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 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. [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 been

* cacate0129@iphy.ac.cn

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

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 reversible instruction or a function call in our eDSL NiLang changes inputs "inplace".

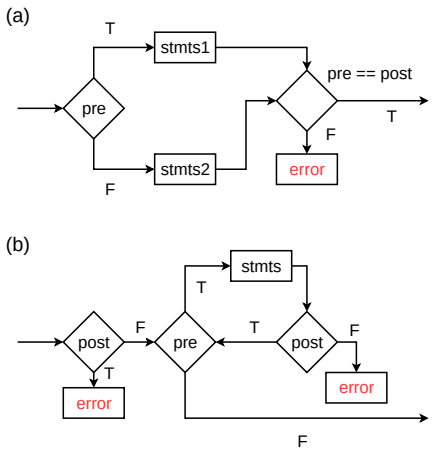

Figure 1. Flow chart for reversible (a) `if` statement and (b) `while` statement. "stmts", "stmts1" and "stmts2" are statements, statements in true branch and statements in false branch respectively. "pre" and "post" are precondition and postconditions respectively. "error" refers to `InvertibilityError`.

NiLang is a reversible eDSL in Julia that simulates reversible Turing machine. The grammar is shown in Appendix A. Its main feature is contained in a single macro `@i`. It interprets a function to a finite set of "instructions" written in julia language. 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. To design such a eDSL, we first introduce a reversible IR that used in NiLang.

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 statement can be defined easily as shown in in Table I. An instruction `y! += f(args...)` is interpreted as a julia function call `PlusEq(f)(y!, args...)` or `⊕(f)(y!, args...)` as a shorthand. Here, the "!" after a variable is used as a convension to indicate that a variable is changed after the call of an instruction or function. The detailed specification of instructions is listed in Appendix B. The function call is same as the host language, except every function `f` has a `~f` that binded to an object of type `Inv{typeof(f)}`. `~f` invokes the compiled inverse functions of `f`.

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. A postcondition is a boolean expression that being evaluated after the body expressions being executed. For the `if` statement as shown in Fig. 1 (a), 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

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.

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 reverse of `for` statement is straightforward. The program first stores the loop informations, start, step and stop, after the execusion of the loop, the program checks the values of these variables 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>` (or restored), 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 stages

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> using NiLangCore, MacroTools

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

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

julia> ex2 = NiLangCore.dual_ex(ex1);

julia> ex2 |> prettify  # after stage 2
:(if (x > 3, x > 3)
      grad((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 grad((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. 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, 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

The constructor of a type is also a reversible function. The inverse function is a "destructor", which does not deallocate memory directly but unpacks data.

```
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
```

