Differentiate Everything with a Reversible Domain-Specific Language

Jin-Guo Liu
Institute of Physics, Chinese Academy of Sciences,
Beijing 100190, China
cacate0129@iphy.ac.cn

Taine Zhao
Department of Computer Science, University of Tsukuba

Abstract

Traditional machine instruction level reverse mode automatic differentiation (AD) faces the problem of having a space overhead that linear to time in order to trace back the computational state, which is also the source of bad time performance. In reversible programming, a program can be executed bi-directionally, which means we do not need extra design to trace back the computational state. This paper answers the question that how practical it is to implement a machine instruction level reverse mode AD in a reversible programming language. The answer is a strong possitive. We developed an open source reversible eDSL NiLang in Julia that can differentiate a general reversible program while being compatible with Julia's ecosystem. It empowers users the flexibility to tradeoff time, space, and energy rather than caching data into a global tape. With examples, we show how one can use reversible programming AD to obtain gradients and Hessians for a wide class of functions in scientific programming and machine learning, from elementary mathematical functions, sparse matrix operations to machine learning applications. Manageable memory allocation makes it a good tool to differentiate GPU kernels. Through benchmarks, we demonstrate that the AD implemented in a reversible programming language can achieve a state-of-the-art performance with negligible space overhead. Finally, we will discuss the challenges that we face towards rigorous reversible programming, mainly from the instruction and hardware perspective.

1 Introduction

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

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

where $x^0 \in R^m$, $x^L \in R^n$, L is the depth of computing. The Jacobian of this program is a $n \times m$ matrix $J_{ij} \equiv \frac{\partial x_i^L}{\partial x_j^0}$, where x_j^0 and x_i^L are single elements from inputs and outputs. Computing the Jacobian or part of the Jacobian automatically is what we called automatic differentiation (AD). It

Preprint. Under review.

can be classified into three classes, the forward mode AD, the backward mode AD and the mixed mode AD Hascoet and Pascual (2013). The forward mode AD computes the Jacobian matrix elements related to a single input using the chain rule $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}} \frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$ with j the column index, while a backward mode AD computes Jacobian matrix elements related to a single output using the chain rule in the reverse direction $\frac{\partial \mathbf{x}_i^L}{\partial x^{k-1}} = \frac{\partial \mathbf{x}_i^L}{\partial \mathbf{x}^k} \frac{\partial \mathbf{x}^k}{\partial x^{k-1}}$ with i the row index. In variational applications where the loss function always outputs a scalar, the backward mode AD is preferred. However, implementing backward mode AD is harder than implementing its forward mode counterpart, because it requires propagating the gradients in the inverse direction of computing the loss. The backpropagation of gradients requires

1. an approach to trace back the computational process,
2. caching variables required for computing gradients.

Most popular AD packages in the market implements the computational graph to solve above issues at the tensor level. In Pytorch Paszke *et al.* (2017) and Flux Innes *et al.* (2018), every variable has a tracker field. When applying a predefined primitive function on a variable, the variable's tracker field keeps track of this function as well as data needed in backpropagation. TensorFlow Abadi *et al.* (2015) also implements the computational graph, but it builds a static computational graph as a description of the program before actual computation happens. These frameworks sometimes fail to meet the diverse needs in research, for example, in physics research,

1. People need to differentiate over sparse matrix operations that are important for Hamiltonian engineering Hao Xie and Wang, like solving dominant eigenvalues and eigenvectors Golub and Van Loan (2012),
2. People need to backpropagate singular value decomposition (SVD) function and QR decomposition in tensor network algorithms to study the phase transition problem Golub and Van Loan (2012); Liao *et al.* (2019); Seeger *et al.* (2017); Wan and Zhang (2019); Hubig (2019); Wan and Zhang (2019),
3. People need to differentiate over a quantum simulation where each quantum gate is an inplace function that changes the quantum register directly Luo *et al.* (2019).

To solve these issues better, we need a hardware instruction level AD. Source code transformation based AD packages like Tapenade Hascoet and Pascual (2013) and Zygote Innes (2018); Innes *et al.* (2019) are closer to this goal. They read the source code from a user and generate a new code that computes the gradients for users. However, these packages have their own limitations too. In many practical applications, an elementary level differentiable program that might do billions of computations will cache intermediate results to a global storage. Frequent caching of data slows down the program significantly, and the memory usage will become a bottleneck as well. With these AD tools, it is still nearly impossible to automatically generate the backward rules for BLAS functions and sparse matrix operations with a performance comparable to the state-of-the-art.

We propose to implement hardware instruction level AD on a reversible (domain-specific) programming language Perumalla (2013); Frank (2017). So that the intermediate states of a program can be traced backward with no extra effort. The overhead of reverse mode AD becomes the overhead of reversing a program, where the later has the advantage of efficient and controllable memory management. There have been many prototypes of reversible languages like Janus Lutz (1986), R (not the popular one) Frank (1997), Erlang Lanese *et al.* (2018) and object-oriented ROOPL Haulund (2017). In the past, the primary motivation of studying reversible programming is to support reversible computing devices Frank and Knight Jr (1999) like adiabatic complementary metal–oxide–semiconductor (CMOS) Koller and Athas (1992), molecular mechanical computing system Merkle *et al.* (2018) and superconducting system Likharev (1977); Semenov *et al.* (2003), where a reversible computing device is more energy-efficient from the perspective of information and entropy, or by the Landauer's principle Landauer (1961). After decades of efforts, reversible computing devices are very close to providing productivity now. As an example, adiabatic CMOS can be a better choice in a spacecraft Hänninen *et al.* (2014); DeBenedictis *et al.* (2017), where energy is more valuable than device itself. Reversible programming is interesting to software engineers too, because it is a powerful tool to schedule asynchronious events Jefferson (1985) and debug a program bidirectionally Boothe (2000). However, the field of reversible computing faces the issue of having not enough funding in recent decade Frank (2017). As a result, not many people

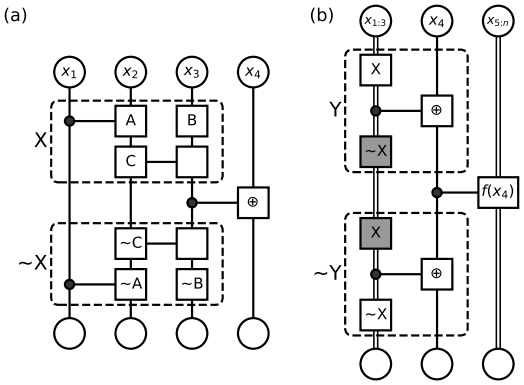


Figure 1: Two computational processes represented in memory oriented computational graph, where (a) is a subprogram in (b). In these graphs, a vertical single line represents one variable, a vertical double line represents multiple variables, and a parallel line represents a function. A dot at the cross represents a control parameter of a function and a box at the cross represents a mutable parameter of a function.

studying AD know the marvelous designs in reversible computing. People have not connected it with automatic differentiation seriously, even though they have many similarities. This paper aims to break the information barrier between the machine learning community and the reversible programming community in our work and provide yet another strong motivation to develop reversible programming.

In this paper, we first introduce the language design of a reversible programming language and introduce our reversible eDSL NiLang in Sec. 2. In Sec. 3, we explain the implementation of automatic differentiation in this eDSL. In Sec. 4, we show several examples. In Sec. 5, we benchmark the performance of NiLang with other AD packages and explain why reversible programming AD is fast. In Sec. 6, we discuss several important issues, the time-space tradeoff, reversible instructions and hardware, and finally, an outlook to some open problems to be solved. In the appendix, we show the grammar of NiLang and other technical details.

2 Language design

2.1 Introductions to reversible language design

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

2.1.1 Memory management

A distinct feature of reversible memory management is, the content of a variable must be known when it is deallocated. We denote the allocation of a zero emptied memory as $\mathtt{x} \leftarrow \mathtt{0}$, and the corresponding deallocation as $\mathtt{x} \rightarrow \mathtt{0}$. A variable x can be allocated and deallocated in a local scope, which is called an ancilla. It can also be pushed to a stack and used later with a pop statement. This stack is similar to a traditional stack, except it zero-clears the variable after pushing and presupposes that the variable being zero-cleared before popping.

