Function is Class, and Call Frame is Instance. An Introduction to ago Programming Language

Ming Xu

Ming Xu, Independent researcher, inshua@gmail.com

Abstract

Through systematic analysis of technologies such as animation systems, coroutines, and workflow engines, this paper identifies that they essentially attempt to formally represent "actions" in the real world. Traditional programming languages face significant limitations when describing persistent, asynchronous actions that can be suspended by external events; and this limitation is overcome by re-conceptualising the Call Frame. By extracting CallFrame from low-level machine mechanisms into an object-oriented viewpoint, I find that an objectified CallFrame can serve as a complete representation of an action, it leads to the novel proposition: "Function is Class, and CallFrame is Instance.". Based on this idea, I design and implement the ago programming language (named after the Esperanto word Action). Its core idea treats actions as resumable, persistable CallFrames, with prototype verification completed in Java runtime. This paper elaborates on ago's architecture, and key designs including parameterized classes, type scalarization, extended boxing, and call-time overloading, and provides reference solutions for several cases.

1 INTRODUCTION

In 2022, while reviewing technologies such as animation frameworks, coroutines, and workflow engines, I realized that they all essentially aim to formalise the concept of an "*action*" in the real world. Through deeper reflection, I discovered a serious shortfall in traditional programming languages when it try to modeling real-world actions. The breakthrough lies precisely in rethinking Call Frame semantics. By extracting CallFrame[1] from low-level stack mechanics into an object-oriented perspective, I found that an objectified CallFrame can serve as a complete representation of an Action. This observation led to the proposition:"**Function is Class, CallFrame is Instance"**. Based on this principle, I created a new object-oriented programming language - **ago** (derived from Esperanto's Action). The compiler and runtime source code are available at http://github.com/inshua/ago.

In a programming model based on the Turing machine, a program can be viewed as a continuous sequence of state transitions, and it is exactly the function calls are carrying those transitions out.

An idealised Turing machine assumes that a function call is an instantaneous, indivisible atomic evaluation; from the perspective of a mathematical function its execution contains no temporal dimension, and we cannot discuss how long it takes to compute cos(x) tan(x) - they're logical activities. In reality, however, every state transition unfolds over time: it may have to wait for external events, gather information, and process results. The traditional Turing-machine model treats function call as an indivisible, uninterruptible process, which means a single function cannot represent an asynchronous and persistable action with interruptions (e.g., bank transfers, approval workflows, or a game sprite jumping from point A to point B).

From a philosophical standpoint, this shortcoming is unsurprising. Aristotle's conception of motion treats **kinesis** as a transient transitional state, emphasizing the endpoint while overlooking the process; it focuses on results rather than continuous dynamics. In Buddhist philosophy, the concept of *Saṅkhāra(行) [3]* regards movement as the primary reality; similarly, Whitehead's Process Philosophy[4] holds that the universe is composed of interrelated events and processes, each process has complete life cycles—from emergence to demise. These divergent perspectives inspire how we perceive actions with temporal properties in the real world. As an object-oriented programming language ago remains rooted primarily in the Platonic–Aristotelian framework, but also incorporates aspects of process philosophy.

Based on these reflections, this paper proposes the design principle "Function is Class, CallFrame is Instance" and realizes it in the ago language, providing a new perspective and tool for object-oriented programming.

2 EXISTING SOLUTIONS ABOUT ACTIONS

Now I'll show the cocos2d game engine, coroutine, and workflow engine as examples to examine how traditional programming languages represent **actions**. Game worlds are often seen as simulations of the real world. Their core mechanism lies in the computational modeling of continuous spatio-temporal behaviors; coroutines, on the other hand, emerged from real-world scenarios that frequently handle asynchronous I/O, differing from the pure functions advocated by functional programming. Workflow engine was created to solve cross-service, multi-step business process. The commonalities and differences among these three technologies in representing Actions provide a valuable problem domain for re-examining traditional function-call models.

2.1 The Cocos2d Action System

The cocos2d game framework models animation actions through an *Action* class hierarchy. Its subclasses (*MoveTo*, *MoveBy*, *Flip*, *Rotate*, *ScaleTo*, *ScaleBy*, etc.) correspond to different movement types[6]; developers can combine these basic actions into more complex animation with *Sequence* and *Parallel* actions, those still were subclasses of *Action*. For example:

```
MoveTo(800, 320, 2) # Move at constant speed to (800,320) over 2 seconds
ScaleTo(2, 2) # Scale up by a factor of 2 over 2 seconds
Sequence(MoveTo(...), ScaleTo(...)) # First move then scale; total duration 4 seconds
Parallel(MoveTo(...), ScaleTo(...)) # Two actions performed in parallel
```

Cocos2d also supports binding any function as an *Action* (*CallAction(lambda)*), allowing business logic to intertwine with animation without callbacks. Game sprites perform sequences of these actions to "act out" stories.

The core code of the *Action* base class looks like this:

```
class Action(object):
    def __init__(self, *args, **kwargs):
        self.init(*args, **kwargs)
        self.target = None # The performer bound to the action; a CocosNode target (its main subclass is the sprite class)
        self._elapsed = None
```

```
    def init(self): pass
    def start(self): pass
    def stop(self): pass

    def step(self, dt):
        """
        Gets called every frame. `dt` is the number of seconds that elapsed
        since the last call. If there was a pause and resume in the middle,
        the actual elapsed time may be bigger.
        """
        ...
```

cocos Basic Action. (https://github.com/los-cocos/cocos/blob/master/cocos/actions/base_actions.py)

Cocos2d encapsulates *Action* as a core class, it repeatedly invokes *step(dt)* to update its state for each animation frame. For instance, inside a *MoveTo* action, the *step* function calculates the current position based on elapsed time and writes it back to the target cocos node, thereby achieving movement animation.

This design of treating actions as particle-like objects and composing them can well correspond to activity diagrams in UML—thus, activity diagrams are often transformed into a big state machine for implementation. The Cocos2d Action architecture provides intuitive and composable semantics for animation, but still falls short when dealing with complex business processes involving flow branches (e.g., expressing *if*-like semantics within the Action framework proves challenging).