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`. Finally call the default constructor `DVar(xx, gg)`. It is easy to find the inverse procedure that transform a `DVarT` instance to a `T` instance. With the flexibility to operate 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

```
grad(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 "inplace".

As we have mensioned, `grad(arr[3].value) += x * y` is translated to `@instr grad(arr[3].value) += x * y` at the thrid stage. To execute the instruction, `@instr` translate the statement to

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

`PlusEq(*)(grad(arr[3].value), x, y)` computes the output, which is a tuple of length 3. `chfield` is used to

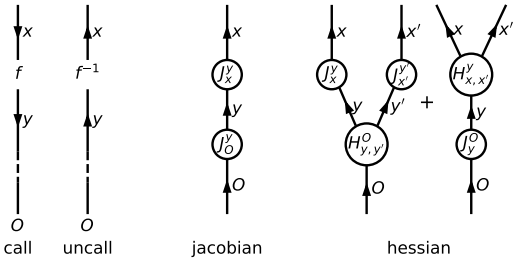

Figure 2. Adjoint rules for Jacobians and Hessians in tensor network language.

a dataview. The second and third arguments can be assigned back directly. 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. If the default constructor of a type is not overwritten by user, NiLang can modify a field of that type automatically. For a bijective mapping of a field, user need to specify the behavior of a dataview by overloading `chfield` function.

III. TAYLOR PROPAGATION ON A REVERSIBLE TURING MACHINE

Taylor propagation is exponentially (as the order) 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 manually defined. The exhaused support requires much more effort than the first order gradient propagation. Instruction level automatic differentiation is more flexible in obtaining higher order gradients like Hessian.

A. First order gradient

Given a node $\vec{y} = f(\vec{x})$ in a computational graph, we can propagate the Jacobians in tangent mode like

$$J^O_{x_i} = J^O_{y_j} J^{y_j}_{x_i} \tag{1}$$

and the adjoint mode

$$J^{y_j}_I = J^{y_j}_{x_i} J^{x_i}_I \tag{2}$$

Here, I is the inputs and 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, here we focus on the adjoint mode. The backward rule can be described in tensor network [] language as shown in Fig. 2.

In reversible programming, we have the following implementation

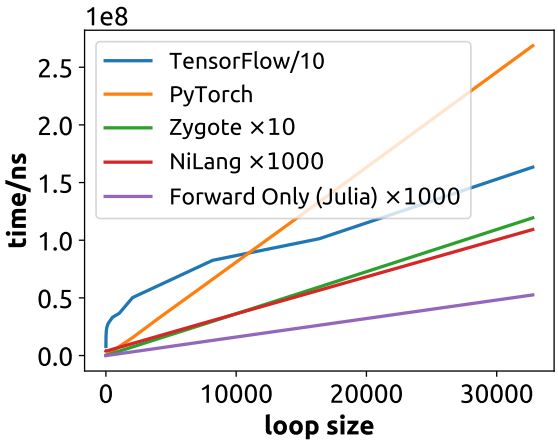

Figure 3. The time for obtaining gradient as function of loop size. $\times n$ in lengend represents a rescaling of time.

Algorithm 1: Reversible programming AD

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

Here, "." is the broadcast operations in Julia. `GVar` is a "reversible type". `GVar(x)` attaches a zero gradient field to a variable, which is similar to the dual number in tangent mode automatic differentiation. The gradient can be accessed by `grad` function. Its inverse ~`GVar` deallocates the gradient field safely and returns its value field. Here, "safely" means it will check the gradient field to make sure it is in 0 state. When a `instruct` meets a `GVar`, besides computing its value field `value`$(\vec{x}) = f^{-1}$(`value`$(\vec{y})$), it also updates the gradient field `grad`$(\vec{x}) = J^{\vec{x}}_{\vec{y}}$`grad`$(\vec{y})$. Since f^{-1} is bijective, $J^{\vec{y}}_{\vec{x}}$ can be easily obtained by inspecting its inverse function f. The final output is stored in the gradient field, when then gradient is not used anymore, the faithful reversible programming style to compute gradients would be uncomputing the whole process to obtain gradient, which increases the hyrachy by 1. Whenever the hyrachy increase by 1, the computational overhead doubles comparing with its irreversible counter part.

Due to the multiple dispatch and just in time compiling, most runtime overhead can be removed. 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
```

From the benchmark result shown in Fig. 3. One can see that the NiLang implementation is unreasonablely fast, even faster than two times the forward pass written in Julia. The code for benchmark is open for review in our paper github repository [29]. This benchmark does show the actual performance in real applications. 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

The second order gradient can also be back propagated

$$\begin{aligned} H^f_{y_L,y'_L} &= 0 \\ H^f_{y_{i-1},y'_{i-1}} &= J^{y_i}_{y_{i-1}} H^f_{y_i,y'_i} J^{y'_i}_{y'_{i-1}} + J^f_{y_i} H^{y_i}_{y_{i-1},y'_{i-1}} \end{aligned} \tag{3}$$

In tensor network language, it can be represented as in Fig. 2. This approach can be easily extended to higher orders, or taylor propagation. However, this is not the widely adopted approach to compute higher order gradients. Althought backpropagating higher order gradients directly is exponentially faster than back propagating the computational graph for computing lower order gradients for computing higher order gradients, one has to extending the backward rules for each primitive rather than reusing existing ones. Here, we emphasis that with instruction level AD, rewritting backward rules for primitives turns out to be not so difficult. An example is provided in Sec. V B.

C. Gradient on ancilla problem

Ancilla can also carry gradients during computation, sometimes these gradients can not be uncomputed even if their parents can be uncomputed regoriously. 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. does not have effect on rest parts of program.

