Gouki Scheme: An embedded Scheme implementation for Async Rust

Matthew Plant
maplant@protonmail.com

Abstract

GoukiScheme Abstract here

ACM Reference Format:
Matthew Plant. 2025. Gouki Scheme: An embedded Scheme implementation for Async Rust. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Over the last decade Rust has become an increasingly popular choice for systems programming. Since the introduction of async runtimes to the Rust language, async Rust programs have also become increasingly popular for writing highly IO bound applications such as web servers. While Rust is extremely performant for such applications, async Rust applications tend to be difficult to develop and debug with. Part of the problem is long compilation times; Rust applications must be stopped, rebuilt, and restarted, extending development time and reducing the ability for developers to iterate.

Glue code written in a dynamic language has long been a solution for enabling rapid prototyping and interopability in a language with long build times. Under this model, the performance critical code that does not often need to be rebuilt is written in the slower compiled language, while a dynamic perhaps interpreted language is embedded into the built to allow for gluing components together without the need for slow rebuilds. Changes to the glue code can be seen just be re-loading the applications, or perhaps a hot reloading mechanism can bring them into the application simply by saving to a file. In either case, iteration speeds are improved dramatically.

Another advantage of embedding a dynamic language in an application is that the application can provide a Read-Eval-Print-Loop, or REPL. Interactive prompts can expand the debugging capabilities of a application by allowing for inspection of the application while it is live. The application can be debugged, inspected, and orchestrated as a plastic system rather than a rigid daemon that can only be started, stopped or interacted with in small fixed languages.

Scheme has shown success as a embedded dynamic language, but its use has been limited to synchronous (but perhaps multithreaded) applications. Scheme implementations exist for embedding within Rust, but they are limited to running sync Rust code and cannot be async themselves. Since async is a Rust language feature that touches all parts of the code base, a new implementation of Scheme is required that is built with async Rust in mind to take full advantage of the async runtime and more importantly integrate frictionlessly into the async application.

GoukiScheme is a Scheme implementation designed to integrate flawlessly with async Rust; GoukiScheme code can execute asynchronously embedded in an async Rust application and can in turn execute async Rust functions. Abitrary Rust objects can be stored in GoukiScheme variables and passed to GoukiScheme functions with little modification. This is done without sacrificing potential performance by architecting GoukiScheme as a tree-walking AST; indeed GoukiScheme is fully JIT compiled and takes advantage of a CPS based mid-level IR to compile to LLVM SSA. GoukiScheme does this while providing a completely Safe API.

This integration allows for Scheme programs to be written that take advantage of the Rust async ecosystem, such as this example of an echo server adapted from tokio:

```
(define (echo socket)
  (let ((buff (await (read-all socket))))
    (await (write-all socket buff))
    (echo socket)))

(define (listen listener)
  (let-values (((socket _) (await (accept listener))))
    (spawn (lambda () (echo socket)))
    (listen listener)))

(listen (await (bind-tcp "127.0.0.1:8080")))
```

In this example, the functions `read-all`, `write-all`, `accept`, and `bind-tcp` are all Rust functions accessible via Scheme that call the appropriate tokio functions. `await` is also a builtin that checks that its argument is a Future and awaits it.

Conference'17, July 2017, Washington, DC, USA
© 2025 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While there are some solutions for embedding Scheme in Rust, such as through Guile foreign function interface bindings, or one of the few native-Rust implementations that rely on an internal bytecode virtual machine, none provide a tight integration with the Rust async system while also providing a JIT compiler.

2 Memory Management

In order to properly implement the Scheme programming language, some form of garbage collection must be implemented by the system. One very popular algorithm for garbage collection is known as tracing, in which objects that are not determined to be reachable from so-called "root" objects are retained while the remaining objects are considered garbage and deallocated[4].

The root objects of a Scheme program are local and global variables active on the current Scheme call-stack, which is easy for our implementation to collect. However, when values escape into Rust code, the root objects for the entire program must be expanded to include that of the Rust code as well. Unfortunately, determining the roots of Rust program is not feasible[3].

A common Rust technique for automatic memory management is to use reference counting, which does not require knowing the roots of a program. However, this by itself this insufficient for our implementation due to the inability of reference counted systems to collect cycles[2]. We can extend this technique to support all valid Scheme programs by utilizing concurrent cycle collection[1].

The API for garbage collection in GoukiScheme is simple. GoukiScheme provides a Gc type that implements the same interface as Arc:

```
pub struct Gc<T>
where
  T: ?Sized,
{
  // private fields
}
```

Fundamentally the Gc type intends to mimic the behavior of a scheme variable. It can be read from and written to and arbitrarily passed around as a reference. Scheme variables can be passed to Rust code as a Gc.

Allocating a new garbage-collected T is done via the `Gc::new` function, and creating a new copy of the pointer to that object is done via the `Clone` method.

Additionally, the `Gc` type provides a `write` and `read` method in order to provide thread-safe mutations and reads to the allocated data. Since GoukiScheme can arbitrarily spawn threads that reference shared or global variables, it is important for all writes and reads to variables to be safe and atomic. This is acheived by embedded `RwLock` in the Gc type.

After the `init_gc` function is called once, GoukiScheme spawns a task dedicated to collecting garbage. This task is shut down automatically when the main function returns. This provides a smooth and intuitive interface to the memory manager that is also performant.

```
#[tokio::main]
fn main() {
  init_gc();
  let a = Gc::new("hello");
  let b = a.clone();
  {
    *a.write() = "world";
  }
  assert_eq!(b.read(), "world");
}
```

No signal needs to be sent to shutdown the collector thread. This interface is only possible with async Rust.

Whenever a `Gc` is cloned or dropped, the change in reference count is sent to the collector task over an unbounded channel called the mutation buffer. The collector task receives those mutations in a loop, in what essentially amounts to the following code:

```
fn init_gc() {
  // Spawn the collector task:
  let _ = spawn(async move {
    let mut mutations = Vec::new();
    loop {
      BUFFER.recv_many(
        &mut mutations
      ).await;
      process_mutation(mutation);
      mutations.clear();
    }
  });
}
```

The collector can be an asynchronous coroutine (in tokio what is called a task) because its loop is receiving from an asynchronous channel. Since it is a task, that means that it is subject to cancellation when it yields at `.await` points. One common cause of cancellation is when the tokio runtime is shutdown, i.e. when the main function returns.

References

[1] David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–235.
[2] Kevin G. Cassidy. 1985. *The Feasibiliy of Automatic Storage Reclamation with Concurrent Program Execution in a LISP Environment.* Master's thesis. Naval Postgraduate School.

[3] Felix S Klock II. 2016. *GC and Rust Part 2: The Roots of the Problem.* https://blog.pnkfx.org/blog/2016/01/01/gc-and-rust-part-2-roots-of-the-problem/

[4] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine - I, Vol. 3. ACM, 184–195.