Feng XY, Fu RX, Shi L *et al.* An overview of Cangjie programming language. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY, 2026. DOI: 10.1007/s11390-025-5978-7, CSTR: 32374.14.s11390-025-5978-7

An Overview of Cangjie Programming Language

Xin-Yu Feng[1, 2] (冯新宇), *Member, CCF*, Rong-Xiao Fu[2] (傅荣枭), Lei Shi[2] (史　磊), Le Tu[2] (涂　功) and Yong-Yong Yang[2] (杨勇勇)

[1] *State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing 210023, China*

[2] *Huawei Central Software Institute, Hangzhou 310000, China*

E-mail: xyfeng@nju.edu.cn; furongxiao@huawei.com; shilei167@huawei.com; tule1@huawei.com; yangyongyong@huawei.com

Received September 24, 2025; accepted December 23, 2025.

Abstract Cangjie is a modern programming language designed for application-level software development. It is statically typed and statically compiled, utilizing a tracing garbage collector (GC) for automated memory management. The language's design strives to balance simplicity, safety, extensibility, and performance. Cangjie supports multi-paradigm programming with a concise syntax. With a strong emphasis on extensibility, it features type extension, macro programming, and a simple, efficient C Foreign Function Interfaces (C-FFI). The compiler frontend leverages modern techniques to support robust type inference and introduces a language-specific high-level intermediate representation, Cangjie high-level intermediate representation (CHIR), to facilitate efficient analysis and semantics-aware optimization. Additionally, the language runtime and LLVM-based backend implement lightweight user-mode threads and a concurrent compacting GC. The standard LLVM backend has been extended with GC-specific intrinsics and custom compilation passes for Cangjie-specific optimizations. Preliminary experimental results demonstrate that Cangjie achieves performance competitive with other application-level languages.

Keywords programming language design and implementation, multi-paradigm programming, high-level intermediate representation, compiler optimization, language runtime

1 Introduction

Cangjie[①] is a modern programming language built at Huawei for application-level software development. It is introduced mainly for mobile application development for the HarmonyOS Next operating system, but as a general-purpose language, it is also applicable to other scenarios such as micro-services. Instead of being an academic practice to implement novel language features, Cangjie is designed to promote the ecosystem of HarmonyOS Next. Its primary purpose is to deliver an outstanding programming experience and exceptional application performance to developers. To achieve this, Cangjie introduces several key innovations in its compiler and runtime architecture. Below we introduce the core design principles and technical breakthroughs of Cangjie.

1) *Simplicity and Productivity.* As an application-level language, Cangjie operates at a high abstraction level and tries to keep the syntax concise and simple. A key innovation is the advanced type inference system, which allows developers to omit type annotations without sacrificing the benefits of static typing. Based on a bidirectional inference framework[1], the compiler employs a novel algorithm to resolve type arguments of generic function calls, where argument expressions have inter-dependent type information and inference involves spiral information propagation — a well-known challenge to the well-structured "checking mode" and "synthesis mode" for bidirectional inference. Cangjie also supports efficient inference with function overloading, and inference of argument types of lambda expressions. For concurrency programming, Cangjie adopts user-level "stackful" threads rather

Regular Paper

Special Issue: Celebrating the 40th Anniversary of JCST

[①]Pronounced like "Tsangjie". It is named after the legendary figure Cangjie in Chinese mythology, who is known as the inventor of Chinese characters.

than "stackless" coroutines, preventing the infectious "colored function" problem[②] often associated with async/await syntax.

2) *Safety.* Cangjie ensures program safety through a comprehensive set of static and dynamic checking mechanisms. As a statically typed language, it conducts rigorous type checking at compilation time to detect errors early. The type system enforces null-safety statically, avoiding common null-pointer exceptions. However, Cangjie tries to avoid an overly complicated type system to keep the language reasonably simple for industry developers, therefore, it also relies on runtime mechanisms to enforce safety. Beyond static checks, Cangjie ensures memory safety through a combination of automatic memory management and runtime bounds checking. It also incorporates overflow checking and string integrity checking at runtime to bolster application robustness.

3) *Expressiveness and Extensibility.* Expressiveness and extensibility are critical for framework developers building abstractions for clients. Cangjie is a multi-paradigm language seamlessly blending procedural, object-oriented, and functional programming. It supports macro programming—similar to Rust[2] as a powerful meta-programming mechanism for building embedded domain-specific languages (DSLs). With annotations and reflection, these mechanisms make Cangjie highly extensible for domain-specific scenarios, such as declarative UI programming (e.g., ArkUI[③]) and agent programming (e.g., Cangjie Magic[④]).

4) *Performance.* While focusing on high-level abstractions, Cangjie aggressively optimizes for the resource constraints of mobile devices, targeting low latency and rapid response. The language implementation features various optimizations at different stages of compilation and runtime to achieve the following.

- *Semantics-Aware Optimization with Cangjie High-Level Intermediate Representation (CHIR).* The compiler frontend introduces a language-specific high-level intermediate representation, CHIR. Unlike flat low-level intermediate representations (IRs), CHIR preserves high-level program structures (such as nested control flow and class hierarchies), enabling precise, semantics-aware optimizations like devirtualization and bounds-check elimination before lowering to LLVM IR[⑤].
- *Concurrent Compacting Garbage Collection.* To minimize UI stutter (jank), Cangjie implements a high-performance concurrent compacting garbage collector (GC). A key innovation here is the use of lightweight semi-Stop-The-World (semi-STW) barriers rather than full STW pauses, allowing the GC to reclaim memory and reduce fragmentation with sub-millisecond tail latency.
- *Lightweight Runtime.* The runtime supports lightweight user-mode threads that enable massive concurrency with low overhead. Furthermore, the support for value types (structs) reduces expensive heap allocations and lowers memory footprint compared with traditional managed languages.

In the rest of this paper, we introduce the main ingredients and key features of Cangjie, including the language design (Section 2), the compiler frontend (Section 3), the LLVM-based compiler backend (Section 4), and the language runtime (Section 5). We also provide a preliminary performance evaluation in Section 6, and then conclude the paper in Section 7.

2 Language Design

Cangjie is a statically typed, multi-paradigm programming language. It supports procedural, object-oriented programming (OOP) and functional programming (FP). To give readers a flavor of Cangjie, Listing 1 shows a function `collatz` (lines 3–13) which verifies the Collatz conjecture[3] for an integer `start` and returns the corresponding Collatz sequence.

The program starts from the `main` entry (lines 15–17), which computes and prints the Collatz sequences for `0` to `19`. Line 16 creates an array of size 20 and initializes it using the function `collatz`, which is called (with the array index as its argument) to produce the corresponding array element. It shows Cangjie's support of functions as first-class citizen.

Cangjie supports algebraic data types (ADT) through its `enum` type, which is a tagged union where each tag takes a product of types. Listing 2 gives the declaration of the abstract syntax tree (AST) of elementary arithmetic expressions. Then the AST of the arithmetic expression `3 * (5 + 1)`, for instance, can be expressed as `Mult(Const(3), Plus(Const(5), Const(1)))`.

[②]Nystrom B. What color is your function? https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/, Dec. 2025.

[③]ArkUI. https://developer.huawei.com/consumer/en/arkui/, Dec. 2025.

[④]Cangjie Magic. https://gitcode.com/Cangjie-TPC/CangjieMagic, Dec. 2025.

[⑤]LLVM contributors. LLVM language reference manual, https://llvm.org/docs/LangRef.html, Dec. 2025.

Listing 1

```
import std.collection.ArrayList

func collatz(start:Int): ArrayList<Int> {
  var n = start
  let s = ArrayList<Int>()
  s.add(n)
  while (n > 1) {
      if (n % 2 == 0) { n /= 2 }
      else { n = n * 3 + 1 }
      s.add(n)
  }
  return s
}

main() {
  println(Array(20, collatz))
}
```

Listing 2

```
enum Expr {
    Const(Int)
  | Plus(Expr, Expr)
  | Mult(Expr, Expr)
}
```

Correspondingly, Cangjie supports pattern matching to deconstruct values of ADTs, as in functional languages such as Haskell[4] and Scala[5]. The `eval` function in Listing 3 maps an expression `Expr` to its `Int` value.

Listing 3

```
func eval(e: Expr): Int {
  match (e) {
    case Const(n) => n
    case Plus(a, b) => eval(a) + eval(b)
    case Mult(a, b) => eval(a) * eval(b)
  }
}
```

Besides, Cangjie also has a builtin pipeline operator `|>` (for reversed function applications) and composition operator `~>` (for reversed function compositions) to makes it more convenient to work with functions. The generic function `interleave` in Listing 4 uses the pipeline operator, together with the functions from the standard library (`std.collection`) to arrange the elements from two arrays in an alternating way. The lambda expression `{p => [p[0], p[1]]}` is an argument of `flatMap`. It is used as a trailing lambda expression so that the parentheses of function application can be omitted to simplify the notation.

Listing 4

```
func interleave<T>(a:Array<T>, b:Array<T>)
  : Array<T> {
      a |> zip(b) |>
      flatMap {p => [p[0], p[1]]} |>
      collectArray
}
```

For OOP, Cangjie supports classes and interfaces. For instance, `ArrayList<T>` used in the `collatz` example is a generic class in the standard library of Cangjie, which implements the interface `List<T>`. Listing 5 gives another simple example.

Listing 5

```
public interface Pet {
  func greeting(): Unit
}
public open class Cat <: Pet {
  public func greeting() { println("meow") }
}
public open class Dog <: Pet {
  public func greeting() { println("woof") }
}
public class Husky <: Dog {
  public func howl() { println("awoo") }
}
main() {
  let pets:Array<Pet> = [Cat(), Dog(), Husky()]
  for (p in pets) {
    p.greeting()
  }
}
```

As in Java, each Cangjie class can implement one or more interfaces, but can have only one super class. Here we unify the syntax of interface implementation and class inheritance using the `<:` notation to indicate the induced subtyping relations. Different from Java, Cangjie class cannot be inherited by default, unless it is modified by the `open` keyword. The same restriction applies to instance member functions, which cannot be overridden unless they are `open`. This reflects Cangjie's stance against abusing inheritance.

For the rest of this section, we highlight several key language features and introduce them in detail.

2.1 Type Extensions

It is a common situation during software development that the developer would like to add extra functionalities to existing types, but the source code of these types may not be available (for example, when these types are imported from binary dependencies). To address this issue, Cangjie allows extending types with member functions and properties without changing the original type definitions. Listing 6 extends the builtin `Int` type with an instance member function `show`, which takes a `String` and prints it `N` times where `N` is the integer used to invoke `show`.

Listing 6

```
extend Int {
  public func show(s: String) {
    for (_ in 0..this) { println(s) }
  }
}
```

Then we can write `5.show("Hello")` to print `Hello` 5 times.

In addition to the direct extension introduced above, Cangjie allows a type to implement interfaces and become a subtype of these interfaces after the type has been defined. We call this feature "interface extension". Similar features also present in other languages, such as `protocol` conformance via extension in Swift[6], `trait` implementation in Rust[2], and type `class` instantiation in Haskell[4]. For example, Listing 7 declares an interface `Typable` containing an instance member function `ofType` which allows the instance to return the name of its type, and by type extensions, the builtin types `Int` and `String` implement `Typable`.

With these declarations, the expression `5.ofType()` evaluates to "Int". More importantly, `Int` and `String` now become the subtypes of `Typable`, and calls of `ofType` on `Typable` instances will be dynamically dispatched to the corresponding implementation in the interface extension. Thus, using the `showType` function, `showType("Hello")` will print `String`. Besides, if `Typable` is the upper bound in a type constraint, as shown in the function `showType2`, `Int` and

Listing 7

```
interface Typable {
  func ofType(): String
}
extend Int <: Typable {
  public func ofType(): String { "Int" }
}
extend String <: Typable {
  public func ofType(): String { "String" }
}
func showType(x: Typable) {
  println(x.ofType())
}
func showType2<T>(x: T) where T <: Typable {
  println(x.ofType())
}
```

`String` can be used to instantiate the corresponding generic type parameter.

Type extensions also support generic types with type constraints. This allows Cangjie to express programs like conditional `protocol` conformance in Swift[6] or type `class` instantiation with contexts in Haskell[4]. For example, Listing 8 shows the extension of `Array<T>` with the `ToString` interface, which requires the element type `T` to implement `ToString` too.

Listing 8

```
public interface ToString {
  func toString(): String
}
extend<T> Array<T> <: ToString
  where T <: ToString
{
  public func toString(): String {
    //implementation omitted
  }
}
```

Note that this extension is different from implementing the `ToString` interface (with the same type constraint) when the type `Array<T>` is defined, which would be overly restrictive because it prevents `Array<T>` from containing elements of type which does not implement `ToString`, even when we do not need to call `toString`. In contrast, our example does

[6]Doug Gregor. Conditional conformances, https://github.com/swiftlang/swift-evolution/blob/main/proposals/0143-conditional-conformances.md, Dec. 2025.

not affect the original type declaration of `Array<T>`.

2.2 Meta-Programming

Cangjie supports various meta-programming mechanisms: macros, annotations, and reflections. Macro declarations are similar to function declarations, but using the keyword `macro` and working on the `Tokens` type, which represents parsed token sequences. Listing 9 shows the declaration of a macro which encloses its input tokens in a forever running `while` loop.

Listing 9

```
public macro Forever(input: Tokens): Tokens {
  println("Creating a Forever loop...")
  quote(while (true) { $input })
}
```

The resulting tokens are generated by a `quote` expression, which converts (but not evaluates) the quoted code into a `Tokens` object. The `input` here is interpolated using the `$` symbol in the `quote` expression; thus, it is treated as the `Tokens` value it represents, instead of the `input` identifier itself.

Macros are called using the `@` symbol, and all macro expansions happen during compilation. For example, the macro call `@Forever(println("Hello"))` is expanded to `while (true) { println("Hello") }`. When macro calls take declarations as arguments, the parentheses can be omitted, looking similar to annotations (but the semantics is different).