Ancilla is the fixed point of a function, which means

$$\begin{aligned} &b, y \leftarrow f(x, a) \text{ , where } b == a \\ &\frac{\partial b}{\partial x} = 0 \end{aligned} \tag{4}$$

During the computation, the gradient field does not have any effect to the `value` field of variables. The key question is will the loss of gradient part in ancilla affect the reversibility of the gradient part of argument variables. The gradient of argument variable is defined as $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} + \frac{\partial L}{\partial b}\frac{\partial b}{\partial x}$, where the second term vanish naturally.

D. Implementation

The automatic differentiation engine 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
```

The program first checks the input parameters and locate the loss variable. Then `Loss` unwraps the loss variable, the location of loss variable is transferred to the ancilla `iloss` of integer type.

`GVar`

```
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
```

Broadcasting is supported. To avoid possible confusing, tuple indexing is forbidden delebrately, one can use `tget(tuple, 2)` to get the second element of a tuple.

IV. LEARN BY CONSISTENCY

Consider training data consist 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 $\mathcal{L} = \text{dist}(\vec{y}^*, f(\vec{x}^*, \vec{p}_x))$ and minimize it with gradient $\frac{\partial L}{\partial \vec{p}_x}$. 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 $\vec{y} \napprox \vec{y}^*$ **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, $\texttt{garbage}(\cdot)$ is a function for taking the garbage space. This algorithm utilizes the self-consistency relation

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

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(out!, x, y!)
    y! += identity(x)
    out! -= exp(x)
    out! += exp(y!)
end

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

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

Functions `f1` and `f2` computes $f(x, y) = e^{(y+x)} - e^x$ and stores the output in a new memory `out!`. The only difference is f_2 "uncompute" x arithmetically. The task of training is to

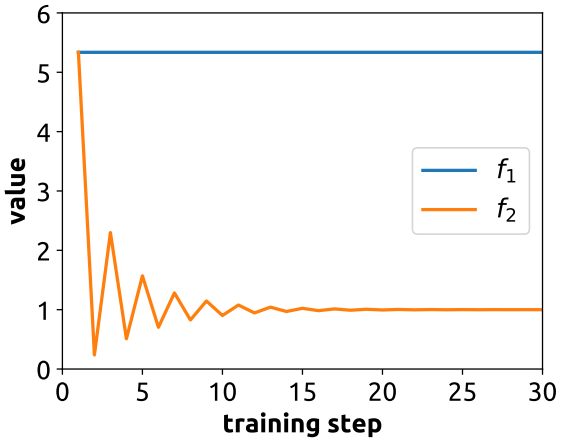

Figure 4. The value of x as a function of self-consistent training step.

find a y that make the output value equal to target value 1. After 200 steps, f_2 runs into the fixed point with x equal to 1 upto machine precision. However, f_1 does not do any training. The training of f_2 fails because this function actually computes $f_1(out!, x, y!) = out! + e^{(y!+x)} - e^x, x, x + y!$, where the training parameter y is completely determined by the garbage space on output side x and $x+y!$. As a result, shifting $out!$ directly is the only approach to satisfy the consistency relation. On the other side, $f_2(out!, x, y!) = out! + e^{(y!+x)} - e^x, (x - \log(e^x)), x + y!$, parameters y is not completely determined by the garbage space $(x - \log(e^x)), x + y!$.

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.*

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

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

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

```
@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
```

To compute the first Fibonacci number that is greater or equal to 100

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

Here, `fib` and `rfib` are defined in Appendix V A.

B. exp function

An exp function can be computed using Taylor expansion

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

This is a recursive algorithm that mimics pebble game. Define the term for accumulation $s_n \equiv \frac{x^n}{\text{factorial}(n)}$, 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 $s_{n-1} = \frac{n s_n}{x}$ to deallocate memory. By allowing loss of several digit precision of result, implementing the constant memory reversible exp function is possible

```
using NiLang, NiLang.AD

@i function iexp(out!, 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)

    out! += 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

    out! += 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`, this uncomputation is only true arithmeticly, but not for floating point number due to the rouding error. 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> out!, x = 0.0, 1.6
(0.0, 1.6)

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

julia> grad(x)
4.9530324244260555

julia> out!, x = 0.0, 1.6
(0.0, 1.6)

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

`iexp'` returns an object of type `Grad{typeof(exp)}`, it returns input variables with updated gradient field. The loss variable is specified by a wrapper `Loss`, notice we don't distinguish input and output in reversible programming. The gradient functions are implemented reversibally so that the gradients can be differentiated again to obtain Hessians as shown in Fig. 5.

