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 the instruction level adjoint mode differential programming, i.e. knowing only the backward rules of basic instructions like +, -, * and /, obtain the gradient of parameters in an arbituary differentiable program with proper performance. In this paper, we review briefly why instruction level automatic differentiation is hard for current machine learning frameworks and propose an implementation of reversible Turing machine as a solution. We show how reversible programming can differentiate a general program to an arbituary order automatically by viewing basic instructions as the computational graph.

I. INTRODUCTION

[**JG: TODOs: quantum example?**] There are two modes of automatic differentiation (AD) [1], the tangent mode [2] and the adjoint mode. Consider a multi-in multi-out function $\vec{y} = f(\vec{x})$, the tangent mode computes one column of its Jacobian $\frac{\partial \vec{y}}{\partial x_i}$ efficiently, where x_i is one of the input variables. Whereas the adjoint mode computes one row of Jacobian $\frac{\partial y_i}{\partial \vec{x}}$ efficiently. Most popular automatic differentiation package implements the adjoint mode AD. Because the adjoint mode is computational more efficient in variational applications, where the loss as output is always a scalar. However, implementing adjoint mode AD is harder than implementing the tangent mode AD. It requires a program's intermediate state for back propagation, including

1. the computation graph,
2. and input variables of nodes in computation graph.

A computational graph is a directed acyclic graph (DAG) that records the relation between data (edges) and functions (nodes). In Pytorch [3] and Flux [4], every variable has a tracker field that stores its parent information, i.e. the input data and function that generate this variable. TensorFlow [5] implements a static computational graph as a description of the program before actual computation happens. Source to source automatic differentiation package Zygote [4, 6] use a intermediate representation (IR) of a program the static single assignment (SSA) form as the computation graph in order to propagate a native julia code. To cache the intermediate state, it uses a global storage.

Several limitations are observed in these AD implementions due to the recording and caching. First of all, these package requires a lot primitive functions with programmer defined backward rules. This is not nessesary given the fact that, at the lowest level, these primitive functions are compiled to a finite set of instructions including '+', '-', '*', '/' and conditional jump statements. By defining backward rules for these basic instructions, AD should just works. These machine learning packages can not use instructions as the computational graph for practical reasons. The cost of memorizing the computational graph and caching intermediate states is huge. It can decrease the performance for more than two orders when a program contains loops (as we will show latter). Even more, the memory consumption for caching intermediate results increases linearly as time. In many deep learning models like recurrent neural network [7] and residual neural networks [8], the depth can reach several thousand, the memory wall [9] can be big problem. Secondly, inplace functions are not handled properly in the diagram of computation graph. Even in source to source AD engine Zygote, it is not trivil to handle inplace functions. Most functions in BLAS and LAPACK are implemented as inplace functions. The lack of automatic differentiation support to inplace functions make the memory wall problem even more sevire. It is also harmful to code reusing since all packages using BLAS functions should define their own backward rules for their non-inplace wrappers. Thridly, obtaining higher order gradients are not efficient in these packages. For example, in most machine learning packages, people back propagate the whole program of obtaining first order gradient to obtain the second order gradients. The repeated use of back propagation cause exponential overhead with respect to the order of gradients. A better approach to obtain higher order gradients is through Taylor propagation like in JAX [10]. However Taylor propagation requires writing rules for all primitives. Besides the exponential overhead, the sorce to source AD engine Zygote suffers from the significant overhead of just in time compiling in Julia language [11].

Our solution to these issues is making a program time reversible. Making use of reversibility has been used in machine learning as a promising approach to save memory. People use information buffer [12] and reversible activation functions to reduce the memory allocations in recurrent neural network [13] and residual neural networks [14]. However, the use of reversiblity in these cases are not general purposed.

Hence we develop a embeded domain specific language (eDSL) in Julia language that implements reversible Turing machine (RTM). [15, 16]. The gradient of any program writting in this eDSL can be obtained in comparable time with the forward computation. The implementation of AD is similar to `ForwardDiff` [2] but runs backward. There has

* cacate0129@iphy.ac.cn

been some prototypes of reversible languages like Janus [17], R (not the popular one) [18], Erlang [19] and object oriented ROOPL [20]. These languages have reversible control flow that allowing user to input an additional postcondition in control flows to help programs run backward. In the past, the main motivation of making a program time reversible is to support reversible devices. Reversible devices do not have a lower bound of energy consumption by Landauer principle [21]. However CMOS devices still has two orders [16] space to optimize regarding this lower bound. The main contribtion of our work is breaking the information barrier between machine learning community and reversible programming community, providing yet another strong motivation to develop a reversible programming. Our eDSL borrows the design of reversible control flow in the Janus, meanwhile provides multiple dispatch based abstraction. With these additional features, the AD engine differentiating a general program could be implemented in less than 100 lines. Our eDSL generates native julia code, and is completely compatible with Julia language. Potensial applications includes

1. generate AD rules for primitive functions like `exp`,
2. control problem in robotics [22] where tensor is not the dominating data type,
3. differentiating over reversible integrators [23] without intermediate state caching,
4. Stablize linear algebras functions backward rules. Current backward rules for singular value decomposition (SVD) and eigenvalue decomposition (ED) [24–26] are vulnarable to spectrum degeneracy. The development of backward rules for these linear algebra functions can greatly change the researches in physics [27, 28].

In this paper, we first introduce the design of this eDSL in Sec. II. On this eDSL, we show how to back propagation Jacobians and Hessians. Then we propose a training strategy in Sec. IV that uses reversibility, rather than gradients. In Sec. V, we show several examples. In Sec. VI, we discuss on several important issues, how time space tradeoff works, reversible instructions and hardwares and finally an outlook to some open problems to be solved.

II. LANGUAGE DESIGN

We instroduce NiLang, a eDSL in Julia that simulates RTM. Its grammar is shown in Appendix A. Its main feature is contained in a single macro `@i`. It interprets a NiLang function to a native Julia function. At the same time, it generates the inverse of this function. The target instructions are closed under the inverse operation "∼", hence all functions defined in NiLang are also closed under "∼" operations.