Knowing the contents in the memory when deallocating is not easy. Hence Charles H. Bennett introduced the famous compute-copy-uncompute paradigm Bennett (1973) for reversible programming. To explain how it works, we introduce the memory oriented computational graph, as

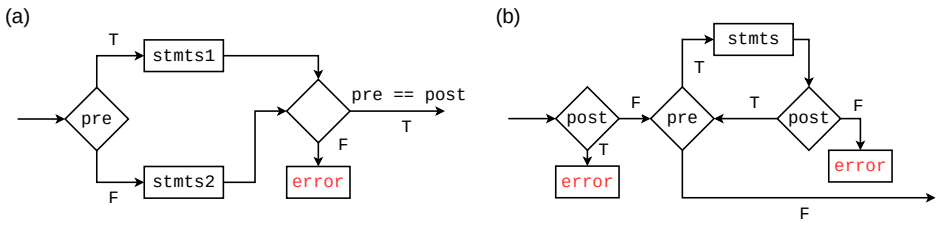


Figure 2: The flow chart for reversible (a) `if` statement and (b) `while` statement. "pre" and "post" represents precondition and postconditions respectively.

shown in Fig. 1. Notations are highly inspired by the quantum circuit representation. A vertical line is a variable and a horizontal line is a function. When a variable is used by a function, depending on whether its value is changed or not, we put a box or a dot at the line cross. It is different from the computational graph for being a hypergraph rather than a simple graph, because a variable can be used by multiple functions now. In panel (a). The subprogram in dashed box `X` is executed on space $x_{1:3}$ represents the computing stage. In the copying stage, the content in x_3 is read out to a pre-emptied memory x_4 through inplace add +=. Since this copy operation does not change contents of $x_{1:3}$, we can use the uncomputing operation `~X` to undo all the changes to these registers. Now we computing the result x_4 without modifying the contents in $x_{1:3}$. If any of them is in a known state, it can be deallocated immediately. In panel (b), we can use the subprogram defined in (a) maked as Y to generate $x_{5:n}$ without modifying the contents of variables $x_{1:4}$. It is easy to see that although this uncompute-copy-uncompute design pattern can restore memories to known state, it has computational overhead. Both X and $\sim X$ are executed twice in the program (b), which is not necessary. We can cancel a pair of X and $\sim X$ (the gray boxes). By doing this, we are not allowed to deallocate the memory $x_{1:3}$ during computing $f(x_{5:n})$. This is the famous time-space tradeoff that playing the central role in reversible programming. The trade off strategy will be discussed in detail in Sec. 6.1.

2.1.2 Control flows

One can define reversible `if`, `for` and `while` statements in a slightly different way comparing with its irreversible counterpart. The reversible `if` statement is shown in Fig. 2 (a). Its condition statement contains two parts, a precondition and a postcondition. The precondition decides which branch to enter in the forward execution, while the postcondition decides which branch to enter in the backward execution. After executing the specific branch, the program checks the consistency between precondition and postcondition to make sure they are consistent. The reversible `while` statement in Fig. 2 (b) also has two condition fields. Before executing the condition expressions, the program preassumes the postcondition is false. After each iteration, the program asserts the postcondition to be true. In the reverse pass, we exchange the precondition and postcondition. The reversible `for` statement is similar to irreversible ones except that after executing the loop, the program checks the values of these variables to make sure they are not changed. In the reverse pass, we exchange `start` and `stop` and inverse the sign of `step`.

2.1.3 Arithmetic instructions

Every arithmetic instruction has a unique inverse that can undo the changes.

- For logical operations, `y` $\veebar$`= f(args...)` is self reversible.
- For integer and floating point arithmetic operations, we treat `y += f(args...)` and `y -= f(args...)` as reversible to each other. Here `f` can be an arbitrary pure function such as `identity`, `*`, `/` and `^`. Let's forget the floating point rounding errors for the moment and discuss in detail in Sec. 6.2.
- For logartihmic number and tropical number algebra Speyer and Sturmfels (2009), `y *= f(args...)` and `y /= f(args...)` as reversible to each other. Notice the zero element $(-\infty)$ in the Tropical algebra is not considered here.

Besides the above two types of operations, `SWAP` operation that exchanges the contents in two memory spaces is also widely used in reversible computing systems.

2.2 Differetiable Reversible eDSL: NiLang

We develop an embedded domain-specific language (eDSL) NiLang in Julia language Bezanson *et al.* (2012, 2017) that implements reversible programming. One can write reversible control flows, instructions, and memory managements inside this macro. Julia is a popular language for scientific programming. We choose Julia as the host language for multiple purposes. The most important consideration is speed that crucial for a machine instruction level AD. Its clever design of type inference and just in time compiling provides a C like speed. Also, it has a rich ecosystem for meta-programming. The package for pattern matching MLStyle allow us to define an eDSL conveniently. Last but not least, its multiple-dispatch provides the polymorphism that will be used in our AD engine. The main feature of NiLang is contained in a single macro `@i` that compiles a reversible function. The allowed statements in this eDSL are shown in Appendix A. We can use `macroexpand` to show the compiling a reversible function to the native Julia function.

```
julia> using NiLangCore, MacroTools

julia> MacroTools.prettify(@macroexpand @i function f(x, y)
           SWAP(x, y)
       end)
quote
    $(Expr(:meta, :doc))
    function $(Expr(:where, :(f(x, y))))
        dove = wrap_tuple(SWAP(x, y))
        x = dove[1]
        y = dove[2]
        (x, y)
    end
    if NiLangCore._typeof(f) != _typeof(~f)
        function $(Expr(:where, :((hummingbird::_typeof(~f))(x, y))))
            toad = wrap_tuple((~SWAP)(x, y))
            x = toad[1]
            y = toad[2]
            (x, y)
        end
    end
end
```

Here, the version of NiLang is v0.4.0. Macro `@i` generates two functions that reversible to each other `f` and `~f`. `~f` is an callable of type `Inv{typeof(f)}`, where the type parameter `typeof(f)` stands for the type of the function `f`. In the body of `f`, `NiLangCore.wrap_tuple` is used to unify output data types to tuples. The outputs of `SWAP` are assigned back to its input variables. At the end of this function, this macro attaches a return statement that returns all input variables.

The compilation of a reversible function to native Julia functions is consisted of three stages: *preprocessing*, *reversing* and *translation*. Fig. 3 shows the compilation of the complex valued log function body, which is originally defined as follows.

Listing 1: Reversible implementation of the complex valued log function.

```
@i function (:+=)(log)(y!::Complex{T}, x::Complex{T}) where T
    @routine begin
        n ← zero(T)
        n += abs(x)
    end
    y!.re += log(n)
    y!.im += angle(x)
    ~@routine
end
```

In the *preprocessing* stage, the compiler pre-processes human inputs to reversible NiLang IR. The preprocessor removes redundant grammars and expands shortcuts. In the left most code box in

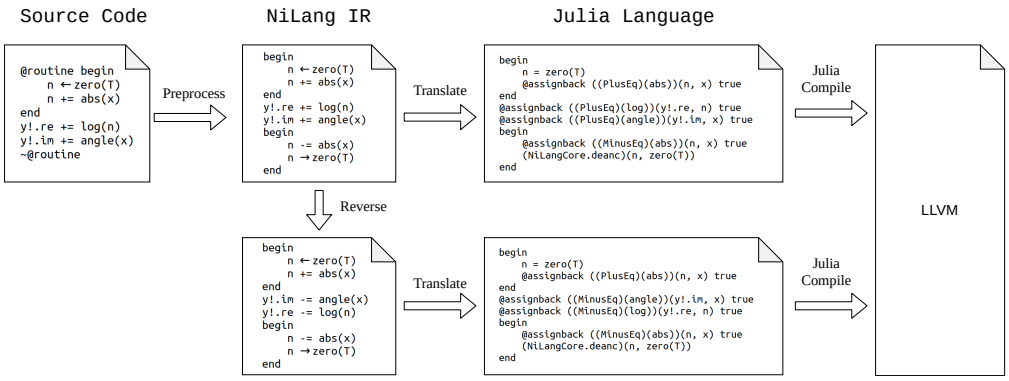


Figure 3: Compiling the body of the complex valued log function defined in Listing. 1.

Fig. 3, one uses `@routine <stmt>` statement to record a statement, and `~@routine` to insert the corresponding inverse statement for uncomputing. The computing-uncomputing macros `@routine` and `~@routine` is expanded in this stage. Here, one can input "←" and "→" by typing "\leftarrow[TAB KEY]" and "\rightarrow[TAB KEY]" respectively in a Julia editor or REPL.

In the *reversing* stage, based on this symmetric and reversible IR, the compiler generates reversed statements according to table Table 1.

