Instruction level automatic differentiation on a reversible Turing machine

Jin-Guo Liu[1,*] and Taine Zhao[2]

[1]*Institute of Physics, Chinese Academy of Sciences, Beijing 100190, China*
[2]*Department of Computer Science, University of Tsukuba*

This paper considers source to source automatic differentiation (AD) on a reversible Turing machine. We start by reviewing why adjoint mode AD is hard for traditional machine learning frameworks and propose a solution to existing issues by writing a program reversiblly. We developed an reversible eDSL NiLang in Julia that can be used to generate backward rules. We demonstrate its power by differentiating over the reversible implementations of exp function, and linear algebra functions unitary matrix multiplication and QR decomposition. It is also a promising direction towards solving the notorious memory wall problem in machine learning. We also discuss the challenges that we face towards rigorous reversible programming from the instruction and hardware perspective.

I. INTRODUCTION

Computing the gradients of a numeric model $f : \mathbb{R}^m \to \mathbb{R}^n$ plays a crutial role in scientific computing. Consider a computing process

$$
\begin{aligned}
\mathbf{x}^1 &= f_1(\mathbf{x}^0)\\
\mathbf{x}^2 &= f_2(\mathbf{x}^1)\\
&\ldots\\
\mathbf{x}^L &= f_L(\mathbf{x}^{L-1})
\end{aligned}
$$

where $x^0 \in R^m$, $x^L \in R^n$, L is the depth of computing. The Jacobian of this program is a $n \times m$ matrix $J_{ij} \equiv \frac{\partial x_i^L}{\partial x_j^0}$, where x_j^0 and x_i^L are single elements of inputs and outputs. Computing part of the Jacobian automatically is what we called automatic differentiation (AD). It can be classified into three classes, the tangent mode AD, the adjoint mode AD and the mixed mode AD. [1] The tangent mode AD computes the Jacobian matrix elements that related to a single input using the chain rule $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}} \frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$, while a tangent mode AD computes Jacobian matrix elements that related to a single output using $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}} \frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$. Mixed mode AD is a mixture of both. In variational applications where the loss function always outputs a scalar, the adjoint mode AD is perfered,. However, implementing adjoint mode AD is harder than implementing its tangent mode counterpart, because it requires propagating the gradients in the inverse direction of computing the loss. The back propagation of gradients requires intermediate information of a program that includes

1. the computational process,
2. and variables used in computing gradients.

The computational process is often stored in a computational graph, a directed acyclic graph (DAG) that represents the relationship between data and functions. In Pytorch [2] and Flux [3], every variable has a tracker field that stores its parent information, i.e., the input data and function generating this variable. TensorFlow [4] implements a static computational graph as a description of the program before actual computation happens. The required variables are also recorded in this graph. For source to source AD package, Tapenade [1] uses source code as the computational graph and Zygote [5, 6] uses an intermediate representation (IR) of a program, the static single assignment (SSA) form, as the computational graph. To cache intermediate states, they use a global stack.

Several limitations are observed in these AD implementations due to the recording and caching. First of all, most packages require a lot of primitive functions with programmer-defined backward rules. For example, the backward rule of `exp` should be provided although it is composed of basic instructions '+', '-', '*', '/', and conditional jumps. Defining backward rules for these basic instructions in the computational graph scheme suffers from the overhead of memorizing the computational graph and caching intermediate states. Even in Tapenade, the program has to remember the control at each place where the flow merges in the forward sweep. Secondly, the memory consumption is significant, also known as the memory wall problem. [7]. The overhead of naive caching every input of instructions is linear to the computing time. In many deep learning models like recurrent neural network [8] and residual neural networks [9], the depth can reach several thousand, where the memory is often the bottleneck of these programs. Another important source of memory overhead is from the fact that inplace functions are forbidden delibrately in a computational graph based AD scheme in order to protect the cached data. Thirdly, obtaining higher-order gradients are not efficient in most of these packages. For example, in most machine learning packages, people back-propagate the whole program of obtaining first-order gradients to obtain second-order gradients. The repeated use of back-propagation algorithm causes an exponential overhead concerning the order of gradients. A better approach is using Taylor propagation like in JAX [10] and beautiful differentiation [11]. However, Taylor propagation in the adjoint mode AD requires tedious implemention of higher order backward rules for primitives.

We tackle these issues by making a program reversible. In the machine learning field, reversibility has been used in reduce the memory allocations in recurrent neural networks [12] and residual neural networks [13]. These techinics

* cacate0129@iphy.ac.cn

include information buffer [14] and reversible activation functions [15, 16]. Our approach is general purposed. We develop an embedded domain-specific language (eDSL) NiLang in Julia language [17, 18] that implements reversible programming. [19, 20]. This eDSL provides a macro to generate reversible functions, and is completely compatible with Julia ecosystem. One can write reversible control flows, instructions and memory managements in this macro. Combining it with Julia's type system, we implement the AD engine within 100 lines that differentiate any program written in this eDSL, including linear algebra functions.

In history, there have been some prototypes of reversible languages like Janus [21], R (not the popular one) [22], Erlang [23] and object-oriented ROOPL [24]. In the past, the primary motivation of making a program reversible is to support energy efficient reversible computing devices [25] like adiabatic complementary metal–oxide–semiconductor (CMOS) [26], molecular mechanical computing system [27] and superconducting system [28, 29]. These devices either implements reversible logical gates or is able to recover signal energy, where the latter is also called generalized reversible computing. Both schemes do not have a lower bound of energy consumption from information and entropy perspective, which is known as the Landauer's principle [30] After decades of efforts, reversible computing devices are very close to providing productivity now. For exmaple, adiabatic CMOS is more energy efficient than a traiditonal CMOS and can be used in a spacecraft [31], where energy is more valuable than device itself. From the software engineering perspective, reversible programming is a powerful tool to schedule asynchronious events [32] and debug a program bidirectionally [33]. These applications are interesting on them own, but not appealing enough to motivate the reversible software ecosystem. Our work aims to breaks the information barrier between the machine learning community and the reversible programming community, and provides yet another strong motivation to develop reversible programming.

In this paper, we first introduce the language design of NiLang in Sec. II. In Sec. III, we explain the back-propagation algorithm of Jacobians and Hessians in this eDSL. In Sec. IV, we show several examples including Fobonacci number, `exp` function, unitary matrix multiplication and QR decomposition [34]. We show how to generate first order and second order backward rules for these functions. In Sec. V, we discuss several important issues, how time-space tradeoff works, reversible instructions and hardware, and finally, an outlook to some open problems to be solved. In the appendix, we show the grammar of NiLang and a gradient free self-consistent training strategy.

II. LANGUAGE DESIGN

A. Intruductions to reversible language design

In a modern programming language, functions are pushed to a global stack for scheduling. The memory layout of a function consists of input arguments, a function frame with information like the return address and saved memory segments, local variables, and working stack. After the call, the function clears these runtime information, only stores the return value. In the reversible programming style, this kind of design pattern is no longer the best practice. One can not discard input variables and local variables easily after a function call, since discarding information may ruin reversibility. For this reason, reversible functions are very different from irreversible ones from multiple perspectives.

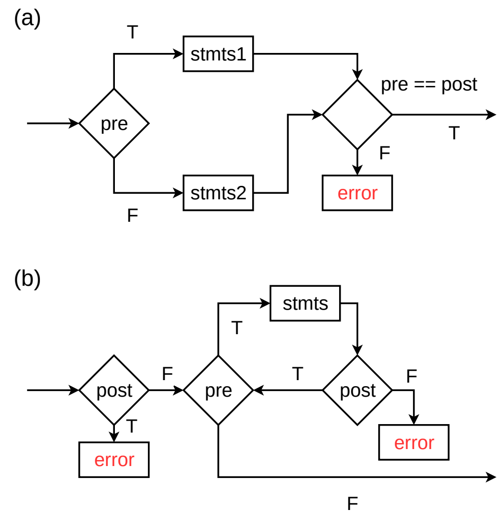

Figure 1. Flow chart for reversible (a) `if` statement and (b) `while` statement. "stmts", "stmts1" and "stmts2" are statements, statements in true branch and statements in false branch respectively. "pre" and "post" are precondition and postconditions respectively.

First of all, the memory management in a reversible language is different. The key difference is when a variable in a reversible program is discarded, its contents should be known. We denote the allocation of a zero emptied memory to variable `x` as `x` $\leftarrow$ `0`, and deallocate a zero emptied variable `x` as `x` $\rightarrow$ `0`. A variable allocated and deallocated in a local scope is called an ancilla, it does not occupy the memory for long period. A variable can also be pushed to a stack and used later with a pop statement. This is similar to a traditional stack operation except it zero-clears the variable after pushing and presupposes the variable being zero-cleared before poping.

