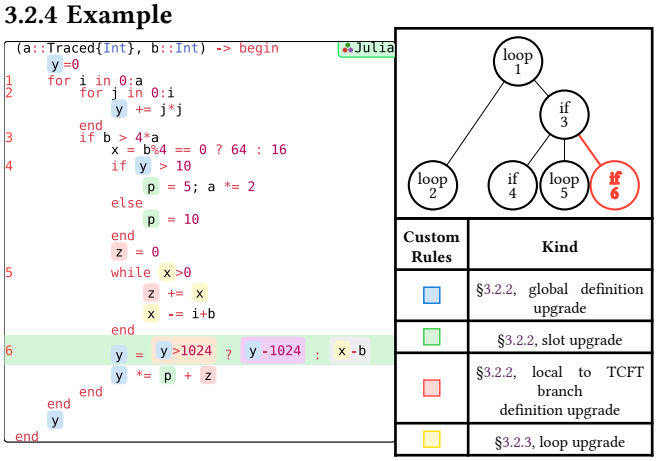
An important point is that the last two cases modify non-local type inference, so type inference must be restarted. This behavior is achieved by Listing 8 -(4).

3.2.3 Loop promotion

Another benefit of modifying the type inference algorithm is that one can detect loops which could be upgraded to

traced loops: if a loop body type infers an expression which returns a `Traced` type, as seen in Listing 8 (5), then this loop and all the ones in the current TCFT branch can be promoted. In practice, this promotion consists of adding an upgrade function call to the loop iterator. This transformation, however, interferes with type inference, which must subsequently be recomputed.

3.2.4 Example

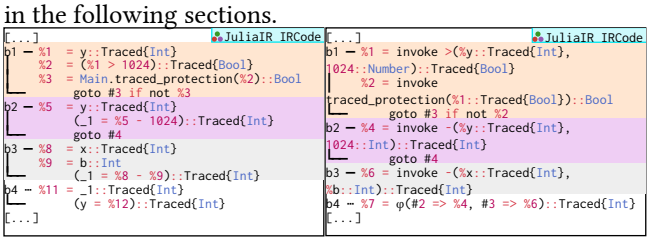


Listing 9: Example showing the different kinds of type inference customization and associated TCFT.

3.3 New trace entry

At this point, the type system is aware of the traced control flow. However, this alone is not sufficient to produce a fully traced MLIR program: currently, not all branches are evaluated because the tracing system follows a trace. Additional work must be done on the IR to achieve that goal by creating new tracing calls which generate `StableHLO` control-flow. To take advantage of TCFG, we must work on the same CFG, so these transformations are handled before the optimizer on the `IRCode` IR.

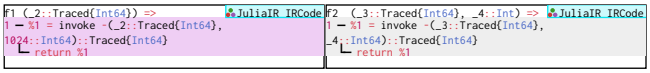
We focus on the ternary operation in green at Listing 9 in the following sections.



Listing 10: `CodeInfo` and `IRCode` before extraction of Listing 9

3.3.1 IR extraction

A first step is to extract bodies and branches using TCFG information see Listing 10: the idea is to create independent sub-representations. To extract a region of an IR, a new IR is created containing only the blocks inside this region, with a new return: all SSA in the region used in the original IR. A similar idea is used to deal with SSA defined before the extracted region: new arguments are added to fill the gap, as shown in Listing 11.



Listing 11: Extracted branches.

3.3.2 New JIT call

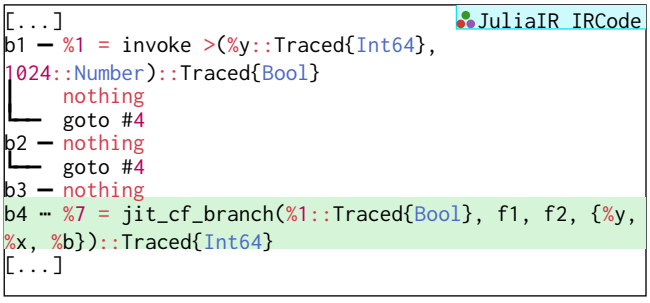
Every traced branch can now be executed independently of the IR. So we create a new function that creates the `StableHLO` operation and fills its regions. To achieve this, extracted IRs are compiled and executed at runtime and generate sub-traces, which are added to the operation region. Runtime checks must be added to verify the integrity of types following `StableHLO` control flow structure semantics.