Statement	**Meaning**	**Inverse**
`<f>(<args>...)`	function call	`(~<f>)(<args>...)`
`<f>.(<args>...)`	broadcast a function call	`<f>.(<args>...)`
`<y> += <f>(<args>...)`	inplace add instruction	`<y> -= <f>(<args>...)`
`<y> ⊻= <f>(<args>...)`	inplace XOR instruction	`<y> ⊻= <f>(<args>...)`
`<a> ← <expr>`	allocate a new variable	`<a> → <expr>`

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

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

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

`for <i>=<m>:<s>:<n>`
`  <stmts>`
`end`
	for statement	`for <i>=<m>:-<s>:<n>`
		`  ~(<stmts>)`
		`end`

Table 1: Basic statements in NiLang IR. "~" is the symbol for reversing a statement or a function. "." is the symbol for the broadcasting magic in Julia, `<pre>` stands for precondition, and `<post>` stands for postcondition "`begin <stmts> end`" is the code block statement in Julia. It can be inverted by reversing the order as well as each element in it.

In the *translation* stage, the compiler translates this reversible IR as well as its inverse to native Julia code. It adds `@assignback` before each function call, inserts codes for reversibility check, and handle control flows. We can expand the `@assignback` macro to see the compiled expression.

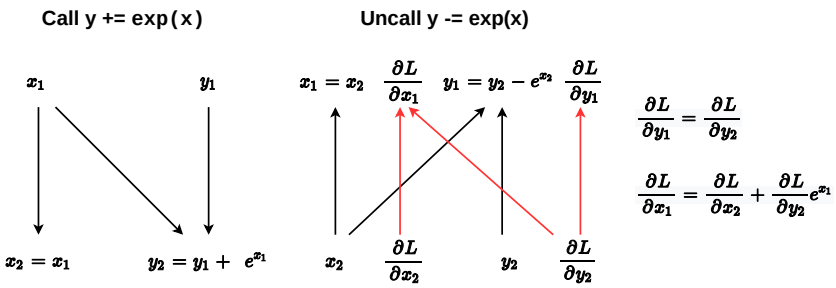


Figure 4: Binding the adjoint rule of `y+=exp(x)` to its uncomputing program.

```
julia> macroexpand(Main, :(@assignback PlusEq(log)(y!.re, n)))
quote
    var"##277" = (PlusEq(log))(y!.re, n)
    begin
        y! = chfield(y!, Val{:re}(), ((NiLangCore.wrap_tuple)(var"##277"))[1])
        n = ((NiLangCore.wrap_tuple)(var"##277"))[2]
    end
end
```

Here, the function `chfield` returns a complex number with an updated `re` field. This updated value is then assigned back to `y!`. In other words, this macro simulates "inplace" operations on immutable types. Except immutable fields, mappings and indexing can also be modified. We call an expression that directly modifiable in NiLang a *dataview*, it can be a variable itself, a field or an element of a dataview, or a bijective mapping of a dataview. The following are some examples of dataviews

- real(x)
- x.re
- x[3]
- x'
- real.(x)
- (x, y, z)
- tget(x, 2) # tuple get index
- -x[2].re'

As a final step, the compiler attaches a return statement that returns all updated input arguments at the end of a function definition. Now, the function is ready to execute on the host language. One can also define a reversible constructor and destructor, we put this part in Appendix C.

3 Reversible automatic differentiation

3.1 First order gradient

The computation of gradients in NiLang contains two parts, computing and uncomputing. In the computing stage, the program marches forward and computes outputs. In the uncomputing stage, we attach each scalar and array element with an extra gradient field and feed them into the inverse function. To composite data type with a gradient field is called `GVar`. As shown in Fig. 4, when an instruction is uncalled, we first uncompute the value field of `GVar`s to x_1 and y_1, using the input information, we then update the gradient fields according to the formula in the right panel. The binding utilizes the multiple dispatch in Julia, where a function can be dynamically dispatched based on the run time type of more than one of its arguments. Here, we dispatch a inverse instruction with input type `GVar` to the `(:-=)(exp)` instruction.

```
@i @inline function (:-=)(exp)(out!::GVar, x::GVar{T}) where T
    @routine @invcheckoff begin
        anc1 ← zero(value(x))
        anc1 += exp(value(x))
    end
    value(out!) -= identity(anc1)
    grad(x) += grad(out!) * anc1
    ~@routine
end
```

Here, the first four lines is the `@routine` statement that computes e^{x_2} and store the value into an ancilla. The 5th line updates the `value` dataview of `out!`. The 6th line updates the gradient fields of `x` and `y` by applying the adjoint rule of `(:+=)(exp)`. Finally, `@routine` uncomputes `anc1` so that it can be returned to the "memory pool". One does not need to define the similar function on `(:+=)(exp)` because macro `@i` will generate it automatically. Notice that taking inverse and computing gradients commute McInerney (2015).

3.2 Second-order gradient

Combining the uncomputing program in NiLang with dual-numbers is a simple yet efficient way to obtain Hessians. The dual number is the scalar type for computing gradients in the forward mode AD, it wraps the original scalar with a extra gradient field. The gradient field of a dual number is updated automatically as the computation marches forward. By wrapping the elementary type with `Dual` defined in package ForwardDiff Revels *et al.* (2016) and throwing it into the gradient program defined in NiLang, one obtains one row/column of the Hessian matrix straightforward. We will show a benchmark in Sec. 5.3.

3.3 Differentiating complex numbers

To differentiate complex numbers, we re-implemented complex instructions reversiblly. For example, with the reversible function defined in in Listing. 1, we can differentiated complex valued log with no extra effort.

3.4 Differentiating CUDA kernels

CUDA programming is playing a significant role in high-performance computing. In Julia, one can write GPU compatible functions in native Julia language with KernelAbstractions Besard *et al.* (2017). Since NiLang does not push variables into stack automatically for users, it is safe to write differentiable GPU kernels with NiLang. We will show this feature in the benchmarks of bundle adjustment (BA) in Sec. 5.4. Here, one should notice that the shared read in forward pass will become shared write in the backward pass, which may result in incorrect gradients. We will review this issue in Sec. 6.5.

4 Examples

In this section, we introduce several examples.

- sparse matrix dot product,
- first kind bessel function and memory oriented computational graph,
- solving the graph embedding problem.

All codes for this section and the next benchmark section are available in the paper repository.

4.1 Sparse Matrices

Differentiating sparse matrices is useful in many applications, however, it can not benefit directly from generic backward rules for the dense matrix because the generic rules do not keep the sparse structure. In the following, we will show how to convert a irreversible Frobenius dot product code

to a reversible one to differentiate it. Here, the Frobenius dot product is defined as `trace(A'B)`. In SparseArrays code base, it is implemented as follows.

```
function dot(A::AbstractSparseMatrixCSC{T1,S1},
     B::AbstractSparseMatrixCSC{T2,S2}
     ) where {T1,T2,S1,S2}
  m, n = size(A)
  size(B) == (m,n) || throw(DimensionMismatch("
     matrices must have the same dimensions"))
  r = dot(zero(T1), zero(T2))
  @inbounds for j = 1:n
    ia = getcolptr(A)[j]
    ia_nxt = getcolptr(A)[j+1]
    ib = getcolptr(B)[j]
    ib_nxt = getcolptr(B)[j+1]
    if ia < ia_nxt && ib < ib_nxt
      ra = rowvals(A)[ia]
      rb = rowvals(B)[ib]
      while true
        if ra < rb
          ia += oneunit(S1)
          ia < ia_nxt || break
          ra = rowvals(A)[ia]
        elseif ra > rb
          ib += oneunit(S2)
          ib < ib_nxt || break
          rb = rowvals(B)[ib]
        else # ra == rb
          r += dot(nonzeros(A)[ia],
                nonzeros(B)[ib])
          ia += oneunit(S1)
          ib += oneunit(S2)
          ia < ia_nxt && ib < ib_nxt || break
          ra = rowvals(A)[ia]
          rb = rowvals(B)[ib]
        end
      end
    end
  end
  return r
end
```