Secondly, a reversible control flow is sepcially designed. The reversible `if` statement is shown in Fig. 1 (a), the program enters the branch specified by precondition. After executing that branch, the program checks the consistency of precondition and postcondition to make sure they are the same. In the reverse pass, the program enters the branch specified by the postcondition. For the reversible `while` statement shown in Fig. 1 (b), before executing the condition expressions, the program preassumes the postcondition is false. After each iteration, the program asserts the postcondition to be true. In the reverse pass, we exchange the precondition and postcondition. The reversible `for` statement is similar to irreversible ones except after executing the loop, the program checks the values of these variables to make sure they are not changed. In the reverse pass, we exchange start and stop and inverse the sign of step.

Lastly, the reversible arithmetic and boolean instructions are also different. Every instruction has a unique inverse that can undo the changes. For logical expressions, we have `y ⊻= f(args...)` self reversible. In the following discussion, we assume `y += f(args...)` and `y -= f(args...)` reverse to each other although it is not true for floating point numbers considering rounding error. Here `f` can be `identity`, `*`, `/` and `^` et. al. We will discuss the number system in detail later in Sec. V B

B. NiLang's Reversible IR

In the last subsection, we have reviewed basic building blocks of a typical reversible language. In order to insert the code of obtaining gradients into the reversed program, the reversible language design should have related abstraction power. This motivates us to design a new reversible language NiLang to fit this task. NiLang is an eDSL in Julia. We choose Julia as the host language for multipile purposes. Julia's meta programming and its package for pattern matching MLStyle.jl [35] allow us to define an eDSL conveniently. Meanwhile, the type inference and just in time compiling can remove most overheads introduced in our eDSL, providing a reasonable performance. Most importantly, the multiple dispatch provides the polymorphism that will be used in our autodiff engine.

The main feature of NiLang is contained in a single macro `@i` that compiles a reversible function. The allowed statements in this eDSL are shown in Appendix A. The following is a minimal example of compiling a NiLang function to native julia function.

```
julia> using NiLangCore, MacroTools

julia> macroexpand(Main, :(@i function f(x, y)
           SWAP(x, y)
       end)) |> MacroTools.prettify
quote
    $(Expr(:meta, :doc))
    function $(Expr(:where, :(f(x, y))))
        gaur = SWAP(x, y)
        x = (NiLangCore.wrap_tuple(gaur))[1]
        y = (NiLangCore.wrap_tuple(gaur))[2]
        return (x, y)
    end
    if typeof(f) != typeof(~f)
     function $(Expr(:where, :((  #$ TODO: remove this comment
                mongoose::typeof(~f))(x, y))))
            mandrill = (~SWAP)(x, y)
            x = (NiLangCore.wrap_tuple(mandrill))[1]
            y = (NiLangCore.wrap_tuple(mandrill))[2]
            return (x, y)
        end
    end
    if !(NiLangCore._hasmethod1(
              NiLangCore.isreversible, typeof(f)))
        NiLangCore.isreversible(::typeof(f)) = true
    end
end
```

Macro `@i` generates three functions `f`, `~f` and `NiLangCore.isreversible`. `f` and `~f` are a pair of functions that reverse to each other, where `~f` is an callable of type `Inv{typeof(f)}`. In the body of `f`, `NiLangCore.wrap_tuple` is used to unify output data types, it will wrap any non-tuple variable to a tuple. The outputs of `SWAP` are assigned back to its input variables, in other words, a function modifies inputs inplace. At the end this function, this macro attaches a return statement that returns all input variables. `NiLangCore.isreversible` is a function to mark the reversibility trait of `f`.

To understand the design of reversibility, we first introduce a reversible IR that plays a central role in NiLang. In this IR, a statement can be an instruction, a function call, a control flow, a memory allocation/deallocation, or the inverse statement "~". Any statement is this IR has a unique inverse as shown in Table I.

"←" and "→" are symbols for memory allocation and deallocation, one can input them by typing "\leftarrow" and "\rightarrow" respectively followed by a Tab key in a Julia editor or REPL. "`begin <stmts> end`" is the block statement in Julia, it represents a code block. It can be inverted by reversing the order of `<stmts>` as well as each element in it. The conditional expression in `if` or `while` statements is a tuple of precondition and postcondition. Finally, the special macro `@safe` allows users to use external statements that do not break reversibility. For example, one can use `@safe @show <var>` for debugging.

statement	inverse
<f>(<args>...)	(~<f>)(<args>...)
<y> += <f>(<args>...)	<y> -= <f>(<args>...)
<y> .+= <f>.(<args>...)	<y> .-= <f>.(<args>...)
<y> ⊻= <f>(<args>...)	<y> ⊻= <f>(<args>...)
<y> .⊻= <f>(<args>...)	<y> .⊻= <f>(<args>...)
<a> ← <expr>	<a> → <expr>
(<T1> => <T2>)(<x>)	(<T2> => <T1>)(<x>)
begin <stmts> end	begin ~(<stmts>) end

if (<pre>, <post>)
<stmts1>
else
<stmts2>
end
	if (<post>, <pre>)
	~(<stmts1>)
	else
	~(<stmts2>)
	end

while (<pre>, <post>)
<stmts>
end
	while (<post>, <pre>)
	~(<stmts>)
	end

for <i>=<m>:<s>:<n> <stmts> end	for <i>=<m>:-<s>:<n> ~(<stmts>) end
@safe <expr>	@safe <expr>

Table I. A collection of reversible statements. “.” is the symbol for broadcasting magic in Julia, “~” is the symbol for reversing a statement or a function. <...> represents a non-keyword, where <pre> stands for precondition, <post> stands for postcondition, <args>... stands for the argument list of a function, <stmts> stands for statement, <exprs> stands for expression, <T1> and <T2> stand for types and the reset are variables.

,

C. Compiling

The compilation of a reversible function contains three stages.

The first stage preprocess human inputs to a reversible IR. The preprocessor expands the symbol “~” in the postcondition field of `if` statement by copying the precondition, adds missing ancilla “←” statements to ensure “←” and “→” appear in pairs inside a function, a `while` statement or a `for` statement, and expands the uncomputing macro `~@routine`. Since the "compmute-copy-uncompute" design pattern is extensively used in reversible programming for uncomputing ancillas. One can use `@routine <stmt>` statement to record a statement, and `~@routine` to insert `~<stmt>` for uncomputing. The following example preprocesses an `if` statement to the reversible IR.

```
julia> using NiLangCore, MacroTools

julia> MacroTools.prettify(
       @code_preprocess if (x > 3, ~)
           @routine z += x * y
           ~@routine
       end)
:(if (x > 3, x > 3)
      z += x * y
      z -= x * y
  else
  end)
```

In this example, since the precondition “`x > 3`” is not change after execution of the specific branch, we omit the postcondition by putting a “~” in this field. “`@routine`” records a statement, the statement can also be a “`begin <stmts> end`” block as a senquence of statements.

The second stage generates the reversed code according to table Table I. For example,

```
julia> MacroTools.prettify(
       @code_reverse if (pre, post)
           z += x * y
       else
           z += x / y
       end)
:(if (post, pre)
      z -= x * y
  else
      z -= x / y
  end)
```

The third stage is translating this IR and its inverse to native Julia code. It explains all functions as inplace and insert codes about reversibility check. At the end of a function definition, it attaches a return statement that returns all input arguments. After this, the function is ready to execute on the host language. The following example shows how an `if` statement is transformed in this stage.

```
julia> MacroTools.prettify(
       @code_interpret if (pre, post)
           z += x * y
       else
           z += x / y
       end)
quote
    bat = pre
    if bat
        @assignback (PlusEq(*))(z, x, y)
    else
        @assignback (PlusEq(/))(z, x, y)
    end
    @invcheck post bat
end
```

The compiler translates the instruction according to Table II and adds `@assignback` before each instruction and function call statement. The macro `@assignback` assigns the output of a function back to the arguments of that function. `@invcheck post bat` checks the consistency between preconditions and postconditions to ensure reversibility. This statement will throw an `InvertibilityError` error if target variables `bat` and `post` are not "equal" to each other up to a certian torlerance.