In a mordern programming language, functions are pushed to a global stack for scheduling. The memory layout of a function is consisted of input arguments, a function frame with informations like return address and saved memory segments, local variables and working stack. After each call, the function clears the input arguments, function frame, local variables and working stack, only stores the return value. In the reversible programming style, this kind of design pattern is nolonger the best practise, input variables can not be easily discarded after a function call, since discarding information may ruin reversibility. Hence, a function or a instruction changes inputs "inplace" in NiLang. To design such a eDSL, we first introduce a reversible IR that plays a central role in NiLang.

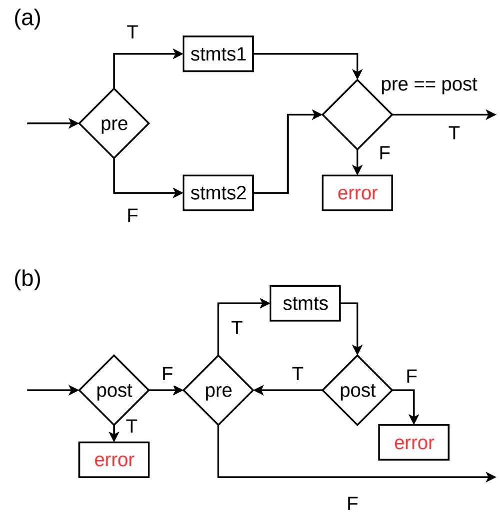

Figure 1. Flow chart for reversible (a) `if` statement and (b) `while` statement. "stmts", "stmts1" and "stmts2" are statements, statements in true branch and statements in false branch respectively. "pre" and "post" are precondition and postconditions respectively. "error" refers to `InvertibilityError`.

A. Reversible IR

In NiLang's IR, a statement can be an instruction, a function call, a controlflow, a macrocall or the inverse statement ∼. With the reversible IR, the inverse of a statement can be defined easily as shown in in Table I.

The reversible control flow is different from the irreversible one that a condition expression in a `if` or a `while` statements is a two-element tuple that consist of a precondition and a postcondition. This design allows user putting additional postcondition in control flows to help reverse the program. For the `if` statement as shown in Fig. 1 (a), the program enters the branch specified by precondition. After executing this branch, the program checks the consistency of precondition and postcondition to make sure they are same. In the reverse pass, the program enters the branch specified by the postcondition. For the `while` statement as shown in Fig. 1 (b), before entering, the program check the postcondition to make sure it is false. After each iteration, the program asserts the postcondition to be true. The inverse function exchanges the precondition and postcondition. The definition of reversible `for` statement is similar to irreversible ones except it is more restrictive. The

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>...)`
`@anc <a> = <expr>`	`@deanc <a> = <expr>`

`begin`
`  <stmts>`
`end`
	`begin`
	`  ~(<stmts>)`
	`end`

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

program first stores the loop informations, start, step and stop. After the executing the loop, the program checks the values of these variables to make sure they are not changed. The reverse program exchanges start and stop and inverse the sign of step.

There is no assign statements in a reversible language, a reversible replacement is the macro `@anc`. `@anc a = <expr>` binds variable `a` to an initial value specified by `<expr>`. Its inverse `@deanc a = <expr>` deallocates the variable `a`. Before deallocating the variable, the program checks that the value of variable is same as the value of `<expr>`, otherwise throws an `InvertibilityError`. `@anc` and `@deanc` must appear in pairs inside a function call, a while statement or a for statement. `@deanc` will be added automatically. Similar designs in Janus and R are `local/delocal` statement and `let` statement. The additional check underlines the difference between the irreversible assign statement and reversible ancilla statement. The `@safe` macro can be followed by an arbituary statement, it allows user to use external statements that does not break reversibility. For example, one can use `@safe @show var` for debugging.

B. Compiling

The interprecation of a reversible function consists three stages. The first stage preprocess human inputs to a reversible IR, The second stage generates the reversed IR according to table Table I. The third stage is translating this reversible IR to native Julia code. The following example shows how to compile an `if` statement

```julia
julia> using NiLangCore, MacroTools

julia> ex0 = :(if (x > 3, ~)
            -arr[3].value += x * y
        end);

julia> ex0 |> prettify
:(if (x > 3, ~)
      -(arr[3]).value += x * y
  end)

julia> ex1 = NiLangCore.precom_ex(ex0,
        NiLangCore.PreInfo());

julia> ex1 |> prettify  # after stage 1
:(if (x > 3, x > 3)
      -(arr[3]).value += x * y
  else
  end)

julia> ex2 = NiLangCore.dual_ex(ex1);

julia> ex2 |> prettify  # after stage 2
:(if (x > 3, x > 3)
      -(arr[3]).value -= x * y
  else
  end)

julia> ex3 = NiLangCore.interpret_ex(ex2);

julia> ex3 |> prettify  # after stage 3
quote
    wren = x > 3
    if wren
        @instr -(arr[3]).value -= x * y
    else
    end
    @invcheck x > 3 wren
end
```

In the first stage, the preprocessor expands the symbol ~ in postcondition field of `if` statement to the precondition as shown above. Besides, it adds missing `@deanc` to ensure `@anc` and `@deanc` statements appear in pairs and expands `@routine` macro. `@routine r <stmt>` records a statement to symbol `r`. When `~@routine r` is called, the inverse statement is inserted to that position for uncomputing. We will use macro extensively in the example in Sec. V. In the last stage, the compiler adds `@instr` before each instruction and function call statement. The macro `@instr` assign the output of a function to the argument list of a function. We will explain this macro in detail in next subsection. It also adds statements to check the consistency between preconditions and postconditions to ensure reversibility. In the above example, `@invcheck x > 3 wren` will throw an error if the variable `wren` and `x > 3` are not equivalent. Finally, at the end of a function body, it attaches a return statement that uses input variables as the output. Now the function is ready to execute on the host language.

C. Types and Dataviews

So far, the language design is not too different from a tranditional reversible language. Next, we introduce type and dataviews that are important to the implementation of adjoint mode AD.