Annotations in Cangjie are special syntactic constructs attached to declarations to provide extra information during compilation or at runtime. By declaring an `@Annotation` class, the developer is able to use that class as an annotation so that an object of that class is attached to the annotated declaration. For example, Listing 10 shows the declaration and usage of a customized annotation `@Version` storing an integer version number.

Listing 10

```
@Annotation
public class Version {
  public const Version(let version: Int) {}
}
@Version[5]
class Example {}
```

To allow compile-time access to the annotated information, an `@Annotation` class must provide a `const` constructor, so that an object of that class can be created via constant evaluation during compilation.

Reflection is a mechanism allowing a program to inspect or modify its own information or behaviors at runtime. Cangjie provides the support for reflection, and allows operations such as reading runtime type information of instances, searching for members by their names, and invoking member functions acquired from runtime type information. However, restrictions still exist for the current reflection mechanism in Cangjie: there is almost no support for modifying the behaviors of types/instances via reflection; the reflection mechanism can only access entities with `public` visibility; and reflection of function types, tuple types, and `enum` types is yet to be supported.

2.3 Cross-Language Interoperation

For working with existing ecosystems built by other languages, Cangjie supports cross-language interoperation with various languages either natively or as libraries. In particular, the interoperation with C is natively supported in Cangjie as part of the language.

By annotating function and type declarations with `@C`, the binary implementations (e.g., calling convention of functions or data layout of structs) of the annotated declarations conform to C and can be used directly from C code. Listing 11 shows declarations annotated with `@C`.

Listing 11

```
@C
func printCPointer(p: CPointer<Int>): Unit {
  unsafe {
    let n: Int =
      if (p.isNull()) { 0 } else { p.read() }
    println(n)
  }
}
@C
struct Point {
  Point(var x: Float64, var y: Float64) {}
}
@C //produces compilation error
func notCType(x: Option<String>): Unit {
  println("No C counterpart exists")
}
foreign func getPtr(): CPointer<Int>
```

For the `printCPointer` function declaration, it

takes a `CPointer<Int>` parameter and returns `Unit`: both types satisfy the built-in `CType` constraint (for Cangjie types with C counterparts), therefore, the function can be used for C interoperation, and `printCPointer` itself has type `CFunc<(CPointer<Int>)->Unit>`. Besides, in the body of `printCPointer`, due to the unsafe nature of C, pointer operations like `read` can only be used within an `unsafe` context such as the `unsafe` block here. Actually, all interoperation with C in Cangjie must happen within the `unsafe` context.

Similarly, for the `Point` struct type declaration, the types of its both instance member variables (`x` and `y`) satisfy the `CType` constraint. Therefore, `Point` can be used for C interoperation, and usually it corresponds to a struct with the same memory layout declared in C.

However, not all types or functions in Cangjie can be used for C interoperation. As its name suggests, `notCType` takes an `Option<String>` parameter which does not satisfy the `CType` constraint. Thus, annotating this function with `@C` results in a compilation error.

The example also includes a `foreign` function declaration, which provides a Cangjie function signature for a function implemented in C with the same function name and the corresponding parameter/return type. Since a `foreign` function is obviously for C interoperation, its `@C` annotation can be omitted.

3 Compiler Frontend

As shown in Fig.1, Cangjie compiler consists of two parts, the language-dependent frontend and the (mostly) language-independent backend. We introduce the frontend in this section, and the backend in Section 4.

The frontend takes Cangjie source code as input and generates LLVM IR[7] (with extensions of intrinsics) as output, which is in turn passed to the backend. The lexical and syntax analysis phase generates abstract syntax tree (AST) from the source code. At the AST level, we do macro expansion, semantic analysis and desugaring. Then we translate AST to the Cangjie-specific intermediate representation, namely CHIR, based on which we can do language-dependent program analysis and optimization. The last stage of the frontend lowers CHIR to LLVM IR.

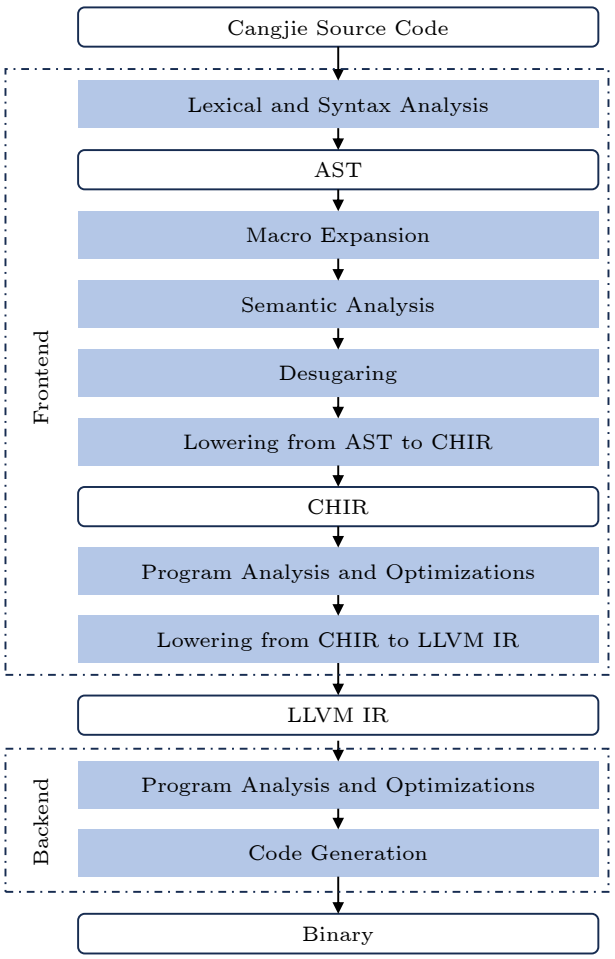

Fig.1. Cangjie compiler overview.

3.1 Semantic Analysis

The semantic analysis phase plays a crucial role in validating a program's correctness beyond its mere syntactic structure. This involves a range of essential checks. For example, as a language with visibility control, Cangjie requires every pair of definition and usage of a name to conform to the visibility rules set by the programmer. Another example is that Cangjie allows the initialization of a variable to be separated from its declaration, but it must ensure that every variable is initialized before its first use. Semantic analysis is responsible for enforcing these rules.

Additionally, semantic analysis is responsible for augmenting the AST with vital contextual information. An obvious example is type information, which must be fixed and validated for every expression and declaration before the program is translated to a

[7] LLVM contributors. LLVM language reference manual, https://llvm.org/docs/LangRef.html, Dec. 2025.

type-safe binary executable. Another example is name resolution. Since Cangjie allows different definitions to share the same name through features like shadowing, overloading, and overriding, the compiler must link every usage of a name to its proper definition.

The exceptionally challenging part of semantic analysis in Cangjie is type inference, which is more powerful than prior work in a number of aspects. Cangjie allows programmers to omit type annotations for variables, function return types, generic arguments, and lambda's parameter types, under a rich type system and complex features such as subtyping, overloading, generics, and mutability of variables and fields. The compiler efficiently gives reasonable type inference results in a number of scenarios that prior work fails to handle.

Note that, as the compiler for an industrial language, our consideration on the type inference algorithm is not the same as in typical research projects. Although soundness must be guaranteed, completeness is not always considered worth pursuing. For example, when two candidate types for the generic argument of a function call are both valid, and no one is more specific than the other, we will simply report an error instead of making a blind guess of the programmer's intention. In general, if the context indicates ambiguous inference results, or the clue for the type has to be found from not closely relevant places, we will choose to ask the programmer to write explicit type annotation, so as to avoid possible mistakes and to keep the program more readable.

Performance of type inference is another major concern. Although type inference for Cangjie is a very hard problem in the worst case, we still hope users would not experience hours or days of compilation time for a single file in practice. Fortunately, with a few exceptions, the worst case rarely occurs in practical programs. A notable exception is nested overloading resolution scenarios, which actually occurs quite often in practice and may lead to exponential time complexity. We will discuss the strategies to mitigate this issue below.

The rest of this section will focus on the overall design of Cangjie's type inference algorithm, and discuss the solutions to some of the most challenging problems in detail. We introduce the base framework of the type inference algorithm in Subsection 3.1.1, and then discuss the details of our solution to some of the most challenging type inference problems in Subsections 3.1.2, 3.1.3, and 3.1.4.

3.1.1 *Bidirectional Type Inference*

Cangjie's type inference algorithm is based on the framework of bidirectional type inference, also known as "local type inference", which was originally described by Pierce *et al.*[1] and later widely adopted by research work on type inference[7–10]. The strategy gives up the traditional Hindley-Milner style of global constraint collection and generalization of type variables[11]. Instead, it passes type information only to neighboring nodes, and focuses on inferring type annotations that are natural to be omitted, such as generic arguments of function calls.

In the bidirectional type inference strategy, if an expression or declaration can provide hints on the expected type of its sub-expressions, then we check a sub-expression with respect to its expected type. This is called "checking mode". Here type information is passed top-down. In the example below,

```
let v: Byte = if (someCond) { 1 } else { 2 };
```

the declaration of `v` explicitly says the expected type is `Byte`. Then the type is passed to the `if` expression, and then distributed down to its branches. Eventually, we know the literals `1` and `2` in both branches must be of type `Byte`, which is a valid choice for integer literals. The program as a whole passes type checking.

If there is no expected type passed down to an expression, then the type information may instead flow from the expression itself to its super-expression, and help the inference of the super-expression's type. We call this case "synthesis mode". The code below gives an example.

```
func id<T>(x: T): T { x }
let s = id("hello")
```

Here we define an generic identity function `id`. When we start the check of the variable `s`, we know neither the type of `s` nor the generic argument of the function call `id("hello")`. Therefore we start from the innermost expression, the string literal `"hello"`, and decide it is of `String`. Then the information is passed up to the function call and help fix the generic type argument `T` to be `String`. Finally, the return type of the function call, which we already know is `String`, is passed up to the variable declaration and fixes the type of `s` to be `String`.

3.1.2 Type Argument Inference

The bidirectional type inference algorithm aims to solve type arguments for generic function calls. The idea is to collect constraints from neighboring nodes of the call and do a typical unification to solve the type arguments.

Beyond the basic ability of bidirectional inference, Cangjie's compiler can also handle a difficult case of type argument inference. It happens when the argument expressions of the function call have inter-dependencies of type information, as shown in the following example.

```
func map<T, R>(a: Array<T>,
               f: (T) -> R): Array<R> { ... }
let s = map([1, 2, 3], {i => i.toString()})
```

When checking the function call to `map`, the type of the lambda expression's parameter `i` relies on the solution of type argument `T`, which in turn relies on the type of the argument $[1, 2, 3]$, `Array<Int>`. With this information, we can figure out the return type of the lambda, `String`, which in turn can be used to solve the type argument `R`. The type information flows in a spiral way. Neither a single top-down pass nor a single bottom-up pass can figure out all the information.

The original algorithm by Pierce *et al.*[1] would fail in such cases. The follow-up work within local type inference framework tries to give a simple solution to this problem. A typical example is Scala 2's "colored local type inference" approach[12], which depends on a heavy-weight type system: beyond the syntax level bidirectional inference, it further gives each type argument a "direction", to mark where this part of the type should be derived from. It can then simulate the spiral propagation to some extent with this finer-grained control of information flow, but still with limitation. For example, it would fail if we swap the order of the two parameters of `map` and place the array as the second argument.

Cangjie adopts a simple solution to overcome the aforementioned limitation in Scala 2. We notice that there are essentially two operations to propagate all necessary information around. 1) Try to type check more argument expressions through the check mode, with the corresponding parameter types as expected types and the already solved type arguments substituted into these parameter types. 2) Try to solve more type arguments by the actual argument expressions that can pass type check. By repetitively applying these two steps, we can eventually reach a fixed point where no new argument expression can pass type check and no new type argument can be solved. If there is no unsolved type argument left, then the inference is successful, otherwise the inference fails and reports a compilation error.

3.1.3 Overloading Resolution

The time complexity of overloading resolution varies greatly depending on what other language features it is combined with. The most simple case is when the type inference is entirely bottom-up, like in C++ and Java. It is not hard to prove that overloading resolution in such cases only takes polynomial time with regard to the program's size, since a local decision never backtracks. The hardest case is when it is combined with the Hindley-Milner style type system, which is proven to be an undecidable problem[13, 14]. Standing in-between is when it is used along with bidirectional type inference, which can be proven to be NP-hard[15].

Although bidirectional inference claims to make decisions only by local information, overloading can still lead to exponential time complexity. Listing 12 demonstrates the problem.

Listing 12

```
func widen(x: Int8): Int16 { Int16(x) }
func widen(x: Int16): Int32 { Int32(x) }
func widen(x: Int32): Int64 { Int64(x) }
let v = widen(widen(widen(1)))
```

Here we have three overloaded functions, each converting an integer of a narrower type to a wider type. Then we have three nested calls to `widen`. The program only makes sense if the three calls, starting from the innermost one, are resolved to return `Int16`, `Int32`, and `Int64` respectively. However, there are actually $3^3 = 27$ different combinations for the three calls. If the nesting level grows deeper, the search space will soon grow to an unmanageable size.

Though looking peculiar, such scenario is actually not rare in practical programs. One typical case is UI programs, where one has nested UI components shaped like a tree. For flexibility, each component may have multiple overloaded constructors, and we have to search for a correct combination of all the constructors so that they are compatible with each

other. Similar problems also occur when we compile very complex binary expressions, since they will also be parsed to trees and the number literals and the operators can be overloaded. The Swift compiler has suffered from similar issues and may fail to compile some programs due to overloading[8–10].

Cangjie gives a partial solution to this problem. We observe that in most practical programs, resolving nested overloaded function calls is actually a dynamic programming problem with optimal substructures, and thus can be solved in polynomial time.