D. Types and Dataviews

So far, the language design is not too different from a traditional reversible language. To implement the adjoint mode AD, we introduce types and dataviews. The type that used in the reversible context is just a normal Julia type with an extra requirement of having reversible constructors. The inverse of a constructor is called a "destructor", which unpacks data and deallocates derived fields. Data packing is implemented by reinterpreting the `new` function in Julia. For example,

```
x ← new{TX, TG}(x, g)
```

Here, the "←" statement followed by a `new` function is treated specially that it deallocates `g`. This makes sense because the output of `new` keeps all information in input argument list. Its inverse is

```
x → new{TX, TG}(x, g)
```

It unpacks `x` and allocates a new ancilla `g`. The following example shows how to define as reversible type `GVar`.

```
julia> using NiLangCore

julia> @i struct GVar{T,GT} <: IWrapper{T}
           x::T
           g::GT
           function GVar{T,GT}(x::T, g::GT) where
                                          {T,GT}
               new{T,GT}(x, g)
           end
           function GVar(x::T, g::GT) where {T,GT}
               new{T,GT}(x, g)
           end
           @i function GVar(x::T) where T
               g ← zero(x)
               x ← new{T,T}(x, g)
           end
           @i function GVar(x::AbstractArray)
               GVar.(x)
           end
       end

julia> GVar(0.5)
GVar{Float64,Float64}(0.5, 0.0)

julia> (~GVar)(GVar(0.5))
0.5

julia> (~GVar)(GVar([0.5, 0.6]))
2-element Array{Float64,1}:
 0.5
 0.6
```

This piece of code is copied from the autodiff submodule of NiLang. `GVar` is a type used to store gradient information of a variable. Here, we put `@i` macro before both `struct` and `function` statements. The ones before functions mark reversible functions, while the one before `struct` keyword is used to handle the function scope issue. It moves `~GVar` functions outside of this type definition.

The reversible cast between two types can be defined conveniently with the macro `@icast`.

```
julia> @pure_wrapper A

julia> @icast A(x) => GVar(x, g) begin
           g ← zero(x)
           g += identity(1)
       end

julia> x = A(0.5)
A(0.5)

julia> @instr (A=>GVar)(x)
GVar{Float64,Float64}(0.5, 1.0)

julia> @instr (GVar=>A)(x)
A(0.5)
```

Here, we first define a simple reversible wrapper `A` using macro `@pure_wrapper`, and then the cast rule between `A` type and GVar type. The body of cast is a reversible mapping that transforms `x` to `(x, g)`. The compiler appends a default constructor `DVar(xx, gg)` at the end of program to instantiate a new object as the return value. Its inverse that coverts an object of type `GVar` to type `A` is automatically generated by reversing the above statements.

The fields of an object can be accessed and manipulated by dataviews. A dataview can be an object, a field of a dataview, an array element of a dataview, or a bijective mapping of a dataview. Let us first consider the following example.

```
julia> arr = [GVar(3.0), GVar(1.0)]
2-element Array{GVar{Float64,Float64},1}:
 GVar{Float64,Float64}(3.0, 0.0)
 GVar{Float64,Float64}(1.0, 0.0)

julia> x, y = 1.0, 2.0
(1.0, 2.0)

julia> @instr -arr[2].g += x * y
2.0

julia> arr
2-element Array{GVar{Float64,Float64},1}:
 GVar{Float64,Float64}(3.0, 0.0)
 GVar{Float64,Float64}(1.0, -2.0)
```

Here, both `-arr[2].g`, `x` and `y` are dataviews. In Julia language, the statement `-grad(arr[2]) += x * y` should throw a syntax error because the function call "`-`" can not be assigned, and `GVar` is an immutable type. In our eDSL, we wish it works because a memory cell is assumed to be modifiable in our eDSL. The secret of how it works lies in the macro `@assignback`, it translates the above statement to

```
1 res = (PlusEq(*))(-arr[2].g, x, y)
2 arr[2] = chfield(arr[2], Val(:g),
3     chfield(arr[2].g, -, res[1]))
4 x = res[2]
5 y = res[3]
```

The first line `PlusEq(*)(-arr[3].g, x, y)` computes the output, which is a tuple of length 3. At lines 2-3, `chfield(x, Val{:g}, val)` modifies the `g` field of `x` and `chfield(x, -, res[1])` returns `-res[1]`. Here, modifying a field requires the default constructor of a type not overwritten. The assignments in lines 4 and 5 are straightforward.

III. AUTOMATIC DIFFERENTIATION

A. First order gradient

Consider a computation $\mathbf{x}^{i-1} = f_i^{-1}(\mathbf{x}^i)$ in a reversed program, the adjoint mode AD propagates the Jacobians in the reversed direction like

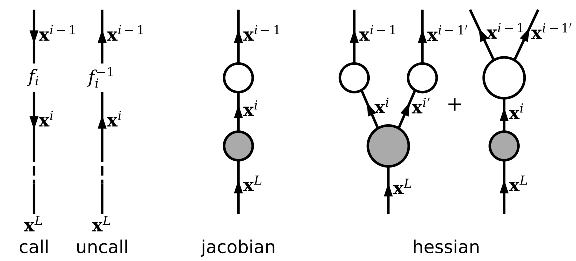

Figure 2. Computional processes in the tensor network diagram, a big circle with three legs represents a Hessian, a small circle with two legs represents a Jacobian. Dangling edges and connected edges stands for unpaired and paired labels respectively in the Einstein's notation. From left to right, the diagrams represent computing, uncomputing, back propagating Jacobians and back propagating Hessians.

$$
\begin{aligned}
J^{\mathbf{x}^L}_{\mathbf{x}^{L'}} &= \delta_{\mathbf{x}^L, \mathbf{x}^{L'}}, \\
J^{\mathbf{x}^L}_{\mathbf{x}^{i-1}} &= J^{\mathbf{x}^L}_{\mathbf{x}^{i}} J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}},
\end{aligned}
\tag{1}
$$

where $\mathbf{x}^L$ represents the outputs of the program, $J^{\mathbf{x}^L}_{\mathbf{x}^i} \equiv \frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^i}$ is the Jacobian to be propagated, and $J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}}$ is the local Jacobian matrix. The Einstein's notation is used here so that the duplicated index $\mathbf{x}^i$ is summed over. Eq. (1) can be rewritten in the diagram of tensor networks [36] as shown in Fig. 2.

The algorithm to compute the adjoint mode AD can be summarized as follows.

Algorithm 1: Reversible programming AD

Result: grad.($\mathbf{x}_g$)
let `iloss` be the index of loss variable in $\mathbf{x}$
$\mathbf{y} = f(\mathbf{x})$
$\mathbf{y}_g$ = GVar.($\mathbf{y}$)
grad($\mathbf{y}_g$[iloss]) += 1.0
$\mathbf{x}_g = f^{-1}(\mathbf{y}_g)$

The program first computes the forward pass, and then wrap each output variable with `GVar`. The constructor `GVar` attaches a zero gradient field to a variable. If an input varaible is an array, `GVar` will be broadcasted to each array element automatically. The line `grad(`$\mathbf{y}_g$`[iloss]) += 1.0` adds one to the gradient field of loss to initialize a single row of Jacobian as in the first line of Eq. (1). Finally, execute the inverse program f^{-1}, the gradients are stored in the `grad` dataview of output variables. The computation of gradients are implemented with multiple dispatch, that is, when an instruction has a `GVar` type in its argument list, it calls a different routine. The same trick is used in the dual number implementation of tangent mode AD [37]. In the following, we examplify the design by binding the adjoint rules to instructions $\oplus(*)$ and $\ominus(*)$

```
@i function ⊖(*)(out!::GVar, x::GVar, y::GVar)
    value(out!) -= value(x) * value(y)
    grad(x) += grad(out!) * value(y)
    grad(y) += value(x) * grad(out!)
end
```

Here, we adopt a convension that only variables ended with ! will be changed after the function call. Although the backward rule is defined on $\ominus(*)$, the compiler generates the backward rules on $\oplus(*)$ too. This reflects the fact that taking inverse and computing gradients commute to each other [38]. Hence for a general reversible function f, one can bind backward rule on either f or its inverse f^{-1}. We can check the correctness of our definition like follows.

```
julia> using NiLang, NiLang.AD

julia> a, b, y = GVar(0.5), GVar(0.6), GVar(0.9)
(GVar(0.5, 0.0), GVar(0.6, 0.0), GVar(0.9, 0.0))

julia> @instr grad(y) += identity(1.0)

julia> @instr y += a * b
GVar(0.6, -0.5)

julia> a, b, y
(GVar(0.5, -0.6), GVar(0.6, -0.5), GVar(1.2, 1.0))

julia> @instr y -= a * b
GVar(0.6, 0.0)

julia> a, b, y
(GVar(0.5, 0.0), GVar(0.6, 0.0), GVar(0.899999, 1.0))
```