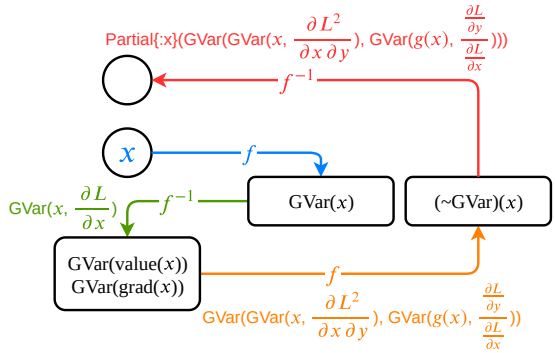

Figure 5. Obtaining the second order gradient with the reversive differentiation approach. Colors blue, green, orange and red indicate the four stages, data types and fields used for computation are marked in the same color.

Their are four stages in computing Hessian in the naive approach, the first two stages are obtaining gradients, the thrid stage is wrapping each field in `GVar` with `GVar` and compute forward function. Then we pick a variable y to compute a row of Hessian by adding 1 to `grad(`y`)`. At the final stage, the `~GVar` operation does not unwrap `GVar` directly because the gradient field of gradients may be non-zeros. Instead, we use `Partial{:FIELD}(·)` to savely invert `GVar` on data type `GVar{<:GVar, <:GVar}`. It take a field without discarding information. In NiLang, `simple_hessian` obtains the Hessian matrix in this way by iterating over different ys.

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

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

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

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

`iexp''` computes the second order derivative by wrapping the variable into type `BeijingRing`. This is a special data structure to store Hessians. Whenever an n-th variable is created, we push a ring of size $2n-1$ in 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 $H_{i,j} = H_{j,i}$ 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

Not only simple functions, linear algebra functions

```
@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
```

where `idot` and `inorm2` 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 implemen-

tation 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 `NiLangCore.ADCore`.

D. Unitary Matrices

Recurrent networks with a unitary parametrized network ease the gradient exploding and vanishing problem [30–32]. One of the simplest way to parametrize a unitary matrix is representing a unitary matrix as a product of two-level unitary operations [32]. A unitary matrix of size $N \times N$ can be parametrized by $k = N(N-1)/2$ two level unitary matrices [33]. A two-level unitary matrices can be applied in $O(1)$ time. A real unitary matrix can be parametrized compactly by k rotation operations `ROT(a!, b!, θ)`, where θ is the rotation angle, `a!` and `b!` are target registers.

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

```
@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
```

Its backward rule of a `ROT` instruction is

$$\begin{aligned} \bar{\theta} &= \sum \frac{\partial R(\theta)}{\partial \theta} \odot (\bar{y} x^T) \\ &= \text{Tr}\left[\frac{\partial R(\theta)}{\partial \theta}^T \bar{y} x^T \right] \\ &= \text{Tr}\left[R\left(\frac{\pi}{2} - \theta\right) \bar{y} x^T \right] \end{aligned} \tag{9}$$

We bind the adjoint function of `ROT` to its reverse `IROT`, and define a new function that dispatch to `GVar` variables

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

VI. DISCUSSION AND OUTLOOK

In this paper, we introduce differential programing on a reversible Turing machine implemented as a Julia eDSL. It is able to differentiate over any program consisting of instructions and control flows to any order reliablely without sophisticated design of computational graph and intermediate state caching. Besides automatic differentiation, we introduce a new training strategy learn by consistency that does not rely on gradients.

In the following, we discussed the paractical side of writting reversible programs, and several future directions to go. Notablely, we introduce the concept of “arithematic

uncomputing" to reduce the overhead of recursive reversible algorithms.

A. Time Space Tradeoff

So far, we have introduced the eDSL. There are many other designs of reversible language and instruction set. The reversible Turing machine may have either a space overhead propotional to computing time T or a computational overhead that sometimes can be even exponential given limited space. In the simplest g-segment trade off scheme [34, 35], it takes $Time(T) = T^{\log_g(4g-2)}$ and $Space(T) = (g-1)S\log_g T$. This section, we try to convince the reader that the overhead of reversible computing is not as terrible as people thought.

