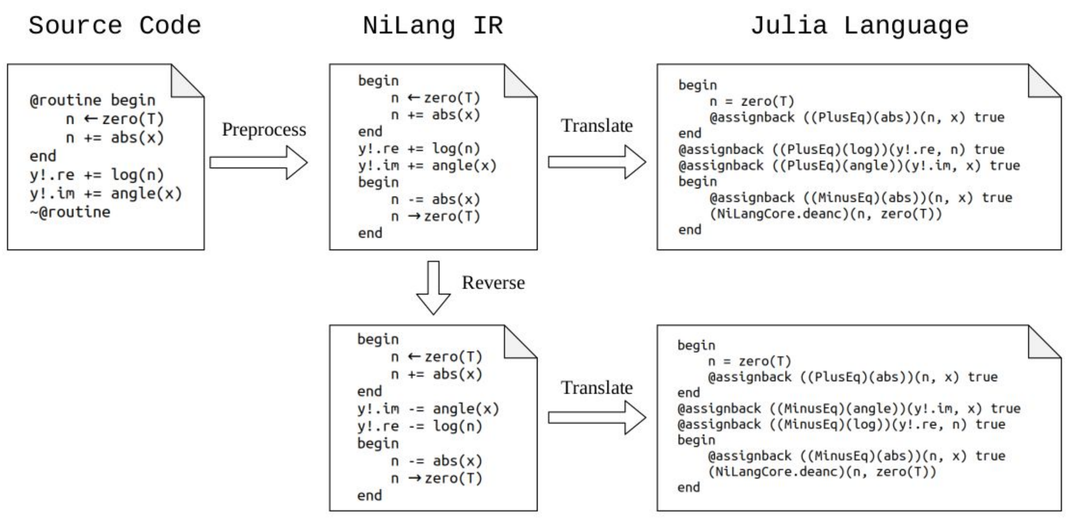
NiLang.jl - Reversible computing for future

Jin-Guo Liu (Institute of Physics, Chinese Academy of Sciences, Beijing 100190, China)
Taine Zhao (Department of Computer Science, University of Tsukuba)

What is reversible computing?

A computing style that does not erase information, hence supports bidirectional execution.



In NiLang.jl, a reversible function is defined like the "Source Code" on the left, basic building blocks are

- arithmetic instructions: `a += b`
- memory allocation: `n ← zero(T)`
- computing-uncomputing macro: `@routine` and `~@routine`

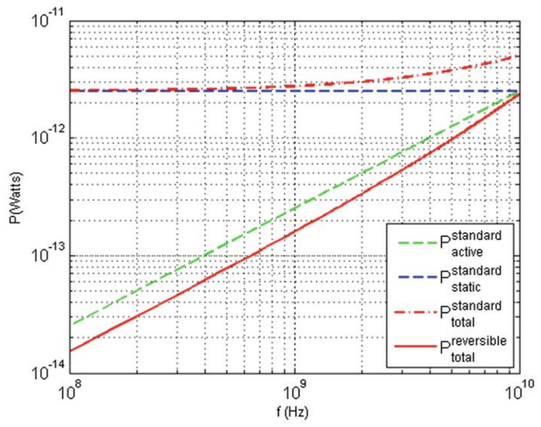
The compiler first transform the source code to a **reversible intermediate representation (IR)** by unrolling the `@routine` macro and other eye candies.

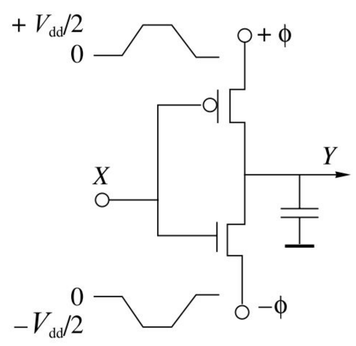
Then we can generate the inverse program based on this IR.

This IR can be transformed to native Julia language directly.

Why do we need reversible computing?

1. It is energy efficient,





(left) Power dissipation for standard and reversible CMOS logic. (SPICE simulation). (right) An adiabatic inverter.
Hänninen, Ismo, Gregory L. Snider, and Craig S. Lent. 2014.

A huge amount of energy dissipation is from discarding signal energy. Adiabatic CMOS turns on and off the signal voltage slowly (or adiabatically), and store the signal energy in a capacitance so that the signal energy can be recovered during uncomputing. It was proven that **adiabatic logic requires reversibility**.

The connection between reversibility and energy efficiency is universal by the **Landauer's principle**. When one erase a bit information, there is a minimum amount of energy dissipation.

Classical reversible computing has came to the winter more than a decade ago, because **people believe the overhead of making a program reversible is not acceptable** in practise. It is not true when it comes to automatic differentiation, where "not discarding intermediate information" is the minimum requirement of backpropagation.

Nowadays, 2.5% of global energy is used for computing, while an increasing amount of them are for AI industry.

