# ChocoPy v2.3: RISC-V Implementation Guide

University of California, Berkeley

December 5, 2020

## 1 Introduction

This document is intended to accompany the ChocoPy language reference manual, to serve as a guide for developers wishing to implement a ChocoPy compiler that targets the RISC-V instruction-set architecture.

Specifically, this guide assists with the task of generating RV32IM[1] assembly code for a semantically valid and well-typed ChocoPy program. This guide is not a complete specification; it is the developer's responsibility to implement the full operational semantics listed in the language manual. The design decisions described in this guide mirror the design of the official reference implementation, which is not optimized for maximum performance. Developers are free to tweak any or all of these design choices.

## 2 Naming conventions

The RISC-V assembly program generated for a ChocoPy program uses a single global namespace. To ensure unique naming, each such program entity is referred to by its fully-qualified name (FQN). FQNs are defined as follows. A class with name `C` has a FQN of `C`. A global variable with name `v` has FQN of `v`. A function `f` defined in global scope has a FQN of `f`. These names do not collide since they are distinct in the global namespace of the ChocoPy program as well. A method `m` defined in class `C` has FQN of `C.m`. A nested function `g` defined inside a function or method with FQN `F` has a FQN of `F.g`. A local variable `v` defined in a function or method with FQN `F` has a FQN of `F.v`. An attribute `a` defined in a class `C` has a FQN of `C.a`. As an example, consider the program:

```
class C(object):
  def f(self:"C") -> int:
    def g() -> int:
      x:int = 1
      return x
    return g()
C().f()
```

Here, the local variable `x` has a FQN of `C.f.g.x`.

[1] RV32IM is the 32-bit version of RISC-V with integer-only arithmetic, including multiplication (and division) instructions.

FQNs are used in creating global symbols in the generated RISC-V program. Global symbols are declared using the `.globl` directive and defined by specifying a label with the symbol name. In most cases, the symbol names prefix FQNs with a dollar sign to avoid conflicts in the global namespace with identifiers that the linker understands (such as `main` and `exit`). The following are the conventions for global symbols in the assembly program:

- Global variables with FQN of `N` have a global symbol of `$N`.
- Functions and methods with FQN of `N` have a global symbol of `$N`.
- Dispatch tables (ref. Section 4.1) for classes with FQN of `N` have a global symbol of `$N$dispatchTable`.
- Prototype objects (ref. Section 4.3) of classes with FQN of `N` have a global symbol of `$N$prototype`.

Attributes and local variables do not have global symbols dedicated to their definition.

# 3 Register conventions

The following RISC-V registers have special meaning:

- `a0` contains the returned value at the end of a function call. This register is also used to pass arguments to built-in routines (ref. Section 6). The value of this register is not preserved across function calls.
- `ra` contains the return address upon function invocation. The value of this register is not preserved across function calls.
- `fp` (alias for register `s0`) contains the current frame pointer. The value of this register is preserved across function calls.
- `sp` contains the current stack pointer. In particular, `sp` points to the top of the stack. The value of this register is preserved across function calls.
- `gp` points to the next free address in the heap. The value in this register is updated only when a new object is allocated, or during garbage collection.
- `s10` is reserved for pointing to the beginning of the heap. The value of this register is never modified by code generated for ChocoPy statements.
- `s11` is reserved for pointing to the end of the heap. The value of this register is never modified by code generated for ChocoPy statements.

The registers `a0`–`a7` and `t0`–`t6` are available for use as temporaries; their contents are not preserved across function calls. As per RISC-V convention, the values in registers `s1`–`s9` are preserved across function calls; ChocoPy functions may use these registers as long as their old contents are restored before returning.

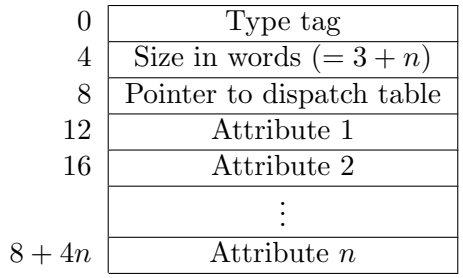



Figure 1: Object layout for an object with $n$ attributes. The numbers on the left denote the offset in bytes from the start of the object record.

# 4 Objects

Program values in ChocoPy are always objects, with the exception of the special value `None`. Objects are usually represented by their 32-bit address in memory. The `None` value is represented by the special address `0`.

## 4.1 Object layout

Figure 1 depicts the layout for a ChocoPy object in memory. An object is a record containing 3 or more slots, each the size of a 32-bit word. The first 3 slots have a common meaning to objects of all types, and together are referred to as the object's *header*.