The constructor of a type is a reversible function. Its inverse function is a "destructor", which does not deallocate memory directly but unpacks data. One can use `@iconstruct` to define a reversible constructor. For example, we can define a type that copies the input data to a new field.

```
using NiLangCore, Test

struct DVar{T}
    x::T
    g::T
end

@iconstruct function DVar(xx, gg=zero(xx))
    gg += identity(xx)
end

@test (~DVar)(DVar(0.5)) == 0.5
```

Here, the `@iconstruct` generates a reversible constructor with single parameters `xx` as input. The statement `gg = zero(xx)` initializes a new memory to be used. The body of function is a reversible program that modifies `xx` and `gg` reversiblely. Finally it calls the default constructor `DVar(xx, gg)`. It is easy to define the inverse procedure that transform a `DVar{T}` instance to a `T` instance by reversing the statements. With the flexibility to use types, it is not nessesary to use global stacks in our eDSL.

Before introducing dataviews, let's first consider the following line that appear in the last subsection

```
-arr[3].value += x * y
```

In Julia, this statement will raise a syntax error, since a function call `(-)` can not be assigned. Meanwhile `arr[3]` might be a immutable type. In our eDSL, we wish it works because every memory cell should be modifiable "in-place". As we have mensioned, `-arr[3].value += x * y` is translated to `@instr -arr[3].value += x * y` at the thrid stage. `@instr` translate the statement to

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

`PlusEq(*)(-arr[3].value, x, y)` computes the output, which is a tuple of length 3. The assignments in line 4 and 5 can be handled directly while the assignment in line 2-3 are not so, where `chfield` is used to modify a dataview. A dataview of a data can be data itself, a field of its view, an array element of its view, or a bijective mapping of its view. `chfield(x, Val{:value}, val)` can modify the `value` field of an instance given the default constructor of a type is not overwritten. A bijective mapping of a field can also be modified. To change `-x`, one simply overwrite the `chfield(x, -, res[1])` function.

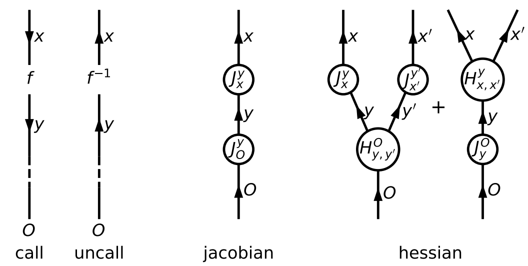

Figure 2. Adjoint rules for Jacobians and Hessians in tensor network language.

III. AUTOMATIC DIFFERENTIATION

A. First order gradient

Given a node $\vec{y} = f(\vec{x})$ in a computational graph, the adjoint mode AD propagates the Jacobians in the reversed direction like

$$\begin{aligned} J^O_{O'} &= \delta_{O,O'} \\ J^O_x &= J^O_y J^y_x, \end{aligned} \tag{1}$$

Here, O is the outputs. Einstein's notation is used so that duplicated indices are summed over. Tagent mode instruction level automatic differentiation can be implemented easily in a irreversible language with dual numbers [2]. Here we focus on the adjoint mode. Its propagation rule can be rewritten in the language of tensor networks [29] as shown in Fig. 2.

In reversible programming with multiple dispatch, the adjoint mode AD algorithms can be implemented as

Algorithm 1: Reversible programming AD

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

Here, `GVar` is a reversible type. The constructor attaches a zero gradient field to a variable, which is similar to the dual number in tangent mode automatic differentiation. If `x` is an array, `GVar` will be broadcasted to each array element automatically. The gradient field of a `GVar` instance can be accessed by the `grad` dataview. Its inverse `~GVar` deallocates the gradient field safely and returns its value field. Here, "safely" means before deallocation, the program will check the gradient field to make sure its value is 0. When an instruction `instruct` meets a `GVar`, besides computing its value field $\texttt{value}(\vec{y}) = \texttt{instruct}(\texttt{value}(\vec{x}))$, it also updates the gradient field $\texttt{grad}(\vec{x}) = \left[J_{\vec{x}}^{\vec{y}}\right]^{-1} \texttt{grad}(\vec{y})$, where $\left[J_{\vec{x}}^{\vec{y}}\right]^{-1}$ is the Jacobian of $\texttt{instruct}^{-1}$. A detailed example could be found in Sec. V D. When gradients are not used anymore, the reversible way to deallocate gradients is uncomputing the whole process of obtaining them.

```
julia> using NiLang, NiLang.AD

julia> x, y = GVar(0.5), GVar(0.6)
(GVar(0.5, 0.0), GVar(0.6, 0.0))

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

julia> @instr x += identity(y)

julia> y
GVar(0.6, -1.0)

julia> @instr grad(x) -= identity(1.0)

julia> @instr (~GVar)(x)

julia> x
1.1
```

The implementation of Algorithm 1 is so short that we present the function defintion as follows