2.2 Coroutines (CPS) and Actions

In concurrent programming, coroutines are often used to describe actions that require multiple discontinuous stages of execution. Here is a typical example in Kotlin:

```
fun main() = runBlocking {
    launch {
        delay(1000)
        println(", coroutines!")
    }
    print("Hello")
}
// Hello, coroutines!
```

Kotlin implements coroutines on the JVM in CPS (Continuation-Passing Style)[8]. The compiler transforms the *launch* block into an anonymous class inherited from *kotlin.coroutines.Continuation*; this class internally maintains an

integer field *label* to mark the current execution point and preserves the coroutine's local state between each suspension and resumption.

The core pseudo-code is as follows:

```
when (label) {
    0 -> {            // Initial entry
        delay(1000)   // This point will generate a suspension
        label = 1     // Mark the next stage
        return SUSPENDED
    }
    1 -> {            // Execute after resuming
        println(", coroutines!")
        return Unit
    }
}
```

The above code shares similarities with the *step* function in Cocos2d's *Action* class: it repeatedly calls the same subroutine to update state at current state. The difference is that a coroutine's *Continuation* is completely opaque to developers—it is auto-generated, anonymous, and not directly accessible.

Other languages also have their own coroutine implementations: C# has *Task<T>/async–await*, Python has *asyncio.coroutine/await*, JavaScript has *Promise*, etc., each encapsulating suspension and resumption as continous objects in different ways. Coroutines in Go, Erlang[5], and other languages differ further; we won't elaborate here. These languages often provide syntactic sugar for coroutines that make them feel like ordinary function calls and integrate seamlessly with mainstream language features. However, at their core they are still just wrappers around a single function body—they lack lifecycle operations (e.g., *start*, *stop*, *cancel*) akin to Cocos Actions. Furthermore, coroutine subclasses cannot be derived or extended to add custom methods. A bigger issue is that they run entirely in memory, preventing them from truly handling real-world business scenarios.[2]

2.3 Workflow Engines and Actions

Workflow engines typically abstract business processes as Directed Acyclic Graphs (DAGs), with each node representing an **activity**. During execution, the current state is persisted for recovery upon subsequent triggers. Workflow eingines excels at declaring cross-service collaboration: edges represent event-driven triggers, while nodes encapsulate specific business units implemented in programming languages.[10] However, workflow engines are not a compile/interpretation technology—the code of workflow engine consists only of declarations of nodes and edges, lacking modularity or reuse principles from software engineering.

Compared with workflow engines, using coroutines instead of a workflow engine to model business actions (such as a leave-approval chain) would lead to serious issues. Although CSP coroutines are also objects, they parasitize the function call stack; their suspension points, resumption contexts, and related information must reside in memory, which is costly. Moreover, if the program crashes, all unfinished coroutine states are discarded, causing the business process to

be lost. This means that coroutines cannot faithfully represent real-world actions, even though they may perform well in I/O domains.

Moreover, other technologies such as work queues and event buses can also provide similar workflow engine functionality, leading to analogous conclusions.

3 FUNCTION IS CLASS, AND CALL FRAME IS INSTANCE

If program language functions could be used to formalise real-world actions, many limitations discussed above—such as lack of a temporal dimension, inability to suspend and resume, difficulty with persistence—can be solved uniformly and fundamentally. I decided to treat CallFrame as Object; once this premise is established, what is function? Function must be class! and CallFrames become instances of function classes. functions are no longer stateless mathematical mappings in traditional terms but computational entities with lifecycle semantics, state management capabilities, and closure characteristics.

From the perspective of "Function is Class, CallFrame is Instance", the following conclusions can be drawn:

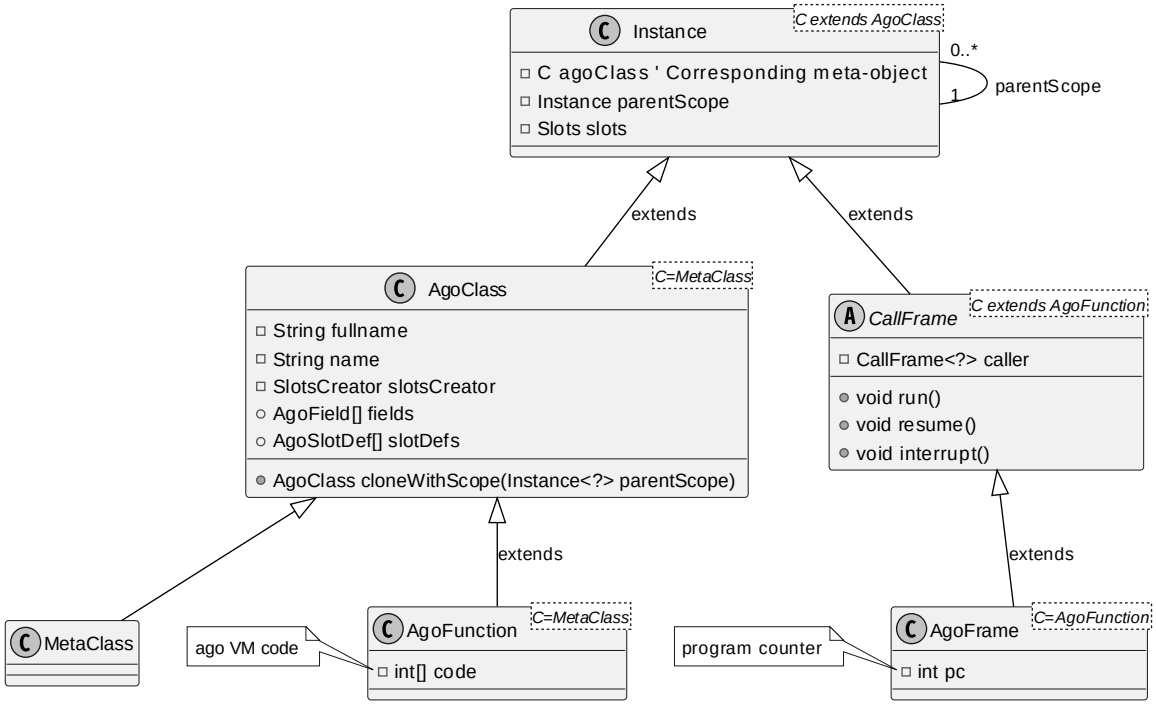
1. Object-oriented programming (OOP) theory achieves completion. Traditionally, OOP and functional programming (FP) are difficult to reconcile; by elevating functions to classes, functions have types in OO world now, therefore OOP and FP are no longer exclusive paths but instead unified within a single paradigm, thereby eliminating the debate over which is first-class.[11]
2. Unified concept of closures. Whether a function or a class, both can possess their own contextual environment. Internally defined sub-functions are semantically equivalent to methods of the class. Similarly, functions can declare fields and methods while integrating into hierarchical object-oriented structures.
3. Scope unified as Object. Since function instances (CallFrame) are also objects, both the scope of a classes and scope of functions are regarded as Object. For classes, include functions; within a class the scope's type is Object, and within a function the scope's type is CallFrame—both of which are Objects.
4. Interface and Trait extended to functions. After becoming classes, functions can implement interfaces or traits just like classes, got the same feature of composability.
5. CallFrames serve as coroutines naturally. An objectified CallFrame satisfies the definition of a "continuous object"—it can be suspended, resumed at any time, and carries a complete execution context with all coroutine features, making it capable of acting as a coroutine directly. By placing the CallFrame's execution environment on an event loop or virtual thread, lightweight concurrent programming is achieved. Moreover, after decoupling from stack structure, a CallFrame can even be deployed on distributed executors to run across nodes and services naturally, achieving semantic-level distribution of actions.
6. Unified storage mechanism. Since CallFrames share the same memory layout as ordinary objects, they can be stored directly in any medium that supports object serialization—including binary files, relational databases, NoSQL databases, and even distributed key-value stores or blockchains.
7. One-to-one mapping from action to function. A business Action can be fully described by a single function instance; the function may nest multiple sub-action sequences and, when necessary, wait for external events before continuing execution. Any real-world multi-stage action—such as bank transfers, approval flows, game animations, or distributed transaction handling—can be modeled with a single function. Ordinary code suffices to express cross-service collaboration processes.

8. Unified generics. Function generics and class generics are unified, sharing the same semantic system and eliminating redundancy in generic handling.

Based on these insights, I implemented a new language named **ago** in Java.

4 AGO LANGUAGE CORE

Below is the UML diagram of core class structures in the ago runtime (Java implementation).



Explanation

- ***Instance*** is the base class for all objects (including ordinary objects, *CallFrames*, ordinary classes, functions, and meta-classes). It holds a reference to its own class (*agoClass*), a collection of slots *slots*, and a scope, thereby allowing any object to possess closure capabilities. It's mapped to *lang.Object* in ago language.
- ***AgoClass***. Ordinary ago classes, derives from *Instance*; Ordinary ago class is the instance of the meta-class *MetaClass*
- ***Function classes*** derive from *AgoClass* and possess their own *code* attribute, which stores the compiled ago VM instruction sequence.
- ***CallFrame*** describes the runtime frame for function calls. It holds a reference to its parent frame (*caller*) and provides methods such as *run*, *resume*, and *interrupt*.
- ***AgoFrame*** is a subclass of *CallFrame*; besides basic fields it also maintains a program counter *pc* used to interpret VM instructions one by one.

All objects (including *Instance*, *CallFrame*, and *AgoClass*, etc.) read and write state by implementing the *Slots* interface. Below is a piece of the definition of *Slots*:

```
public interface Slots {
    default int getInt(int slot) { /* ... */ }
    default long getLong(int slot) { /* ... */ }
    default void setInt(int slot, int value) { /* ... */ }
    default void setLong(int slot, long value) { /* ... */ }
    default Instance<?> getObject(int slot) { /* ... */ }
    default void setObject(int slot, Instance<?> value) { /* ... */ }
}
```

Slots.java (https://github.com/inshua/ago/blob/master/ago-engine/src/main/java/org/siphonlab/ago/Slots.java)

The runtime dynamically generates an anonymous *Slots* implementation for each ago class. For example, a class with two integer slots would generate the following code:

```
class SlotsImpl implements Slots {
    private int a_0;
    private int b_1;

    @Override
    public int getInt(int slot) {
        switch(slot){
            case 0: return a_0;
            case 1: return b_1;
           default: throw new IllegalArgumentException("Unsupported slot access: " + slot);
        }
    }

    @Override
    public void setInt(int slot, int value) {
        switch(slot){
            case 0: a_0 = value; break;
            case 1: b_1 = value; break;
           default: throw new IllegalArgumentException("Unsupported slot access: " + slot);
        }
```

```
    }
}
```

CallFrame shares the same *Slots* structure as ordinary objects. In ago's runtime, all frames are allocated on the heap rather than relying on stack space, simplifying memory layout. Based on this unified slot model, a register-based VM instruction set was designed; ago code will be compiled to corresponding instruction sequences.[7]

Slots is a Java interface that Java developers can implement differently for various underlying storage backends. Leveraging this feature, I created a *Slots* implementation that writes object state directly to PostgreSQL's JSON column. Objects (including *CallFrame*) are loaded from the database o demanded; upon completion or suspension they are written back and released; it's a effective alternative to traditional workflow engines.

Additionally, I attempted to map ago vm instructions directly to SQL statements—for example, a *mov* instruction corresponds to an *UPDATE* operation, allowing *Slots* to be updated without being loaded into memory. A lightweight ORM-style *Slots* implementation was also created: each class maps to a table and slots map to columns; the ORM supports both cached and uncached modes (details omitted).