The intuition can be explained with the above `widen` example (Listing 12). In a naive approach, we may do a depth-first search of all possible combinations of the three calls. For each call, we check if a certain candidate definition can satisfy the expected return type from its outer-layer call, and if the inner-layer expression can match the candidate's parameter type. In this way, we reach the innermost call (`widen(1)`) $3 \times 3 = 9$ times, since the two outer-layer calls each have three candidates. However, not all these nine visits constitute different questions for the innermost call. Though the resolution result of the call does depend on the expected return type given by the outer-layer calls, there are only three different possible expected return types in total: `Int8`, `Int16`, and `Int32`. Therefore, we can remember the resolution result of the innermost call under a certain expected return type, and reuse the result when the expression is checked again with the same expected type. The computation will be done only three times instead of nine times. The results for two other calls may also be cached similarly.

For overloading resolution of the entire program, each function call and each possible expected return type uniquely identify a sub-question, whose answer can be cached. Therefore, there are only $O(nm)$ different questions to be answered in total, where n is the size of the program and m is the number of different expected return types in the program, which is no more than n. A memoized search can avoid the exponential blow-up of search space and finish within polynomial time.

There is only one special case that falsifies the claim that a function call expression and an expected return type can uniquely identify a sub-question. It is when the symbol table may be changed by the resolution result of an outer-layer call. The symbol table maps names to definitions and types. Therefore, its change is crucial for determining the resolution result. In Cangjie, the symbol table may be changed only when the following conditions are satisfied simultaneously. 1) There is a lambda expression whose parameters have no type annotations. 2) The lambda is used as a direct argument of an overloaded function call. 3) The parameter of the lambda expression, or any local variable whose type is inferred from that parameter, is used in an argument of another overloaded function call. The following code gives an example.

```
let v = f({x => g(x)})
```

Suppose both `f` and `g` have overloaded declarations, then `x` is the key parameter that meets the above conditions. The type of `x` is affected by which definition `f` resolves to, and in turn its type affects how `g` is resolved. In this case, we are no longer able to claim that resolution of `g(x)` is uniquely determined by its expected return type.

This special hard case is not common in practice. If it does occur, the cache as we described above may no longer be valid. Cache entries that may be affected by the symbol table change must be invalidated, and the search may regress to an exponential time enumeration in the worst case.

3.1.4 Lambda Parameter Type Inference

Cangjie allows omitting parameter types for lambda expressions to support concise notation. In this case, it needs to support global type inference for parameter types. Listing 13 demonstrates the ability. The lambda expression calculates the maximum value in a list `l` of integers. Its type can be inferred to be `(Iterable<Int>) ->Int`, which relies on some global type inference. As was discussed before, Cangjie has a rich type system, including ADT, OOP style subtyping, function and literal overloading, generics, etc. No known prior work has discussed

[8] Diggory. The compiler is unable to type-check this expression in reasonable time. String [edit] addition with an Int in the mix (solved), https://forums.swift.org/t/the-compiler-is-unable-to-type-check-this-expression-in-reasonable-time-string-edit-addition-with-an-int-in-the-mix-solved/64 526, Dec. 2025.

[9] Slappy. Why does this SwiftUI code causing compiler to timeout, https://stackoverflow.com/questions/64 639 286/why-does-this-swiftui-code-causing-compiler-to-timeout, Dec. 2025.

[10] Daniel Hooper. Why Swift's Type Checker Is So Slow, https://danielchasehooper.com/posts/why-swift-is-slow/, Dec. 2025.

Listing 13

```
let findMax = {l =>
        let i = l.iterator()
        var max = 0
        while (let Some(v) <- i.next()) {
                    if (v > max) { max = v }
        }
        max
}
```

global type inference under this complex combination of features.

We infer the parameter types of lambdas with a best-effort algorithm. The basic idea is still to give each parameter an unsolved type variable, and collect constraints about it from the function body. There are three notable difficulties in the inference: subtyping, member access, and overloading.

We handle subtyping in a way similar to the recent work by Dolan[16] and Parreaux[17]. That is, whenever there is a subtyping judgment between a type variable and any other type, we add that type as an upper-bound or a lower-bound for that type variable. Unlike prior work that can represent the constraints about a type variable by union and intersection type, or equivalently by generic constraints, Cangjie limits how these features can be used. Therefore, we record each pair of the subtyping relation as an individual constraint. Eventually we try to find a single concrete type that satisfies all these constraints as the solution, which may not exist and leads to a type error.

Member access provides another dimension of information other than subtyping. If following the approach of prior work, we would have added an imaginary structured interface with the type of the accessed member inferred to be the type variable's upper bounds[18]. However, Cangjie enforces nominal subtyping only. There is a gap between the information carried by the imaginary interface and the actual types defined in the program. In practice, as the type inference goes through the lambda's body, we do a real-time filtering of all possible types with the accessed members known so far. And if there is only one candidate left, we will greedily fix the type variable to be that concrete type. This way, we can get rid of the uncertainty introduced by type variables as early as possible.

The challenge introduced by overloading is that overloading resolution involves subtyping judgment, and may need to backtrack. Since subtyping judgment may add bounds to the type variables, the bounds may also need to be rolled back too. Therefore, we cannot just have a global set of constraints, but instead have to maintain a stack of constraints, and pop the stack top when backtracking happens. The following code gives an example.

```
func f(a: Int, b: Int): Int { b }
func f(a: String, b: String): String { b }
let g = {x => f(x, "hello")}
```

When checking the body of the lambda, we first try the version of `f` with type `(Int, Int) ->Int`, and add `Int` to the upper bound of `x`'s type, and later find this version wrong and have to backtrack. The constraints of `x`'s type are rolled back to an empty set too. Then we try the version of `f` with type `(String, String) ->String`, and add `String` to the upper bound of `x`'s type. This time we find no conflict, and the type of `x` is finally inferred to be `String`.

3.2 CHIR: Cangjie High-Level IR

Cangjie compiler introduces an extra intermediate representation, Cangjie high-level intermediate representation (CHIR), between Cangjie AST and LLVM IR[11].

3.2.1 *Purposes of CHIR*

CHIR is introduced mainly for the following purposes.

- *More Precise Program Analysis.* A typical example is the compile-time check on unexpected program behaviors including arithmetic overflow, shifting overflow, division by zero, out-of-bounds data access, etc. Cangjie implements a series of run-time check on these behaviors for program safety. With more precise program analysis such as constant propagation and value range analysis, in certain cases, we can report these behaviors earlier in compile-time and eliminate the overhead of the runtime check.
- *More Aggressive Optimizations.* Performance is always important to a programming language. As for Cangjie, its rich type systems, multi-paradigm support, and other high-level abstractions bring more

[11] LLVM contributors. LLVM language reference manual, https://llvm.org/docs/LangRef.html, Dec. 2025.

challenges on the compiler. For instance, the usage of OOP features like `class` and `interface` leads to extra operations for producing and accessing metadata, along with garbage collection overheads. More aggressive optimizations targeting Cangjie-specific scenarios are required to achieve better execution performance, code size and memory footprint.

- *Platform for Implementing Advanced Language and Compiler Features Such As Constant Evaluation and Compiler Plugin.*

AST is not suitable to fulfill the objectives above since it lacks explicit control-flow information which is essential for much program analysis and optimizations. As for LLVM IR, it is in the form of control-flow graph[19], and there are various program analysis and many optimizations which can be reused in LLVM community. However, it is still not enough for us because it is a language agnostic low level IR and a lot of Cangjie-specific high-level semantics information is lost at this level. Without such information, LLVM only performs conservative program analysis and optimizations. Besides, from the perspective of engineering, injecting too many Cangjie-specific codes into LLVM will leads to a huge burden when we want to upgrade to new version of LLVM.

In fact, these issues are not specific to Cangjie only. Many programming languages including Swift[12] and Rust[13] also develop their own IR for similar reasons. Besides, the MLIR project[20] also aims to provide infrastructure support of new abstraction levels in programming language implementation.

3.2.2 Key Design Principles

Inspired by the prior work, we design and use CHIR to address the issues. A brief overview of CHIR is shown in Fig.2. The design follows the following key principles.

1) *Nested Control-Flow Graph*

Many existing IRs, including LLVM IR, are in the form of a flat control-flow graph. In the graph, each node represents a basic block where the operations inside are executed sequentially, and edges are used to denote the control-flow transitions between basic blocks. However, such kind of IR will have semantic information loss when representing complex expressions in Cangjie. Listing 14 shows a `for-in` expression which performs iterations in the range of $[0, 10]$ with stride of 2.

Listing 14

```
for (i in 0..10:2) {
  let localVar: String = "iteration num: "
  println(localVar + i.toString())
}
println("iteration finish")
```

The representation of this `for-in` in a flat control-flow graph is shown in Fig.3. Two kinds of semantic information are lost here. First, the `for-in` in source code creates an inner scope where the variable `localVar` declared inside is not visible outside. This information can be used to implement lifetime analysis, but now it vanishes in such flat control-flow

Package	*pkg*	::=	`[`*is*`,...,`*is*`] [`*ty*`,...,`*ty*`] [`*gv*`,...,`*gv*`] [`*f*`,...,`*f*`]`
Imported Symbol	*is*	::=	`import symbol` *id* `from package` *id*
User-Defined Type	*ty*	::=	*typeKind id* `{...}`
	typeKind	::=	`enum` | `struct` | `class`
Global Var	*gv*	::=	*id* | *id* `= literals`
Function	*f*	::=	`func` *id*`(`*param*`, ..., `*param*`) { `*bg*` }`
	param	::=	*id*
Value	*v*	::=	`literals` | *id*
Identifier	*id*	::=	`string literals`
Block Group	*bg*	::=	`a CFG constructed by a set of` *bb*
Basic Block	*bb*	::=	`a sequentially executed` *op* `list with` *id*
Operation	*op*	::=	*trivialOp* | *hyperOp*
Trivial Operation	*trivialOp*	::=	*id* `=` *trivialOpCode*`(`*v*`, ..., `*v*`)`
	trivialOpCode	::=	`Add` | `Sub` | `Mul` | `Div` | `Apply` | `Invoke` | `Allocate` | `Store` | `Load` | `InstanceOf` | `Exit` | `Goto` | `Branch` | `...`
Hyper Operation	*hyperOp*	::=	*id* `=` *hyperOpCode*`(`*v*`, ..., `*v*`, `*bg*`, ..., `*bg*`)`
	hyperOpCode	::=	`If` | `While` | `ForInRange` | `...`

Fig.2. Brief overview of the CHIR in BNF form.

[12]Joe Groff, Chris Lattner. Swift intermediate language, https://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf, Dec. 2025.

[13]Rust contributors. The MIR (Mid-level IR), https://rustc-dev-guide.rust-lang.org/mir/index.html, Dec. 2025.

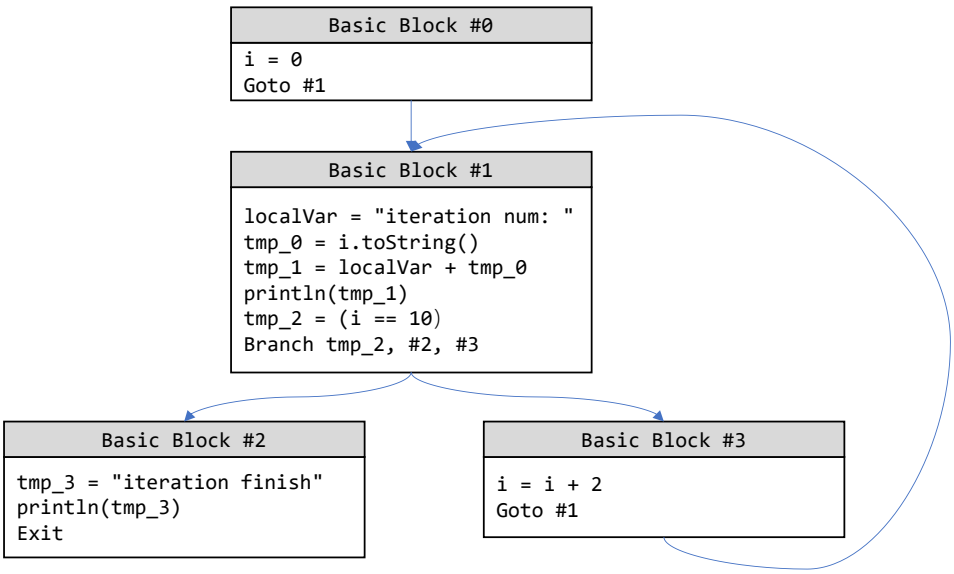

Fig.3. Flat control-flow graph representation of for-in expressions.

graph. Second, the `for-in` expression exposes the iteration variable and boundary condition explicitly in the syntax. Yet the flat control-flow graph is not expressive enough for this. For loop-related optimizations, the compiler has to analyze the cyclic edges in IR to reconstruct the information, which is complex and maybe not precise in certain cases.

In CHIR, we introduce a special kind of operations named as "Hyper Operation", where one or more "Block Group"s can be embedded. Each "Block Group" itself is a control-flow graph thus makes CHIR become a nested control-flow graph. With the "Hyper Operation"s, complex Cangjie expressions can be represented in CHIR without issues with semantic information loss. Fig.4 shows the CHIR of aforementioned `for-in` expression example. The loop variable, iteration start point, iteration end point, and the stride are all directly recorded in the "Hyper Operation" highlighted in bold.

As for the extra `flag_0` and related operations highlighted in blue, they are introduced to ensure that the execution of each "Basic Block" always reaches the end of the block and cannot be shortcut by non-local jumps such as `return`, `break`, and `continue` (although they are not shown in this example). This property of basic block is very important for the data-flow analysis in compiler. Therefore, we impose the following restrictions to enforce it.