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

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

    ~@routine getiloss
end
```

In NiLang, broadcasting and array indexing are supported. To avoid confusion, tuple indexing is forbidden delebrately, e.g. one can use `tget(tuple, 2)` to get the second element of a tuple. This program first checks the input parameters and locate the loss variable. Then `Loss` unwraps the loss variable, the information about location of loss variable is transferred to the ancilla `iloss`. After computing the forward pass and backward pass, `@routine getiloss` uncomputes `iloss` and returns the location information to the target variable.

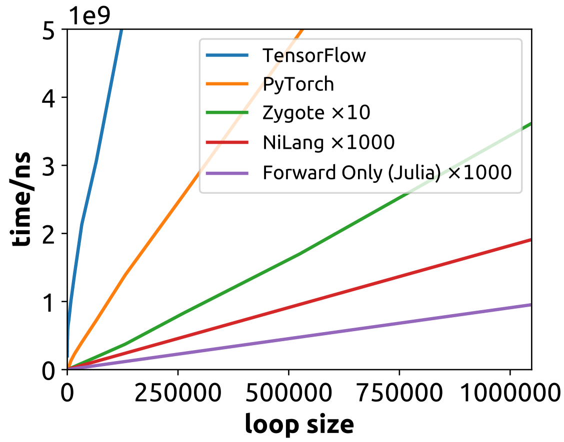

Figure 3. The time for obtaining gradient as function of loop size. $\times n$ in lengend represents a rescaling of time.

The overhead of using `GVar` type can be removed thanks to Julia's multiple dispatch and type inference. Let's consider a simple example that acccumulate 1.0 to a target variable x for n times

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

We implement the same function with TensorFlow, PyTorch and Zygote for comparison. The codes could be found in our paper's github repository [30]. 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. In real application, the reversible program can have memory or computing time overhead. We will discuss the details of time and space tradeoff in Sec. VI A.

B. Second order gradient

Second order gradients can be obtained with 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. Given the function space is closed under gradient operation, one can obtain higher order gradients recursively without defining new backward rules.

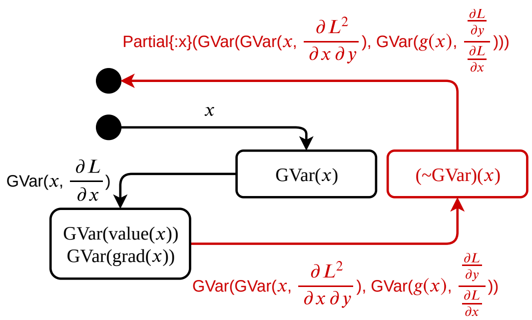

Figure 4. Obtaining the second order gradient with the reversive differentiation approach. Black lines are computing gradients, red lines are back propagating the process of obtaining the first order gradients. Annotations on lines are data types and their fields used in the computation.

Fig. 4 show the four passes in computing Hessian. The first two passes (black lines) are obtaining gradients. Before entering the third pass, the program wraps each field in `GVar` with `GVar`. Then we pick a variable x_i and add 1 to `grad(grad(`x_i`))` to compute one row of Hessian. At the final stage, the `~GVar` operation does not unwrap `GVar` directly because the gradient fields may not be zero in this case. Instead, we use `Partial{:x}(·)` to safely compute `~GVar` on data type `GVar{<:GVar, <:GVar}`. It takes the `x` field without actually deallocating memory. By iterating over different x_i, one can obtains the Hessian matrix.

2. *Taylor propagation*

A probably more efficient approach is back propagating Hessians directly [31]

$$
\begin{aligned}
H^{O}_{O',O''} &= \mathbf{0}\\
H^{O}_{x,x'} &= J^{y}_{x} H^{O}_{y,y'} J^{y'}_{x'} + J^{O}_{y} H^{y}_{x,x'}
\end{aligned}
\tag{2}
$$

The Hessian tensor $H^{O}_{x,x'}$ is a rank three, where the top index is often takes as a scalar and omitted. In tensor network language, the above equation can be represented as in 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. The later requires traversing the computational graph repeatedly. In JAX, in order to support Taylor propagation, the propagation rules for part of primitives should be manually defined. The exhaused support requires much more effort than the first order gradient propagation, this is why most AD packages use the recursive approach. Instruction level automatic differentiation has the advantage of having very limited primitives. It is more flexible in obtaining higher order gradients like Hessian. An example is provided in Sec. V B.

C. Gradient on ancilla problem

Ancilla can also carry gradients during computation. As a result, even if an ancilla can be uncomputed regoriously in the original program, its `GVar` version can not be safely uncomputed. In these case, we simply "drop" the gradient field instead of raising an error. In this subsection, we prove doing this is safe, i.e. dropping the gradient field of an ancilla when deallocating does not have side effect to the output variables of a function.

Consider a reversible function $\vec{y}, b = f(\vec{x}, a)$, where a and b are the input and output values of an ancilla. The reversibility requires $b = a$ for any $\vec{x}$. So that

$$
\frac{\partial b}{\partial \vec{x}} = \vec{0}. \tag{3}
$$

In the backward pass, we discarded the gradient field of b. So the question becomes does `grad(b)` have effect on the $\vec{x}$? The gradient fields are derived from the values of variables, they should not have any effect to the `value` fields. The rest is to show `grad(b)` does not have effect on `grad.(`$\vec{x}$`)` too. It can be seen from the expression the back propagation rule

$$
\frac{\partial O}{\partial \vec{x}} = \frac{\partial O}{\partial \vec{y}}\frac{\partial \vec{y}}{\partial \vec{x}} + \frac{\partial O}{\partial b}\frac{\partial b}{\partial \vec{x}}, \tag{4}
$$

where the second term with $\frac{\partial O}{\partial b}$ vanishes naturally.

IV. LEARN BY CONSISTENCY

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

Here we provide an alternative by making use of reversibility. We construct a reversible program $\vec{y}, \vec{p}_y = f_r(\vec{x}, \vec{p}_x)$, where $\vec{p}_x$ and $\vec{p}_y$ are "garbage" spaces which include parameters. The algorithm can be summarized as

---

Algorithm 2: Learn by consistency

Result: $\vec{g}_x$

Initialize $\vec{x}$ to $\vec{x}^{*}$, garbage space $\vec{g}_x$ to random.

if $\vec{g}_y$ *is* `null` **then**

  $\vec{x}, \vec{g}_x = f_r^{-1}(\vec{y}^{*})$

else

  $\vec{y}, \vec{g}_y = f_r(\vec{x}, \vec{g}_x)$

  while $\vec{y} \not\approx \vec{y}^{*}$ **do**

    $\vec{y} = \vec{y}^{*}$

    $\vec{x}, \vec{g}_x = f_r^{-1}(\vec{y}, \vec{g}_y)$.

    $\vec{x} = \vec{x}^{*}$

    $\vec{y}, \vec{g}_y = f_r(\vec{x}, \vec{g}_x)$

Here, garbage(·) is a function for taking the garbage space. This algorithm utilizes the self-consistency relation

$$\vec{g}_x^* = \texttt{garbage}(f_r^{-1}(\vec{y}^*, \texttt{garbage}(f_r(\vec{x}^*, \vec{g}_x^*)))), \qquad (5)$$

Similar idea of training by consistency is used in self-consistent meanfield theory [] in physics. The difficult part of self-consistent training is to find a self-consistency relation, here the reversibility provides a natural self-consistency relation. Learn by consistency can be used to handle descrete optimization. However, it is not a silver bullet, and should be used with causion. Let's consider the following example

```
@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 "uncompute" x arithmetically. The task of training is to find a p that make the output value equal to target value 1. After 200 steps, f2 runs into the fixed point with x equal to 1 upto machine precision. However, f1 does not do any training. The training of f2 fails because this function actually computes $\texttt{f1}(y, x, p) = y + e^{(p+x)} - e^x, x, x + p$, where the training parameter y is completely determined by the garbage 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, $\texttt{f2}(y, x, p) = y + e^{(p+x)} - e^x, (x - \log(e^x)), x + p$, the garbage space can not uniquely determine the input garbage space p and y.