It is conceivable that when *Slots* are stored in a database, it is even possible to inject transaction semantics into *CallFrame*, making ago a *persistence language*.[13]

ago introduces a container called ***RunSpace*** that runs *CallFrames*, performing the work traditionally handled by threads. The responsibilities of *RunSpace* include tracking the active frame and driving its execution, while providing a shared result store for all frames within the same space. Unlike traditional threads, RunSpace does not need to maintain a full call stack—CallFrame objects keep the call chain via the *caller* field. RunSpace only holds a reference to the currently active frame and continues with the next one after it exits, until its state is 'exit' or 'suspend' or no more active call frame.

In implementation, the *RunSpace* class implements the *Runnable* interface, so it can be submitted directly to any Java executor (thread pool, event loop such as Netty's EventLoop, virtual threads, or a custom scheduler). Its core code is:

```
class RunSpace implements Runnable{
    protected CallFrame<?> currCallFrame;
    protected Instance<?> exception;
      protected ResultSlots resultSlots = new ResultSlots(); // share result store for
callframes in this RunSpace

    protected byte runningState = RunningState.PENDING;

    public void run() {
        this.setRunningState(RunningState.RUNNING);
        while (this.currCallFrame != null &&
            !RunningState.isPausingOrWaitingResult(this.getRunningState())) {
                this.currCallFrame.run(); // execute VM instructions of AgoFrame, or
native functions for NativeFrame.
```

```
        }
        tryComplete();
    }
}
```

RunSpace.java. (https://github.com/inshua/ago/blob/master/ago-engine/src/main/java/org/siphonlab/ago/RunSpace.java#L139)

The *run* method of *RunSpace* iteratively calls the current frame's *run()*, continuing until the current frame become null, or complete or suspended, after complete cleanup logic is executed, and wait for restore when suspended, however, this time *run* over.

4.1 Function Calls in the ago Language

In ago, function calls retain the same syntax as traditional programming languages: *f(arguments)*. However, from an implementation standpoint this is actually split into two steps: first a *CallFrame* object is created, then that *CallFrame* is e xecuted. This can be expressed with the following two statements:

```
var t = new f(arguments)    // Create a CallFrame object and store it in slot t
t()                         // Execute that CallFrame
```

Now, *f(arguments)* can be regarded as syntactic sugar for the creator and invoker statements above. When direct access to the CallFrame object is needed, these steps can be still separated alone.

Below is an example of the compiled VM instruction sequence for the simplest addition function:

```
fun add(a as int, b as int) as int{
    return a + b
}
fun main(args as string){
     var t = add(5, 4)
     Trace.print(t)
}
```

main compiles to the following VM code:

```
// statement: var t = add(5, 4)

     0    new_vC    2,15                  // create instance of `add`, put to slot 2, 15 is
class id of `add`
    3    const_fld_i_ovc    2,0,5        // set parameter 1 of `add` to 5, slot 0
    7    const_fld_i_ovc    2,1,4        // set parameter 2 of `add` to 4, slot 1
```

```
    11  invoke_v        2               // invoke slot 2. runspace.currCallFrame set to instance of `add`, and `main.run` yields, so that the runspace shifts to the add callframe and calls `add.run()` to evaluate the opcode of `add`
    13  accept_i_v      1              // after `add` is invoked, the runSpace will shift back to `main` callframe and resume `main.run` from here; this instruction copies the result from `runspace.resultSlots` into slot 2 of the main callframe (`slots.setInt(1, runspace.resultSlots.getInt())`)
```

The *add* function compiles to:

```
// statement: var t = add(5, 4)

    0   add_i_vvv   2,0,1          // invoke add instruction; result placed in slot 2
    4   return_i_v  2              // runspace.resultSlots.setInt(slots.getInt(2)), and set `this.caller.runspace.currFrame` to `this.caller`; now the runspace's current frame restores to the instance of `main`
```

As can be seen, invoking functions is implemented by iterating over *RunSpace.currCallframe*. When *main* calls *add*, the main callframe switches *currCallframe* to the *add* callframe when evaluate instruction *invoke_v*; at this point the *main* callframe is suspended but retained in memory. After *add* completes, it writes its result into *runspace.resultSlots* and restores *currCallframe* to the caller, *main* callframe; then the *main* callframe executes the *accept_i_v* instruction, copying the call result back into its slots.

The entire process aligns with coroutine suspend/resume semantics: the caller yields, activates the callee; upon the callee's exit, the caller resumes from where it left off. The completeness of context preservation in CallFrames enables this coroutines-like behavior.

4.2 Structured Concurrency in ago

From dissecting the function-call mechanism we've seen that ago's CallFrame embodies coroutine features, and ago's function-call flow already leverages these features. The following examples of *NativeFrame* and the *await* statement will show how ago to implement structured concurrency with corountine-like call frame.

4.2.1 NativeFrame and Callback Adaptation

NativeFrame is a subclass of *CallFrame* that wraps Java native code; besides wrapping synchronous Java methods it also supports wrapping callback-driven asynchronous APIs.

Example: the *sleep* function implementation for ago language:

```
// java implementation for ago function `fun sleep(milliseconds as int) native "org.siphonlab.ago.lang.RunSpaceAware.sleep"`
// invoke ago functon 'sleep', i.e. sleep(2000), creates a NativeFrame, and the `NativeFrame.run()` enter this Java method

public static void sleep(NativeFrame nativeFrame, int millisecond) {
    var runSpaceHost = nativeFrame.getRunSpace().getRunSpaceHost();
    nativeFrame.beginAsync(); // Set runspace state to WAITING_RESULT
            runSpaceHost.setTimer(millisecond, nativeFrame::finishVoidAsync); // Setup callback; when finishVoidAsync is called, runspace.run() will resume execution by restoring 'nativeFrame.caller' and runspace
     // After NativeFrame.run finishes, runspace state becomes WAITING_RESULT, temporarily exits(yield)
}
```

sleep, RunSpaceAware.java(https://github.com/inshua/ago/blob/master/ago-engine/src/main/java/org/siphonlab/ago/lang/RunSpaceAware.java#L132)

Through *beginAsync* and *finishXxxAsync*, a callback-based Java async program is transformed into an ordinary ago function.