Here, $J(\ominus(*)) = J(\oplus(*))^{-1}$, hence consecutively applying them restores gradient fields of variables. The implementation of Algorithm 1 is so short that we present the function definition as follows.

```
@i function (g::Grad)(args...; kwargs...)
    @safe @assert count(x -> x isa Loss, args) == 1
    iloss ← 0
    @routine for i=1:length(args)
        if (tget(args,i) isa Loss, iloss==i)
            iloss += identity(i)
            (~Loss)(tget(args,i))
        end
    end

    g.f(args...; kwargs...)
    GVar.(args)
    grad(tget(args,iloss)) += identity(1.0)
    (~g.f)(args...; kwargs...)

    ~@routine
end
```

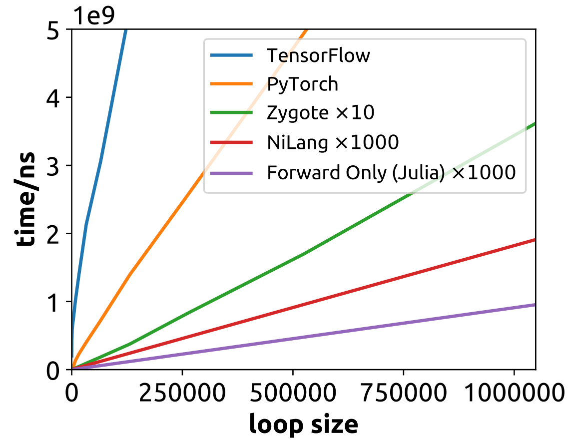

Figure 3. The time to obtain gradient as function of loop size. $\times n$ in lengend represents a rescaling of time.

The program first checks variables contain exactly one `Loss`, where `Loss` is a reversible wrapper used to mark the loss variable. Then we locates the loss variable as `iloss` and use `~Loss` unwraps the loss variable. After computing the forward pass and backward pass, `~@routine` uncomputes the ancilla `iloss` and returns the location information to the loss variable. Here, `tget(args, i)` returns the i-th element of a tuple. We forbid tuple indexing deliberately in order to avoid possible ambiguity in supporting array indexing.

The overhead of using `GVar` type is negligible thanks to Julia's multiple-dispatch and type inference. Let us consider a simple example that accumulates 1.0 to a target variable x for n times.

[**JG: Grammarly here!**]

```
julia> using NiLang, NiLang.AD, BenchmarkTools

julia> @i function prog(x, one, n::Int)
           for i=1:n
               x += identity(one)
           end
       end

julia> @benchmark prog'(Loss(0.0), 1.0, 10000)
BenchmarkTools.Trial:
  memory estimate:  1.05 KiB
  allocs estimate:  39
  --------------
  minimum time:     35.838 μs (0.00% GC)
  median time:      36.055 μs (0.00% GC)
  mean time:        36.483 μs (0.00% GC)
  maximum time:     185.973 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

We implement the same function with TensorFlow, PyTorch and Zygote for comparison. The code could be found in our paper's github repository [39]. Benchmark results on CPU

Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz are shown in Fig. 3. One can see that the NiLang implementation is unreasonably fast, it is approximately two times the forward pass written in native Julia code. Reversible programming is not always as fast as its irreversible counterparts. In practical applications, a reversible program may have memory or computation overhead. We will discuss the details of time and space trade off in Sec. V A.

B. Second-order gradient

Second-order gradients can be obtained in two different approaches.

1. *Back propagating first-order gradients*

Back propagating the first-order gradients is the most widely used approach to obtain the second-order gradients. Suppose the function space is closed under gradient operation, one can obtain higher-order gradients by recursively differentiating lower order gradient functions without defining new backward rules.

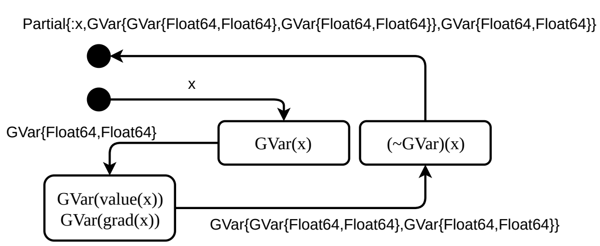

Figure 4. Data flow in obtaining the second-order gradient with the recursive differentiation approach. Annotations on lines are data types used in the computation.

Fig. 4 shows the data flow in the four passes of computing Hessian. The first two passes obtains the gradients. Before entering the third pass, the program wraps each field in `GVar` with another layer of `GVar`. Then we pick a variable x_i and add 1 to the gradient field of its gradient `grad(grad(`x_i`))` in order to compute the i-th row of Hessian. Before entering the final pass, the `~GVar` is called. We can not unwrap `GVar` directly because although the values of gradients have been uncomputed to zero, the gradient fields of gradients may be nonzero. Instead, we use `Partial{:x}(obj)` to take field `x` of an object without erasing memory. By repeating the above procedure for different x_i, one can obtains the full Hessian matrix.

2. *Hessian propagation*

A probably more efficient approach is back-propagating Hessians directly [40] using the relation

$$
\begin{aligned}
H^{\mathbf{x}^L}_{\mathbf{x}^{L'},\mathbf{x}^{L''}} &= \mathbf{0},\\
H^{\mathbf{x}^L}_{\mathbf{x}^{i-1},\mathbf{x}^{i-1'}} &= J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}} H^{\mathbf{x}^L}_{\mathbf{x}^i,\mathbf{x}^{i'}} J^{\mathbf{x}^{i'}}_{\mathbf{x}^{i-1'}} + J^{\mathbf{x}^L}_{\mathbf{x}^i} H^{\mathbf{x}^i}_{\mathbf{x}^{i-1},\mathbf{x}^{i-1'}}.
\end{aligned}
\tag{2}
$$

Here, the Hessian tensor $H^{\mathbf{x}^L}_{\mathbf{x}^{i-1},\mathbf{x}^{i-1'}}$ is rank three, where the top index is often taken as a scalar and omitted. In tensor network diagram, the above equation can be represented as the right panel of Fig. 2. Hessian propagation is a special case of Taylor propagation. With respect to the order of gradients, Taylor propagation is exponentially more efficient in obtaining higher-order gradients than differentiating lower order gradients recursively. Comparing with operator overloading, source to source automatic differentiation has the advantage of having very limited primitives, exhausted implementation of Hessian propagation is possible. An example to obtain Hessians is provided in Sec. IV B.

C. Gradient on ancilla problem

In this section, we introduced an easily overlooked problem in our reversible AD framework. An ancilla sometimes can carry a nonzero gradient when it is going to be deallocated. As a result, even if an ancilla can be uncomputed rigorously in the original program, its `GVar` wrapped version is not nessesarily safely deallocated. In NiLang, we simply "drop" the gradient field of ancillas instead of raising an error. In the following, we justify our approach by proving the following theorem

***Theorem* 1.** *Deallocating an ancilla with emptied value field and nonzero gradient field does not harm the reversibility of a function.*

Proof. Consider a reversible function $\mathbf{x}^i, b = f_i(\mathbf{x}^{i-1}, a)$, where a and b are the input and output values of an ancilla. Since the ancilla is emptied for any input $\mathbf{x}^{i-1}$, we have

$$\frac{\partial b}{\partial \mathbf{x}^{i-1}} = \mathbf{0}. \tag{3}$$

Since the gradient fields are derived from the value fields of variables, discarding gradients should not have any effect to the `value` fields. The rest is to show `grad(b)` $\equiv \frac{\partial \mathbf{x}^L}{\partial b}$ does appear in the backward rule of this function. It can be seen from the back-propagation rule

$$\frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^{i-1}} = \frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^i}\frac{\partial \mathbf{x}^i}{\partial \mathbf{x}^{i-1}} + \frac{\partial \mathbf{x}^L}{\partial b}\frac{\partial b}{\partial \mathbf{x}^{i-1}}, \tag{4}$$

where the second term with $\frac{\partial \mathbf{x}^L}{\partial b}$ vanishes naturally. $\square$

IV. EXAMPLES

A. Computing Fibonacci Numbers

The following is an example that everyone likes, computing Fibonacci number recursively.

```
using NiLang

@i function rfib(out!, n::T) where T
    n1 ← zero(T)
    n2 ← zero(T)
    @routine begin
        n1 += identity(n)
        n1 -= identity(1)
        n2 += identity(n)
        n2 -= identity(2)
    end
    if (value(n) <= 2, ~)
        out! += identity(1)
    else
        rfib(out!, n1)
        rfib(out!, n2)
    end
    ~@routine
end
```

The time complexity of this recursive algorithm is exponential to input `n`. It is also possible to write a reversible linear time for loop algorithm. A slightly non-trivial task is computing the first Fibonacci number that greater or equal to a certain number z, where a `while` statement is required.

```
@i function rfibn(n!, z)
    @safe @assert n! == 0
    out ← 0
    rfib(out, n!)
    while (out < z, n! != 0)
        ~rfib(out, n!)
        n! += identity(1)
        rfib(out, n!)
    end
    ~rfib(out, n!)
end
```

In this example, the postcondition `n!=0` in the `while` statement is false before entering the loop, and becomes true in later iterations. In the reverse program, the `while` statement stops at `n==0`. If executed correctly, a user will see the following result.