The first slot in the header contains a *type tag*. The type tag is a 32-bit integer that uniquely represents the dynamic type of the object. The following type tags are fixed for every ChocoPy program[2]:

| **Type tag** | **Type** |
|---:|---|
| `0` | (reserved) |
| `1` | `int` |
| `2` | `bool` |
| `3` | `str` |
| `-1` | `[T]` |

The type tag for lists is the same for lists of any element type. ChocoPy lists do not need to store their statically declared element types at run-time. The dynamic type of objects contained in the list can be determined by accessing their corresponding type tags. Objects of user-defined classes use type tags that are not listed in the table above.

The second slot in the object's header is the size of the object in words. For objects of classes with $n$ attributes (including inherited attributes), the size is always $3 + n$. For lists containing $n$ elements, the size is $4 + n$. For strings of length $k$, the object size is $4 + \lceil (k+1)/4 \rceil$. The expression $\lceil (k + 1)/4 \rceil$ computes the number of words required to store $k$ bytes of the string's characters, including a null-byte terminator.

The third slot in the object's header is the address of the object's dispatch table in memory. The dispatch table for a class having $m$ methods contains $m$ words, each word representing the

[2]These type tags are recognized by the reference implementation of the predefined functions `print` and `len`.

address of the method's code. For a class named `C`, the global symbol for its dispatch table is `$C$dispatchTable`.

The object's header is followed by its attributes. Each attribute takes up one slot. For objects of user-defined classes, attributes store addresses of the objects they reference, or the address `0` to denote the `None` value. Objects of predefined classes and of list type have special attributes:

- `int` objects have a single attribute called `__int__`, which contains the 32-bit integer value that the object represents.
- `bool` objects have a single attribute called `__bool__`, which contains the 32-bit integer 1 or 0, depending on whether the object represents the value `True` or `False` respectively.
- `str` objects have two attributes. The first attribute is called `__len__` and it contains the 32-bit integer length of the string. The second attribute is called `__str__` and is of variable size. This attribute contains the sequence of bytes that make up the string's characters, followed by a terminating null byte[3]. This sequence is then appended with zero or more null bytes as padding until the total number of bytes is a multiple of 4.
- List objects have one attribute called `__len__`, which contains $n$, the 32-bit integer length of the list. This attribute is then followed by $n$ word-sized slots containing the addresses of list elements (or the address `0` for `None` values) in sequence.

### 4.2 Unwrapped Values

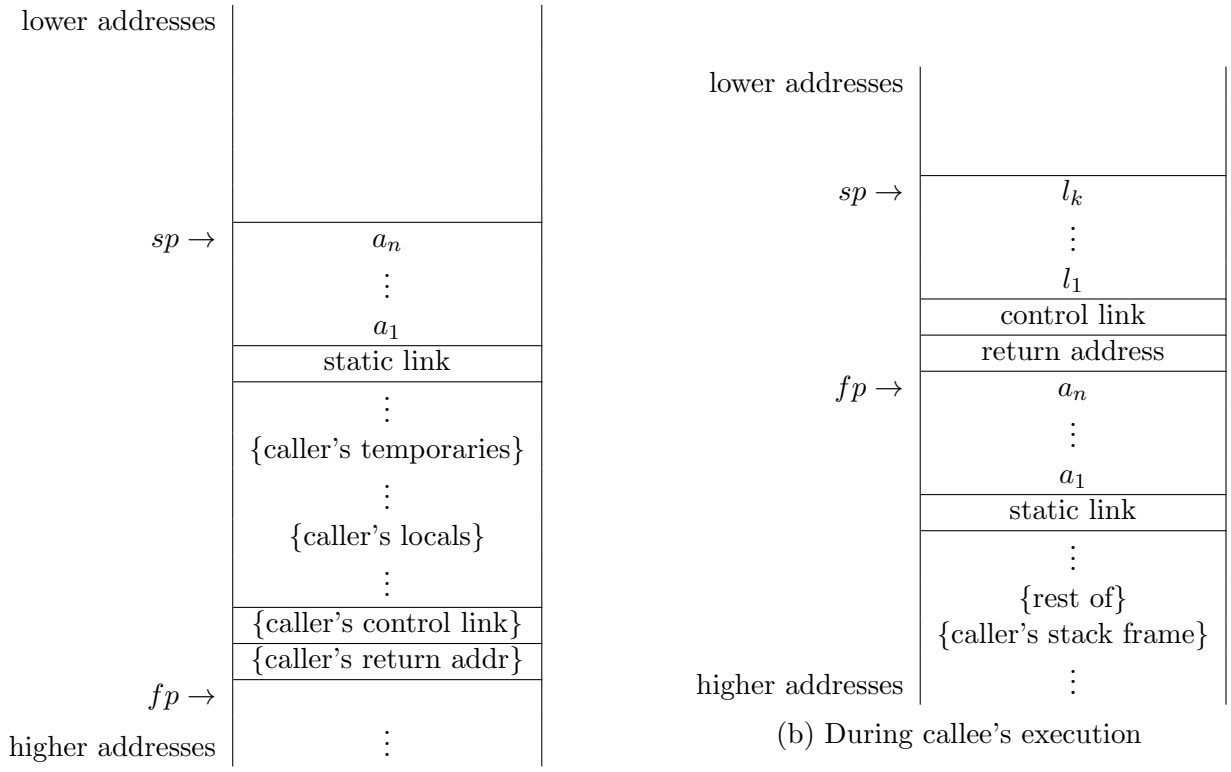
Parameters, local variables, global variables, and attributes whose static types are `int` or `bool` are represented by simple integer values. This is possible because of the rule in ChocoPy that `None` is not a value of either type, so that there can be no confusion between 0 or false on the one hand, and `None` on the other. We say that these two types are usually *unwrapped* or *unboxed*. Only when assigning them to variables of type `object` is it necessary to "wrap" or "box" them into the object representations described in Section 4.1 so that their actual types can be recovered by functions that expect to receive pointers to objects. The unwrapped values are the same as those that would be stored in the `__int__` or `__bool__` attributes of the object forms. This unwrapped representation considerably speeds up the execution of code that manipulates integer and boolean values.