It is easy to rewrite it in a reversible style with NiLang without sacrificing much performance.

```
@i function dot(r::T, A::SparseMatrixCSC{T}, B::
     SparseMatrixCSC{T}) where {T}
    m ← size(A, 1)
    n ← size(A, 2)
    @invcheckoff branch_keeper ← zeros(Bool,2*m)
    @safe size(B) == (m,n) || throw(
     DimensionMismatch("matrices must have the
     same dimensions"))
    @invcheckoff @inbounds for j = 1:n
        ia1 ← A.colptr[j]
        ib1 ← B.colptr[j]
        ia2 ← A.colptr[j+1]
        ib2 ← B.colptr[j+1]
        ia ← ia1
        ib ← ib1
        @inbounds for i=1:ia2-ia1+ib2-ib1-1
            ra ← A.rowval[ia]
            rb ← B.rowval[ib]
            if (ra == rb, ~)
                r += A.nzval[ia]'*B.nzval[ib]
            end
            # b move -> true, a move -> false
            branch_keeper[i] ⊻= ia==ia2-1 ||
                ra > rb
            ra → A.rowval[ia]
            rb → B.rowval[ib]
            if (branch_keeper[i], ~)
                ib += identity(1)
            else
                ia += identity(1)
            end
        end

        ~@inbounds for i=1:ia2-ia1+ib2-ib1-1
            # b move -> true, a move -> false
            branch_keeper[i] ⊻= ia==ia2-1 ||
                A.rowval[ia] > B.rowval[ib]
            if (branch_keeper[i], ~)
                ib += identity(1)
            else
                ia += identity(1)
            end
        end
    end
    @invcheckoff branch_keeper → zeros(Bool, 2*m
     )
end
```

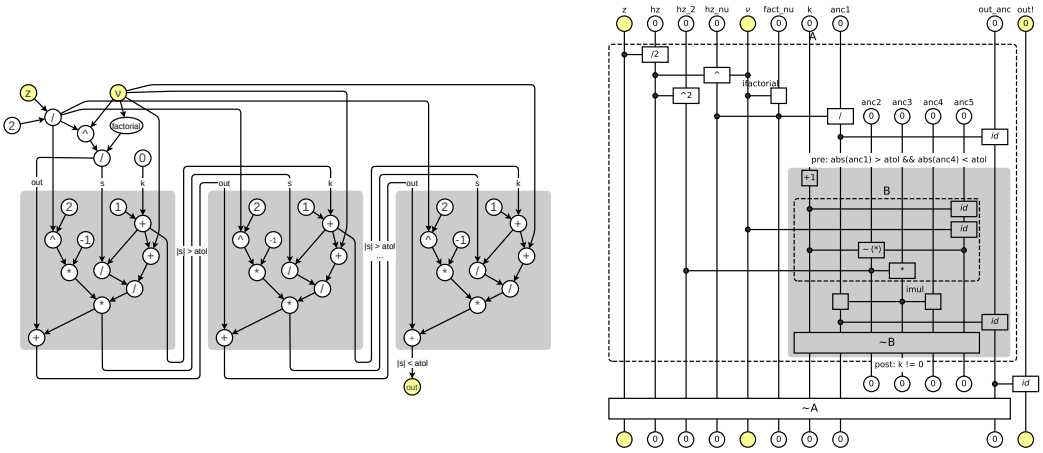
Here, all assignments are replaced with ← to indicate that the values of these variables must be returned at the end of this function scope. We put a "~" symbol in the postcondition field of if statements to indicate this postcondition is a dummy one that takes the same value as the precondition, i.e. the condition is not changed inside the loop body. If the precondition is changed by the loop body, one can use a `branch_keeper` vector to cache branch decisions. The value of `branch_keeper` can be restored through uncomputing (the "~" statement above). Finally, after checking the correctness of the program, one can turn off the reversibility checks by using the macro `@invcheckoff` macro to achieve better performance. We provide the benchmark of this function in Sec. 5.1, where the reversible sparse matrix multiplication is also benchmarked.

4.2 The first kind Bessel function

A Bessel function of the first kind of order ν can be computed via Taylor expansion

$$J_\nu(z) = \sum_{n=0}^{\infty} \frac{(z/2)^\nu}{\Gamma(k+1)\Gamma(k+\nu+1)}(-z^2/4)^n \tag{1}$$

where $\Gamma(n) = (n-1)!$ is the Gamma function. One can compute the accumulated item iteratively as $s_n = -\frac{z^2}{4}s_{n-1}$. The irreversible implementation is



(a) (b)

Figure 5: (a) The traditional computational graph for the irreversible implementation of the first kind Bessel function. A vertex (circle) is an operation, and a directed edge is a variable. The gray regions are the body of the unrolled while loop. (b) The memory oriented computational graph for the reversible implementation of the first kind Bessel function. Notations are explained in Fig. 1. The gray region is the body of a while loop. Its precondition and postcondition are positioned on the top and bottom, respectively.

```
function besselj(ν, z; atol=1e-8)
    k = 0
    s = (z/2)^ν / factorial(ν)
    out = s
    while abs(s) > atol
        k += 1
        s *= (-1) / k / (k+ν) * (z/2)^2
        out += s
    end
    out
end
```

This computational process could be diagrammatically represented as a computational graph as shown in Fig. 5 (a). The computational graph is a directed acyclic graph (DAG), where a node is a function and an edge is a data. An edge connects two nodes, one generates this data, and one consumes it. A computational graph is more likely a mathematical expression. It can not describe inplace functions and control flows conveniently because it does not have the notation for memory and loops.

Before showing the reversible implementation, we introduce how to obtain the product of a sequence of numbers reversiblly. Consequtive multiplication requires an increasing size of tape to cache an intermediate state $x_1 x_2 \dots x_n$, since one can not deallocate the previous state $x_1 x_2 \dots x_{n-1}$ directly since *= and /= are not considered as reversible here. To mitigate the space overhead, the standard approach in reversible computing is the pebble game model Perumalla (2013) (or the checkpointing technique in machine learning), where the cache size can be reduce in the cost of increasing the time complexity, however, constant cache size is not achievable in this scheme. Hence, we introduce the following reversible approximate multiplier.

```
@i @inline function imul(out!, x, anc!)
    anc! += out! * x
    out! -= anc! / x
    SWAP(out!, anc!)
end
```

Here, the thrid argument `anc!` is a *dirty ancilla*, which should be a value ≈ 0. Line 2 computes the result and accumulates it to the dirty ancilla, so that we have an approximate output in `anc!`. Line 3 removes the content in `out!` approximately using the information stored in `anc!`. Line 4 swaps the contents in `out!` and `anc!`. Finally, we have an approximate output and a dirtier ancilla. The reason why this trick works here lies in the fact that *= and /= are mathematically reversible (except the zero point) to each other. One can approximately uncomputing the contents in the register at the cost of rounding errors. This rounding error introduced in such a way only affects the function output, and does not sacrifice the reversibility. With this approximate multiplier, we implement the reversible J_ν as follows.

```
using NiLang, NiLang.AD

@i function ibesselj(out!, ν, z; atol=1e-8)
    k ← 0
    fact_nu ← zero(ν)
    halfz ← zero(z)
    halfz_power_nu ← zero(z)
    halfz_power_2 ← zero(z)
    out_anc ← zero(z)
    anc1 ← zero(z)
    anc2 ← zero(z)
    anc3 ← zero(z)
    anc4 ← zero(z)
    anc5 ← zero(z)

    @routine begin
        halfz += z / 2
        halfz_power_nu += halfz ^ ν
        halfz_power_2 += halfz ^ 2
        ifactorial(fact_nu, ν)

        anc1 += halfz_power_nu/fact_nu
        out_anc += identity(anc1)
        while (abs(unwrap(anc1))>atol && abs(
     unwrap(anc4)) < atol, k!=0)
            k += identity(1)
            @routine begin
                anc5 += identity(k)
                anc5 += identity(ν)
                anc2 -= k * anc5
                anc3 += halfz_power_2 / anc2
            end
            imul(anc1, anc3, anc4)
            out_anc += identity(anc1)
            ~@routine
        end
    end
    out! += identity(out_anc)
    ~@routine
end

@i function ifactorial(out!, n)
    anc ← zero(n)
    out! += identity(1)
    for i=1:n
        imul(out!, i, anc)
    end
end

@i @inline function imul(out!::T, x::T, anc!::T)
       where T<:Integer
    anc! += out! * x
    out! -= anc! ÷ x
    SWAP(out!, anc!)
end
```