- Trivial operations which are in the end of a basic block can only transfer the control flow to a basic block in the current block group or the exit and back to the hyper operation which the current block group

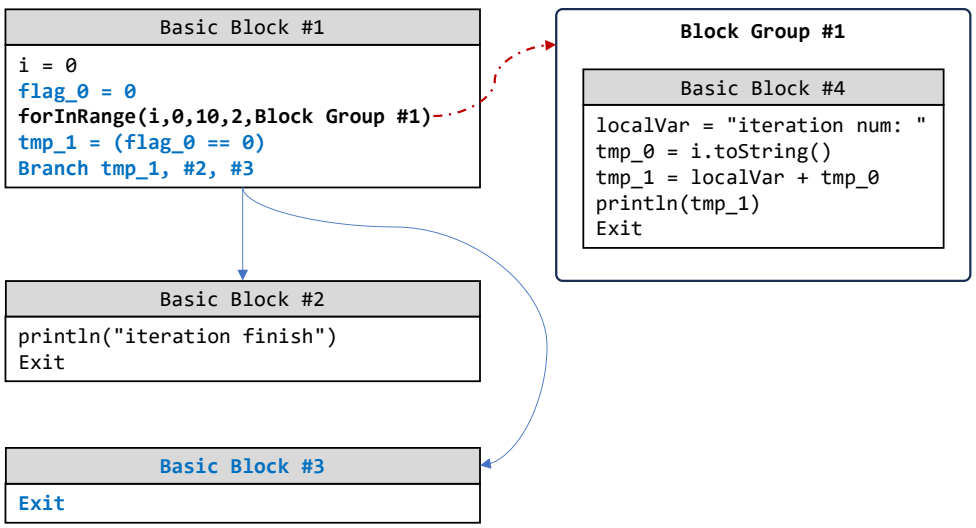

Fig.4. CHIR representation of `for-in` expressions.

belongs to.

• Hyper operations can either transfer the control flow to its subordinate block groups or the next operation in the current block, depending on its semantics.

Under these restrictions, some Cangjie expressions, such as `return`, `break`, and `continue`, need to be equivalently transformed to break their whole control flow transition into several parts. Listing 15 shows a more complicated example to illustrate this.

Listing 15

```
func f1(): Unit {
    i = 0
    while (i < 10) {
        if (i < 5) {
            println("iteration num: " +
i.toString())
        } else {
            return
        }
        i = i + 2
        println("continue")
    }
    println("iteration finish")
}
```

Its CHIR representation is shown in Fig.5. The function body of `f1` is a control-flow graph constructed by `basic block` #1, #2 and #3. The `while` and `if` expression are represented as hyper operations (highlighted in bold). As for the `return` expression, it will trigger a control-flow transition to exit the function. This control-flow transition is now divided into multiple parts. First, for each hyper operation, an extra `flag` variable and related operations (highlighted in blue color) are generated to capture the control-flow transition inside of the hyper operation and further transfer it to outer block group or exit the function. Then the `return` expression is represented by an `exit` operation along with the value set of those `flag` variables.

Moreover, when CHIR-related analysis and optimization are done, the hyper operations can be lowered to combinations of trivial operations. Then a simple round of constant propagation and dead code elimination is able to eliminate all the extra `flag` variables and related operations. As a result, the CHIR becomes a traditional flat control-flow graph.

2) *High-Level Abstractions*

Compared with low-level IRs like LLVM IR, CHIR has higher abstraction. First, there is no pointer arithmetic operation in CHIR. Second, operations with high-level abstraction are introduced to express semantics without relying on the concrete data layout. A typical example here is the "Invoke" operation. It represents a function call with dynamic dispatch semantics, without touching the implementation details including those where the virtual method table (VMT)[21] is stored.

3) *Rich Type Information*

Type declarations in Cangjie source code are represented in a similar form in CHIR to retain information including subtyping, and member variables and methods. Besides, CHIR provides built-in representation of VMT (in an abstracted form, not concrete implementation) for better analysis and optimizations

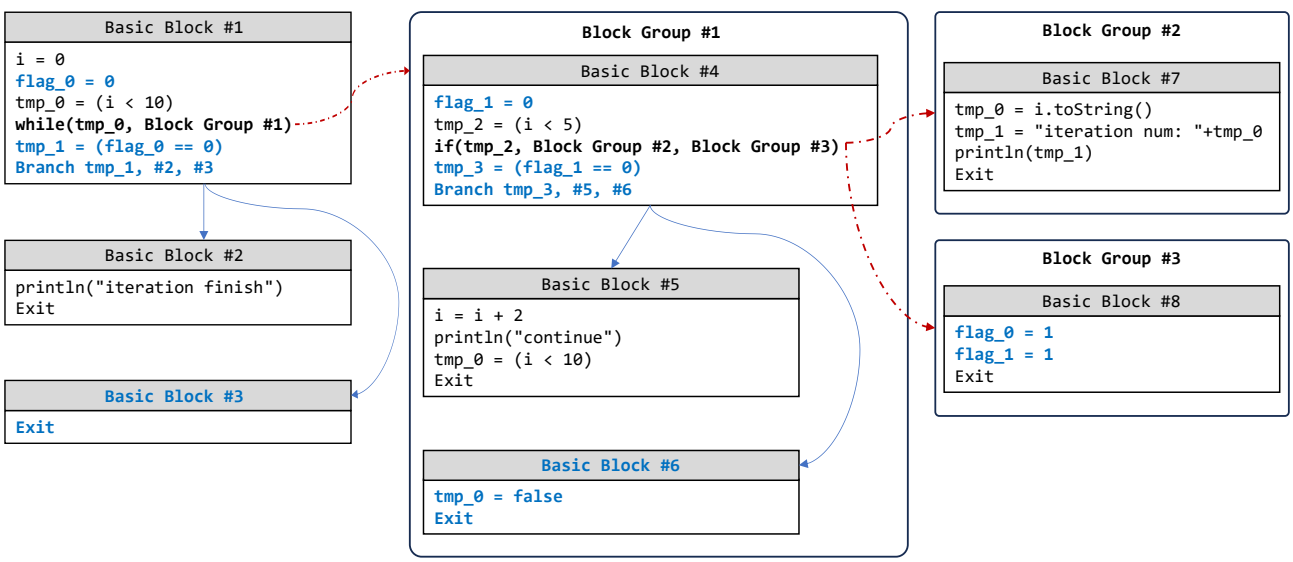

Fig.5. Break the control flow transition of `return` expression into several parts.

targeting the dynamic dispatch feature in OOP programming. For example, Listings 16 and 17 show some class declarations in Cangjie and the corresponding CHIR, respectively.

Listing 16

```
open class Base {
    let x: Int64
    let y: String
    open func foo(): Int64 {...}
    static func bar(): String {...}
}
open class Sub <: Base {
    open func foo(): Int64 {...}
}
```

With CHIR, the compiler frontend can perform a series of analysis (e.g., overflow check, division by zero check, and out-of-bound array access check) and optimizations (e.g., constant propagation, function inlining, devirtualization, and loop invariant code motion).

3.3 Lowering from CHIR to LLVM IR

After all the program analysis and optimizations on CHIR, the compiler further lowers the CHIR to LLVM IR. Since we will reuse the code generation capability of LLVM in the backend, the application binary interface (ABI)[22] of Cangjie, including type layout, calling convention and name mangling rule, is partially determined in this process.

The key challenge here is to balance the performance and back-compatibility when choosing proper type layout, calling convention and name mangling rule implementation. We will use the case of a generic function to illustrate the issue. In programming languages like C++ and Rust, the support of generics is based on compile-time instantiations. The generic functions are cloned with the generic type parameters replaced by concrete types. As a result, the execution time performance will be better since more optimizations are enabled with the concrete type information. The clone of generic functions will also leads to poor code size. Moreover, any changes on the generic functions will be a back-compatibility break. As illustrated in Fig.6, the function call in `main` will still use the cloned instantiated version of `foo<T>` in

Listing 17

```
class Base {
        "parent class": {}
        "instance member variables": {
                let x: Int64
                let y: String
        }
        "static member variables": {}
        "instance methods": {
                Base::foo
        }
        "static methods": {
                Base::bar
        }
        "VMT": {
                "foo": () -> Int64 => Base::foo
        }
}
class Sub {
        "parent class": {
                Base
        }
        "instance member variables": {}
        "static member variables": {}
        "instance methods": {
                Sub::foo
        }
        "static methods": {}
        "VMT": {
                "foo": () -> Int64 => Sub::foo
        }
}
func Base::foo(this: Base): Int64 {...}
func Base::bar(): String {...}
func Sub::foo(this: Sub): Int64 {...}
```

version 1.0 even though it has been changed.

However, the back-compatibility is important in the application development scenario for decoupling the upgrade of libraries and application. Thus, C#[23] performs specialization of generics via just-in-time compilation, and Swift[14] generates extra runtime type metadata for each generic type parameter. In Cangjie, we also treat the generic type as run-time data. As shown in Fig.7, a value with a generic type has simi-

[14] Slava Pestov. Compiling Swift generics, https://download.swift.org/docs/assets/generics.pdf, Dec. 2025.

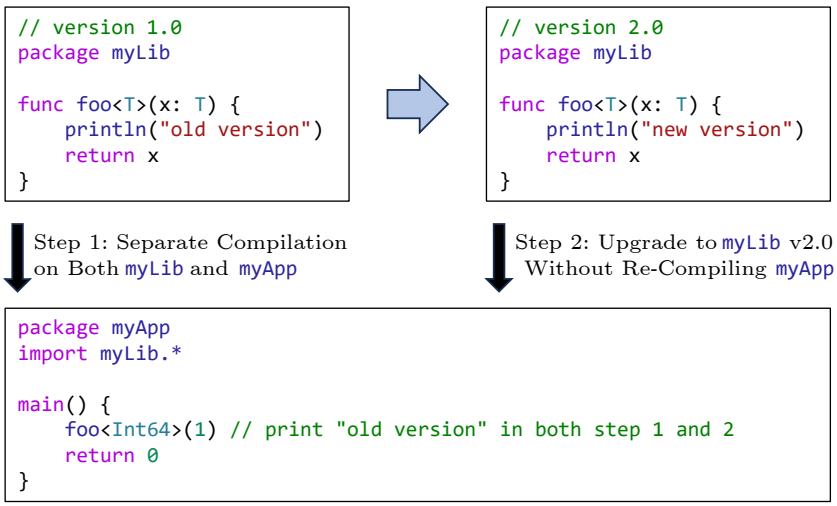

Fig.6. Example of back-compatibility break caused by changes in generic functions.

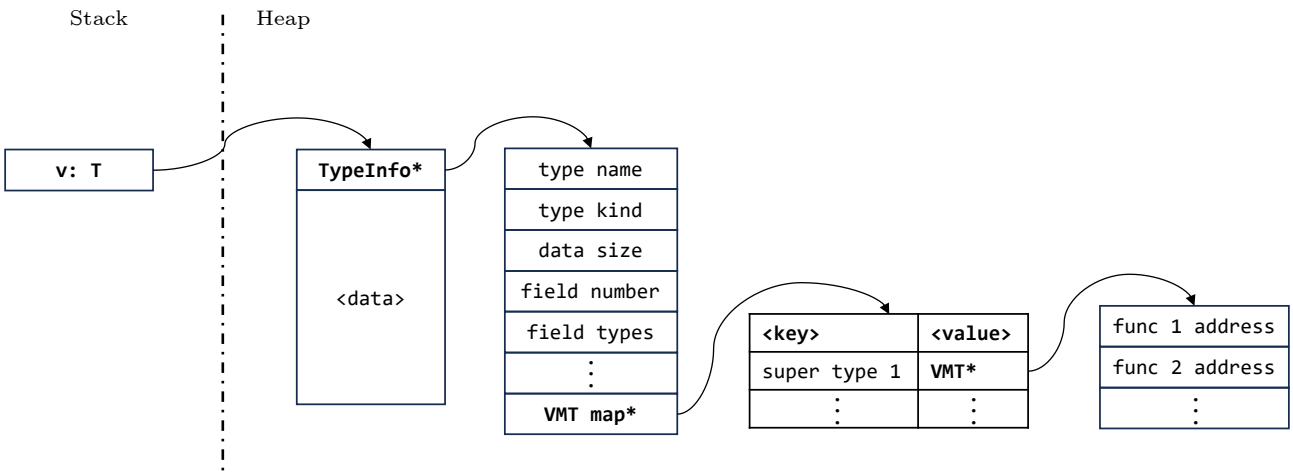

Fig.7. Layout of a value v with the generic type T.

lar layout to class objects. The TypeInfo* field points to a memory chunk which stores the information about type name, type kind, layout, VMT, etc.

Then the calling convention rule ensures the following.

• The caller of a generic function will convert the argument and return value to the required layout by box and un-box operations if generics are involved.

• The call site will construct correct TypeInfo* value of each generic type parameter and pass them as extra arguments to the callee.

Therefore, the operations in the callee generic function, including memory allocation, data copy, and method invocation can be done by querying the corresponding TypeInfo*.

4 LLVM-Based Compiler Backend

Cangjie compiler backend (as shown in Fig.8) is based on the open source LLVM platform, which is a prosperous ecosystem and provides mature compiler infrastructures. LLVM supports modular pass mechanisms, which make it easy to add new analysis or transformation passes to implement the Cangjie compiler. It also provides a powerful code generator to produce optimized machine code for multiple hardware platforms such as aarch64 and x86_64, which can be reused to save significant amount of engineering work.

Most of our extensions to LLVM are GC-related. Although LLVM does provide some infrastructural supports for GC, such as statepoints and safepoints, some of the transformations are unsound or less optimized for moving GC.