3.3.3 Filling the semantics disparity

Another important semantic difference is the handling of mutation: as hinted in §2.1, `StableHLO` is a dialect working on tensors and which cannot handle mutation. Julia, however, can mix tensor and buffer semantics, so specific mechanisms must be introduced to support them for control flow. Reactant supports mutation by changing the `Value` inside a traced object: for instance, a mutation of an array `%a` in the MLIR representation creates operations to the muted array `%a'` and the traced value now points to `%a'`. This idea can be reused for control flow: inside sub-trace, Julia SSA mutation can be detected by checking `Value` before and after sub-trace execution. Then, these muted `Value` must be integrated inside the MLIR operation region. An example can be found in Listing 13 and generated MLIR in Listing 15.

3.3.4 IR & φ-nodes simplification

This new function must be inserted inside the original IR. To achieve that, all traced functions inside TCFG must be cleaned, bodies and branches must be removed, and the CFG simplified. An important part of this cleanup is to handle phi nodes properly: the IR is a mix between traced and non-traced control flow and only TCF is removed and replaced by the new control flow call. Listing 12 presents the IR after this step.



Listing 12: Modified IR able to trace the ternary operation.

3.4 Usage

After theses changes, TCF is replaced with new calls recursively and so a trace of a program can catch all CF informations for MLIR generation. Thus, the standard Reactant tracing can be used to generate the control flow enabled MLIR.

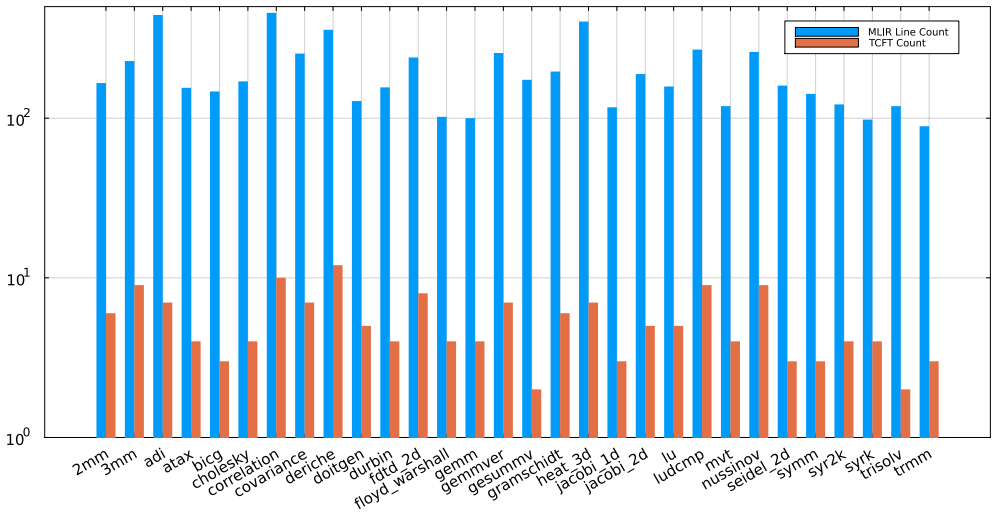


Figure 6: Front-end result on Polybench test suite

4 Validations

4.1 Basic Examples

We show the interest of our approach with two examples:

```julia
function first_row_to_column(A)
  for i in axes(A,1)
      A[i, 1] = A[1, i]
  end
  A
end
```

Listing 13: A simple kernel showing loop promotion and mutable semantics.

```julia
function test(cond, value::Int)
  value = cond ? value * 2 : 0
end
```

Listing 14: An program which cannot be traced by Reactant where cond is a `Traced{Bool}`.

We compare the obtained tracing between Reactant and our approach using `first_row_to_colomn` (Listing 13) with an 2d `Int32` matrix of size 4096. The considered metric is the number of lines of the generated MLIR program.