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

***Theorem* 1.** *Only if the the conditional entropy $S(\vec{y}|\vec{p}_y)$ is nonzero, algorithm 2 is trainable.*

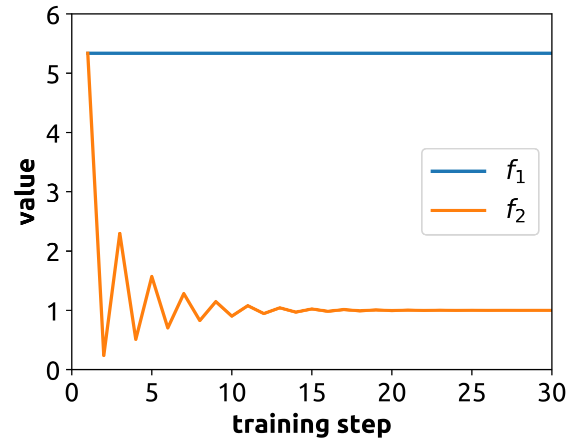

Figure 5. The value of x as a function of self-consistent training step.

Proof. The above example reveals a fact that the training can not work when the output $\vec{p}_y$ completely determines $\vec{p}_x$, that is

$$\begin{aligned} S(\vec{p}_x|\vec{p}_y) &= S(\vec{p}_x \cup \vec{p}_y) - S(\vec{p}_y) \\ &\leq S\left((\vec{p}_x \cup \vec{x}) \cup \vec{p}_y\right) - S(\vec{p}_y), \\ &\leq S\left((\vec{p}_y \cup \vec{y}) \cup \vec{p}_y\right) - S(\vec{p}_y), \\ &\leq S(\vec{y}|\vec{p}_y). \end{aligned} \qquad (6)$$

The third line uses the bijectivity $S(\vec{x} \cup \vec{p}_x) = S(\vec{y} \cup \vec{p}_y)$. □

This inequality shows that when the garbage space on the output side satisfies $S(\vec{y}|\vec{p}_y) = 0$, i.e. contains all information to determine the output field, the input parameters are also completely determined by this garbage space. In the above examples, it corresponds to the case $S\left(e^{(x+y)-e^x}|x \cup x + y\right) = 0$ in f_1. One should remove the redundancy of information by uncomputing to make training by consistency work properly.

V. EXAMPLES

A. Computing Fibonacci Numbers

An example that everyone likes

```julia
@i function rfib(out, n::T) where T
    @anc n1 = zero(T)
    @anc n2 = zero(T)
    @routine init begin
        n1 += identity(n)
        n1 -= identity(1.0)
        n2 += identity(n)
        n2 -= identity(2.0)
    end
    if (value(n) <= 2, ~)
        out += identity(1.0)
    else
        rfib(out, n1)
        rfib(out, n2)
    end
    ~@routine init
end
```

The following example shows how to construct a reversible `while` statement. It computes the first Fibonacci number that greater or equal to 100

```julia
@i function rfib100(n)
    @safe @assert n == 0
    while (fib(n) < 100, n != 0)
        n += identity(1.0)
    end
end
```

In this example, the postcondition `n!=0` is false before entering the loop, and becomes true in later iterations. In the reverse program, the `while` statement stops at `n==0`.

B. exp function

An exp function can be computed using Taylor expansion

$$y! + = \sum_n \frac{x^n}{\mathrm{factorial}(n)} \tag{7}$$

This is a recursive algorithm that mimics the famous pebble game [15]. In each iteration, the accumulated term is $s_n \equiv \frac{x^n}{\mathrm{factorial}(n)}$, the recursion relation is written as $s_n = \frac{x s_{n-1}}{n}$. There is no known constant memory and polynomial time algorihm to pebble game. Here the case is different. Notice $*=$ and $/=$ are arithmetically reversible to each other, we can "uncompute" with the previous state with the relation $s_{n-1} = \frac{n s_n}{x}$ to deallocate the memory.

```julia
using NiLang, NiLang.AD

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

    y! += identity(1.0)
    @routine r1 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 r1
end
```

The definition of SWAP instruction can be found in Appendix B. The two lines bellow the comment "`# arithmetic uncompute`" "uncomputes" variables `anc1` and `anc2` approximately. This uncomputation is only arithmeticly true. As a result, the final output is not exact due to the rounding error. On the other side, the reversibility is not harmed since the inverse call at the last line of function uncomputes all ancilla bits rigorously. The `while` statement takes two conditions, the precondition and postconditoin. Precondition `val(anc1) > atol` indicates when to break the forward pass and post condition `!isapprox(iplus, 0.0)` indicates when to break the backward pass.

To obtain the gradient, one can wrap the loss with `Loss` type and feed it into `iexp'` function

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

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

julia> grad(x)
4.9530324244260555

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
```

`iexp'` is a callable instance of type `Grad{typeof(iexp)}`. `iexp'(·)` returns input variables with updated gradient field. The gradient function is implemented reversibly so that the gradient field of output can be differentiated again to obtain Hessians as shown in Fig. 4.

The back propagation approach can be more efficient in obtaining higher order gradients.

```
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 by wrapping variables with type `BeijingRing` [32]. 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 $H_{i\leq n,n}$ and $H_{n,i<n}$. We didn't use the symmetry relation $H_{i,j} = H_{j,i}$ to save memory here. To simplify the implementation of backward rules described in the right most panel of Fig. 2. The final result can be collected by calling a global function `collect_hessian`.