4.2.2 Fork, Suspend, and Resume

ago provides a parameterless *await* statement to explicitly yield control. The following code demonstrates how to combine *await* with *fork* to synchronize parent and child runspaces:

```
fun f() {
    await;          // Explicitly yield: RunSpace enters waiting state
}
fun main() {
    var c = fork f();   // Start a child RunSpace and obtain the CallFrame object
    sleep(2000)
```

```
    c.notify();         // Manually wake up the child RunSpace, resume execution
}
```

In this example, *fork f()* creates a new child runspace and returns its entry *CallFrame* to the caller. When *f* encounters the *await* statement it immediately suspends that child runspace; after *main* sleeps for 2 seconds, *c.notify()* signals the child to continue.

The runspace created by *fork* implements structured concurrency: when the parent runspace finishes its own work, it checks whether any child runspaces remain unfinished. If so, the parent automatically enters a waiting state until every child completes before proceeding.

4.2.3 race Example

Leveraging the above mechanisms, ago provides concurrency primitives such as *race()* and *awaitMany()*, etc. Below is complete *race* source code that returns the first completed result among all supplied functions and aborts remaining tasks immediately:

```
fun race<R>(functions as Function<R>...) as R{
    var result as R;
    var b = new AtomicBoolean()
    var self = fun.this;

    fun wrap(f as Function<R>){
        var r = f();
        if(b.compareAndSet(false, true)){
            result = r;
            for(var other in functions){
                if(other != f)
                    other.runspace.interrupt()
            }
            self.notify();
        }
    }
    for(var f in functions){
        fork wrap(f)
    }
    await;
    return result
}
```

race, runspace.ago(https://github.com/inshua/ago/blob/master/ago-sdk/src/lang/runspace.ago#L39)

This implementation uses a CAS-based concurrency mechanism (*AtomicBoolean*) to ensure only the first returned result is recorded, and it aborts other child tasks via *interrupt()*. The parent runspace automatically resumes after all children finish, finally returning the earliest result.

4.2.4 Semantic Equivalence Between Coroutines and ago CallFrame

The table below shows the semantic equivalence between ago's CallFrame and traditional coroutines:

Coroutine	ago CallFrame	Java Runtime
coroutine(V)	f(V)	cf = new f(V); cf.run()

yield	await	cf.state = PAUSE;
		exit cf.run();
		runspace.run exits

resume	cf.notify()	cf.runspace.currentFrame.resume(); runspace submits on executor again

Through this mapping, ago's CallFrame is semantically identical to a conventional coroutine; the difference lies only in the invocation style and host-runtime details. Because a CallFrame itself is an ordinary object, it can be serialized, persisted, or even transmitted across network, naturally supporting distributed concurrency and recoverable execution.

4.3 Meta-Class and Parameterized Class

ago provides a limited meta-class mechanism. In ago types you can declare a class's meta-class via a *metaclass{}* block inside the class body; the meta-class can define attributes and methods for the instance class. A meta-class itself is an anonymous class that developers cannot access directly ; it always placed as a top-level class and does not share scope with the instance class. In ago, a *metaclass* may nest another *metaclass*, supporting only two levels of meta-class structure from the instance class.

First-level meta-class can has a parameterized constructor for generating **parameterized classes**:

```
class VarChar from String with ValidationSupport{
    metaclass{
        fun new(field maxLength as int){}
    }
    fun validate() throws ValidationException {
        if(this.length() > maxLength) throw new ValidationException('...');
    }
}
class Person{
    name as VarChar::(200)
}
```

```
fun main(){
    var p = new Person();
     p.name = 'John'    // ago extends the boxing concept; any class that derived from the Boxer<T> class can be boxed.
                        // VarChar derives from String, which is a box type for string, so it is also a box class of string.
                       // Assigning to p.name therefore creates an action that boxes the value into a VarChar::(200) instance.
}
```

The *VarChar::(200)* shown above is a **parameterized class** of *VarChar*. Parameterized classes exist from compile time, so the constructor arguments of the meta-class are limited to constants. A parameterized class is not a template; it is an actual class and whose field *maxLength* belongs to the slots of class.

Interfaces and traits can also have meta-classes, thus supporting parameterization. Using a parameterized interface allows you to achieve an effect similar to annotations, and this implementation is entirely object-oriented. For example:

```
fun article(id as long) with Restful::('/article/{id}'){
}
```

Functions can also implement traits, enabling functions to access members of the trait.

```
fun someJob() with ProgressReporter::('report destination'){ // ProgressReporter is a trait
    for(var i = 0; i < 100; i++){
        this.updateProgress(i)  // invoke trait method
    }
}
```

Parameterized classes allow types to carry specific values, and the extended boxing concept greatly enriches type expressiveness, providing a more powerful tool for domain modeling and other scenarios.

4.4 Scalarizing Classe, Callbacks, and Generic in go

ago supports scalarizing classes; this basic type is called ***classref***. For example:

```
class Animal{}


fun main(){
    var c as classref = Animal
}
```

To give *classref* richer semantics, ago provides a set of special boxing classes. The core wrapper type is *ClassRef*, which derived from the generic *Boxer<classref>* class and internally holds an actual *AgoClass* instance when boxing. Its implementation is as follows:

```
// Box type for classref
class ClassRef from Boxer<classref>{
    protected final value as classref;
     protected final boxedClass as Class;        // the agoClass instance, value set when
boxing classref
    fun new(value as classref) {
        this.value = value;
    }
    final fun getBoxedClass() as Class {
        return boxedClass;
    }
}
```

ClassRef, lang.ago(https://github.com/inshua/ago/blob/master/ago-sdk/src/lang/lang.ago#L41)

ClassInterval is a subclass of *ClassRef*, specifically designed describe the upper/lower bounds of type scalars, thereby forming a "type interval". Its parameterized class can record these bounds:

```
class ClassInterval from ClassRef{
    metaclass{
        public final lBound as classref
        public final uBound as classref
        fun new(lBound as classref, uBound as classref){
            this.lBound = lBound
            this.uBound = uBound
        }
    }

    fun new(value as classref){
        this.value = value;
    }
}
```

ClassInterval, lang.ago(https://github.com/inshua/ago/blob/master/ago-sdk/src/lang/lang.ago#L57)

Now, the parameters of parameterized class *ClassInterval::(L, U)* is designed to carry the bound, and the instance of *ClassInterval::(L, U)* carrying the exact class as its boxing value.