### 4.3 Protoype objects

For each class with name `C`, the global symbol `$C$prototype` points to a *protoype* object of that class. A prototype is an object in memory whose attribute slots contain their initial values, as defined in the ChocoPy program. For predefined classes `int`, `bool`, and `str`, the initial values of their attributes correspond to the values `0`, `False`, and the empty string `""` respectively.

Object prototypes are useful for constructing new objects. Instead of executing code to initialize an object's header and attributes, the prototype can simply be copied to the next free address in the heap. Section 6 describes the built-in routines that perform this operation.

[3]The string representation contains redundant information due to the combination of the length attribute and the terminating null byte. However, this enables fast bounds checks as well as fast I/O operations.




(a) Before/after invocation

(b) During callee's execution

Figure 2: Stack layouts during the invocation of a nested function with $n$ arguments $a_1, \ldots, a_n$ and $k$ local variables having initial values $l_1, \ldots, l_k$. The diagram on the left shows the state of the stack just before the function is invoked (and just after it returns). The diagram on the right shows the state of the stack just before executing the called function's first statement. The stack layouts are similar during invocations of global functions and methods, except that the static link is not present.

### 4.4 Constants

The reference compiler emits constant values corresponding to integer, boolean, and string literals found in a ChocoPy program to the global data section. Since objects of type `int`, `bool`, and `str` are immutable, multiple occurrences of the same literal can refer to the same constant object in memory. The emitted constants are therefore globally unique. The global symbols corresponding to constants have names with prefix `const_` and a unique integer as suffix. For example, `const_0` and `const_1` usually refer to the objects corresponding to the values `False` and `True` respectively.

## 5 Functions and methods

### 5.1 Calling convention

The ChocoPy implementation on RISC-V specifies the following calling convention to enable interoperability between predefined functions, user-defined functions, and externally defined functions.

- At function invocation, the register `sp` points to the element last pushed on the stack by the

caller.

- At function invocation, the register `ra` contains the return address, i.e., the address from which program execution must continue after the called function returns.
- The caller expects the value of `sp` and `fp` to be preserved across a function call. The value of `ra` and temporary registers need not be preserved.
- The called function (a.k.a. the callee) returns the result in register `a0`.

### 5.2 Activation records

The reference compiler implements the calling convention in the following way. While executing a function's body, the register `fp` contains a pointer to a data structure called the *activation record*, which stores the contents of the function's parameters and local variables. On function invocation, the callee saves the contents of registers `fp` and `ra` in its own activation record before updating register `fp` with the address of this new record. When returning to its caller, the callee restores the old value of `fp` by retreiving it from its own activation record.

The saved value of the address of the caller's activation record is called the *control link*[4], since it points to the activation record of the function where program control will return once the current function completes its execution.