2. It supported machine instruction level automatic differentiation.

```
res ← 0.0
y ← 0.0
for i = 1:N
    y += x[i] ^ 2
end
res += sqrt(y)
```

1. Write the program in a reversible style.

```
res -= sqrt(y)
for i = N:-1:1
    y -= x[i] ^ 2
end
y → 0.0
res → 0.0
```

2. Reverse the program.

```
∂res ← 1.0
∂y ← 0.0
res -= sqrt(y)
∂y += ∂res/2/sqrt(y)
for i = N:-1:1
    y -= x[i] ^ 2
    ∂x[i] += 2 * x[i] * ∂y
end
y → 0.0
res → 0.0
```

3. Insert gradient codes.

Comparing with traditional computational graph based approach

Advantages

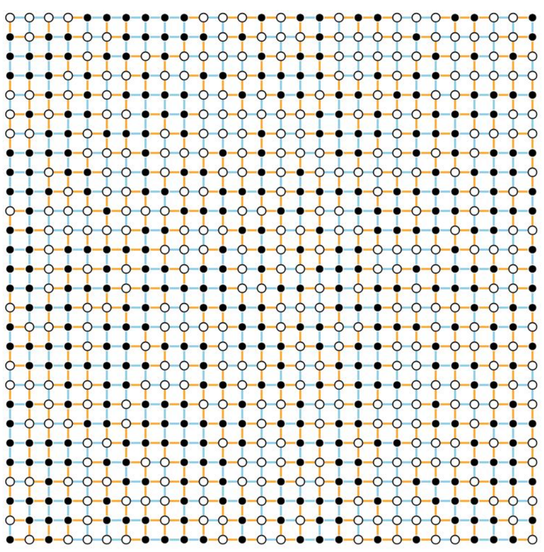
1. Does not slow down the program by accessing cached data in the stack.
2. Friendly to **GPUs**.
3. Inplace functions are supported.

Disadvantages

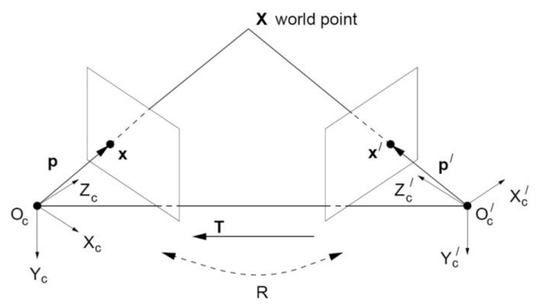
1. Most people are not familiar about writing reversible code.
2. Writing a program reversibly may also cost overheads.

Applications of reversible programming

Application 1. Finding the optimal configuration of a 28 x 28 spinglass problem



Application 2. Obtaining the jacobian for bundle adjustment



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

Play with NiLang.jl

Accelerate the gradient function of squared norm

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

julia> function norm2(x::AbstractArray{T}) where T
           out = zero(T)
           for i=1:length(x)
               @inbounds out += x[i]^2
           end
           return out
       end
norm2 (generic function with 1 method)

julia> x = randn(1000);

# obtaining the gradients with Zygote's magic prime
julia> original_grad = norm2'(x);

# however, it is slow
julia> @btime norm2'($x) seconds=1;
  3.240 ms (19091 allocations: 15.99 MiB)

julia> @btime norm2($x) seconds=1;
  1.563 μs (0 allocations: 0 bytes)

# the reversible version, define the function with `@i`
julia> @i function r_norm2(out::T, x::AbstractArray{T}) where T
           for i=1:length(x)
               @inbounds out += x[i]^2
           end
       end

# `GVar` wraps the output variables with gradient fields. When an
instruction meets a `GVar` type, it inserts the gradient codes to
that place.
julia> @btime (~r_norm2)(GVar($(norm2(x)), 1.0), $(GVar(x)))
seconds=1;
  1.597 μs (0 allocations: 0 bytes)

# customize the gradient of the `norm2` function with the one
generated by NiLang.
julia> Zygote.@adjoint function norm2(x::AbstractArray{T}) where
T
           out = norm2(x)
           out, δy -> (grad((~r_norm2)(GVar(out, δy),
GVar(x))[2]),)
       end

julia> @assert norm2'(x) ≈ original_grad

# you see, faster!
julia> @btime norm2'(x) seconds=1;
  3.714 μs (2 allocations: 23.69 KiB)
```

Inspect the compiler of NiLang

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

# execute a single statement with `@instr`
julia> MacroTools.prettify(@macroexpand @instr x[2] += y)
quote
    mandrill = wrap_tuple(((PlusEq)(identity))(x[2], y))
    x[2] = mandrill[1]
    y = mandrill[2]
end

# the `if` statement contains a precondition and a post
condition
julia> MacroTools.prettify(@macroexpand @instr if (x < 0,
x > 0) NEG(x) end)
quote
    mandrill = x < 0
    if mandrill
        hawk = wrap_tuple(NEG(x))
        x = hawk[1]
    end
    if !(mandrill === (x > 0) || almost_same(mandrill, x
> 0))
        throw(InvertibilityError("$(Symbol("##364"))
(=$(mandrill)) ≮ $($(QuoteNode(:(x > 0)))) (=$(x > 0))"))
    end
end

# the inverse (~) of an `if` statement
julia> MacroTools.prettify(@macroexpand @instr ~(if (x <
0, x > 0) NEG(x) end))
quote
    mandrill = x > 0
    if mandrill
        hawk = wrap_tuple((~NEG)(x))
        x = hawk[1]
    end
    if !(mandrill === (x < 0) || almost_same(mandrill, x
< 0))
        throw(InvertibilityError("$(Symbol("##366"))
(=$(mandrill)) ≮ $($(QuoteNode(:(x < 0)))) (=$(x < 0))"))
    end
end

# the inverse of the `for` statement, use `@invcheckoff`
to turn off invertibility check.
julia> MacroTools.prettify(@macroexpand @instr
@invcheckoff ~(for i=1:10 f(x, y) end))
:(for mandrill = 1:10
    i = (10 - mandrill) + 1
    hawk = wrap_tuple((~f)(x, y))
    x = hawk[1]
    y = hawk[2]
end)
```

← Documentation

Advertisement: QuEra computing is hiring Julia programmers