First, one should notice that even in the worst case, the overhead of a reversible program is not more than a traditional machine learning package. In pytorch, a tensor memorize every input of a primitive. The program is apparently reversible since it does not discard any information. For deep neural networks, people used checkpointing trick to trade time with space [36], which is also a widely used trick in reversible programming [15]. Reversible programming AD is sometimes more memory efficient. Comparing with logging computational graph, reversible programming has the advantage of supporting inplace functions, which is difficult is both tranditional and source to source AD framework. The example of parametrizing unitary matrice costs zero memory allocation.

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.

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. In this eDSL, we can just deallocate the memory irreversibly, i.e. trade energy with time. This underlines the fact that a reversible program is more suited in a program with small granularity. We can quantify it by introducing

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

$$\log_2 \frac{Time(T)}{T} \tag{10}$$

In the lowest granuality, instruction design, we need ancilla bits. Defining primitive functions like `iexp` requires uncomputing ancillas. Deallocating the gradient further increase the granularity by ~ 1. After the training, the inverse training of whole programs should be done to deallocate all the memory used for training. As a result, the program complexity increase exponentially as the granuality increase. The granularity can be decreased by flattening the functions, since the uncomputing of ancillas can be executed at any level of granularity.

One should notice the memory advantage of reversible programming to machine learning does comes from reversibility itself, but from a better data tracking strategy inspired from invertible programming. Normally, a reversible program is not as memory efficient as its irreversible couterpart due to the additional requirement of no information loss. A naive approach that keeping track of all informations will cost an additional space $O(T)$, where T stands for the excution time in a irreversible TM, the longer the program runs, the larger the memory usage is. This is exactly the approach to keeping reversibility in most machine learning packages in the market. The point it, an reversible Turing Machine is able to trade space with time. In some cases, it may cause polynomial overhead than its irreversible counterpart.

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 executable on a reversible devices. For example, the control flow defined in this NiLang can be compiled to reversible instructions like conditioned `goto` statement. It is designed in such a way that the target instruction is a `comefrom` statement which specifies the postcondition. [37]

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 [38, 39] that introduces garbage bits to ensure reversibility. In other words, to represent a 64 bit floating point number requires more than 64 bits in storage. Reversible multiplier is also possible in similar approach. [40] With these infrastructure, a reversibile program can be executed without suffereing from the irreversibility from rounding error. In machine learning field, people using information buffer in multiplication operations [12] in an approach to enforce invertibility in a memory efficient way.

Reversible programming is not nessesarily related to reversible hardwares, reversible programs is a subset of irreversible programs hence can be simulated efficiently with CMOS technologies [37]. However, only using reversible hardwares [] can break the energy efficiency barrier by Landauer principle. 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. [41, 42] In the following, we comment briefly on a special type of reversible device Quantum computer.

C. 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 [43]

is very hard to design and implement, it is known to have many caveats [44]. 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 from entanglement, nor the quantum disadvantages from non-cloning and decoherence. Only the limitation of reversibility is retained, reversibility comes from the fact that micro scopic processes are all unitary. In microscopic world, irreversibility is rare, it can come from the interaction with classical devices, like environment induces 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 microscopic reversible nature. 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 like a reversible adder. The quantum fourier transformation provides a shortcut to the adder bt introducing only one additional gate, the CPHASE gate even though it is a classical in classical out algorithm. [] Interestingly, by introducing rotation gates $Ry(\theta)$ and $Rz(\theta)$ 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 computers.

D. Outlook

So far NiLang is not full ready for productivility. It can be improved from multiple perspectives, compiling support to merge the uncomputing can decrease granularity and hence reduce overhead. Additional instructions like stack operations and logical operations are under consideration. It is also interesting to see how it can be combined with a high performance quantum simulator like `Yao`. It is able to provide control flow to `Yao`'s QBIR. By porting a quantum simulator. it is interesting to see how quantum simulator can improve the instruction design. Notice a quantum fourier transformation (QFT) based quantum adder and multiplier is sometimes more efficient than a classical adder [45] [**JG: Is this true?**]. Reversible programming is known to have the advantage in parallel computing [46] and debugging [47], it is interesting to see how it combines with other parts of Julia packages like CUDAnative [48] and Debugger, it would be interesting to see our eDSL running on a GPU with little synchronization overhead [**JG: Is this even possible?**] and using an interative debugging with bidirectional move. This could be used to reduce the momory cost in normalizing flow, time reversible integrator, recurrent neural network and residual neural network.