```
julia> rfib(0, 10)
(55, 10)

julia> rfibn(0, 100)
(12, 100)

julia> (~rfibn)(rfibn(0, 100)...)
(0, 100)
```

This examplifies how an addition postcondition provided by user can help reversing a control flow without caching controls.

B. exp function

An exp function can be computed using Taylor expansion

$$y+=\sum_n \frac{x^n}{\mathrm{factorial}(n)} \tag{5}$$

One can compute the accumulated item $s_n \equiv \frac{x^n}{\mathrm{factorial}(n)}$ iteratively as $s_n = \frac{x s_{n-1}}{n}$. Intuitively, this problem mimics the famous pebble game [19] considering the fact that product and division are considered as irreversible in NiLang, one can not deallocate s_{n-1} after computing s_n. However, here the case is different because $*=$ and $/=$ are arithmetically reversible to each other at nonzero points. We can "uncompute" previous state s_{n-1} by $s_{n-1} = \frac{n s_n}{x}$ approximately, and use the dirty ancilla in next iteration. The implementation is

```julia
using NiLang, NiLang.AD

@i function iexp(y!, x::T; atol::Float64=1e-14)
        where T
    anc1 ← zero(T)
    anc2 ← zero(T)
    anc3 ← zero(T)
    iplus ← 0
    expout ← zero(T)

    y! += identity(1.0)
    @routine begin
        anc1 += identity(1.0)
        while (value(anc1) > atol, iplus != 0)
            iplus += identity(1)
            anc2 += anc1 * x
            anc3 += anc2 / iplus
            expout += identity(anc3)
            # arithmetic uncompute
            anc1 -= anc2 / x
            anc2 -= anc3 * iplus
            SWAP(anc1, anc3)
        end
    end

    y! += identity(expout)

    ~@routine
end
```

Here, the definition of SWAP instruction can be found in Appendix B. The two lines bellow the comment "`# arithmetic uncompute`" uncompute variables `anc1` and `anc2` approximately. The rounding errors are transfered to the output, while the reversibility is not affected since the inverse call at the last line of function uncomputes all ancillas rigorously. This "arithmetic uncomputing" trick can be used extensively in many other application. The reason why this trick works here is because from the mathematic perspective the state in nth step $\{\frac{x^n}{\text{factorial}(n)}, x\}$ contains the same amount of information as the state in the $n-1$th step $\{\frac{x^{n-1}}{\text{factorial}(n-1)}, x\}$, it is highly possible to find a route to compute the previous state from the current state. This fact is not even related to the number system that we adopt.

To obtain gradients, one can wrap the variable `y!` with `Loss` type and feed it into `iexp'`

```julia
julia> y!, x = 0.0, 1.6
(0.0, 1.6)

julia> @instr iexp'(Loss(y!), x)

julia> grad(x)
4.9530324244260555
```

Here, `iexp'` is a callable instance of type `Grad{typeof(iexp)}`. Its implementation is shown in Sec. III A. This function itself is reversible and differentiable, one can back-propagate this function to obtain Hessians as introduced in Sec. III B 1. In NiLang, it is implemented as `simple_hessian`.

```julia
julia> y!, x = 0.0, 1.6
(0.0, 1.6)

julia> simple_hessian(iexp, (Loss(y!), x))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  4.95303
```

To obtain Hessians, we can also use the Hessian propagation approach as introduced in Sec. III B 2.

```julia
julia> y!, x = 0.0, 1.6
(0.0, 1.6)

julia> @instr iexp''(Loss(y!), x)

julia> collect_hessian()
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  4.95303
```

`iexp''` computes the second-order gradients. It wraps variables with type `BeijingRing` [41] in the backward pass. `BeijingRing` records Jacobians and Hessians for a variable, where Hessians are stored in a global storage. Whenever an n-th variable or ancilla is created, we push a ring of size $2n-1$ to a global tape. Whenever an ancilla is deallocated, we pop a ring from the top. The n-th ring stores Hessian elements $H_{i\leq n,n}$ and $H_{n,i<n}$.The final result can be collected by calling `collect_hessian()`, which will read out the Hessian matrix that stored in the global storage.

C. Unitary Matrices

A unitary matrices features uniform eigenvalues and reversibility. It is widely used as an approach to ease the gradient exploding and vanishing problem [42–44] and the memory wall problem [45]. One of the simplest way to parametrize a unitary matrix is representing a unitary matrix as a product of two-level unitary operations [44]. A real unitary matrix of size N can be parametrized compactly by $N(N-1)/2$ rotation operations [46]

$$\text{ROT}(a!, b!, \theta) = \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}\begin{bmatrix}a!\\ b!\end{bmatrix}, \qquad (6)$$

where θ is the rotation angle, `a!` and `b!` are target registers.

```julia
using NiLang, NiLang.AD

@i function umm!(x!, θ)
    @safe @assert length(θ) ==
            length(x!)*(length(x!)-1)/2
    k ← 0
    for j=1:length(x!)
        for i=length(x!)-1:-1:j
            k += identity(1)
            ROT(x![i], x![i+1], θ[k])
        end
    end

    k → length(θ)
end
```

Here, the ancilla `k` is deallocated manually by specifying its value, because we know the loop size is $N(N-1)/2$. We define the test functions in order to check gradients.

```julia
julia> @i function isum(out!, x::AbstractArray)
           for i=1:length(x)
               out! += identity(x[i])
           end
       end

julia> @i function test!(out!, x!::Vector, θ::Vector)
           umm!(x!, θ)
           isum(out!, x!)
       end

julia> out, x, θ = Loss(0.0), randn(4), randn(6);

julia> @instr test!'(out, x, θ)

julia> x
4-element Array{GVar{Float64,Float64},1}:
 GVar(1.220182125326287, 0.14540743042341095)
 GVar(2.1288634811475937, -1.3749962375499805)
 GVar(1.2696579252569677, 1.42868739498625)
 GVar(0.1083891125379283, 0.2170123344615735)

julia> @instr (~test!')(out, x, θ)

julia> x
4-element Array{Float64,1}:
 1.220182125326287
 2.1288634811475933
 1.2696579252569677
 0.10838911253792821
```

In the above testing code, `test'` attaches a gradient field to each element of `x`. `~test'` is the inverse program that erase the gradient fields. Notablly, this reversible implementation costs zero memory allocation although it changes the target variables inplace.

D. QR decomposition

Let's consider a naive implementation of QR decomposition from scratch. We admit this implementation is just a proof of principle which does not even consider reorthogonalization.

```julia
using NiLang, NiLang.AD

@i function qr(Q, R, A::AbstractMatrix{T}) where T
    anc_norm ← zero(T)
    anc_dot ← zeros(T, size(A,2))
    ri ← zeros(T, size(A,1))
    for col = 1:size(A, 1)
        ri .+= identity.(A[:,col])
        for precol = 1:col-1
            dot(anc_dot[precol], Q[:,precol], ri)
            R[precol,col] +=
                identity(anc_dot[precol])
            for row = 1:size(Q,1)
                ri[row] -=
                    anc_dot[precol] * Q[row, precol]
            end
        end
        norm2(anc_norm, ri)

        R[col, col] += anc_norm^0.5
        for row = 1:size(Q,1)
            Q[row,col] += ri[row] / R[col, col]
        end

        ~begin
            ri .+= identity.(A[:,col])
            for precol = 1:col-1
                dot(anc_dot[precol], Q[:,precol], ri)
                for row = 1:size(Q,1)
                    ri[row] -= anc_dot[precol] *
                        Q[row, precol]
                end
            end
            norm2(anc_norm, ri)
        end
    end
end
```

Here, in order to avoid frequent uncomputing, we allocate ancillas `ri` and `anc_dot` as vectors. The expression in ~ is used to uncompute `ri`, `anc_dot` and `anc_norm`. `dot` and `norm2` are reversible functions to compute dot product and vector norm. They are implemented as follows.