The above algorithm uses a constant number of ancillas, while the time overhead is also a constant factor. Ancilla `anc4` plays the role of *dirty ancilla* in multiplication, and it is uncomputed rigorously in the uncomputing stage marked by `~@routine`. This reversible program can be diagrammatically represented as a memory oriented computational graph as shown in Fig. 5 (b). This diagram can be used to analyse variables uncomputing. In this example, routine "B" uses `hz_2`, ν and `k` as control parameters, and changes the contents in `anc2`, `anc3` and `anc5`. The following `imul` and `(:+=)(identity)` copies the result to output without changing these variables. Hence we can apply the inverse routine `~B` to safely restore contents in `anc2`, `anc3` and `anc5`, and this examplifies the compute-copy-uncompute paradigm.

```
julia> out!, x = 0.0, 1.0
(0.0, 1.0)

julia> Grad(ibesselj)(Val(1), out!, 2, x)
(Val{1}(), GVar(0.0, 1.0), 2, GVar(1.0, 0.2102436))
```

One can obtain gradients of this function by calling `Grad(ibesselj)`. Here, `Grad(ibesselj)` returns a callable instance of type `Grad{typeof(ibesselj)}`. The first parameters `Val(1)` specifies the position of loss in argument list. The Hessian can be obtained by feeding dual-numbers into this gradient function.

```
julia> out!, x = 0.0, 1.0
(0.0, 1.0)

julia> Grad(ibesselj)(Val(1), out!, 2, x)
(Val{1}(), GVar(0.0, 1.0), 2, GVar(1.0, 0.2102436))

julia> using ForwardDiff: Dual

julia> _, hxout!, _, hxx = Grad(ibesselj)(Val(1),
        Dual(out!, zero(out!)), 2, Dual(x, one(x)));

julia> grad(hxx).partials[1]
0.13446683844358093
```

Here, the gradient field of `hxx` is defined as $\frac{\partial \text{out!}}{\partial x}$, which is a Dual number. It has a field partials that store the derivative for `x`. It corresponds to the Hessian $\frac{\partial \text{out!}^2}{\partial x^2}$ that we need.

4.3 Solving a graph embedding problem

Graph embedding can be used to find representation for an order parameter Takahashi and Sandvik (2020) in condensed matter physics. Ref. Takahashi and Sandvik (2020) considers a problem of finding the minimum Euclidean space dimension k that a Petersen graph can fit into, with extra requirements that the distance between a pair of connected vertices has the same value l_1, and the distance between a pair of disconnected vertices has the same value l_2 and $l_2 > l_1$. The Petersen graph is ten vertices graph, as shown in Fig. 6. Let us denote the set of connected and disconnected vertex pairs as L_1 and L_2, respectively. This problem can be variationally solved by differential programming by designing the subsequent loss.

$$\begin{aligned}\mathcal{L} = &\text{Var}(\text{dist}(L_1)) + \text{Var}(\text{dist}(L_2)) \\ &+ \exp(\text{relu}(\overline{\text{dist}(L_1)} - \overline{\text{dist}(L_2)} + 0.1))) - 1\end{aligned} \tag{2}$$

The first line is a summation of distance variances in two sets of vertex pairs, where Var$X$ means taking the variance of samples in X. The second line is used to guarantee $l_2 > l_1$, where $\overline{X}$ means taking the average of samples in X. Its reversible implementation could be found in our benchmark repository.

We repeat the training for each dimension k from 1 to 10 and search for possible solutions by variationally optimizing the positions of vertices. In each training, we fix two of the vertices and train the rest. Otherwise, the program will find the trivial solution with overlapped vertices. For $k = 5$, we can get a loss close to machine precision with high probability, while for $k < 5$, the loss is always much higher than 0. From the solution, it is easy to see $l_2/l_1 = \sqrt{2}$ is the solution. For $k = 5$, an Adam optimizer with a learning rate 0.01 Kingma and Ba requires $\sim$ 2000 steps training. The trust region Newton's method converges much faster, which requires $\sim$ 20 computations of Hessians to reach convergence. Although training time is comparable, the converged precision of the later is much better.

5 Benchmarks

In the following benchmarks the CPU device is Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz, and the GPU device is Nvidia Titan V. For NiLang benchmarks, we have turned off the reversibility check off to achieve better performance.

5.1 Sparse matrices

We benchmarked the call, uncall and backward propagation time used for sparse matrix dot product and matrix multiplication. Here, we estimate the time for back propagating gradients rather than including both forward and backward, since `mul!` does not output a scalar as loss.

The time used for computing backward pass is approximately 1.5-3 times the Julia's native forward pass. This is because the instruction length of differentiating basic arithmetic instructions is longer than pure computing by a factor of 2 or more.

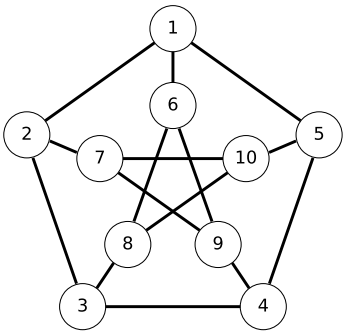


Figure 6: The Petersen graph has 10 vertices and 15 edges. We want to find a minimum embedding dimension for it.

	`dot`	`mul!` (complex valued)
Julia-O	3.493e-04	8.005e-05
NiLang-O	4.675e-04	9.332e-05
NiLang-B	5.821e-04	2.214e-04

Table 2: Absolute runtimes in seconds for computing the objectives (O) and the backward pass (B) of sparse matrix operations. The matrix size is 1000×1000, and the element density is 0.05. The total time used in computing gradient can be estimated as a sum of "O" and "B".

5.2 Bessel Function

We differentiate the first type Bessel function in Sec. 4.2 and show the benchmarks in Table 3. In the table, Julia is the CPU time used for running the irreversible forward program. It is the baseline for benchmarking. NiLang (call/uncall) is the time of reversible call or uncall. Both of them are $\sim$ 3 times slower than its irreversible counterpart. Since Bessel function has only one input argument, forward mode AD tools are faster than reverse mode AD, both source-to-source framework ForwardDiff and operator overloading framework Tapenade have the a comparable computing time with the pure function call.

	T_{min}/ns	Space/KB
Julia-O	18	0
NiLang-O	32	0
Tapenade-O	32	0
ForwardDiff-G	38	0
NiLang-G	201	0
NiLang-G (CUDA)	1.4	0
ReverseDiff-G	1406	1.2
Zygote-G	22596	13.47
Tapenade-G (Forward)	30	0
Tapenade-G (Backward)	111	> 0

Table 3: Time and space used for computing objective (O) and gradient (G) of the first kind Bessel function $J_2(1.0)$. The CUDA time is averaged over a batch size of 4000, which is not a fair comparison but shows how much performance can we get from GPU in the parallel computing context.

NiLang.AD is the reverse mode AD submodule in NiLang, and it takes 13.6 times the native Julia program, and is also 2 times slower than Tapenade. However, the key point is, there is no extra memory allocation like stack operations in the whole computation. The controllable memory allocation of NiLang makes it compatible with CUDA program. In other backward mode AD like Zygote, ReverseDiff and Tapenade, the memory allocation in heap is nonzero due to the checkpointing and possible failure of type inference.

5.3 Graph embedding problem

Since one can combine ForwardDiff and NiLang to obtain Hessians, it is interesting to see how much performance we can get in differentiating the graph embedding program in Sec. 4.3.

k	2	4	6	8	10
Julia-O	4.477e-06	4.729e-06	4.959e-06	5.196e-06	5.567e-06
NiLang-O	7.173e-06	7.783e-06	8.558e-06	9.212e-06	1.002e-05
NiLang-U	7.453e-06	7.839e-06	8.464e-06	9.298e-06	1.054e-05
NiLang-G	1.509e-05	1.690e-05	1.872e-05	2.076e-05	2.266e-05
ReverseDiff-G	2.823e-05	4.582e-05	6.045e-05	7.651e-05	9.666e-05
ForwardDiff-G	1.518e-05	4.053e-05	6.732e-05	1.184e-04	1.701e-04
Zygote-G	5.315e-04	5.570e-04	5.811e-04	6.096e-04	6.396e-04
(NiLang+F)-H	4.528e-04	1.025e-03	1.740e-03	2.577e-03	3.558e-03
ForwardDiff-H	2.378e-04	2.380e-03	6.903e-03	1.967e-02	3.978e-02
(ReverseDiff+F)-H	1.966e-03	6.058e-03	1.225e-02	2.035e-02	3.140e-02

Table 4: Absolute runtimes in seconds for computing the objectives (O), uncall objective (U), gradients (G) and Hessians (H) of the graph embedding program. k is the embedding dimension, the number of parameters is $10k$.