First of all, the current infrastructure cannot track precisely all the addresses of heap memory. Some transformation or optimization may derive pointers pointing to the middle of structs by adding offsets to the base addresses. Although they work for non-moving GC, they are unsound for moving GC

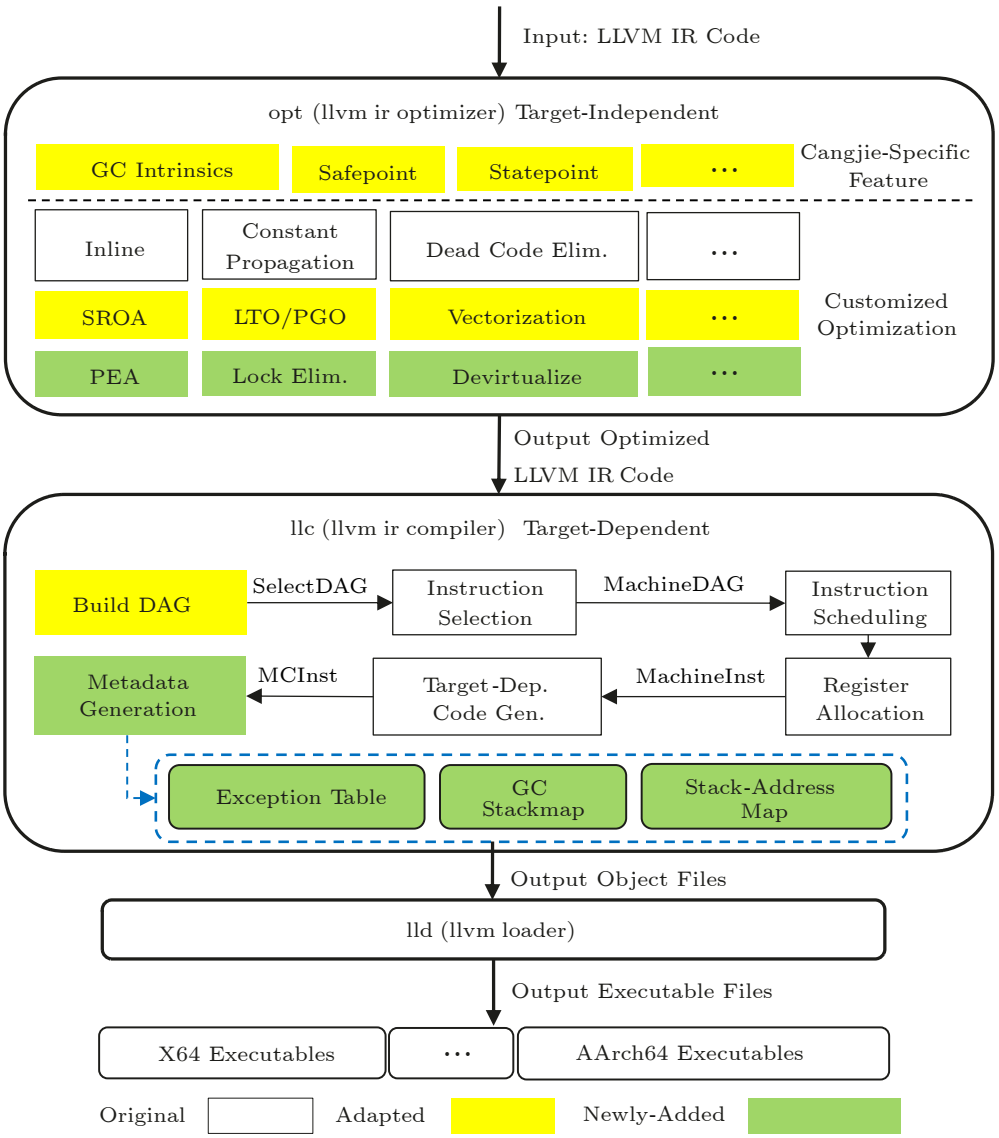

Fig.8. Cangjie compiler backend.

since moving GC needs to update pointers pointing to heap objects when objects are moved. These derived pointers cannot be effectively traced and updated. In order to track all the pointers and derived pointers, we extend the LLVM IR by adding several groups of intrinsics for all heap operations, as described in Subsection 4.1.

Second, due to the precision problem with some of the alias analysis or liveness analysis in LLVM, the analyzed lifetime of a heap pointer could be longer than the actual case. Therefore dead objects may not be reclaimed in time during the upcoming round of GC. We add a few passes to solve this problem (see Subsection 4.2).

Third, LLVM's current encoding of stackmaps is not fully optimized. We try to further reduce the size of stackmaps, as discussed at Subsection 4.3.2.

Furthermore, Cangjie has value types (such as structs), whose local instances are inlined on stack. This makes frame layout more complicated; thus it is error-prone to generate precise and efficient stackmaps for Cangjie. We also add new passes to address this problem, as shown in Subsection 4.2.

To the best of our knowledge, the Cangjie compiler gives the first open source implementation of moving GC on an LLVM-based infrastructure.

4.1 Extension to LLVM IR

Cangjie implements a concurrent compacting GC

algorithm to minimize the latency and memory fragmentation; hence, expressing heap memory operations in LLVM IR becomes important. Officially LLVM uses the decorator `addrspace(1)` to mark the addresses of managed heap. However, the load and store instructions with pointers decorated by `addrspace(1)` may be improperly handled by some optimization passes. For instance, the `addrspace(1)` decorator may be dropped after some transformation. These optimizations are problematic for moving GC since each address of the heap memory must be tracked precisely, including the derived addresses pointing to the middle of structs. To address this problem, we introduce Cangjie-specific intrinsics for memory accesses with `addrspace(1)` decorated addresses, then the default LLVM IR optimizer will keep these operations as untouched. Optimizations on these intrinsics will be done in custom passes. Several groups of intrinsics are added to LLVM IR, which are all about heap memory operations, including:

- allocating memory from heap to create a new Cangjie objects;
- reading from reference fields of heap objects (called read barriers);
- writing to reference fields of heap objects (called write barriers);
- reading and writing of references, such as atomic compare-and-swap operations over references (called fused barriers);
- retrieving the starting address of array payload, which is used for coordinating C foreign function interfaces (C-FFI) and GC.

Note that some intrinsics with similar semantics may be already defined officially in LLVM language[15]. Cangjie backend chooses to add custom intrinsics for its own rather than to reuse these existing intrinsics, because the semantics of these intrinsics may be a little different in a subtle way from Cangjie language, especially these intrinsics may be coupled with some existing optimization passes. Modifying the semantics and the corresponding optimizations for these intrinsics would be very difficult. Moreover, it would be difficult to keep synchronized with the evolution of the official LLVM. Thus we choose to add custom intrinsics for Cangjie, which makes no change to the original intrinsics in LLVM and is easy to maintain. Since all the other LLVM IR instructions used by Cangjie are officially defined in LLVM language, they are highly stable right now. Only a little maintenance work is needed for Cangjie backend to follow the evolution of LLVM IR.

As introduced in Subsection 3.3, CHIR code can be converted to extended LLVM IR code, which is further optimized by LLVM IR optimizer (named as `opt`). These intrinsics are eventually lowered to machine code at the code generation stage (called LLVM `llc`).

4.2 Optimizing LLVM IR

There are about 150 passes altogether executed in LLVM opt to analyze or optimize Cangjie LLVM IR code, which can be divided into the following three classes.

- Twenty-six passes are adapted from existing passes. For instance, `PlaceSafepoints` is adapted to insert safepoints in Cangjie functions so that unnecessary safepoints can be optimized; `LoopUnrollPass`, `LoopVectorizePass`, `SLPVectorizerPass`, `VectorCombinePass`, etc. are enhanced for better optimization of Cangjie-specific intrinsics.
- Fifteen passes are newly added. For instance, `CJPartialEscapeAnalysisPass` is added so that unescaped objects can be allocated on stack, which reduces memory management pressure for GC. `CJBarrierOpt` is added to optimize heap memory access operations expressed in Cangjie-specific intrinsics. `CJRewriteStatepoint` is used to track precisely the addresses of heap memory according to liveness analysis and the layout of Cangjie data structures for GC.
- All the other passes are unchanged.

4.3 Code Generation

The machine code generation stage includes passes for instruction selection, register allocation, target-dependent optimization, etc. Most of the passes are unchanged, except for some Cangjie-specific optimizations.

4.3.1 Machine Code Level ABI

Cangjie machine code ABI follows the rules defined in the standard ABI specifications for each hardware platform, including the calling convention, frame layout, etc. This is the basis to support very

[15] LLVM contributors. LLVM language reference manual, https://llvm.org/docs/LangRef.html, Dec. 2025.

low-cost C-FFI invocations from Cangjie code. In addition, Cangjie introduces the following two customized rules.

- Frame-pointer (the base address of the current stack frame) is always explicitly maintained (similar to `-fno-omit-frame-pointer` for the C/C++ compiler), which is helpful for stack unwinding.
- A hardware register is reserved as the so-called thread register to speed up accesses of thread-local data.

4.3.2 *Metadata Generation*

Metadata is one of the most important parts for the runtime of managed languages. As mentioned in Subsection 3.3, some kinds of metadata are already generated when CHIR code is converted to LLVM IR code, including type information (of both concrete type and generic type), field metadata, method metadata, virtual function tables, interface function tables, etc. There are three kinds of metadata newly generated by the backend as listed below. They all depend on the specific layout of stacks, therefore the generation is postponed to the code generation stage.

- *Exception Table.* The exception table is used by Cangjie runtime when an exception is raised. Runtime queries the exception table with the address of the instruction (usually a function call) which raises the exception.
- *GC Stackmap.* GC stackmap is a sequence of pairs consisting of a program counter (PC) and a bit string. The PC represents the position of a GC safepoint or the beginning of a GC safe-region. The bit string encodes information about registers and slots in the current stack frame which hold pointers or derived pointers pointing to heap memory. These pairs are used to locate GC roots in the register context or stack frame. When GC is triggered at some PC, it uses the corresponding bit string to retrieve roots to trace live objects. These pointers on stack or registers are also updated when the objects they point to are moved in the moving GC. Since the sets of live variables (roots) could be different at different program points (i.e. PCs), we maintain multiple pairs in the stackmap to record this mapping. Adjacent pairs in the stackmap may have the same bit strings, then they are merged as one bit string paired with an instruction address range, which reduces the size of stackmap.
- *Stack-Address Map.* The stack-address map has almost the same format as the stackmap, but the bit strings have different meanings. They record the registers and stack slots that contain pointers pointing to objects on the stack instead of on the heap. Cangjie needs the stack-address map because it supports stack expansion, which moves the current stack to a bigger one. When stack frames are moved, pointers pointing to stack objects need to be updated accordingly.

5 Runtime

Cangjie runtime consists of the following key functionalities.

- *Package Loading.* At the beginning of program execution, the runtime first loads the package containing the program entry, and then loads the dependent packages recursively. For each package, the runtime registers the package information, resolves all the symbols, and relocates all the symbol references.
- *Thread Management.* The functionality includes user-level thread creation, scheduling, and synchronization. We give more details in Subsection 5.1.
- *Stack Unwinding.* It traverses the call stack reversely (from callee to caller) during runtime. Stack unwinding is necessary for GC, exception handling, stack expansion, debugging or testing when stack traverse is needed to collect related data. Since stack unwinding happens quite frequently, we maintain frame-pointer during runtime to quickly rebuild the function call chain.
- *Exception Handling.* Exception handling is a modern structured design for handling occurred errors. Searching for the exception handler during stack traverse is based on stack unwinding as mentioned above. Another critical action for exception handling is restoring the register context. The traditional libunwind traverses stack frames and restores the whole register context according to the DWARF-format[16] debugging data related to each frame, which introduces significant runtime overhead. Since all the registers which need to be restored are maintained in function prologue and epilogue, we encode the information to speed up register context restoration rather than relying on the DWARF data.
- *Reflection.* It provides the fundamental mechanism to implement the reflect package in the Cangjie

[16]DWARF debugging standard. https://dwarfstd.org/, Dec. 2025.

standard library, including manipulation of metadata such as package information (info), type info, field info, method info, etc. Some metadata is dynamically generated if universal generic types are involved.

• *GC*. Cangjie implements a concurrent compacting GC for automatic memory management. It is the most important and complicated functionality of the runtime. We give the details in Subsection 5.2.

5.1 Thread Management

Cangjie supports lightweight user-mode stackful threads, which are called "Cangjie threads" to be distinguished from kernel-mode threads (a.k.a. system threads). Cangjie threads are more lightweight in that thread management (e.g., context switch and scheduling) can be much faster since it avoids the overhead of crossing the user mode and the kernel mode. Moreover, the stack allocated for each Cangjie thread can be much smaller than the kernel-mode thread.

Cangjie thread management includes several important features.

• *User-Mode Context Switch and Scheduling*. The relationship between Cangjie threads and system threads is shown in Fig.9. Each system thread runs a scheduler, which manages multiple Cangjie threads in the thread queue (the ready queue). If a Cangjie thread is blocked, it does not block the underlying system thread. Instead, the system thread puts the blocked Cangjie thread into a blocking queue and schedules another Cangjie thread from the ready queue to execute. If the ready queue becomes empty, the system thread can take a Cangjie thread from the ready queue of another system thread to execute, which is known as work stealing.

• *Preemption*. Unlike system threads, which can be preempted by incoming interrupts and the subsequent scheduling at the end of the interrupt handlers, user-mode threads cannot rely on hardware interrupts to implement preemptive scheduling. Instead, preemption code is inserted into function prologue and at the back edge of loops. Each time a function is invoked, the preemption code is executed to check whether it is proper for the current Cangjie thread to release the CPU resource, and if so, the scheduler puts it back into the ready queue and picks another to execute. Although the scheduling is caused by a voluntary yield of the CPU resource from the Cangjie thread code, the preemption code is automatically inserted and not visible in the user code. Therefore the user still does not have the control of the program points where to yield the control. Therefore, the scheduling is still viewed as preemptive.