```
@i function dot(out!, v1::AbstractVector{T}, v2) where T
    for i = 1:length(v1)
        out! += v1[i]'*v2[i]
    end
end

@i function norm2(out!, vec::AbstractVector{T}) where T
    anc1 ← zero(T)
    for i = 1:length(vec)
        anc1 += identity(vec[i]')
        out! += anc1*vec[i]
        anc1 -= identity(vec[i]')
    end
end
```

In `norm2`, we copied `vec[i]'` to `anc1` to avoid the same variable appear twice in the argument list of $\oplus(*)$, where the prime represents the adjoint dataview. One can easily check the correctness of the gradient function

```
julia> A, q, r = randn(4,4), zero(A), zero(A);

julia> @i function test1(out, q, r, A)
           qr(q, r, A)
           isum(out, q)
       end

julia> check_grad(test1, (Loss(0.0), q, r, A))
true
```

Here, the loss function `test1` is defined as the sum of the output unitary matrix `q`. The `check_grad` function is a gradient checker function defined in module NiLang.AD.

V. DISCUSSION AND OUTLOOK

In this paper, we show a program on an reversible Turing machine can be differentiated to any order reliably and efficiently without sophisticated designs to memorize computational graph and intermediate states. We introduce a reversible Julia eDSL NiLang that implements a reversible AD. In a reversible programming language, we proposed to use "arithmetic uncomputing" trick to avoid the overhead of a reversible program in many practical cases.

In the following, we discussed some practical issues about reversible programming, and several future directions to go.

A. Time Space Tradeoff

In history, there has been many other interesting designs of reversible languages. However, current popular programming languages are all irreversible. In the simplest g-segment trade off scheme [47, 48], a RTM model has either a space overhead that proportional to computing time T or a computational overhead that sometimes can be exponential comparing with a irreversible counter part. The tradeoff between space and time is a crutial issue in the theory of RTM. In the following, we try to convince the readers that the overhead of reversible computing is not as terrible as people thought.

The overhead of reversing a program is bounded the checkpointing [49] strategy used in a traditional machine learning package that memorizes every inputs of primitives because similar strategy can also be used in reversible programming. [19] Reversible programming simply provides more alternatives to reduce the overhead. For example, the overhead in many iterative algorithms can often be removed with "arithematic uncomputing" trick without sacrificing reversibility as shown in the `iexp` example in Sec. IV B.

Clever compiling can also be used to remove most overheads. Often, when we define a new reversible function, we allocate some ancillas at the beginning of the function and deallocate them through uncomputing at the end. The overhead comes from the uncomputing, in the worst case, the time used for uncomputing can be the same as the forward pass. In a hierarchical design, uncomputing can appear in every layer of abstraction. To quantify the overhead of uncomputing, we introducing the concept

***Definition* 1** (program granularity)**.** The log ratio between the execution time of a reversible program and its irreversible counter part

The computing time increases exponentially as the granularity increases. A cleverer compilation of a program can reduce the granularity by merging the uncomputing statements to avoid repeated efforts.

At last, making reversible programming an eDSL rather than an independent language allows flexible choices between reversibility and computational overhead. For example, in order to deallocate the gradient memory in a reversible language one has to uncompute the whole process of obtaining this gradient. In our eDSL, we can just deallocate the memory irreversibly, i.e. trade energy with time.

B. Instructions and Hardwares

So far, our eDSL is compiled to Julia. In the future, it can be compiled to reversible instructions [50] and executed on a reversible device. However, arithmetic instructions should be redesigned to support better reversible programs. The major obstacle to exact reversibility programming is current floating point adders and multipliers used in our computing devices are not exactly reversible. There are proposals of reversible floating point adders and multipliers [51–54], however these designs with allocate garbage bits in each operaton. Alternatives include fixed point numbers [55] and logarithmic numbers [56, 57], where logarithmic number system is reversible under `*=` and `/=`. With these infrastructures, a reversible program can be executed without suffering from the rounding error.

Reversible instructions can be executed on an energy efficient reversible hardware. In the introduction, we mensioned several reversible hardwares. A reversible hardware can be those supporting reversible gates such as the Toffoli gate and the Fredkin gate, or like an adiabatic CMOS with the ability to recover signal energy. The latter is known as the generalized reversible computing. [58, 59] In the near future, there might be energy efficient artificial intellegence (AI) chips as coprocessors that our eDSL can compile to. Since reversible computing is mainly driven by quantum computing in recent years. In the following, we comment briefly on quntum devices.

1. Quantum Computers

Building a universal quantum computer [60] is difficult. The difficulty lies in the fact that it is extremely hard to protect a quantum state. Unlike a classical state, an quantum state is can not be cloned, meanwhile, it losses information by interating with the environment, or decoherence. Classical reversible computing does not enjoy the quantum advantage, nor the quantum disadvantages of non-cloning and decoherence. It is technically more smooth to have a reversible computing device to bridge the gap between classical devices and universal quantum computing devices. By introducing entanglement little by little, we can accelerate some basic components in reversible computing. For example, quantum Fourier transformation provides an interesting alternative to the reversible adders and multipliers by introducing the CPHASE quantum gate. [61] The development of reversible compiling theory can be benefit quantum compiling directly.

C. Outlook

The reversible eDSL NiLang can be used to solve many existing scientific computing problems. First of all, it can be used to generate AD rules for existing machine learning packages like Zygote. For example, one can use NiLang to generate backward rules for singular value decoposition and eigenvalue decomposition functions that extensively used in scientific computing [62, 63]. Although their backward rules [64–66] have been drived in recent years, these backward rules can not handle degenerate eigenvalues properly. Hopefully, the automatically generated backward rules do not have such problems.

Secondly, we can use it to overcome the memory wall problem in some applications. NiLang provides a systematic time-space trade off scheme through uncomputing. A successful related example is the memory efficient domain-specific AD engine in quantum simulator Yao [45]. This domain-specific AD engine is written in a reversible style and solved the memory bottleneck in variational quantum simulations. It also gives hitherto the best performance in differentiating quantum circuit parameters. Similarily, we can write memory efficient normalizing flow [67] with NiLang. Normalizing flow is a successful class of generative model in both computer vision [68] and quantum physics [69, 70], where its building block bijector is reversible. We can use similar idea to differentiate reversible integrators [71, 72]. With reversible integrators, it should be possible to rewrite the control system in robotics [73] in a reversible style, where scalars are first class citizen rather than tensors. Writing a control program reversibly should boost the training performance a lot.

Thirdly, reversibility is a resource for training. For those who are interested in non-gradient based training. In Appendix C, we provide a self-consistency training strategy for reversible programs.

Latstly, the reversible IR is a good starting point to study quantum compiling. Most quantum programming language preassumes a classical coprocessor and use classical control flows [74] in universal quantum computing. However, we believe reversible control flows are also very important to a universal quantum computer.

To solve the above problems better, NiLang can be improved from multiple perspectives. Like we need a more efficient fixed point or log number system to avoid rounding errors. Currently we simulate reversible arithematics with the julia fixed point numebr package. [55] Then we should optimize the compiling to decreases granularity of a program and reduces uncomputing overheads. There are also some known issues to be solved like the type inference problem, we have listed some of them on Github. These improvements needs participation of people from multiple fields.

VI. ACKNOWLEDGMENTS

Jin-Guo Liu thank Lei Wang for motivating the project with possible applications reversible integrator, normalizing flow and neural ODE. Xiu-Zhe Luo for discussion on the implementation details of source to source automatic differetiation, Shuo-Hui Li for helpful discussion on differential geometry. Damian Steiger for telling me the `comefrom` joke. Tong Liu and An-Qi Chen for helpful discussion on quantum adders and multipliers. Chris Rackauckas for helpful discussion on reversible integrators. Mike Innes for reviewing the comments about Zygote. Simon Byrne and Chen Zhao for helpful discussion on floating point and logarithmic numbers. The authors are supported by the National Natural Science Foundation of China under the Grant No. 11774398, the Strategic Priority Research Program of Chinese Academy of Sciences Grant No. XDB28000000.

[1] L. Hascoet and V. Pascual, ACM Transactions on Mathematical Software (TOMS) **39**, 20 (2013).

[2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, in *NIPS*

Autodiff Workshop (2017).
[3] M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah, "Fashionable modelling with flux," (2018), arXiv:1811.01457 [cs.PL].
[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," (2015), software available from tensorflow.org.
[5] M. Innes, "Don't unroll adjoint: Differentiating ssa-form programs," (2018), arXiv:1810.07951 [cs.PL].
[6] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, CoRR **abs/1907.07587** (2019), arXiv:1907.07587.
[7] "Breaking the Memory Wall: The AI Bottleneck," `https://blog.semi.org/semi-news/breaking-the-memory-wall-the-ai-bottleneck`.
[8] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," (2015), arXiv:1506.00019 [cs.LG].
[9] K. He, X. Zhang, S. Ren, and J. Sun, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016), 10.1109/cvpr.2016.90.
[10] J. Bettencourt, M. J. Johnson, and D. Duvenaud, (2019).
[11] C. M. Elliott, ACM Sigplan Notices **44**, 191 (2009).
[12] M. MacKay, P. Vicol, J. Ba, and R. B. Grosse, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 9029–9040.
[13] J. Behrmann, D. Duvenaud, and J. Jacobsen, CoRR **abs/1811.00995** (2018), arXiv:1811.00995.
[14] D. Maclaurin, D. Duvenaud, and R. Adams, in *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 37, edited by F. Bach and D. Blei (PMLR, Lille, France, 2015) pp. 2113–2122.
[15] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, in *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Curran Associates, Inc., 2017) pp. 2214–2224.
[16] J.-H. Jacobsen, A. W. Smeulders, and E. Oyallon, in *International Conference on Learning Representations* (2018).
[17] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, arXiv preprint arXiv:1209.5145 (2012).
[18] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM Review **59**, 65–98 (2017).
[19] K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).
[20] M. P. Frank, IEEE Spectrum **54**, 32–37 (2017).
[21] C. Lutz, "Janus: a time-reversible language," (1986), *Letter to R. Landauer*.
[22] M. P. Frank, *The R programming language and compiler*, Tech. Rep. (MIT Reversible Computing Project Memo, 1997).
[23] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018).
[24] T. Haulund, "Design and implementation of a reversible object-oriented programming language," (2017), arXiv:1707.07845 [cs.PL].
[25] M. P. Frank and T. F. Knight Jr, *Reversibility for efficient computing*, Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and ... (1999).
[26] J. G. Koller and W. C. Athas, in *Workshop on Physics and Computation* (1992) pp. 267–270.
[27] R. C. Merkle, R. A. Freitas, T. Hogg, T. E. Moore, M. S. Moses, and J. Ryley, Journal of Mechanisms and Robotics **10** (2018), 10.1115/1.4041209.
[28] K. Likharev, IEEE Transactions on Magnetics **13**, 242 (1977).
[29] V. K. Semenov, G. V. Danilov, and D. V. Averin, IEEE Transactions on Applied Superconductivity **13**, 938 (2003).
[30] R. Landauer, IBM journal of research and development **5**, 183 (1961).
[31] E. P. DeBenedictis, J. K. Mee, and M. P. Frank, Computer **50**, 76 (2017).
[32] D. R. Jefferson, ACM Transactions on Programming Languages and Systems (TOPLAS) **7**, 404 (1985).
[33] B. Boothe, ACM SIGPLAN Notices **35**, 299 (2000).
[34] G. H. Golub and C. F. Van Loan, *Matrix computations*, Vol. 3 (JHU press, 2012).
[35] "MLStyle.jl," `https://github.com/thautwarm/MLStyle.jl`.
[36] R. Orús, Annals of Physics **349**, 117–158 (2014).
[37] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," (2016), arXiv:1607.07892 [cs.MS].
[38] A. McInerney, *First steps in differential geometry* (Springer, 2015).
[39] "Paper's Github Repository," `https://github.com/GiggleLiu/nilangpaper/tree/master/codes`.
[40] J. Martens, I. Sutskever, and K. Swersky, "Estimating the hessian by back-propagating curvature," (2012), arXiv:1206.6464 [cs.LG].
[41] When people ask for the location in Beijing, they will start by asking which ring it is? We use the similar approach to locate the elements of Hessian matrix.
[42] M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464.
[43] S. Wisdom, T. Powers, J. R. Hershey, J. L. Roux, and L. Atlas, "Full-capacity unitary recurrent neural networks," (2016), arXiv:1611.00035 [stat.ML].
[44] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231.
[45] X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, "Yao.jl: Extensible, efficient framework for quantum algorithm design," (2019), arXiv:1912.10877 [quant-ph].
[46] C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).
[47] C. H. Bennett, SIAM Journal on Computing **18**, 766 (1989).
[48] R. Y. Levine and A. T. Sherman, SIAM Journal on Computing **19**, 673 (1990).
[49] T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174.
[50] C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.
[51] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on Nanotechnology* (2010) pp. 233–237.
[52] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on Nanotechnology* (2011) pp. 451–456.

[53] T. D. Nguyen and R. V. Meter, "A space-efficient design for reversible floating point adder in quantum computing," (2013), arXiv:1306.3760 [quant-ph].
[54] T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, "Quantum circuits for floating-point arithmetic," (2018), arXiv:1807.02023 [quant-ph].
[55] "FixedPointNumbers.jl," `https://github.com/JuliaMath/FixedPointNumbers.jl`.
[56] F. J. Taylor, R. Gill, J. Joseph, and J. Radke, IEEE Transactions on Computers **37**, 190 (1988).
[57] "LogarithmicNumbers.jl," `https://github.com/cjdoris/LogarithmicNumbers.jl`.
[58] M. P. Frank, in *35th International Symposium on Multiple-Valued Logic (ISMVL'05)* (2005) pp. 168–185.
[59] M. P. Frank, in *Reversible Computation*, edited by I. Phillips and H. Rahaman (Springer International Publishing, Cham, 2017) pp. 19–34.
[60] M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," (2002).
[61] L. Ruiz-Perez and J. C. Garcia-Escartin, Quantum Information Processing **16** (2017), 10.1007/s11128-017-1603-1.
[62] J.-G. L. Hao Xie and L. Wang, arXiv:2001.04121.
[63] H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, Physical Review X **9** (2019), 10.1103/physrevx.9.031041.
[64] M. Seeger, A. Hetzel, Z. Dai, E. Meissner, and N. D. Lawrence, "Auto-differentiating linear algebra," (2017), arXiv:1710.08717 [cs.MS].
[65] Z.-Q. Wan and S.-X. Zhang, "Automatic differentiation for complex valued svd," (2019), arXiv:1909.02659 [math.NA].
[66] C. Hubig, "Use and implementation of autodifferentiation in tensor network methods with complex scalars," (2019), arXiv:1907.13422 [cond-mat.str-el].
[67] I. Kobyzev, S. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," (2019), arXiv:1908.09257 [stat.ML].
[68] D. P. Kingma and P. Dhariwal, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 10215–10224.
[69] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real nvp," (2016), arXiv:1605.08803 [cs.LG].
[70] S.-H. Li and L. Wang, Physical Review Letters **121** (2018), 10.1103/physrevlett.121.260601.
[71] P. Hut, J. Makino, and S. McMillan, The Astrophysical Journal **443**, L93 (1995).
[72] D. N. Laikov, Theoretical Chemistry Accounts **137** (2018), 10.1007/s00214-018-2344-7.
[73] M. Giftthaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, Advanced Robotics **31**, 1225–1237 (2017).
[74] K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, and A. Paz, Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018 (2018), 10.1145/3183895.3183901.
[75] M. Bender, P.-H. Heenen, and P.-G. Reinhard, Rev. Mod. Phys. **75**, 121 (2003).

Appendix A: NiLang Grammar

To define a reversible function one can use "@i" plus a normal function definition like bellow