In Table 4, we show the the performance of different implementations by varying the dimension k. As the baseline, (a) shows the time for computing the function call. We have reversible and irreversible implementations, where the reversible program is slower than the irreversible native Julia program by a factor of ~ 2 due to the uncomputing overhead. The reversible program shows the advantage of obtaining gradients when the dimension $k \geq 3$. The larger the number of inputs, the more advantage it shows due to the overhead proportional to input size in forward mode AD. The same reason applies to computing Hessians. The combining NiLang and ForwardDiff gives better performance when $k \geq 3$ comparing with other AD frameworks.

5.4 Gaussian mixture model and bundle adjustment

We reproduced the benchmarks for Gaussian mixture model (GMM) and bundle adjustment (BA) in Ref. Srajer *et al.* (2018) by re-writing the programs in a reversible style. We show the results in Table 5 and Table 6. In our new benchmarks, we also rewrite the ForwardDiff program for a fair benchmark, this explains the difference between our results and the original benchmark. The Tapenade data is obtained by executing the docker file provided by the original benchmark, which is the baseline of our benchmark and the original benchmark.

# parameters	3.00e+1	3.30e+2	1.20e+3	3.30e+3	1.07e+4	2.15e+4	5.36e+4	4.29e+5
Julia-O	9.844e-03	1.166e-02	2.797e-01	9.745e-02	3.903e-02	7.476e-02	2.284e-01	3.593e+00
NiLang-O	3.655e-03	1.425e-02	1.040e-01	1.389e-01	7.388e-02	1.491e-01	4.176e-01	5.462e+00
Tapende-O	1.484e-03	3.747e-03	4.836e-02	3.578e-02	5.314e-02	1.069e-01	2.583e-01	2.200e+00
ForwardDiff-G	3.551e-02	1.673e+00	4.811e+01	1.599e+02	-	-	-	-
NiLang-G	9.102e-03	3.709e-02	2.830e-01	3.556e-01	6.652e-01	1.449e+00	3.590e+00	3.342e+01
Tapenade-G	5.484e-03	1.434e-02	2.205e-01	1.497e-01	4.396e-01	9.588e-01	2.586e+00	2.442e+01

Table 5: Absolute runtimes in seconds for computing the objective (O) and gradients (G) of GMM with 10k data points.

In the GMM benchmark, NiLang's objective function has more overhead than irreversible programs in most cases. Except the uncomputing overhead, it is also because the bottleneck of this program is the matrix-vector multiplication, where our naive reversible implementation is much slower than the highly optimized BLAS function. The forward mode AD suffers from too large input dimension in this case, hence some data are missing. Although ForwardDiff is able to batch the gradient fields, the overhead proportional to input size still dominates. The source to source AD framework Tapenade is faster than NiLang in all scale, but the ratio between computing gradients and objective function are close. The memory consumption of our reversible implementation is quite low. The peak memory is only slightly more than twice of the original program, where the factor 2 comes from wrapping each variable with an extra gradient field.

# measurements	3.18e+4	2.04e+5	2.87e+5	5.64e+5	1.09e+6	4.75e+6	9.13e+6
Julia-O	2.020e-03	1.292e-02	1.812e-02	3.563e-02	6.904e-02	3.447e-01	6.671e-01
NiLang-O	2.708e-03	1.757e-02	2.438e-02	4.877e-02	9.536e-02	4.170e-01	8.020e-01
Tapenade-O	1.632e-03	1.056e-02	1.540e-02	2.927e-02	5.687e-02	2.481e-01	4.780e-01
ForwardDiff-J	6.579e-02	5.342e-01	7.369e-01	1.469e+00	2.878e+00	1.294e+01	2.648e+01
NiLang-J	1.651e-02	1.182e-01	1.668e-01	3.273e-01	6.375e-01	2.785e+00	5.535e+00
NiLang-J (GPU)	1.354e-04	4.329e-04	5.997e-04	1.735e-03	2.861e-03	1.021e-02	2.179e-02
Tapenade-J	1.940e-02	1.255e-01	1.769e-01	3.489e-01	6.720e-01	2.935e+00	6.027e+00

Table 6: Absolute runtimes in seconds for computing the objective (O) and Jacobians (J) in bundle adjustment.

In the BA benchmark, reverse mode AD shows slight advantage over ForwardDiff. The bottleneck of computing this large sparse Jacobian is computing the Jacobian of a elementary function with 15 input arguments and 2 output arguments, where input space is larger than the output space. In this instance, our reversible implementation is even faster than the source code transformation based AD framework Tapenade. With KernelAbstractions, we run the reversible program on GPU, which provides more than 200x speed up. Running codes on GPU (in Julia) requires not introducing any memory allocation at the kernel level. NiLang provides user the flexibility to manage the memory allocation instead of using a global stack.

6 Discussion and outlooks

In this paper, we show how to realize a reversible programming eDSL and implement an instruction level backward mode AD on top of it. It gives the user more flexibility to tradeoff memory and computing time comparing with traditional checkpointing. The Julia implementation NiLang gives the state-of-the-art performance and memory efficiency in obtaining first and second-order gradients in applications, including first type Bessel function, sparse matrix manipulations, solving graph embedding problem, Gaussian mixture model and bundle adjustment.

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

6.1 Time Space Tradeoff

In history, there have been many discussions about time-space tradeoff on a reversible Turing machine (RTM). In the most straightforward g-segment tradeoff scheme Bennett (1989); Levine and Sherman (1990), an RTM model has either a space overhead that is proportional to computing time T or a computational overhead that sometimes can be exponential to the program size comparing with an irreversible counterpart. This result stops many people from taking reversible computing seriously as a high-performance computing scheme. In the following, we try to explain why the overhead of reversible computing is not as terrible as people thought.

First of all, the overhead of reversing a program is upper bounded by the checkpointing Chen *et al.* (2016) strategy used in many traditional machine learning package that memorizes inputs of primitives because checkpointing can be trivially implemented in reversible programming. Perumalla (2013) Reversible programming provides some alternatives to reduce the overhead. For example, accumulation is reversible, so that many BLAS functions can be implemented reversiblly without extra memory. Meanwhile, the memory allocation in some

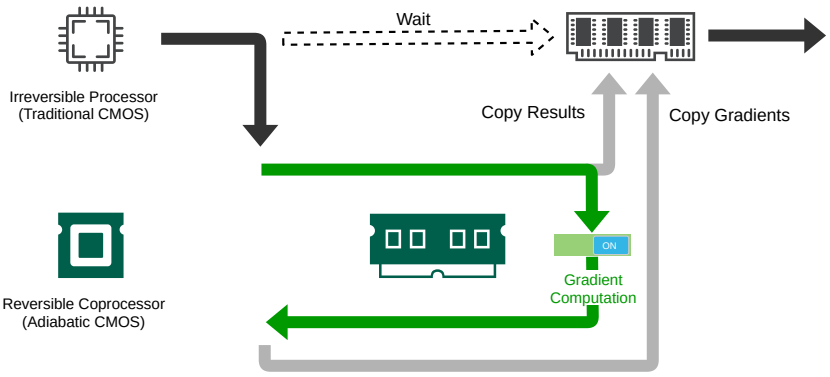


Figure 7: Energy efficient AI co-processor. Green arrows represents energy efficient operations on reversible devices.

iterative algorithms can often be reduced with the "arithmetic uncomputing" trick without sacrificing reversibility, as shown in the `ibesselj` example in Sec. 4.2. Clever compiling based on memory oriented computational graphs (Fig. 1 and Fig. 5 (b)) can also be used to help user tradeoff between time and space. The overhead of a reversible program mainly comes from the uncomputing of ancillas. It is possible to automatically uncompute ancillas by analyzing variable dependency instead of asking users to write `@routine` and `~@routine` pairs. In a hierarchical design, uncomputing can appear in every memory deallocation (or symbol table reduction). To quantify the overhead of uncomputing, we introduce the term uncomputing level as bellow.

***Definition* 1** (uncomputing level)**.** The log-ratio between the number of instructions of a reversible program and its irreversible counterpart.

From the lowest instruction level, whenever we reduce the symbol table (or space), the computational cost doubles. The computational overhead grows exponentially as the uncomputating level increases, which can be seen from some of the benchmarks in the main text. In sparse matrix multiplication and dot product, we don't introduce uncomputing in the most time consuming part, so it is ~ 0. The space overhead is `2*m` to keep the branch decisions, which is even much smaller than the memory used to store row indices. in Gaussian mixture model, the most time consuming matrix-vector multiplication is doubled, so it is ~ 1. The extra memory usage is approximately 0.5% of the original program. In the first kind Bessel function and bundle adjustment program, the most time consuming parts are (nestedly) uncomputed twice, hence their uncomputing level is ~ 2. Such aggressive uncomputing makes zero memory allocation possible.