Usually classes in ago has scope, I introduce *ScopedClassInterval*, which extends *ClassInterval* with an additional *scope* field that records the scope of the current instance.

```
// it's the typically box class of classref; the sugar syntax is 'var T as [Animal to Cat]'
class ScopedClassInterval from ClassInterval{
    metaclass{
        fun new(lBound as classref, uBound as classref){
            super(lBound, uBound)
        }
    }
    public final scope as Object   // assigned by box opcode
}
```

ScopedClassInterval, lang.ago(https://github.com/inshua/ago/blob/master/ago-sdk/src/lang/lang.ago#L73)

To make using *ScopedClassInterval* more convenient, ago provides syntactic sugar *as [L to U]* and *like L*; the latter is equivalent to *as [L to _]*.

ScopedClassInterval implements passing a class with scope information. For example:

```
fun test(){
    var s = 'foo'
    class Animal{
        fun speak(){ return s; }
    }
    class Cat from Animal{}
    var c as [Animal to _] = Cat
}
```

The type-interval check for *ClassInterval* is actually performed at compile time. In a *ScopedClassInterval* instance, the value of *boxedClass* is a copy of the corresponding *agoClass* tailored to the specific scope.

Using *ScopedClassInterval* enable function callbacks in ago. For example:

```
fun add(a as int, b as int) as int{ return a + b }
fun add2(a as int, b as int) as int{ return a + b + a + b }
class C{
```

```
    i as int
    fun new(i as int){ this.i = i;}
    fun add(a as int, b as int) as int{ return this.i + a + b }
}

fun test(op like add, a as int, b as int){
    Trace.print(op(a, b))
}

fun main(){
    test(add, 1, 2)     // 3
    test(add2, 1, 2)    // 6

    var c = new C(100);
    test(c.add, 1, 2)

    var f like add = c.add     // 'like add' is equivalent to `as [Function<int, int>]`; as mentioned earlier, the ago compiler automatically inherits generic class `Function<R>` and implements genric interface `FunctionN<R, T1, ..., TN>` for every function; the `like` keyword selects the `FunctionN` interface, yielding a `ClassInterval`, not the function itself

    Trace.print(f(3, 4))
    test(f, 10, 20)
}
```

Leveraging classref boxing techniques and the concept of parameterized classes, generic type parameters can also be understood as a special *ClassInterval*. Their value does not carry scope information, but the type parameter must specify variance rules. ago's syntax for type parameters is *+-T as [LClass to UClass]*, where + and − represent covariance and contravariance, and default is invariance. ago uses a parameterized class *GenericTypeParameter* to record the variance and class interval of the type parameter.

```
// For G<T>, i.e., G<+T as [Animal to _], -T2 as [_ to Cat]>
class GenericTypeParameter from ClassInterval{
    metaclass{
        public final variance as byte
        fun new(lBound as int, uBound as int, variance as byte){
```

```
            super(lBound, uBound)
            this.variance = variance
        }
    }
}
```

GenericTypeParameter, lang.ago (https://github.com/inshua/ago/blob/master/ago-sdk/src/lang/lang.ago#L83)

This class-scope syntax is consistent in form with *ScopedClassInterval*, and during actual compilation it indeed generates a parameterized class like *GenericTypeParameter::(Variance,L,U)*.

Formally, generic classes can also be understood as template classes parameterized by type parameters. For example, given the following generic class:

```
class G<+T as [Animal to _]> {}
```

It can be considered to have the following logical correspondence:

```
class G{
    metaclass{
        fun new(T as GenericTypeParameter::(Covariant, Animal, Any)){
            // apply the template and return an instantiated class
        }
    }
}
```

Thus, a concrete class *G<Cat>* can be regarded as the parameterized class *G::(Cat)*.

For dynamic languages, this understanding of type parameters is sufficient as an implementation blueprint. However, ago is a static language; its generics are essentially still template technology, so the parameterized class *GenericTypeParameter::(Variance,L,U)* that corresponds to generic type parameters does indeed exist, but *G::(Cat)* does not—what actually exists is still *G<T>*.

Using scalarized *classref* and this set of special boxing classes, rather than meta-classes, is because types in ago typically carry scopes; a meta-class type value cannot encapsulate scope, more impossbile for boundary, and meta-classes are prone to circular dependencies, making them difficult to engineer. ago constrains meta-classes to anonymous classes with at most two levels, helping developers to avoid interacting directly with them.

4.5 Function Overloading

In ago, a function is treated as a class, and class names must be globally unique. Consequently, the traditional approach of using the same identifier with different parameter lists to distinguish multiple implementations (overloading) no longer works. ago introduces **call-time overloading** through **shared/qualified name** distinction:

When declaring functions, developers can assign each implementation a **fully qualified name**: the part before '#' is the *shared name* (the identifier common to this group of overloads), followed by a unique suffix after '#'. The compiler

matches function calls by examining the types and number of arguments automatically from the shared name; and explicit full name specification is required in case of ambiguity.

```
// Declare functions with fully qualified names; the portion before # is the shared name
fun f#1(i as int){

}
fun f#2(i as int, j as int){

}
fun main(){
    f(1)        // The compiler recognizes as f#1
    f(2,3)      // The compiler recognizes as f#2
    f#2(2,3)    // Explicit full name specification
}
```