```
"""
docstring...
"""
@i function f(args..., kwargs...) where {...}
    <stmts>
end
```

where the definition of "<stmts>" are shown in the grammar on the next column. The following is a list of terminologies used in the definition of grammar

- *ident*, symbols
- *num*, numbers
- ϵ, empty statement
- *JuliaExpr*, native Julia expression
- [ ], zero or one repetitions.

Here, all *JuliaExpr* should be pure, otherwise the reversibility is not guaranteed. Dataview is a view of a data, it can be a bijective mapping of an object, an item of an array or a field of an object.

⟨Stmts⟩ ::= ϵ
| ⟨Stmt⟩
| ⟨Stmts⟩ ⟨Stmt⟩
⟨Stmt⟩ ::= ⟨BlockStmt⟩
| ⟨IfStmt⟩
| ⟨WhileStmt⟩
| ⟨ForStmt⟩
| ⟨InstrStmt⟩
| ⟨RevStmt⟩
| ⟨AncillaStmt⟩
| ⟨TypecastStmt⟩
| ⟨@routine⟩ ⟨Stmt⟩
| ⟨@safe⟩ *JuliaExpr*
| ⟨CallStmt⟩
⟨BlockStmt⟩ ::= *begin* ⟨Stmts⟩ *end*
⟨RevCond⟩ ::= *(JuliaExpr , JuliaExpr)*
⟨IfStmt⟩ ::= *if* ⟨RevCond⟩ ⟨Stmts⟩ [*else* ⟨Stmts⟩] *end*
⟨WhileStmt⟩ ::= *while* ⟨RevCond⟩ ⟨Stmts⟩ *end*
⟨Range⟩ ::= *JuliaExpr : JuliaExpr* [*: JuliaExpr*]
⟨ForStmt⟩ ::= *for ident* = ⟨Range⟩ ⟨Stmts⟩ *end*
⟨KwArg⟩ ::= *ident = JuliaExpr*
⟨KwArgs⟩ ::= [⟨KwArgs⟩ ,] ⟨KwArg⟩
⟨CallStmt⟩ ::= *JuliaExpr (* [⟨DataViews⟩] [*;* ⟨KwArgs⟩] *)*
⟨Constant⟩ ::= *num* | π
⟨InstrBinOp⟩ ::= += | -= | $\veebar$=
⟨InstrTrailer⟩ ::= [.] *(* [⟨DataViews⟩] *)*
⟨InstrStmt⟩ ::= ⟨DataView⟩ ⟨InstrBinOp⟩ *ident* [⟨InstrTrailer⟩]
⟨RevStmt⟩ ::= ~ ⟨Stmt⟩
⟨AncillaStmt⟩ ::= *ident* ← *JuliaExpr*
⟨TypecastStmt⟩ ::= *(JuliaExpr => JuliaExpr) (ident)*
⟨@routine⟩ ::= *@routine ident* ⟨Stmt⟩
⟨@safe⟩ ::= *@safe JuliaExpr*
⟨DataViews⟩ ::= ϵ
| ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩ ...
⟨DataView⟩ ::= ⟨DataView⟩ *[JuliaExpr]*
| ⟨DataView⟩ . *ident*
| *JuliaExpr (* ⟨DataView⟩ *)*
| ⟨DataView⟩ *'*
| - ⟨DataView⟩
| ⟨Constant⟩
| *ident*

Appendix B: Instruction Table

The translation of instructions to Julia functions
The list of instructions implemented in NiLang

instruction	translated	symbol
$y \mathrel{+}= f(args...)$	`PlusEq(f)(args...)`	$\oplus$
$y \mathrel{-}= f(args...)$	`MinusEq(f)(args...)`	$\ominus$
$y \mathrel{\veebar}= f(args...)$	`XorEq(f)(args...)`	$\odot$

Table II. Instructions and their compilation in NiLang.

instruction	output
SWAP(a, b)	b, a
ROT(a, b, θ)	$a\cos\theta - b\sin\theta, b\cos\theta + a\sin\theta, \theta$
IROT(a, b, θ)	$a\cos\theta + b\sin\theta, b\cos\theta - a\sin\theta, \theta$
$y \mathrel{+}= a\wedge b$	$y + a^b, a, b$
$y \mathrel{+}= \exp(x)$	$y + e^x, x$
$y \mathrel{+}= \log(x)$	$y + \log x, x$
$y \mathrel{+}= \sin(x)$	$y + \sin x, x$
$y \mathrel{+}= \cos(x)$	$y + \cos x, x$
$y \mathrel{+}= \text{abs}(x)$	$y + \lvert x\rvert, x$
NEG(y)	$-y$
CONJ(y)	y'

Table III. A collection of reversible instructions, "." is the broadcasting operations in Julia.

Appendix C: Learn by consistency

Consider a training that with input $\mathbf{x}^*$ and output $\mathbf{y}^*$, find a set of parameters $\mathbf{p}_x$ that satisfy $\mathbf{y}^* = f(\mathbf{x}^*, \mathbf{p}_x)$. In traditional machine learning, we define a loss $\mathcal{L} = \text{dist}(\mathbf{y}^*, f(\mathbf{x}^*, \mathbf{p}_x))$ and minimize it with gradient $\frac{\partial L}{\partial \mathbf{p}_x}$. This works only when the target function is locally differentiable.

Here we provide an alternative by making use of reversibility. We construct a reversible program $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$, where $\mathbf{p}_x$ and $\mathbf{p}_y$ are "parameter" spaces on the input side and output side. The algorithm can be summarized as

Algorithm 2: Learn by consistency

Result: $\mathbf{p}_x$
Initialize $\mathbf{x}$ to $\mathbf{x}^*$, parameter space $\mathbf{p}_x$ to random.
if $\mathbf{p}_y$ *is* `null` **then**
    $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}^*)$
else
    $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$
    while $\mathbf{y} \not\approx \mathbf{y}^*$ **do**
        $\mathbf{y} = \mathbf{y}^*$
        $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}, \mathbf{p}_y)$.
        $\mathbf{x} = \mathbf{x}^*$
        $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$

Here, `parameter`$(\cdot)$ is a function for taking the parameter space. This algorithm utilizes the self-consistency relation

$$\mathbf{p}_x^* = \texttt{parameter}(f_r^{-1}(\mathbf{y}^*, \texttt{parameter}(f_r(\mathbf{x}^*, \mathbf{p}_x^*)))), \quad \text{(C1)}$$

Similar idea of training by consistency is used in self-consistent meanfield theory [75] in physics. Finding the self-consistent relation is crucial to a self-consistency based training. Here, the reversibility provides a natural self-consistency relation. However, it is not a silver bullet, let's consider the following example