C. QR decomposition

Let's consider a slightly non-trivil function, the QR decomposition

```
@i function iqr(Q, R, A::AbstractMatrix{T}) where T
    @anc anc_norm = zero(T)
    @anc anc_dot = zeros(T, size(A,2))
    @anc ri = zeros(T, size(A,1))
    for col = 1:size(A, 1)
        ri .+= identity.(A[:,col])
        for precol = 1:col-1
            idot(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
        inorm2(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

        ~(ri .+= identity.(A[:,col]);
        for precol = 1:col-1
            idot(anc_dot[precol], Q[:,precol], ri)
            for row = 1:size(Q,1)
                ri[row] -= anc_dot[precol] *
                    Q[row, precol]
            end
        end;
        inorm2(anc_norm, ri)
        )
    end
end
```

This implementatin of QR decomposition is very naive that does not consider reorthogonalization. `idot` and `inorm2` are functions to compute dot product and vector norm. They are implemented as

```
@i function idot(out, v1::AbstractVector{T}, v2)
        where T
    @anc anc1 = zero(T)
    for i = 1:length(v1)
        anc1 += identity(v1[i])
        CONJ(anc1)
        out += v1[i]*v2[i]
        CONJ(anc1)
        anc1 -= identity(v1[i])
    end
end

@i function inorm2(out, vec::AbstractVector{T})
        where T
    @anc anc1 = zero(T)
    for i = 1:length(vec)
        anc1 += identity(vec[i])
        CONJ(anc1)
        out += anc1*vec[i]
        CONJ(anc1)
        anc1 -= identity(vec[i])
    end
end
```

One can easily check the gradient of this naive implementation of QR decomposition is correct

```
using Test
A = randn(4,4)
q = zero(A)
r = zero(A)

@i function test1(out, q, r, A)
    iqr(q, r, A)
    out += identity(q[1,2])
end

@i function test2(out, q, r, A)
    iqr(q, r, A)
    out += identity(r[1,2])
end

@test check_grad(test1, (Loss(0.0), q, r, A);
        atol=0.05, verbose=true)
@test check_grad(test2, (Loss(0.0), q, r, A);
        atol=0.05, verbose=true)
```

Here, the `check_grad` function is a gradient checker function defined in module `NiLangCore.ADCore`.

D. Unitary Matrices

Unitary matrices can be used to ease the gradient exploding and vanishing problem in recurrent networks [33–35]. One of the simplest way to parametrize a unitary matrix is representing a unitary matrix as a product of two-level unitary operations [35]. A real unitary matrix of size N can be parametrized compactly by $N(N-1)/2$ rotation operations [36]

$$\mathrm{ROT}(a!, b!, \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} a! \\ b! \end{bmatrix}, \tag{8}$$

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

```
@i function umm!(x, )
    @anc k = 0
    @anc Nin = size(x, 2)
    @anc Nout = size(x, 1)
    for j=1:Nout
        for i=Nin-1:-1:j
            k += identity(1)
            ROT(x[i], x[i+1], [k])
        end
    end

    # uncompute k
    for j=1:Nout
        for i=Nin-1:-1:j
            k -= identity(1)
        end
    end
end
```

We bind the adjoint function of `ROT` to its reverse `IROT` by defining a new function that dispatch to `GVar`

```
@i function IROT(a!::GVar, b!::GVar, θ::GVar)
    IROT(value(a!), value(b!), value(θ))
    NEG(value(θ))
    value(θ) -= identity(/2)
    ROT(grad(a!), grad(b!), value(θ))
    grad(θ) += value(a!) * grad(a!)
    grad(θ) += value(b!) * grad(b!)
    value(θ) += identity(/2)
    NEG(value(θ))
    ROT(grad(a!), grad(b!), /2)
end
```

VI. DISCUSSION AND OUTLOOK

In this paper, we introduce a Julia eDSL NiLang that simulates a reversible Turing machine (RTM). We developed an instruction level automatic differentiation tool on this eDSL for differential programing. It can differentiate over any program to any order reliablely and efficiently without sophisticated designs to memorize computational graph and intermediate states. Also, we introduce a new training strategy that does not rely on gradients, learn by consistency.

In the following, we discussed the paractical side of writting reversible programs, and several future directions to go.

A. Time Space Tradeoff

In history, there has been many other designs of reversible languages and instruction sets. One of the main reason why RTM is not so popular is it may have either a space overhead propotional to computing time T or a computational overhead that sometimes can be even exponential. In the simplest g-segment trade off scheme [37, 38],

$$Time(T) = \frac{T^{1+\epsilon}}{S^{\epsilon}} \tag{9}$$

$$Space(T) = \epsilon 2^{1/\epsilon}(S + S \log \frac{T}{S}) \tag{10}$$

with T and S the time and space usage on a irreversible Turing machine, ϵ is the control parameter. It is related to the g-segment trade off parameters by $g = k^n, \epsilon = \log_k(2k-1)$ with $n \geq 1$ and $k \geq 1$. This section, we try to convince the readers that the overhead of reversible computing is not as terrible as people thought.

First, at $\epsilon \to 0$, the resource used by a RTM is same as the caching stategy used in a traditional machine learning package that memorizing every inputs of primitives. Memorizing inputs always make a program reversible since it does not discard any information. For deep neural networks, people used checkpointing trick to trade time with space [39], which is also a widely used trick in reversible programming [15]. RTM just provides more alternatives to trade time and space.

Second, some computational overhead of running recursive algorithms with limited space resources can be mitigated by "pseudo uncomputing" without sacrifising reversibility like in the `iexp` example. With reversible floating point $*=$ and $/=$ operations [40], many primitives can be implemented with pure reversible functions, which may significant decrease the computation time and memory usage. We will review this point in Sec. VI B.

Third, making reversible programming an eDSL rather than a independant 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. As a reversible eDSL, we have the flexibility to deallocate the memory irreversibly, i.e. trade energy with time. To quantify the overhead of uncomputing, we introducing the concept

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

$$\log_2 \frac{Time(T)}{T} \tag{11}$$

Whenever the program uncomputes the memory, the program granularity increases by approximately one. In instruction design, defining primitive functions like `iexp`, and deallocating gradients used for training, we need uncomputing ancilla bits and increase the granularity. The overhead increase exponentially as the granuality increase. The granularity can be decreased by cleverer compilation of a program, since the uncomputing of ancillas can be executed at any level of abstraction.