Tracer	Line count
Reactant	45062
CF aware approach	29

Table 2: Listing 13 tracing result.

```
module {
  func.func @first_row_to_column(%arg0: tensor<4096x4096xi64>) -> tensor<4096x4096xi64> {
    %c = stablehlo.constant dense<1> : tensor<i64>
    %0:2 = stablehlo.while(%iterArg = %c, %iterArg_0 = %arg0) : tensor<i64>, tensor<4096x4096xi64>
     cond {
      %c_1 = stablehlo.constant dense<4096> : tensor<i64>
      %1 = stablehlo.compare  LT, %iterArg, %c_1 : (tensor<i64>, tensor<i64>) -> tensor<i1>
      stablehlo.return %1 : tensor<i1>
    } do {
      %c_1 = stablehlo.constant dense<1> : tensor<i64>
      %c_2 = stablehlo.constant dense<1> : tensor<i64>
      %1 = stablehlo.subtract %c_1, %c_2 : tensor<i64>
      %2 = stablehlo.convert %1 : (tensor<i64>) -> tensor<i32>
      %c_3 = stablehlo.constant dense<1> : tensor<i64>
      %3 = stablehlo.subtract %iterArg, %c_3 : tensor<i64>
      %4 = stablehlo.convert %3 : (tensor<i64>) -> tensor<i32>
      %5 = stablehlo.dynamic_slice %iterArg_0, %2, %4, sizes = [1, 1] : (tensor<4096x4096xi64>, tensor<i32>, tensor<i32>) -> tensor<1x1xi64>
      %6 = stablehlo.transpose %5, dims = [1, 0] : (tensor<1x1xi64>) -> tensor<1x1xi64>
      %7 = stablehlo.reshape %6 : (tensor<1x1xi64>) -> tensor<i64>
      %8 = stablehlo.transpose %7, dims = [] : (tensor<i64>) -> tensor<i64>
      %9 = stablehlo.broadcast_in_dim %8, dims = [] : (tensor<i64>) -> tensor<1x1xi64>
      %10 = stablehlo.convert %9 : tensor<1x1xi64>
      %c_4 = stablehlo.constant dense<1> : tensor<i64>
      %11 = stablehlo.subtract %iterArg, %c_4 : tensor<i64>
      %12 = stablehlo.convert %11 : (tensor<i64>) -> tensor<i32>
      %c_5 = stablehlo.constant dense<0> : tensor<i32>
      %13 = stablehlo.dynamic_update_slice %iterArg_0, %10, %12, %c_5 : (tensor<4096x4096xi64>, tensor<1x1xi64>, tensor<i32>, tensor<i32>) ->
      tensor<4096x4096xi64>
      %c_6 = stablehlo.constant dense<1> : tensor<i64>
      %14 = stablehlo.add %iterArg, %c_6 : tensor<i64>
      stablehlo.return %14, %13 : tensor<i64>, tensor<4096x4096xi64>
    }
    stablehlo.return %0#1 : tensor<4096x4096xi64>
  }
}
```

Listing 15: Traced MLIR generated for Listing 13.

Table 2 shows that Reactant tracing of a `for` loop leads to massive code generation, caused by the fact that the loop is unrolled and the loop body is executed 4096 times. The CF aware extension is able to detect upgradable loops, change type inference and generate a compact MLIR representation using the `while` operation as shown in Listing 15. Listing 14 shows an example where Reactant cannot pass type inference because of the branch with non boolean type, as `Traced{Bool}` is not a `Bool`, and can not even start tracing, our method passes inference and traces it.

4.2 Reference benchmark

Polybench [9] is a collection of 30 numerical kernels featuring complex CF with nested loops and conditions, focused on linear algebra computations (`gemm` or `cholesky`), image manipulation (`deriche`), or dynamic programming (`floyd_warshall`). The benchmark was rewritten in Julia to match the C implementation. We test our approach against the entire benchmark by tracing every array arguments. We verify the generated MLIR by running the MLIR verifier infrastructure, which enforces semantics rules such as dominance or dialect-specific restrictions and by using the Reactant MLIR interpreter.

Our approach passes the entire benchmark. For each test, we decide to represent the size of the TCFT and the number of lines in the output MLIR, in practice this is computed by counting MLIR operations.

Figure 6 presents the result for each one, the benches are composed of between 2 and 12 traced control flow structures and between 89 and 457 MLIR lines. Thus, our approach is able to handle the CF of non-trivial programs while producing compact MLIR and can be used by MLIR compilers.