```julia
@i function f1(y!, x, p!)
    p! += identity(x)
    y! -= exp(x)
    y! += exp(p!)
end

@i function f2(y!, x!, p!)
    p! += identity(x!)
    y! -= exp(x!)
    x! -= log(-y!)
    y! += exp(p!)
end

function train(f)
    loss = Float64[]
    p = 1.6
    for i=1:100
        y!, x = 0.0, 0.3
        @instr f(y!, x, p)
        push!(loss, y!)
        y! = 1.0
        @instr (~f)(y!, x, p)
    end
    loss
end
```

Functions `f1` and `f2` computes $f(x, p) = e^{(p+x)} - e^x$ and stores the output in a new memory `y!`. The only difference is `f2` uncomputes x arithmetically. The task of training is to find a p that make the output value equal to target value 1. After 100 steps, `f2` runs into the fixed point with x equal to 1 upto machine precision. However, parameters in `f1` does change at all. The training of `f1` fails because this function actually computes $\texttt{f1}(y, x, p) = y + e^{(p+x)} - e^x, x, x + p$, where the training parameter p is completely determined by the parameter space on the output side $x \cup x + p$. As a result, shifting y directly is the only approach to satisfy the consistency relation. On the other side, $\texttt{f2}(y, x, p) = y + e^{(p+x)} - e^x, \tilde{0}, x + p$, the output parameters $\tilde{0} \cup x + p$ can not uniquely determine input parameters p and x. Here, we use $\tilde{0}$ to denote the zero with rounding error.

By viewing $\mathbf{x}$ and parameters in $\mathbf{p}_x$ as variables, we can study the trainability from the information perspective.

Theorem 2. *Only if the the conditional entropy $S(\mathbf{y}|\mathbf{p}_y)$ is nonzero, algorithm 2 is trainable.*

Proof. The above example reveals a fact that the training can not work when output parameters completely determines input parameters. In other words, if $S(\mathbf{p}_x|\mathbf{p}_y) = 0$, the traning

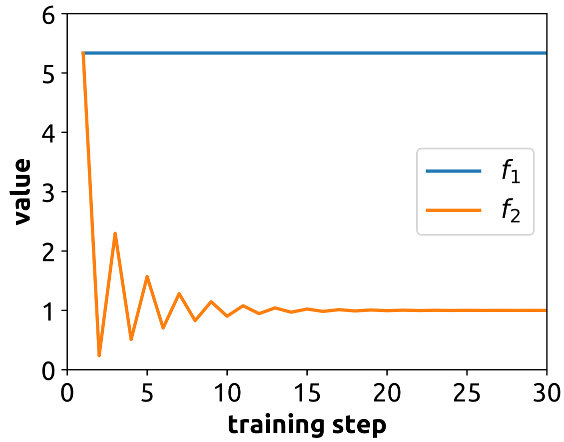

Figure 5. The output value `y!` as a function of self-consistent training step.

can not work.

$$\begin{aligned} S(\mathbf{p}_x|\mathbf{p}_y) &= S(\mathbf{p}_x \cup \mathbf{p}_y) - S(\mathbf{p}_y) \\ &\leq S\left((\mathbf{p}_x \cup \mathbf{x}) \cup \mathbf{p}_y\right) - S(\mathbf{p}_y), \\ &\leq S\left((\mathbf{p}_y \cup \mathbf{y}) \cup \mathbf{p}_y\right) - S(\mathbf{p}_y), \\ &\leq S(\mathbf{y}|\mathbf{p}_y). \end{aligned} \tag{C2}$$

The third line uses the bijectivity $S(\mathbf{x} \cup \mathbf{p}_x) = S(\mathbf{y} \cup \mathbf{p}_y)$. This inequality shows that when the parameter space on the output side satisfies $S(\mathbf{y}|\mathbf{p}_y) = 0$, i.e. contains all information to determine the output field, the input parameters are also completely determined by this parameter space, hence traning can not work. □

In the above example, it corresponds to the case $S\left(e^{(x+y)-e^x}|x \cup x + y\right) = 0$ in $f1$. The solution is to remove the information redundancy in output parameter space through uncomputing as shown in $f2$.