6.2 Differentiability as a Hardware Feature

So far, our eDSL is compiled to Julia language. It relies on Julia's multiple dispatch to differentiate a program, which requires users to write generic programs. A more liable AD should be a hardware or micro instruction level feature. In the future, we can expect NiLang being compiled to reversible instructions Vieri (1999) and executed on a reversible device. A reversible devices can play a role of differentiation engine as shown in the hetero-structural design in Fig. 7. It defines a reversible instruction set and has a switch that controls whether the instruction calls a normal instruction or an instruction that also updates gradients. When a program calls a reversible differentiable subroutine, the reversible co-processor first marches forward, compute the loss and copy the result to the main memory. Then the co-processor execute the program backward and uncall instructions, initialize and updating gradient fields at the same time. After reaching the starting point of the program, the gradients are transferred to the global memory. Running AD program on a reversible device can save energy. Theoretically, the reversible routines do not necessarily cost energy, the only energy bottleneck is copying gradient and outputs to the main memory.

6.3 The connection to Quantum programming

A Quantum device Nielsen and Chuang (2002) is a special reversible hardware that features quantum entanglement. The instruction set of classical reversible programming is a subset of

quantum instruction set. However, building a universal quantum computer is difficult. Unlike a classical state, a quantum state can not be cloned. Meanwhile, it loses information by interacting with the environment. Classical reversible computing does not enjoy the quantum advantage, nor the quantum disadvantages of non-cloning and decoherence, but it is a model that we can try directly with our classical computer. It is technically smooth to have a reversible computing device to bridge the gap between classical devices and universal quantum computing devices. By introducing entanglement little by little, we can accelerate some elementary components in reversible computing. For example, quantum Fourier transformation provides an alternative to the reversible adders and multipliers by introducing the Hadamard and CPHASE quantum gates Ruiz-Perez and Garcia-Escartin (2017). From the programming languages's perspective, most quantum programming language preassumes the existence of a classical coprocessor to control quantum devices Svore *et al.* (2018). It is also interesting to know what is a native quantum control flow like, and does quantum entanglement provide speed up to automatic differentiation? We believe the reversible compiling technologies will open a door to study quantum compiling.

6.4 Gradient on ancilla problem

In this subsection, we introduce an easily overlooked problem in our reversible AD framework. An ancilla can sometimes carry a nonzero gradient when it is deallocated. As a result, the gradient program can be irreversible in the local scope. In NiLang, we drop the gradient field of ancillas instead of raising an error. In the following, we justify our decision by proving the following theorem.

***Theorem* 1.** *Deallocating an ancilla with constant value field and nonzero gradient field does not introduce incorrect gradients.*

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

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

Discarding gradients should not have any effect on the `value` fields of outputs. The key is to show `grad(b)` $\equiv \frac{\partial \mathbf{x}^L}{\partial b}$ does appear in the `grad` fields of the output. It can be seen from the back-propagation rule

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

where the second term with $\frac{\partial \mathbf{x}^L}{\partial b}$ vanishes naturally. We emphasis here, the value part of discarded ancilla must be a constant. □

6.5 Shared read and write problem

One should be careful about shared read in reversible programming AD, because the shared read can introduce shared write in the adjoint program. Let's begin with the following expression.

```
y += x * y
```

Most people will agree that this statement is not reversible and should not be allowed because it changes input variables. We call it the *simultaneous read-and-write* issue. However, the following expression with two same inputs is a bit subtle.

```
y += x * x
```

It is reversible, but should not be allowed in an AD program because of the *shared write* issue. It can be seen directly from the expanded expression.

```
julia> macroexpand(Main, :(@instr y += x * x))
quote
    var"##253" = ((PlusEq)(*))(y, x, x)
    begin
        y = ((NiLangCore.wrap_tuple)(var"##253"))[1]
        x = ((NiLangCore.wrap_tuple)(var"##253"))[2]
        x = ((NiLangCore.wrap_tuple)(var"##253"))[3]
    end
end
```

In an AD program, the gradient field of `x` will be updated. The later assignment to x will overwrite the former one and introduce an incorrect gradient. One can get free of this issue by avoiding using same variable in a single instruction

```
anc ← zero(x)
anc += identity(x)
y += x * anc
anc -= identity(x)
```

or equivalently,

```
y += x ^ 2
```

Share variables in an instuction can be easily identified and avoided. However, it becomes tricky when one runs the program in a parallel way. For example, in CUDA programming, every thread may write to the same gradient field of a shared scalar. How to solve the shared write in CUDA programming is still an open problem, which limits the power of reversible programming AD on GPU.

6.6 Outlook

We can use NiLang to solve some existing issues related to AD. Reversible programming can make use of reversibility to save memory. Reversibility has been used in reducing the memory allocations in machine learning models such as recurrent neural networks MacKay *et al.* (2018), Hyperparameter learning Maclaurin *et al.* (2015) and residual neural networks Behrmann *et al.* (2018). We can use it to generate AD rules for existing machine learning packages like ReverseDiff, Zygote Innes *et al.* (2019), Knet Yuret (2016), and Flux Innes *et al.* (2018). Many backward rules for sparse arrays and linear algebra operations have not been defined yet in these packages. We can also use the flexible time-space tradeoff in reversible programming to overcome the memory wall problem in some applications. A successful, related example is the memory-efficient domain-specific AD engine in quantum simulator Yao Luo *et al.* (2019). This domain-specific AD engine is written in a reversible style and solved the memory bottleneck in variational quantum simulations. It also gives so far the best performance in differentiating quantum circuit parameters. Similarly, we can write memory-efficient normalizing flow Kobyzev *et al.* (2019) in a reversible style. Normalizing flow is a successful class of generative models in both computer vision Kingma and Dhariwal (2018) and quantum physics Dinh *et al.* (2016); Li and Wang (2018), where its building block bijector is reversible. We can use a similar idea to differentiate reversible integrators Hut *et al.* (1995); Laikov (2018). With reversible integrators, it should be possible to rewrite the control system in robotics Giftthaler *et al.* (2017) in a reversible style, where scalar is a first-class citizen rather than tensor. Writing a reversible control program should boost training performance. Reversibility is also a valuable resource for training.

To solve the above problems better, reversible programming should be improved from multiple perspectives. First, we need a better compiler for compiling reversible programs. To be specific, a compiler that admits mutability of data, and handle shared read and write better. Then, we need a reversible number system and instruction set to avoid rounding errors and support reversible control flows better. There are proposals of reversible floating point adders and multipliers, however these

designs require allocating garbage bits in each operation Nachtigal *et al.* (2010, 2011); Nguyen and Meter (2013); Häner *et al.* (2018). In NiLang, one can simulate rigorous reversible arithmetics with the fixed-point number package FixedPointNumbers. However, a more efficient reversible design requires instruction-level support. Some other numbers systems are reversible under `*=` and `/=` rather than `+=` and `-=`, including LogarithmicNumbers Taylor *et al.* (1988) and TropicalNumbers. They are powerful tools to solve domain specific problems, for example, we have an upcoming work about differentiating over tropical numbers to solve the ground state configurations of a spinglass system efficiently. We also need `comefrom` like instruction as a partner of `goto` to specify the postconditions in our instruction set. Finally, although we introduced that the adiabatic CMOS as a better choice as the computing device in a spacecraft DeBenedictis *et al.* (2017). There are some challenges in the hardware side too, one can find a proper summary of these challenges in Ref. Frank (2005).

Solutions to these issues requires the participation of people from multiple fields.

7 Acknowledgments

Jin-Guo Liu thank Lei Wang for motivating the project with possible applications to reversible integrator, normalizing flow, and neural ODE. Johann-Tobias Schäg for deepening the discussion about reversible programming with his mathematicians head. Marisa Kiresame and Xiu-Zhe Luo for discussion on the implementation details of source-to-source automatic differentiation, Shuo-Hui Li for helpful discussion on differential geometry, Tong Liu and An-Qi Chen for helpful discussion on quantum adders and multipliers, Ying-Bo Ma for correcting typos by submitting pull requests, Chris Rackauckas for helpful discussion on reversible integrator, Mike Innes for reviewing the comments about Zygote, Jun Takahashi for discussion about the graph embedding problem, Simon Byrne and Chen Zhao for helpful discussion on floating-point and logarithmic numbers. The authors are supported by the National Natural Science Foundation of China under Grant No. 11774398, the Strategic Priority Research Program of Chinese Academy of Sciences Grant No. XDB28000000.