• *Stack Expansion*. The stack of Cangjie thread

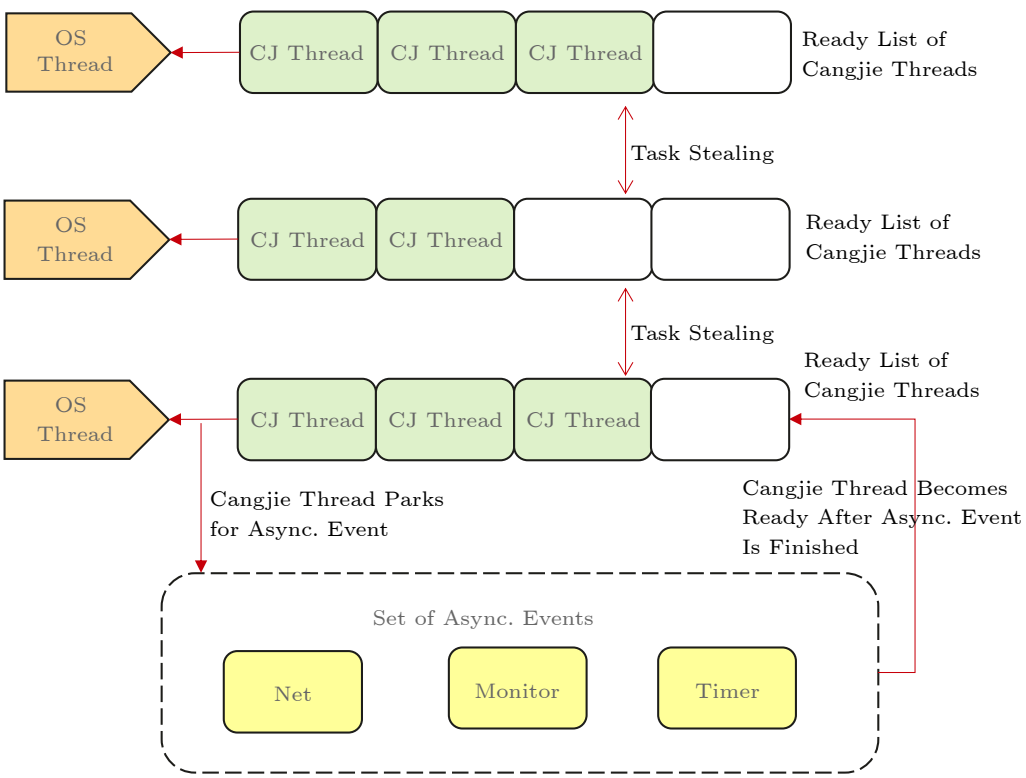

Fig.9. Cangjie thread scheduling.

can be expanded if the stack size reaches its specified limit. During stack expansion, the original stack data is copied to a new stack area, and then all the stack addresses stored in the original stack data are updated accordingly. These addresses are traced by the stack-address map, as explained in Subsection 4.3.2.

For some blocking APIs, especially the network programming libraries, Cangjie runtime intercepts the API invocation. If the execution is blocked, the system thread picks a different Cangjie thread to run. When the blocking condition is satisfied, the blocked thread becomes ready to execute.

The runtime also provides a set of synchronization libraries for Cangjie thread, such as monitors, semaphores, and blocking queues, which may also block a Cangjie thread and trigger scheduling.

5.2 Concurrent Compacting GC

For mobile application development, the compute resource (CPU and memory) is limited, and end-users are sensitive to prompt UI responses. Therefore Cangjie tries to implement a highly optimized concurrent compacting GC to achieve low latency, small memory footprints, and low overhead. It reduces the average GC latency to 1 millisecond or less (see Subsection 6.2), and the average heap memory footprint to 20% less than Java ART (see Subsection 6.3).

- *Concurrent GC Algorithm to Minimize Stop-The-World (STW) Latency.* We introduce a novel pre-compact phase with lighter synchronization with application code. It only requires a semi-STW so that the application can resume execution immediately after processing the roots on stack, without waiting for the GC to finish its processing of the global roots (see Subsection 5.2.4).
- *Heap Compaction Instead of Copying to Reduce Peak Memory Consumption.* The GC does heap compaction to avoid memory fragmentation and enable fast memory allocation. We choose compaction over copying to avoid reserving a separate to-space, which increases the peak memory consumption (see Subsection 5.2.3).
- *Thread-Local Regions and the Bumping-Pointer Allocator.* The whole heap is divided into equally sized blocks called regions. Each thread holds a local region for allocation to avoid contention. Furthermore, heap compaction allows us to apply the bumping-pointer allocator, which allocates memory with an average of 10 machine instructions, much faster than free-list allocators (see Subsection 5.2.2).
- *Optimizing GC Barriers.* Stale pointers caused by object copy are tagged to be distinguished from valid pointers. Based on the distinction, we can optimize the read barrier by introducing a fast path to read valid pointers (see Subsection 5.2.5).

5.2.1 Outline of the GC Algorithm

The GC algorithm consists of five basic phases in its core loop, as briefly described below (Fig.10).

- *Idle Phase.* It is the phase between two rounds of GC execution.
- *Enum (and Fix) Phase.* All global roots and stack roots are enumerated in this phase for the following trace phase.
- *Trace (and Fix) Phase.* This phase traverses the whole heap starting with all GC roots to mark all reachable objects.
- *Pre-Compact Phase.* During this phase, objects pointed by GC roots are moved to the to-regions if these objects are originally in the from-regions.
- *Compacting Phase.* All live objects in each from-region are copied to the to-regions if they are not yet copied. The mapping from stale objects in the from-regions to the corresponding new objects are recorded in the forwarding table.

The execution of GC transits between these phases. Before entering a phase, the GC thread first re-

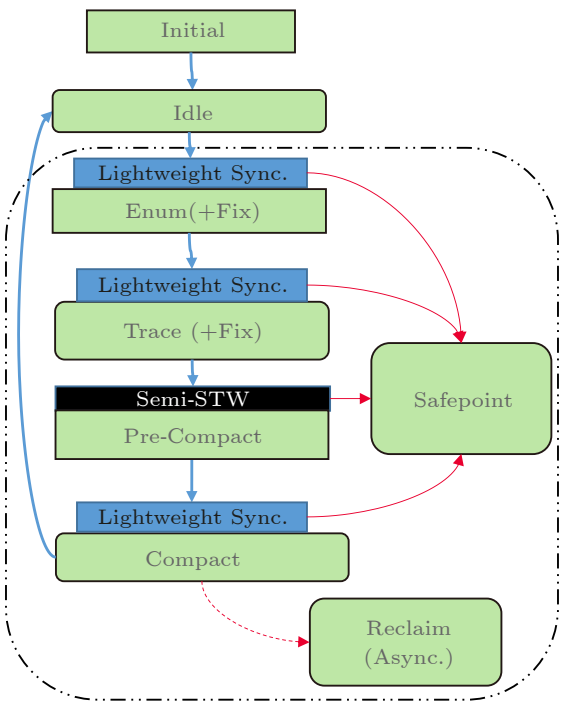

Fig.10. Phases of Cangjie GC.

quests for synchronization with all Cangjie threads by activating safepoints. Code is inserted at certain places in Cangjie threads to check the status of safepoints. If the safepoints are activated, Cangjie threads should respond to the GC request before continuing execution. After all Cangjie threads have responded to the GC request, GC synchronization is finished, and the Cangjie threads continue to execute.

During the pre-compact phase and the compacting phase, live objects in from-regions (from-objects) are copied to to-regions, and then objects in from-regions are dropped. Pointers to these copied from-objects become invalid and should be updated to point to the corresponding to-objects. These invalid pointers are called stale pointers. Updating the stale pointers is called fixing, which happens after the compacting phase is finished. Usually fixing is performed during heap traverse by tracing heap or checking live regions one by one, which is rather costly; thus fixing is delayed to the next GC loop and combined to the enum phase and the trace phase, marked by "(+fix)" in Fig.10. The design is inspired by [24] and [25].

Also note that the reclaim phase is not mandatory in GC's core loop. After a GC loop is finished, unreclaimed regions are reclaimed asynchronously for the sake of performance.

Several critical design decisions of Cangjie GC are further explained as below.

5.2.2 *Heap Layout*

Currently, Cangjie heap consists of one continuous virtual address space, which is then divided into equal-sized blocks named as regions. During its whole lifecycle, a region becomes a free region after initialized, and then it becomes a thread-local region for object allocation. As the number of allocated objects increases, the size of usable memory decreases, which may eventually cause allocation failure. Then a thread-local region becomes a full region, which then becomes a from-region during GC. After a from-region is compacted, it becomes a reclaimed region and ready for later allocation. A reclaimed region becomes a released region if the runtime informs OS that the physical pages cached for this region can be withdrawn. These states are maintained in region types, which is part of the region metadata called region-info. The allocated size for a thread-local region is also maintained in the region-info. The start addresses of regions are linearly mapped to the addresses of the corresponding region-info, which makes it very efficient to check the region which contains a specified object in Cangjie heap.

A Cangjie thread allocates objects from a thread-local region which maintains a pointer in the region info to record the start address of allocatable memory. To allocate an object of a given size, Cangjie runtime moves this pointer to higher address according to the given size. This procedure is very efficient (about 10 machine instructions on average) and named as "bumping-pointer allocation".

5.2.3 *Heap Compaction*

Moving GC algorithms are usually chosen to reduce fragmentation of heap memory. One of the widely used moving GC algorithms is known as copying GC, which copies live objects in the from-space to the to-space, and then update the stale pointers pointing to the original objects with new pointers pointing to their copies in the to-space. After that, the from-space can be reclaimed. As the from-space and the to-space coexist for a short period of time during GC, the peak memory consumption becomes higher during this time (see Fig.11). Eventually, the overall performance of mobile systems is impacted. To eliminate the peak of memory overhead, Cangjie implements a compacting algorithm. It compacts a selected memory region by copying all live objects from this region (the from-region) to an empty target region (the to-region), or to the same region if the in-place strategy is taken. The mapping from the stale objects to the new ones is stored in a so-called "forwarding table". After a selected region is compacted, it can be reclaimed immediately for later allocation. Fig.12 shows why the compaction algorithm can reduce peak memory consumption. The peak memory footprint consists of six regions for copying, whereas, the peak memory footprint consists of four regions for compaction.

With the help of the forwarding table, fixing stale pointer can be delayed until it is done by read barriers or in the enum phase or trace phase of the next

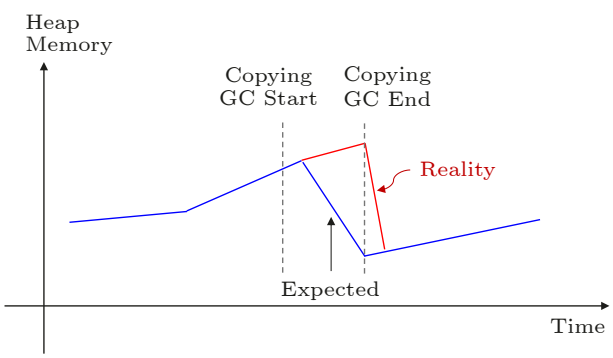

Fig.11. Memory footprint of copying GC.

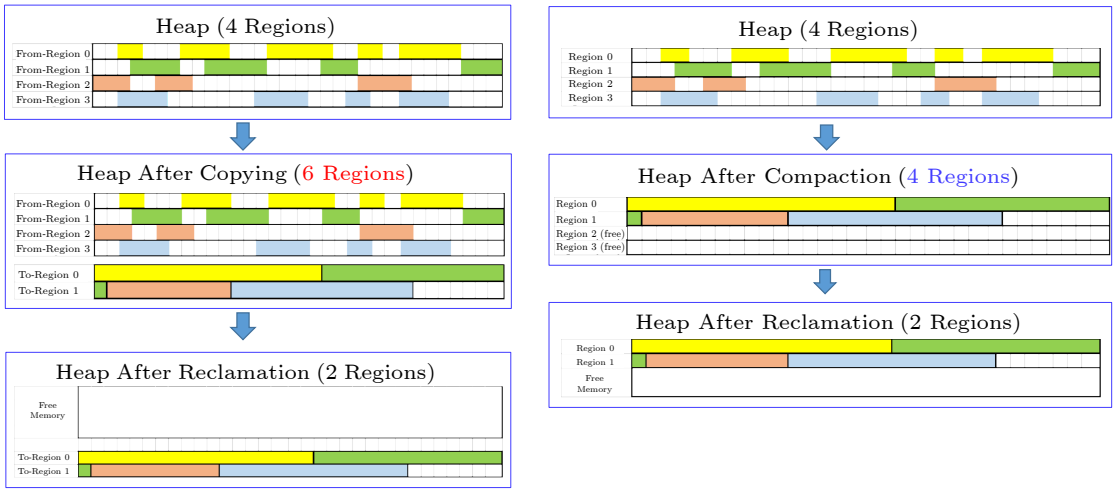

Fig.12. Comparison of copying and compaction. (a) Copying. (b) Compaction.

round of the GC loop. This optimization helps to reduce the GC overhead.