The same strategy applies to implement attributes. An attribute is a pair of getter/setter(setter is optional) with same identifier; in ago it can be defined as functions with the same shared name, suffixed with *#get* and *#set*, respectively. For example:

```
class Person{
    name as string
   fun name#get as string{ return this.name}       // `get name as string()` syntax is also supported
      fun name#set(name as string){this.name = name}  // `set name(name as string)` syntax is also supported
}
fun main(){
    var p = new Person()
    p.name = 'ago'   // access attribute like field
    Trace.print(p.name)
}
```

5 SOME REAL-WORLD CASE STUDIES IN AGO

The current form of ago is not a traditional scripting language for Java developers; instead it's an embeddable, extensible interpreter framework. Unlike conventional interpreters that rely on an explicit interpretation loop, ago has none—each *AgoFrame* executes its own vm code instructions. In other words, it's essentially an interpreter without an interpreter

loop. This approach allows more flexible customization: by subclassing *Slots*, *AgoFrame*, *RunSpace*, and *AgoEngine*, developers can adapt the system for specific business scenarios.

Developers can integrate various middleware into the ago runtime or SDK, enabling ago code to access them without any modification. For instance, you could replace the implementation of *AtomicBoolean* used in the earlier *race* function with a distributed lock; the business-layer ago code would then automatically achieve cross-node consistency without changes. Likewise, by specifying particular service interfaces for functions, a custom *AgoEngine* can automatically map specified functions to RESTful or gRPC endpoints, decoupling business logic from network protocols. This *language-level middleware* design allows developers to focus purely on domain behavior declarations.

Developers can also implement ago's runtime in other host languages, enabling the same ago code to run across multiple language environments. For example, a Python-written crawler and a Java-written content system could be deeply integrated using one same ago unit.

Although ago currently interprets VM instructions one by one via *AgoFrame*, this is not the only option. Developers can translate ago VM code into Java bytecode in a custom class loader, or even compile it with LLVM to native machine instructions for higher performance. Moreover, a *CallFrame* need not be heap-allocated; optimizations could allow it to run on a stack-based runspace (e.g., to be benefited from object elimination).

As a true object-oriented language, ago is fully friendly to software engineering: it provides necessary features such as stack traces and debuggers, integrates well with IDEs and version control tools, and supports robust object-oriented analysis and design.

The following examples from game development and business processes illustrate the characteristics of ago.

5.1 AnimationRunSpace Example

```
class Monkey extends Sprite {
    // sequential movement operations
    fun jump() {
        this.moveTo(2, 300, 500);   // moveTo(duration, x, y)
        this.moveTo(2, 400, 0);     // jump back to the floor.
    }
    // parallel execution of rotation and movement
    fun rotateJump() {
         fork this.repeat(4, new this.rotate(0.5, 180));     // repeat 4s, rotate(duration,
angle)
        this.jump();
    }
}
```

The code above demonstrates the two composition patterns analogous to cocos2d's *SequenceAction* and *ParallelAction*. The basic animation functions such as *moveTo* and *rotate* internally call a function named *awaitNextFrame()* (or overridden *await* statement) to yield execution until *AnimationRunSpace* resumes at the next animation frame. *AnimationRunSpace* orchestrates the execution of these CallFrames, which are equivalent to cocos2d Actions but expressed as ordinary function calls. This lets game designers concentrate solely on writing the narrative script.

5.2 Simple Leave-Request Workflow

The following is a simple leave-request workflow written in ago, intended only to illustrate the implementation idea:

```
class Employee{
    // The leave process is encapsulated as a member method of Staff; compared to traditional BMNP processes, this design aligns better with object-oriented responsibility separation.
    fun requestLeave(){
        var leave = this.submitLeaveRequestForm();           // Request to the form service; the callframe is suspended, the employee receives a form interface; after submission, the program continues via MQ or RPC
        var r = this.manager.approvalLeave(leave);           // Also wait submission on the form service
        if(r) this.scheduler.set(leave.start, leave.end, 'Leave')
    }
}
```

In this example, the custom *RunSpace* and related *Slots* classes persist program state, as described earlier. Each business operation (e.g., form submission/approval) creates a new CallFrame that is immediately suspended; state is persisted before and after each business function call. Even if the program crashes or restarts, the system can resume the corresponding CallFrame and RunSpace upon receiving the next MQ or RPC message and continue execution. In fact, such programs spend most of their time suspended; program state resides in persistent storage and is only activated when an external event occurs.

In real scenarios, user can still work with workflow designers: workflow engine would generate ago code like above from the DAG produced by the designer. For example, a conditional branch can be translated into an *if* statement; a source map between the DAG and ago code enables visual debugging on the flow diagram. I have already built a prototype of an object-oriented visual designer to demonstrate that this approach is feasible.

5.3 Bank Transfer with Distributed Transaction

Here's how ago naturally integrates business logic with distributed transactions using its framework:

```
class Account{
    fun transfer(toAccount as Account, amount as int) with Transaction{
        this.balance -= amount
```

```
        if (this.balance < 0) {
            throw new InsufficientFundsException()
        }
        await toAccount.accept(amount) via anotherDB  // XA transaction on separate DB
    }
}
```

The *XATransactionRunSpace* is a subclass of *RunSpace*, providing database connections and transaction managers:

```
class XATransactionRunSpace extends RunSpace{
   java.sql.Connection connection;    // Obtained via AtomikosDataSourceBean.getConnection()
                                          UserTransaction              utx              =
com.atomikos.icatch.jta.UserTransactionManager.getInstance().getUserTransaction();
   XATransactionRunSpace(){
        utx.begin();
   }
   void commit(){
       utx.commit();
   }
}
```

This creates a two-phase transaction pattern using XA. The actual commit/rollback logic resides within the *XATransactionRunSpace* implementation, providing business code with complete transparency regarding transaction management details.

In this example:

- Source and destination accounts reside in separate databases
- Each has its own *XATransactionRunSpace*
- These spaces share common class definitions for interoperability
- The transaction manager handles final commit at the end of transfer

Any unhandled exception during execution causes the *utx.rollback()* method to be called, automatically rolling back both accounts' operations while maintaining data consistency across systems.