B. Instructions and Hardwares

So far, our eDSL is not really compiled to instructions, instead, it runs on a irreversible host Julia. In the future, it can be compiled to low level instructions and is execute on a reversible devices. For example, the control flow defined in this NiLang can be compiled to reversible instructions like conditioned `goto` instruction. The reversibility requires the target instruction a `comefrom` instruction that specifing the postcondition. [41]

Arithmetic instructions should be redesigned to support better reversible programs. The major obstacle to exact reversibility programming is current floating point adders used in our computing devices are not exactly reversible. There are proposals of reversible floating point adders and multipliers [40, 42–44] that introduces garbage bits to ensure reversibility. With these infrastructure, a reversibile program can be executed without suffereing from the irreversibility from rounding error and reduce the computational and memory overhead significantly. Notabley, in machine learning field, information buffer is used to make multiplication operations [12] reversible and reduce memory cost.

Reversible programming is not nessesarily related to reversible hardwares. Reversible programs are a subset of irreversible programs, hence can be simulated efficiently on CMOS devices [41]. By Using reversible hardwares [], the computation may costs zero energy by Landauer's principle [21]. Reversible hardwares are not nessesarily related to reversible gates such as Toffoli gate and Fredkin gate. Devices with the ability of recovering signal energy is able able to save energy, which is known as generalized reversible computing. [45, 46] In the following, we comment briefly on a special type of reversible device Quantum computer.

1. *Quantum Computers*

One of the fundamental difficulty of building a quantum computer is, unlike a classical state, an unknown quantum state can not be copied. Quantum random access memory [47] is very hard to design and implement, which is known to have many caveats [48]. A quantum state in a environment will decoherence and can not be recovered, this underlines the simulation nature of quantum devices. Reversible computing does not enjoy the quantum advantage, nor the quantum disadvantages of non-cloning and decoherence. The reversibility of quantum computing comes from the fact that microscopic processes are unitary. On the other side, the irreversibility is rare, it can come from interacting with classical devices. Irreversible processes include decaying, qubit state resetting, measurements and classical feedbacks to quantum devices. These are typically harder to implement on a quantum device.

Given the fundamental limitations of quantum decoherence and non-cloning and the reversible nature of microscopic world. It is reasonable to have a reversible computing device to bridge the gap between classical and universal quantum computing. By introducing entanglement little by little, we can accelerate some basic components. For example, quantum

Fourier transformation provides an interesting alternative to the adders and multipliers by introducing one additional CPHASE gate even though it is a classical-in classical-out algorithm. [49] By introducing qantum rotation gates $R_y(\theta)$ and $R_z(\theta)$ in the reversible programming, we make NiLang a path integral based universal quantum simulator as shown in Appendix **??**. The compiling theory developed for reversible programming will have profounding effect to quantum computing.

C. Outlook

So far NiLang is not full ready for productivility. It can be improved from multiple perspectives. We call for better compiling that decreases granularity and hence reduces overhead, rigoriously reversible floating point arithematics to let the reversibility free from rounding error. It is also interesting to see how it can be combined with a high performance quantum simulator like Yao [50]. It can provide control flow to Yao's quantum block intermediate representation, while Yao can provide the quantum features for NiLang. Reversible programming is known to have advantage in parallel computingi in handle asynchronious computing [51] and debugging with bidirectional move [52]. It is interesting to see how NiLang combines with other parts of Julia ecosystem like CUDAnative [53] and Debugger.

VII. 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. 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 and the research funding from Huawei Technologies under the Grant No. YBN2018095185.

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

[2] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," (2016), arXiv:1607.07892 [cs.MS].

[3] 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).

[4] 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].

[5] 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.

[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] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," (2015), arXiv:1506.00019 [cs.LG].

[8] 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.

[9] "Breaking the Memory Wall: The AI Bottleneck," `https://blog.semi.org/semi-news/breaking-the-memory-wall-the-ai-bottleneck`.

[10] J. Bettencourt, M. J. Johnson, and D. Duvenaud, (2019).

[11] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM review **59**, 65 (2017).

[12] 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.

[13] 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.

[14] J. Behrmann, D. Duvenaud, and J. Jacobsen, CoRR **abs/1811.00995** (2018), arXiv:1811.00995.

[15] K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).

[16] M. P. Frank, IEEE Spectrum **54**, 3237 (2017).

[17] C. Lutz, "Janus: a time-reversible language," (1986), *Letter to R. Landauer*.

[18] M. P. Frank, *The R programming language and compiler*, Tech. Rep. (MIT Reversible Computing Project Memo, 1997).

[19] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, Journal of Logical and Algebraic Methods in Programming **100**, 7197 (2018).

[20] T. Haulund, "Design and implementation of a reversible object-oriented programming language," (2017), arXiv:1707.07845 [cs.PL].

[21] R. Landauer, IBM journal of research and development **5**, 183 (1961).

[22] M. Giftthaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, Advanced Robotics **31**, 12251237 (2017).

[23] D. N. Laikov, Theoretical Chemistry Accounts **137** (2018), 10.1007/s00214-018-2344-7.

[24] M. Seeger, A. Hetzel, Z. Dai, E. Meissner, and N. D. Lawrence, "Auto-differentiating linear algebra," (2017), arXiv:1710.08717 [cs.MS].