5.2.4 *Minimizing Stop-The-World Latency*

Cangjie tries to minimize Stop-The-World (STW) latency in different phases. First of all, the STW for the enum phase is optimized with mixed pre-write barriers and post-write barriers as Sapphire GC[26] or Golang GC[17] does. Moreover, the STW for the pre-compact phase is replaced with a semi-STW (see Fig.13), which is a lightweight synchronization: when the GC issues a synchronization request, all the application threads stop at the synchronization point, and then they start simultaneously relocating stack roots pointing to the from-objects. That is, objects pointed by stack roots are copied to some to-regions and then the stack roots are updated to point to these new copies. Once the stack roots are updated, the application threads can resume their execution immediately,

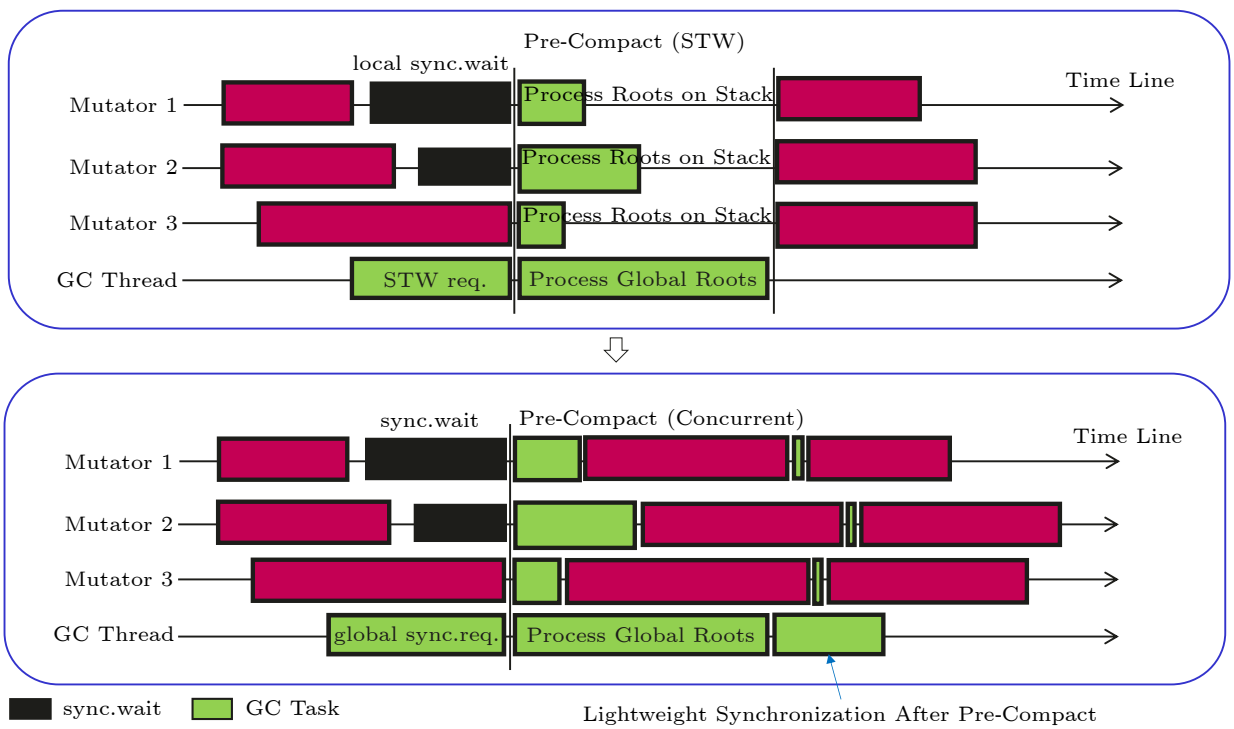

Fig.13. Elimination of Stop-The-World

[17] A guide to the Go garbage collector. https://go.dev/doc/gc-guide, Dec. 2025.

without waiting for the GC thread to finish processing the global roots.

5.2.5 *Optimizing GC Barriers*

As described above, after objects are copied, fixing stale pointers can be delayed to the next GC loop, therefore, the stale pointers and fixed valid pointers coexist for some period of time. Reading the stale pointer is guarded by read barriers' slow path, while reading valid pointers needs nothing more than an ordinary load operation, which is a fast path of read barriers. This distinction allows us to optimize barriers. Therefore, if a pointer to a from-object is visited during the enum and trace phase, it is tagged as a stale pointer in some of the unused higher 16 bits. After the trace phase, if a pointer is not tagged, it must be a valid pointer and access to it takes a fast path of read barriers. Then, during the pre-compact phase, the stale roots are relocated, thus becoming valid pointers. During the compact phase, the from-objects are moved to to-objects. Any stale pointer visited by the read barrier is either relocated if the object it points to is not yet moved, or updated if the object it points to is already moved. This stale pointer becomes a valid pointer. All remaining stale pointers are delayed to the next GC cycle and be fixed during its enum or trace phase.

5.3 Runtime Support for C-FFI

Like JNI[18] for Java, Cangjie has built-in support to invoke C functions. We show the language support of C-FFI in Subsection 2.3. Here we introduce the runtime support.

The default procedure to invoke a C function takes the following steps:

- convert parameters of Cangjie format to C-style;
- save the calling context and mark the start of a safe-region for GC;
- prepare the incoming parameters for the C function by putting them in registers or stack memory;
- execute the C function;
- prepare the return value;
- mark the end of the safe-region for GC, restore the calling context, and return.

If a pointer pointing to a Cangjie object is passed to a C function, this Cangjie object must be pinned and should not be moved during Cangjie GC. The C function executes on the same stack as the calling Cangjie function. During the execution of the C function, Cangjie stack expansion must be disabled until the C function returns.

Cangjie optimizes the above standard process from several aspects. First, data to be passed to C functions need to be annotated with the `@C` annotation. The layout of the data conforms to C ABI. Cangjie functions to be called from C are annotated as well. The calling convention of these functions also conforms to C. This way we can save the overhead of data conversion and preparation of parameters. Second, we introduce the `@fastnative` annotation to annotate C functions whose execution time is very short. There is no need to record events of entering and leaving a GC safe-region for these C functions.

6 Performance Evaluation

The performance of Cangjie language is evaluated from three perspectives in this paper.

- *Execution Time*. We implement test cases proposed by the Programming Language Benchmarks Game[19] in Cangjie and compare the execution time with other languages like Java and Golang. We also evaluate the performance of Cangjie C-FFI.
- *GC Latency*. We design a novel test to estimate the impact of GC latency as shown in Subsection 6.2. It shows that Cangjie behaves very well with most of GC latency less than 1 millisecond.
- *Memory Footprint*. It is evaluated in Subsection 6.3. The test result shows about 20% reduction of memory footprint compared with Java on Android. The design of Cangjie object header and heap compaction makes the major contribution to this advantage.

6.1 Execution Time

We pick the well-known Computer Language Benchmarks Game[19] and implement most of them in Cangjie. Each case is executed five times with the configuration of hardware platform which is described in Table 1. The average execution time is taken as the final result, which is compared with those of Golang and Java on the same platform.

The average execution time (Geo. Mean in Tab-

[18] Java native interface specification. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html, Dec. 2025.
[19] The Computer Language Benchmarks Game. https://madnight.github.io/benchmarksgame/, Dec. 2025.

Table 1. Average Execution Time (ms) over Benchmarks Game

Language	binarytrees	fannkuchredux	fasta	mandelbrot	nbody	pidigits	regexredux	revcomp	spectralnorm	Geo. Mean
Cangjie	5.96	0.23	2.20	1.51	10.50	20.88	9.85	2.73	10.13	4.02
Go	18.48	9.89	3.00	1.82	10.48	1.65	29.35	1.78	10.09	5.93
Java	3.10	1.54	2.81	2.59	12.58	12.41	8.97	4.51	6.67	4.84

Note: Cangjie: compiler v1.0.0, Ubuntu 18.04/ARM64; Go: version 1.19.2, Ubuntu 18.04/ARM64; Java: OpenJDK version 1.8.0_362, Ubuntu 18.04/ARM64.

le 1) of Cangjie is the least. The highly-optimized LLVM platform is a good help of course. Besides, high-level language-aware analysis and optimization done on CHIR and custom LLVM IR passes discussed in Subsection 4.2 are the most significant factors. For some cases, e.g. pidigits, Cangjie uses more time because read barriers are heavily executed in this case. Read barriers are necessary for Cangjie's concurrent compact GC, but not used for Java (G1 GC) or Golang. The overhead of read barriers can be reduced further through some ongoing optimization. Also, at the time of the test, we notice that some of existing optimization passes do not take effects due to those newly added custom intrinsics in Cangjie. For instance, auto-vectorization does not work because load/store operations accessing heap memory are expressed in Cangjie's custom intrinsics which are function calls rather than standard LLVM IR. Inlining cost is also affected by these custom intrinsics. But we do not see these as fundamental problems of the language, which can be addressed through corresponding compiler optimizations.

We also evaluate C-FFI performance with primitive types, strings and C-compliant structs passed as arguments. C-FFI invocation is wrapped in a loop with 100 000 iterations. The execution time is shown in Fig.14. (Due to space limit, we split the data set and show them in two sub-figures.) The average execution time for Cangjie is 113.84 ms, Java 139.72 ms, and Golang 158.25 ms. Cangjie performs slightly bet-

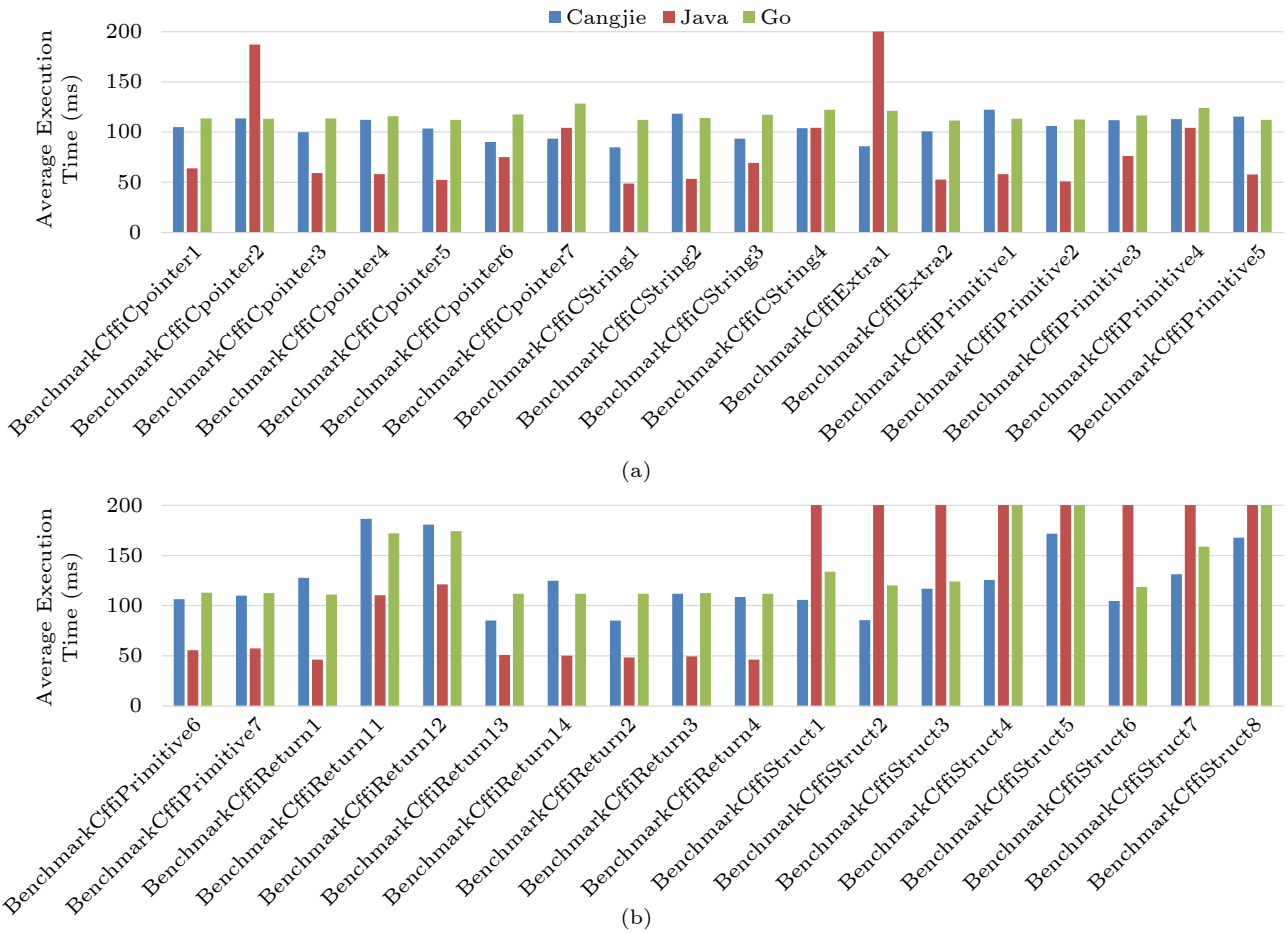

Fig.14. Evaluation of C-FFI

ter than the other two languages. The main reason is discussed in Subsection 5.3.

6.2 GC Latency

We design a simple and efficient way to estimate GC latency. Here is the procedure.

• First of all we choose a special test case in Benchmarks Games called `binarytrees`. This case creates a number of threads, and each thread creates a number of binary trees of different heights. Since the number of nodes is exponential to the height of a tree, the memory pressure in this case is easily configurable and can be set very high.

• Then we create an extra thread which contains an infinite loop. The loop body is almost empty, except for recording the time of entering and leaving each round of the loop. Then the time between leaving the previous round of the loop and entering the current round is calculated as an estimate of latency. Because safepoints for GC and preemption are inserted in the loop's back edge, this latency reflects the delay caused by GC and scheduling. This extra thread is actually an observer of latency.

• After running `binarytrees` alongside with the observer thread, a lot of latency data is produced. Then we sort the data and analyze the tail latency.

Two platforms are tested for tail latency. The first platform is Ubuntu 22/x86_64, with the maximum height of the binary trees set to 20. Equivalent versions written in Go and Java are also tested, respectively. The Java version is executed with two different GC algorithm: G1 GC, and ZGC. During the execution of `binarytrees` modified as mentioned above, about 10 000–50 000 latency samples are generated, which are sorted by values, and then tens of the top latency samples are picked to evaluate the tail latency. The top 30 records of latency are shown in Fig.15 for comparison between these four different language implementations. The result shows that Cangjie performs very well with the maximal latency 5.8 ms which is the best of all the four test objects.

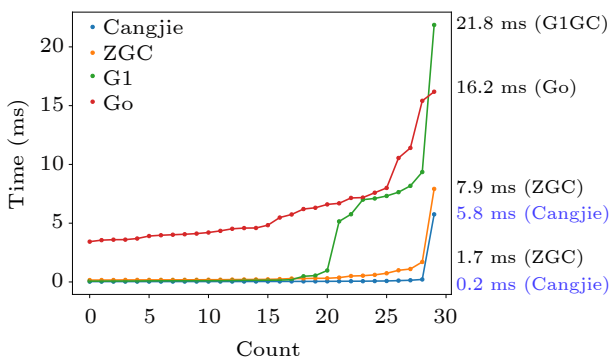

Fig.15. Evaluation of tail latency (top 30) on Ubuntu servers.