References

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

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).

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] .

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.

J.-G. L. Hao Xie and L. Wang, arXiv:2001.04121 .

G. H. Golub and C. F. Van Loan, *Matrix computations*, Vol. 3 (JHU press, 2012).

H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, Physical Review X **9** (2019), 10.1103/physrevx.9.031041.

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

Z.-Q. Wan and S.-X. Zhang, "Automatic differentiation for complex valued svd," (2019), arXiv:1909.02659 [math.NA] .

C. Hubig, "Use and implementation of autodifferentiation in tensor network methods with complex scalars," (2019), arXiv:1907.13422 [cond-mat.str-el] .

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] .

M. Innes, "Don't unroll adjoint: Differentiating ssa-form programs," (2018), arXiv:1810.07951 [cs.PL] .

M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, CoRR **abs/1907.07587** (2019), arXiv:1907.07587 .

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

M. P. Frank, IEEE Spectrum **54**, 32–37 (2017).

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

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

I. Lanese, N. Nishida, A. Palacios, and G. Vidal, Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018).

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

M. P. Frank and T. F. Knight Jr, *Reversibility for efficient computing*, Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and ... (1999).

J. G. Koller and W. C. Athas, in *Workshop on Physics and Computation* (1992) pp. 267–270.

R. C. Merkle, R. A. Freitas, T. Hogg, T. E. Moore, M. S. Moses, and J. Ryley, Journal of Mechanisms and Robotics **10** (2018), 10.1115/1.4041209.

K. Likharev, IEEE Transactions on Magnetics **13**, 242 (1977).

V. K. Semenov, G. V. Danilov, and D. V. Averin, IEEE Transactions on Applied Superconductivity **13**, 938 (2003).

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

I. Hänninen, G. Snider, and C. Lent, "Adiabatic cmos: Limits of reversible energy recovery and first steps for design automation," (2014) pp. 1–20.

E. P. DeBenedictis, J. K. Mee, and M. P. Frank, Computer **50**, 76 (2017).

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

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

C. H. Bennett (1973).

D. Speyer and B. Sturmfels, Mathematics Magazine **82**, 163 (2009).

J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, arXiv preprint arXiv:1209.5145 (2012).

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM Review **59**, 65–98 (2017).

A. McInerney, *First steps in differential geometry* (Springer, 2015).

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

T. Besard, C. Foket, and B. D. Sutter, CoRR **abs/1712.03112** (2017), arXiv:1712.03112 .

J. Takahashi and A. W. Sandvik, "Valence-bond solids, vestigial order, and emergent so(5) symmetry in a two-dimensional quantum magnet," (2020), arXiv:2001.10045 [cond-mat.str-el] .

D. P. Kingma and J. Ba, arXiv:1412.6980 .

F. Srajer, Z. Kukelova, and A. Fitzgibbon, Optimization Methods and Software **33**, 889 (2018).

C. H. Bennett, SIAM Journal on Computing **18**, 766 (1989).

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

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

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

M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," (2002).

L. Ruiz-Perez and J. C. Garcia-Escartin, Quantum Information Processing **16** (2017), 10.1007/s11128-017-1603-1.

K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, and A. Paz, Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018 (2018), 10.1145/3183895.3183901.

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.

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.

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

D. Yuret (2016).

I. Kobyzev, S. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," (2019), arXiv:1908.09257 [stat.ML] .

D. P. Kingma and P. Dhariwal, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 10215–10224.

L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real nvp," (2016), arXiv:1605.08803 [cs.LG] .

S.-H. Li and L. Wang, Physical Review Letters **121** (2018), 10.1103/physrevlett.121.260601.

P. Hut, J. Makino, and S. McMillan, The Astrophysical Journal **443**, L93 (1995).

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

M. Giftthaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, Advanced Robotics **31**, 1225–1237 (2017).

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

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

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] .

T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, "Quantum circuits for floating-point arithmetic," (2018), arXiv:1807.02023 [quant-ph] .

F. J. Taylor, R. Gill, J. Joseph, and J. Radke, IEEE Transactions on Computers **37**, 190 (1988).

M. P. Frank, in *Proceedings of the 2nd Conference on Computing Frontiers* (2005) pp. 385–390.

A NiLang Grammar

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

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

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

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

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

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

⟨Stmt⟩ ::= ⟨BlockStmt⟩
| ⟨IfStmt⟩
| ⟨WhileStmt⟩
| ⟨ForStmt⟩
| ⟨InstrStmt⟩
| ⟨RevStmt⟩
| ⟨AncillaStmt⟩
| ⟨TypecastStmt⟩
| ⟨@routine⟩ ⟨Stmt⟩
| ⟨@safe⟩ *JuliaExpr*
| ⟨CallStmt⟩

⟨BlockStmt⟩ ::= *begin* ⟨Stmts⟩ *end*

⟨RevCond⟩ ::= *(JuliaExpr , JuliaExpr)*

⟨IfStmt⟩ ::= *if* ⟨RevCond⟩ ⟨Stmts⟩ [*else* ⟨Stmts⟩] *end*

⟨WhileStmt⟩ ::= *while* ⟨RevCond⟩ ⟨Stmts⟩ *end*

⟨Range⟩ ::= *JuliaExpr : JuliaExpr* [*: JuliaExpr*]

⟨ForStmt⟩ ::= *for ident* = ⟨Range⟩ ⟨Stmts⟩ *end*

⟨KwArg⟩ ::= *ident* = *JuliaExpr*

⟨KwArgs⟩ ::= [⟨KwArgs⟩ ,] ⟨KwArg⟩

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

⟨Constant⟩ ::= *num* | π | *true* | *false*

⟨InstrBinOp⟩ ::= += | -= | $\veebar$=

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

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

⟨RevStmt⟩ ::= ~ ⟨Stmt⟩

⟨AncillaStmt⟩ ::= *ident* ← *JuliaExpr*
| *ident* → *JuliaExpr*

⟨TypecastStmt⟩ ::= *(JuliaExpr => JuliaExpr) (ident)*

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

⟨@safe⟩ ::= *@safe JuliaExpr*

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

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

B A table of instructions

A table instructions used in the main text

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 += $a * b$	$y + a * b, a, b$
y += a/b	$y + a/b, a, b$
y += a^b	$y + a^b, a, b$
y += identity(x)	$y + x, x$
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)	$y + \|x\|, x$
NEG(y)	$-y$
INC(y)	$y + 1$
DEC(y)	$y - 1$

Table 7: Predefined reversible instructions in NiLang.

C Reversible Constructors

So far, the language design is not too different from a traditional reversible language. To port Julia's type system better, we introduce dataviews. The type used in the reversible context is just a standard Julia type with an additional requirement of having reversible constructors. The inverse of a constructor is called a "destructor", which unpacks data and deallocates derived fields. A reversible constructor is implemented by reinterpreting the `new` function in Julia. Let us consider the following statement.

```
x ← new{TX, TG}(x, g)
```

The above statement is similar to allocating an ancilla, except that it deallocates `g` directly at the same time. Doing this is proper because `new` is special that its output keeps all information of its arguments. All input variables that do not appear in the output can be discarded safely. Its inverse is

```
x → new{TX, TG}(x, g)
```

It unpacks structure `x` and assigns fields to corresponding variables in the argument list. The following example shows a non-complete definition of the reversible type `GVar`.

```
julia> using NiLangCore

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

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

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

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

`GVar` has two fields that correspond to the value and gradient of a variable. Here, we put `@i` macro before both `struct` and `function` statements. The ones before functions generate forward and backward functions, while the one before `struct` moves `~GVar` functions to the outside of the type definition. Otherwise, the inverse function will be ignored by Julia compiler.

Since an operation changes data inplace in NiLang, a field of an immutable instance should also be " modifiable". Let us first consider the following example.

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

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

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

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

In Julia language, the assign statement above will throw a syntax error because the function call "-" can not be assigned, and `GVar` is an immutable type. In NiLang, we use the macro `@assignback` to modify an immutable data directly. It translates the above statement to

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

The first line `PlusEq(*)(-arr[3].g, x, y)` computes the output as a tuple of length 3. At lines 2-3, `chfield(x, Val{:g}, val)` modifies the `g` field of `x` and `chfield(x, -, res[1])` returns `-res[1]`. Here, modifying a field requires the default constructor of a type not overwritten. The assignments in lines 4 and 5 are straightforward. We call a bijection of a field of an object a "dataview" of this object, and it is directly modifiable in NiLang. The definition of dataview can be found in Appendix A.