[25] Z.-Q. Wan and S.-X. Zhang, "Automatic differentiation for complex valued svd," (2019), arXiv:1909.02659 [math.NA].
[26] C. Hubig, "Use and implementation of autodifferentiation in tensor network methods with complex scalars," (2019), arXiv:1907.13422 [cond-mat.str-el].
[27] J.-G. L. Hao Xie and L. Wang, arXiv:2001.04121.
[28] H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, Physical Review X **9** (2019), 10.1103/physrevx.9.031041.
[29] R. Orús, Annals of Physics **349**, 117158 (2014).
[30] "Paper's Github Repository," `https://github.com/GiggleLiu/nilangpaper/tree/master/codes`.
[31] J. Martens, I. Sutskever, and K. Swersky, "Estimating the hessian by back-propagating curvature," (2012), arXiv:1206.6464 [cs.LG].
[32] When people ask the location in Beijing, they will start by asking which ring.
[33] M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464.
[34] 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].
[35] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231.
[36] C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).
[37] C. H. Bennett, SIAM Journal on Computing **18**, 766 (1989), https://doi.org/10.1137/0218053.
[38] R. Y. Levine and A. T. Sherman, SIAM Journal on Computing **19**, 673 (1990).
[39] T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174.
[40] T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, "Quantum circuits for floating-point arithmetic," (2018), arXiv:1807.02023 [quant-ph].
[41] C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.
[42] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on Nanotechnology* (2010) pp. 233–237.
[43] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on Nanotechnology* (2011) pp. 451–456.
[44] 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].
[45] M. P. Frank, in *35th International Symposium on Multiple-Valued Logic (ISMVL'05)* (2005) pp. 168–185.
[46] M. P. Frank, in *Reversible Computation*, edited by I. Phillips and H. Rahaman (Springer International Publishing, Cham, 2017) pp. 19–34.
[47] V. Giovannetti, S. Lloyd, and L. Maccone, Physical Review Letters **100** (2008), 10.1103/physrevlett.100.160501.
[48] S. Aaronson, Nature Physics **11**, 291 (2015).
[49] L. Ruiz-Perez and J. C. Garcia-Escartin, Quantum Information Processing **16** (2017), 10.1007/s11128-017-1603-1.
[50] 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].
[51] D. R. Jefferson, ACM Transactions on Programming Languages and Systems (TOPLAS) **7**, 404 (1985).
[52] B. Boothe, ACM SIGPLAN Notices **35**, 299 (2000).
[53] T. Besard, C. Foket, and B. D. Sutter, CoRR **abs/1712.03112** (2017), arXiv:1712.03112.

Appendix A: NiLang Grammar

Terminologies

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

$$
\begin{aligned}
\langle \text{Stmts} \rangle &::= \epsilon \\
&\mid \langle \text{Stmt} \rangle \\
&\mid \langle \text{Stmts} \rangle \ \langle \text{Stmt} \rangle \\
\langle \text{Stmt} \rangle &::= \langle \text{BlockStmt} \rangle \\
&\mid \langle \text{IfStmt} \rangle \\
&\mid \langle \text{WhileStmt} \rangle \\
&\mid \langle \text{ForStmt} \rangle \\
&\mid \langle \text{InstrStmt} \rangle \\
&\mid \langle \text{RevStmt} \rangle \\
&\mid \langle \text{@anc} \rangle \ \langle \text{Stmt} \rangle \\
&\mid \langle \text{@routine} \rangle \ \langle \text{Stmt} \rangle \\
&\mid \langle \text{@safe} \rangle \ \textit{JuliaExpr} \\
&\mid \langle \text{CallStmt} \rangle \\
\langle \text{BlockStmt} \rangle &::= \textit{begin} \ \langle \text{Stmts} \rangle \ \textit{end} \\
\langle \text{RevCond} \rangle &::= \textit{(JuliaExpr , JuliaExpr)} \\
\langle \text{IfStmt} \rangle &::= \textit{if} \ \langle \text{RevCond} \rangle \ \langle \text{Stmts} \rangle \ [\textit{else} \ \langle \text{Stmts} \rangle] \ \textit{end} \\
\langle \text{WhileStmt} \rangle &::= \textit{while} \ \langle \text{RevCond} \rangle \ \langle \text{Stmts} \rangle \ \textit{end} \\
\langle \text{Range} \rangle &::= \textit{JuliaExpr : JuliaExpr} \ [\textit{: JuliaExpr}] \\
\langle \text{ForStmt} \rangle &::= \textit{for ident} = \langle \text{Range} \rangle \ \langle \text{Stmts} \rangle \ \textit{end} \\
\langle \text{CallStmt} \rangle &::= \textit{JuliaExpr (} \ [\langle \text{DataViews} \rangle] \ \textit{)} \\
\langle \text{Constant} \rangle &::= \textit{num} \mid \pi \\
\langle \text{InstrBinOp} \rangle &::= \mathtt{+=} \mid \mathtt{-=} \mid \veebar= \\
\langle \text{InstrTrailer} \rangle &::= [.] \ \textit{(} \ [\langle \text{DataViews} \rangle] \ \textit{)} \\
\langle \text{InstrStmt} \rangle &::= \langle \text{DataView} \rangle \ \langle \text{InstrBinOp} \rangle \ \textit{ident} \ [\langle \text{InstrTrailer} \rangle] \\
\langle \text{RevStmt} \rangle &::= \sim \langle \text{Stmt} \rangle \\
\langle \text{@routine} \rangle &::= \textit{@routine ident} \ \langle \text{Stmt} \rangle \\
\langle \text{AncArg} \rangle &::= \textit{ident} = \textit{JuliaExpr} \\
\langle \text{@anc} \rangle &::= \textit{@anc} \ \langle \text{AncArg} \rangle \\
&\mid \textit{@deanc} \ \langle \text{AncArg} \rangle \\
\langle \text{@safe} \rangle &::= \textit{@safe JuliaExpr} \\
\langle \text{DataViews} \rangle &::= \epsilon \\
&\mid \langle \text{DataView} \rangle \\
&\mid \langle \text{DataViews} \rangle \ \textit{,} \ \langle \text{DataView} \rangle \\
\langle \text{DataView} \rangle &::= \langle \text{DataView} \rangle \ \textit{[JuliaExpr]} \\
&\mid \langle \text{DataView} \rangle \ \textit{. ident} \\
&\mid \textit{JuliaExpr (} \ \langle \text{DataView} \rangle \ \textit{)} \\
&\mid \langle \text{DataView} \rangle \ ' \\
&\mid \text{-} \ \langle \text{DataView} \rangle \\
&\mid \langle \text{Constant} \rangle \\
&\mid \textit{ident} \\
&\mid \textit{ident} \ ...
\end{aligned}
$$

One can use `@i function` $\langle Stmts \rangle$ `end` to define a function and its inverse. All *JuliaExpr* is should be pure, otherwise the reversibility is not gurantted.

Dataview is a bijective mapping of an object or a field (or item) of an object. When modifying the dataview of an object, it changes the object directly with the `chfield` method.

Appendix B: Instruction Table

The translation of instructions to Julia functions

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

Table II. Instructions and their interpretation in NiLang.

The list of instructions implemented in NiLang

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

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