5 Conclusion

In this paper, we have introduced a new hybrid approach that integrates static analysis into the tracing system of Reactant by making the tracing process aware of control-flow constructs. Our contribution operates on two levels: by extending the type inference to adapt to the new semantics of traced control-flow and by modifying the control-flow graph to apply these changes. As a result, this method enables more precise and expressive MLIR code generation from Julia programs.

References

[1] Sabne Amit. 2020. XLA : Compiling Machine Learning for Peak Performance.

[2] Tim Besard, Pieter Verstraete, and Bjorn De Sutter. 2016. High-Level GPU Programming in Julia.

[3] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (October 2018), 1–23. https://doi.org/10.1145/3276490

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Review* 59, 1 (January 2017), 65–98. https://doi.org/10.1137/141000671

[5] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. https://doi.org/10.48550/arXiv.1209.5145

[6] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive Raising in Multi-level IR. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, February 2021. IEEE, Seoul, Korea (South), 15–26. https://doi.org/10.1109/CGO51591.2021.9370332

[7] Benjamin Chung. 2023. A Type System for Julia.

[8] Valentin Churavy, Leon Shen, Becker McCoy R., and Stephen Neuendorffer. 2020. Brutus.jl. Retrieved from https://github.com/JuliaLabs/brutus/

[9] Polybench Contributors. 2012. Polybench 4.2 benchmarks collection. Retrieved from https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

[10] XLA Contributors. 2025. MLIR StableHLO Dialect. Retrieved from https://openxla.org/stablehlo/spec

[11] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling Machine Learning Programs via High-Level Tracing.

[12] Robert Harper, David MacQueen, and Robin Milner. 1986. *Standard Ml*. Department of Computer Science, University of Edinburgh.

[13] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law.

[14] Chris Lattner. 2008. *LLVM and Clang: Next Generation Compiler Technology*.

[15] Stanis law Zukowski. 1990. Introduction to Lattice Theory. *Formalized Mathematics* 1, 1 (1990), 215–222.

[16] Gaëtan Lounes, Robin Gerzaguet, and Matthieu Gautier. 2025. Flexible Front-End for High Level Synthesis Leveraging Heterogeneous Compilation. In *Workshop on Rapid Simulation and Performance Evaluation for Design Optimization: Methods and Tools (RAPIDO)*, January 2025.

[17] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless Gradients in Numpy.

[18] Jules Merckx. 2025. Building Bridges: Julia as an MLIR Frontend. https://doi.org/10.48550/arXiv.2503.04771

[19] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17, 3 (December 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[20] William S Moses, Ruizhe Zhao, Lorenzo Chelini, and Oleksandr Zinenko. 2021. Polygeist: Affine C in MLIR. (2021).

[21] William S. Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. https://doi.org/10.5555/3495724.3496770

[22] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, November 2021. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3458817.3476165

[23] William Moses, Valentin Churavy, Sergio Sánchez Ramirez, Paul Berg, and Avik Pal. Reactant. Retrieved from https://enzymead.github.io/Reactant.jl/stable/

[24] Artem Pelenitsyn, Julia Belyakova, Benjamin Chung, Ross Tate, and Jan Vitek. 2021. Type Stability in Julia: Avoiding Performance Pathologies in JIT Compilation (Extended Version). *Proceedings of the ACM on Programming Languages* 5, OOPSLA (October 2021), 1–26. https://doi.org/10.1145/3485527

[25] Norman Ramsey. 2022. Beyond Relooper: Recursive Translation of Unstructured Control Flow to Structured Control Flow (Functional Pearl). *Proceedings of the ACM on Programming Languages* 6, ICFP (August 2022), 1–22. https://doi.org/10.1145/3547621

[26] Benedict Short, Ian McInerney, and John Wickerson. 2025. Hardware.Jl - An MLIR-based Julia HLS Flow (Work in Progress). https://doi.org/10.48550/arXiv.2503.09463

[27] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, April 2022. Seoul, Korea, Republic of, 741–755. https://doi.org/10.1109/HPCA53966.2022.00060

[28] Hanchen Ye, Hyegang Jun, and Deming Chen. 2024. HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis. In *In the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, April 2024. 215–230. https://doi.org/10.1145/3617232.3624850