A similar approach can be used for implementing Saga transactions:

```
interface Compensator(){
    void rollback();
}
class Account{
    fun transfer(toAccount as RemoteAccount, amount as int) with Compensator{
```

```
        var subtracted = false
        this.balance -= amount
        if (this.balance < 0) {
            throw new InsufficientFundsException()
        }
        subtracted = true;
          await toAccount.accept(amount)       // `accept` also implements the Compensator
interface
        override rollback(){
            if(subtracted) this.balance += amount
        }
    }
}
```

The *SagaRunSpace* implementation:

```
class SagaRunSpace{
    List<CallFrames> sagaFrames = new LinkedList<>();
    void rollback(Exception e){
      for(frame: frames.reversed()):
          frame.rollback()  // reverse execution order to properly compensate
   }
}
```

In this example:

- Remote accounts are accessed via service stubs implementing the same interface
- Each step records its *CallFrame* into a transaction log maintained by *SagaRunSpace*
- On failure, it triggers compensation in exact reverse order of operations

Each frame is durably persisted using atomic write patterns (e.g., *balance -= amount* must be atomically recorded to avoid data loss from crashes, that means, *inc* instruction and *pc++* action which writing the CallFrame data table must both work in the same transaction). This ensures the state remains consistent even after unexpected system shutdowns or restarts.

These examples demonstrate how ago unifies various domain-specific requirements into its core architecture while maintaining natural syntax and semantics.

Similarly, ago can also easily express TCC transactions; this is not elaborated here.

It should be noted that ago is still in development and not yet production-ready; the above solutions are only technical explorations. Moreover, ago's design concepts are novel, offering ample opportunities for further exploration. While the ideas presented here are somewhat inspiring, they are neither final nor unique implementations. With continued research and practice, more interesting, robust, and general solutions may emerge.

6 CONCLUSION AND OUTLOOK

This paper starting the programmatic representation of **actions** in technologies such as animation, coroutines, and workflow engines, discusses the limitations of traditional function-call models in terms of time extension, asynchronous waiting, and persistence. By re-examining the essence of a CallFrame, it propose the new perspective **Function is Class, CallFrame is Instance**, thereby transforming functions from purely mathematical logic entities into objects capable of expressing actions. This unifies object-oriented (OOP) and functional paradigms at the same perspective.

Based on the above reflections, I designed and implemented an object-oriented language called **ago**. Its key innovations include:

1. **Functions as classes** – unifying the concepts of closure for both classes and functions, generic types for classes and functions, and trait composition.
2. **Objectified CallFrames** – after being made into objects, a CallFrame possesses the ability to yield and persist, naturally expressing long-term actions and supporting concurrent models, thereby eliminating the reliance on callbacks and state machines that traditional programming languages use to express actions.
3. **Parameterized classes** – allow constant values to be directly embedded into types, enhancing type-semantic expressiveness.
    a. Parameterized interfaces provide annotation-like semantics in an object-oriented manner, further enriching domain-modeling capabilities.
4. Class scalarization via *classref*, with its boxing class *ScopedClassInterval* carrying scope propagation;
    a. The boxing class *GenericTypeParameter* offers a more intuitive interpretation of generics within OOP.
    b. Syntax sugar such as *like add* simplify function type references.
5. Call-time overloading through fully qualified and shared names – using a '#' suffix to distinguish different signatures, providing a coherent approach for getters/setters.
6. An embeddable interpreter framework, that can evolve into a persistent programming language or distributed language.

Overall, ago attempts to provide a more thorough object-oriented perspective, offering a natural and easier programming model for real-world actions. Although it has not yet been fully validated in real production environments, an executable prototype has already been completed on the Java runtime, demonstrating the feasibility of its implementation path. As an evolving exploratory effort, ago's value lies not only in what functionalities it can realize but also in the new thinking framework it offers: **Function is Class, CallFrame is Instance**. This perspective provides a valuable direction for future programming model development, especially suited to systems that require long-term execution, state persistence, and cross-environment collaboration.

Currently, ago is still undergoing continuous development; further work is needed to refine the SDK, documentation, IDE support, and build-tool integration to improve developer experience and foster a sustainable ecosystem and community. Moreover, the current garbage-collection strategy largely depends on the host language platform; for GC-free host languages, better GC approach is still being explored.

REFERENCES

[1] Call Stack – Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Call_stack

[2] Coroutine – Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Coroutine

[3] Saṅkhāra – Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Sa%E1%B9%85kh%C4%81ra

[4] Whitehead, A.N. (1929). *Process and Reality*. Macmillan.

[5] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors*.

[6] Actions, Transformations and Effects. - cocos v0.6.9 documentation. Retrieved from https://los-cocos.github.io/cocos-site/doc/programming_guide/actions.html#actions

[7] Dalvik bytecode format (2025-01-07). Android Open Source Project. Retrieved from https://source.android.com/docs/core/runtime/dalvik-bytecode

[8] Kotlin Language Documentation – Coroutines. (2020). JetBrains. Retrieved from https://kotlinlang.org/docs/coroutines-overview.html

[9] Generics – Scala 3 Book – Scala Documentation. Retrieved from https://docs.scala-lang.org/scala3/book/types-generics.html

[10] Camunda BPMN 2.0 Specification (OMG). (2015). Retrieved from https://www.omg.org/spec/BPMN/2.0/

[11] First-class Citizen – Wikipedia. Retrieved from https://en.wikipedia.org/wiki/First-class_citizen

[12] Metaclass – Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Metaclass

[13] Persistent Programming Language – Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Persistent_programming_language

[14] Tuple Space – Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Tuple_space#JavaSpaces

[15] python-trio/trio: *Trio* – a friendly Python library for async concurrency and I/O. GitHub. Retrieved from https://github.com/python-trio/trio

[16] Landin, P.J.H. (1964). “A Structural Approach to Operational Semantics.” *Acta Informatica*, 1(2), 125–144.