We also run the test on Huawei Mate60 Pro. The Cangjie version runs on Android and Harmony OS respectively, and the Java version runs on Android. The top 50 latency samples are picked, as shown in Fig.16. Several samples with latency of more than 200 ms are recorded for Java on Android; meanwhile, the maximal latency recorded for Cangjie on Harmony and Android is less than 20 ms.

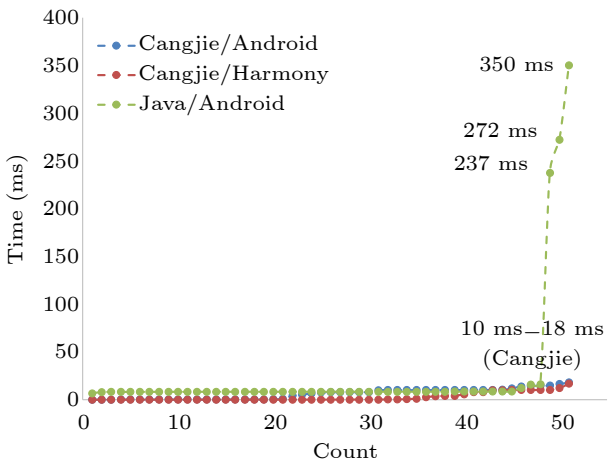

Fig.16. Tail latency (top50) on Huawei Mate60 Pro.

The reason that Cangjie achieves the low latency is discussed in Subsection 5.2. Please also note this method of introducing an observer thread for latency can be easily applied to other test cases rather than `binarytrees`, as long as the test cases can produce high (and configurable) memory pressure.

6.3 Memory Footprint

To evaluate memory efficiency, we develop a test suite consisting of a series of small test cases which create objects or arrays with various member fields of different types. The number of created objects or arrays is configurable; thus this test suite covers common use cases of heap memory, which can be used to evaluate the memory utilization. Eventually, a suite with about 80 test cases is built for both Cangjie and Java, and we run it on Huawei Mate60 Pro with Harmony OS and Android, respectively. The result is shown in Fig.17. (The dataset is split and shown in two sub-figures.)

The memory footprint of Cangjie cases is about 20% less than that of Java. There are two major contributing factors.

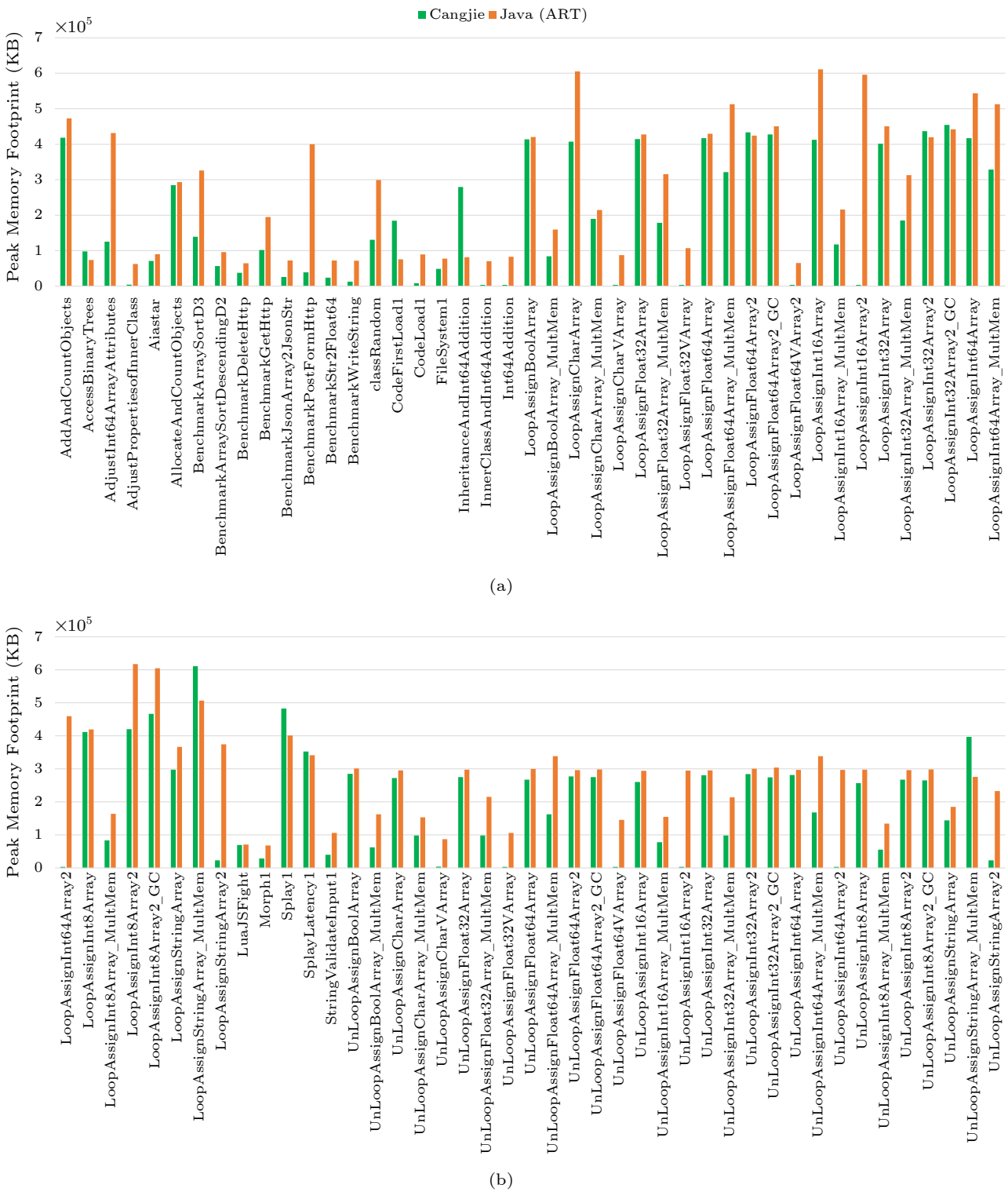

Fig.17. Evaluation of memory efficiency.

• Cangjie 's object header is more compact than Java's.

• Cangjie's compact GC lowers the peak memory overhead (see Subsection 5.2.3).

7 Conclusions

Cangjie has been applied to develop applications and frameworks in various scenarios, including mobile applications on HarmonyOS Next, UI frame-

works, micro-services and frameworks, agent domain-specific languages (DSLs), etc. The language design and implementation try to achieve a nice balance between various expectations, including simplicity, expressiveness, extensibility, safety, and performance.

As a new language, there are still lots of interesting future work to further improve the language, including better ownership and lifetime analysis for safer resource management and concurrency programming, better support of LLM-based program generation, and more systematic support of cross-platform programming and cross-language interoperability.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Pierce B C, Turner D N. Local type inference. *ACM Trans. Programming Languages and Systems*, 2000, 22(1): 1–44. DOI: 10.1145/345099.345100.

[2] Klabnik S, Nichols C. The Rust Programming Language (2nd edition). No Starch Press, 2023. Burlingame, CA 94010-4093, USA

[3] Lagarias J C. The $3x + 1$ problem and its generalizations. *The American Mathematical Monthly*, 1985, 92(1): 3–23. DOI: 10.1080/00029890.1985.11971528.

[4] Hudak P, Peyton Jones S, Wadler P, Boutel B, Fairbairn J, Fasel J, Guzman M M, Hammond K, Hughes J, Johnsson T, Kieburtz D, Nikhil R, Partain W, Peterson J. Report on the programming language Haskell: A non-strict, purely functional language version 1.2. *ACM SIGPLAN Notices*, 1992, 27(5): 1–164. DOI: 10.1145/130697.130699.

[5] Odersky M, Altherr P, Cremet V, Emir B, Maneth S, Micheloud S, Mihaylov N, Schinz M, Stenman E, Zenger M. An overview of the scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004. https://infoscience.epfl.ch/server/api/core/bitstreams/b7b0b562-3978-4af6-8c04-04c8b304935e/content, Jan. 2026.

[6] Racordon D, Buchs D. Featherweight swift: A core calculus for swift's type system. In *Proc. the 13th ACM SIGPLAN International Conference on Software Language Engineering*, Nov. 2020, pp.140–154. DOI: 10.1145/3426425.3426939.

[7] Cui C, Jiang S, Oliveira B C D S. Greedy implicit bounded quantification. *Proceedings of the ACM on Programming Languages*, 2023, 7(OOPSLA2): Article No. 295. DOI: 10.1145/3622871.

[8] Dunfield J, Krishnaswami N. Bidirectional typing. *ACM Computing Surveys (CSUR)*, 2022, 54(5): Article No. 98. DOI: 10.1145/3450952.

[9] Dunfield J, Krishnaswami N R. Complete and easy bidirectional typechecking for higher-rank polymorphism. *ACM SIGPLAN Notices*, 2013, 48(9): 429–442. DOI: 10.1145/2544174.2500582.

[10] Zhao J, Oliveira B C D S, Schrijvers T. A mechanical formalization of higher-ranked polymorphic type inference. *Proceedings of the ACM on Programming Languages*, 2019, 3(ICFP): Article No. 112. DOI: 10.1145/3341716.

[11] Damas L, Milner R. Principal type-schemes for functional programs. In *Proc. the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1982, pp.207–212. DOI: 10.1145/582153.582176.

[12] Odersky M, Zenger C, Zenger M. Colored local type inference. *ACM SIGPLAN Notices*, 2001, 36(3): 41–53. DOI: 10.1145/373243.360207.

[13] Smith G S. Polymorphic type inference for languages with overloading and subtyping [Ph.D. Thesis]. Cornell University, Ithaca, 1991.

[14] Volpano D M, Smith G S. On the complexity of ml typability with overloading. In *Lecture Notes in Computer Science 523*, Hughes J (ed.), Springer, 1991, pp.15–28. DOI: 10.1007/3540543961_2.

[15] Charguéraud A, Bodin M, Riboulet L. Typechecking of overloading in programming languages and mechanized mathematics. In *Proc. the 36es Journées Francophones des Langages Applicatifs (JFLA 2025)*, Jan. 2025.

[16] Dolan S. Algebraic Subtyping: Distinguished Dissertation. BCS Learning & Development Ltd., 2017.

[17] Parreaux L. The simple essence of algebraic subtyping: Principal type inference with subtyping made easy (functional pearl). *Proceedings of the ACM on Programming Languages*, 2020, 4(ICFP): 1–28. DOI: 10.1145/3409006.

[18] Parreaux L, Chau C Y. MLstruct: Principal type inference in a Boolean algebra of structural types. *Proceedings of the ACM on Programming Languages*, 2022, 6(OOPSLA2): 449–478. DOI: 10.1145/3563304.

[19] Adve V, Lattner C, Brukman M, Shukla A, Gaeke B. LLVA: A low-level virtual instruction set architecture. In *Proc. the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003, pp.205–216. DOI: 10.1109/MICRO.2003.1253196.

[20] Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proc. the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 27–Mar. 3, 2021, pp.2–14. DOI: 10.1109/CGO51591.2021.9370308.

[21] Milton S, Schmidt H W. Dynamic dispatch in object-oriented languages. Technical Report TR-CS-94-02, The Australian National University, Canberra, Australia, 1994. https://scispace.com/pdf/dynamic-dispatch-in-object-oriented-languages-4s64vkql74.pdf, Jan. 2026.

[22] Atkinson K, Flatt M, Lindstrom G. ABI compatibility through a customizable language. In *Proc. the 9th International Conference on Generative Programming and Component Engineering*, Oct. 2010, pp.147–156. DOI: 10.1145/1868294.1868316.

[23] Kennedy A, Syme D. Design and implementation of generics for the .NET common language runtime. In *Proc. the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2001, pp.1–12. DOI: 10.1145/378795.378797.

[24] Click C, Tene G, Wolf M. The pauseless GC algorithm.

In *Proc. the 1st ACM/USENIX International Conference on Virtual Execution Environments*, Jun. 2005, pp.46–56. DOI: 10.1145/1064979.1064988.

[25] Tene G, Iyengar B, Wolf M. C4: The continuously concurrent compacting collector. In *Proc. the 2011 International Symposium on Memory Management*, Jun. 2011, pp.79–88. DOI: 10.1145/1993478.1993491.

[26] Hudson R L, Moss J E B. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 2003, 15(3/4/5): 223–261. DOI: 10.1002/cpe.712.

Xin-Yu Feng received his B.S. and M.E. degrees from the Department of Computer Science and Technology, Nanjing University, Nanjing, in 1999 and 2002, respectively, and his Ph.D. degree from the Department of Computer Science, Yale University, New Haven, in 2007. He is a professor with the School of Computer Science and the State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing. He is also a chief scientist on programming languages in Huawei. His research interests include design and implementation of programming languages, programming language semantics, and formal program verification.

Rong-Xiao Fu received his B.S. degree from Glasgow College, University of Electronic Science and Technology of China, Chengdu, in 2017, and his B.E. degree from the University of Glasgow, Glasgow, in 2017, and his M.S. and Ph.D. degrees from the School of Informatics, University of Edinburgh, Edinburgh, in 2018 and 2024, respectively. He is a senior engineer in the Programming Languages Lab, Central Software Institute, Huawei, Beijing. His research interests include programming language design and type systems.

Lei Shi received his B.E. degree from School of Software, Shanghai Jiao Tong University, Shanghai, in 2016, and his Ph.D. degree from the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, in 2022. He is a senior engineer in the Programming Languages Lab, Central Software Institute, Huawei, Beijing. His research interests include program analysis and type systems.

Le Tu received his B.S. and Ph.D. degrees from Sun Yat-Sen University, Guangzhou, in 2013 and 2018, respectively. He is the director of the Programming Languages Lab in Huawei, Beijing. His research interests include programming language design and implementation.

Yong-Yong Yang received his B.S. degree from Beihang University, Beijing, in 2008 and his Ph.D. degree from University of Chinese Academy of Sciences, Beijing, in 2013. He is a technology expert of system software in Language Virtual Machine Lab in Huawei, Beijing. His research interests include language runtime, compilation, optimization, automatic memory management and other related topics.