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] "Paper's Github Repository," `https://github.com/GiggleLiu/nilangpaper/tree/master/codes`.

[30] M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464.

[31] 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].

[32] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231.

[33] C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).

[34] C. H. Bennett, SIAM Journal on Computing **18**, 766 (1989), https://doi.org/10.1137/0218053.

[35] R. Y. Levine and A. T. Sherman, SIAM Journal on Computing **19**, 673 (1990).

[36] T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174.

[37] C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.

[38] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on Nanotechnology* (2011) pp. 451–456.

[39] 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].

[40] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on Nanotechnology* (2010) pp. 233–237.

[41] M. P. Frank, in *35th International Symposium on Multiple-Valued Logic (ISMVL'05)* (2005) pp. 168–185.

[42] M. P. Frank, in *Reversible Computation*, edited by I. Phillips and H. Rahaman (Springer International Publishing, Cham, 2017) pp. 19–34.

[43] V. Giovannetti, S. Lloyd, and L. Maccone, Physical Review Letters **100** (2008), 10.1103/physrevlett.100.160501.

[44] S. Aaronson, Nature Physics **11**, 291 (2015).

[45] T. Haener, M. Soeken, M. Roetteler, and K. M. Svore, Lecture Notes in Computer Science , 162174 (2018).

[46] D. R. Jefferson, ACM Transactions on Programming Languages and Systems (TOPLAS) **7**, 404 (1985).

[47] B. Boothe, ACM SIGPLAN Notices **35**, 299 (2000).

[48] 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.

⟨Stmts⟩ ::= ϵ
| ⟨Stmt⟩
| ⟨Stmts⟩ ⟨Stmt⟩

⟨Stmt⟩ ::= ⟨BlockStmt⟩
| ⟨IfStmt⟩
| ⟨WhileStmt⟩
| ⟨ForStmt⟩
| ⟨InstrStmt⟩
| ⟨RevStmt⟩
| ⟨@anc⟩ ⟨Stmt⟩
| ⟨@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*

⟨CallStmt⟩ ::= *JuliaExpr (* [⟨DataViews⟩] *)*

⟨Constant⟩ ::= *num* | π

⟨InstrBinOp⟩ ::= += | -= | ⩒=

⟨InstrTrailer⟩ ::= [.] *(* [⟨DataViews⟩] *)*

⟨InstrStmt⟩ ::= ⟨DataView⟩ ⟨InstrBinOp⟩ *ident* [⟨InstrTrailer⟩]

⟨RevStmt⟩ ::= ~ ⟨Stmt⟩

⟨@routine⟩ ::= @*routine ident* ⟨Stmt⟩

⟨AncArg⟩ ::= *ident = JuliaExpr*

⟨@anc⟩ ::= @*anc* ⟨AncArg⟩
| @*deanc* ⟨AncArg⟩

⟨@safe⟩ ::= @*safe JuliaExpr*

⟨DataViews⟩ ::= ϵ
| ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩

⟨DataView⟩ ::= ⟨DataView⟩ *[JuliaExpr]*
| ⟨DataView⟩ . *ident*
| *JuliaExpr (* ⟨DataView⟩ *)*
| ⟨DataView⟩ ′
| - ⟨DataView⟩
| ⟨Constant⟩
| *ident*

One can use `@i function` ⟨*Stmts*⟩ `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 += $f(args...)$	$\oplus(f)(args...)$	`PlusEq`
y −= $f(args...)$	$\ominus(f)(args...)$	`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
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 += $a \wedge b$	$y + a^b, a, b$
y += exp(x)	$y + e^x, x$
y += log(x)	$y + \log x, x$
y += sin(x)	$y + \sin x, x$
y += cos(x)	$y + \cos x, x$
y += abs(x)	$y + \lvert x\rvert, x$
NEG(y)	$-y$
CONJ(y)	y'

Table III. A collection of reversible instructions.