The parameters for a nested function include an additional parameter called the *static link*. The static link points to the activation record of the latest dynamic instance of the nearest statically enclosing function or method. For example, if function `g` is nested inside function `f`, then the static link in the activation record of `g` points to the activation record of the latest execution of `f`. In ChocoPy, the static link is used to implement static scoping within nested functions, giving access to the proper instances of nonlocal variables.

### 5.3 Stack frames

The activation record is implemented on the stack in a layout called the *stack frame*, as shown in Figure 2(b). The frame pointer (register `fp`) points to the bottom of the frame, specifically to the topmost element of the preceding stack frame. The stack frame contains the following information in top-to-bottom order:

1. actual parameters to the function it is calling (if applicable),
2. the static link for the function it it calling (if applicable),
3. temporary storage for intermediate results of expression evaluation or other operations,
4. the values of the local variables of the function,
5. the saved frame pointer of its caller (a.k.a the control link or dynamic link),
6. the saved return address. Its own actual parameters and static link are in the frame immediately below it.

[4]In some languages, this link is also known as the *dynamic link* since it can be used to implement dynamic scoping.

The local variables and parameters are stored in reverse order; that is, the variables or parameters declared earlier in the ChocoPy program are stored towards the bottom of the stack. As per RISC-V convention, the stack grows towards lower addresses; that is, the top-of-stack has the smallest address.

Figure 2(a) shows the layout of the stack at the time of function invocation. The caller pushes the arguments on the stack in left-to-right order. When calling nested functions, the static link is pushed on the stack before the first argument. The layout of the stack is exactly the same when the called function returns. On return, the caller pops the arguments off the stack.

The static link is not passed to global functions or to methods. For methods, the first argument is the address of the object whose method is being invoked.

# 6 Execution environment

The execution environment for a ChocoPy program consists of predefined functions and methods as well as built-in routines that are always available. The following global symbols are always generated for predefined functions and methods: `$print`, `$input`, `$len`, and `$object.__init__`. The assembly code for these functions is hand-written in the reference compiler. These functions are invoked using standard ChocoPy calling conventions (ref. Section 5).

The reference compiler also defines the following built-in routines, whose argument-passing convention is slightly different:

- `alloc`: Allocates a new object on the heap. The routine expects the address of the object's prototype (ref. Section 4.3) to be provided as argument in register `a0`. The routine returns with the address of the newly allocated object in register `a0`. This routine updates the `gp` register. This routine may invoke garbage collection during its execution.

- `alloc2`: Allocates a new object on the heap with custom size. Like `alloc`, this routine expects the address of the object's prototype in register `a0`. Additionally, this routine takes a 32-bit integer as a *size* argument in register `a1`. The newly allocated object will be trimmed or extended to fit in *size* words. The value of *size* should be at least 3, in order to accommodate the object's header. This routine is useful in implementing string and list operations such as concatenation. Like `alloc`, this routine updates the `gp` register and may invoke garbage collection during its execution.

- `abort`: Exits the program with an erroneous-exit code after printing an error message.

  The error message should be provided as argument in register `a1`, as an address of a null-terminated string of characters in memory. The exit code should be provided in register `a0` as a 32-bit integer. The following is an exhaustive list of error messages printed by ChocoPy programs compiled using the reference implementation, along with their corresponding exit code:

| **Exit code** | **Error message** |
|---:|---|
| 1 | Invalid argument |
| 2 | Division by zero |
| 3 | Index out of bounds |
| 4 | Operation on `None` |
| 5 | Out of memory |
| 6 | Unsupported operation |

The "Invalid argument" error is raised only by the predefined functions `print` and `len`, when their arguments are not printable or iterable respectively. The "Unsupported operation" error is reserved for future use.

The built-in routines do not have a dollar sign in front of their names; therefore, the names of these routines do not conflict with global ChocoPy functions. ChocoPy programs cannot invoke these routines directly; in fact, the arguments expected by these routines are not necessarily ChocoPy objects. Apart from the argument-passing convention, the routines should be treated similarly to function calls. Values held in temporary registers must be saved by the caller before invoking a built-in routine, since the routines themselves make use of several temporary registers. Just like standard calling convention, the value of registers `fp`, `sp`, and `s1`–`s11` are preserved across the routine's invocation.

# 7 Garbage collection

ChocoPy does not provide means for programmers to manually free memory. An object is live if it reachable from any value in program registers, on the stack, in the global data area, or from an attribute of a live object. Objects that are not live may be destroyed and their memory reclaimed by a process known as *garbage collection*.

Garbage collection has not yet been implemented in the current reference implementation. However, the object layout and heap-register conventions are sufficient for implementing a conservative GC.