Gouki Scheme: An Embedded Scheme Implementation for Async Rust

Anonymous Author(s)

Abstract

GoukiScheme is the first-of-its-kind JIT compiler for the R6RS dialect of Scheme that supports call-with-current-continuation while being embeddable within and seamlessly integrating with a compiled language with first class support for async/await (Rust).

CCS Concepts: • **Software and its engineering → Just-in-time compilers**.

Keywords: Scheme, Rust, JIT Compilation

ACM Reference Format:
Anonymous Author(s). 2025. Gouki Scheme: An Embedded Scheme Implementation for Async Rust. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 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 or embedded programming. While Rust is extremely performant for such applications[12], async Rust applications tend to be difficult to develop and debug with[1] . Part of the problem is long compilation times[1]; 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 interoperability in a language with long build times. Under this model, the performance critical code that does not need to be rebuilt as often is written in the slower compiled language, while a dynamic perhaps interpreted language is embedded into the build to allow for gluing components together without the need for slow rebuilds. Changes to the glue code can be seen just by re-loading the application, or perhaps a hot reloading mechanism can bring them into the application whenever changes are committed 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 an 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 a small number of fixed commands.

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 or they require the overhead of an interpreter or bytecode virtual machine. 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 application while also maximizing performance with JIT compilation.

GoukiScheme[18] 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. Arbitrary 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 interpreter or bytecode VM; 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[7]:

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

[1]Biffle [5] claims: the debugging story is not great. In particular, answering the question "why isn't my program currently doing anything" is very hard.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
(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 Rust that checks that its argument is a `Future`[9] and awaits it.

2 Alternatives

There are several solutions currently available for embedding Scheme in Rust, one of the most straightforward is to use the C bindings provided by Guile, of which there are several bindings available in the Rust package registry, although most of them are fairly rudimentary and incomplete. Additionally, none of these implementations provide any support for async Rust, since no such support is available from Guile.

Steel[17] is a pure Rust interpreter for Scheme that has support for calling async functions and utilizing async runtimes. Gouki and Steel differ in two primary ways. The first is in implementation: Steel is implemented via a bytecode interpreter, while Gouki is JIT compiled. The second is to what degree async is required by the implementation: Steel allows for an async runtime to be provided optionally, while Gouki requires an async runtime as it is entirely built around it.

At the time of publication, Gouki Scheme is the only JIT compiled implementation of a language with support for `call-with-current-continuation` that can be embedded in and integrate with a compiled language with support for async programming.

3 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 determined to be reachable from so-called "root" objects are retained while the remaining objects are considered garbage and deallocated[16].

The root objects of a Scheme program are the local and global variables active on the current Scheme call-stack, which are easy for our implementation to collect. However, when values escape into Rust code, the root objects must be expanded to include that of the Rust code as well. Unfortunately, determining the roots of a Rust program is not feasible[14].

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[6]. We can extend this technique to support all valid Scheme programs by utilizing concurrent cycle collection[4].

The API for garbage collection in GoukiScheme is simple. GoukiScheme provides a `Gc` type that implements the same interface as `Arc`[8], the thread-safe reference counted smart pointer provided by the Rust standard library, with a few additions:

```
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, 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 type 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 `write` and `read` methods in order to provide thread-safe mutations and accesses 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 achieved by and embedded reader-writer lock 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. Subsequent calls to `init_gc` are a no-op.

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

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, which is a `Future` (also known as a promise), it is subject to cancellation[22] 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. Therefore, no manual intervention is required by the user to shut down the garbage collection task, as it will properly shut down at the end of the program's lifetime.

3.1 Tracing and Finalizing

One limitation of the cycle collection scheme (and indeed even tracing garbage collection schemes) and therefore the `Gc` smart pointer is the requirement to enumerate every reference from within a single `Gc` in order to properly determine cycles. Additionally, properly finalizing a cycle is non-trivial and needs information about the contents of the type. This limits us to storing types in a Gc that implement the `Trace` trait:

```
unsafe trait Trace: 'static {
  unsafe fn visit_children(
    &self,
    visitor: unsafe fn(OpaqueGcPtr)
  );

  unsafe fn finalize(&mut self);
}
```

Implementing this function is tedious and requires the user to implement potentially memory unsafe code. However, it is not difficult to implement based on the syntactic form of the data type being implemented for. For example, here is the corresponding `Trace` implementation for an example struct:

```
struct Example {
  a: Gc<Example>,
  b: OtherStruct,
}

unsafe impl Trace for Example {
  unsafe fn visit_children(
    &self,
    visitor: unsafe fn(OpaqueGcPtr)
  ) {
    visitor(self.a.as_opaque());
    self.b.visit_children(visitor);
  }

  unsafe fn finalize(&mut self) {
    self.b.finalize();
  }
}
```

The rules for implementing a Trace for a function is simple. First, we must assume that each field implements `Trace` or is a `Gc`. If the fields do not implement `Trace` and are not a `Gc`, then a compilation error will surface later when we attempt to call the `visit_children` and `finalize` functions. After we assume this, we can implement `Trace` as follows:

for $field \in fields$ **do**
  if `typeof`($field$) = `Gc` **then**
    `Visitor(AsOpaque(`$field$`))` ;
  else `VisitChildren(`$field$`)` ;
end

Algorithm 1: visit children

for $field \in fields$ **do**
  if `typeof`($field$) ≠ `Gc` **then**
    `Finalize(`$field$`)`
  end
end

Algorithm 2: finalize

Since these algorithms can be clearly implemented from the syntactic form of the data type (or rather it nearly can; we can assume the type of a field is equal to a `Gc` if it is spelled `Gc` - while this is not true in some pathological cases, such as when a local `Gc` type is defined and shadows ours, it is a good enough heuristic for nearly all cases), they are a prime candidates to be implemented with a procedural macro, or more specifically a derive macro[11]:

```
#[derive(Trace)]
struct NamedCell {
  name: &'static str,
  cell: Gc<Value>,
}

let foo = Gc::new(NamedCell { .. });
```

While this covers a large number of cases, some types can never implement `Trace`. One such example is types defined in external crates that do not already implement `Trace`; such types cannot implement `Trace` due to the orphan rule[10]. Storing these types requires an extra level of indirection through Rust's built-in reference counted type, the `Arc`. Extra care must be taken when using this escape hatch as previously mentioned these types do not provide any way to collect cyclical references.

3.2 Enabling Back-Pressure from the Garbage Collector

Because we are running the garbage collector concurrently and are sending mutations to the collector over an unbounded channel, the allocating tasks receive no back pressure from the collector task. As a result should the pace of new allocations outpace the collectors ability to free garbage, the process would exhibit what appears to be a memory leak. To address this, we provide a a way to convert the unbounded channel into backpressure:

```
async fn yield_until_gc_cleared() {
  while PENDING_MUTATIONS > MAX_ALLOWED {
    tokio::task::yield_now().await
  }
}
```

This function is automatically called in the evaluation trampoline to prevent runaway memory allocations. This means that while the collection scheme is always concurrent, it is only *parallel* for some amount of pending mutations.

4 Values

The `Value` type represents a Scheme value. It has the dual responsibility of storing any possible Scheme value while also allowing for the storage of any Rust value. Additionally, if a Scheme value has a reasonable native equivalent in Rust (which almost all of them do), conversion should be convenient and efficient. The `Value` type should also be convenient to pass to and use in our JIT compiled functions. To do this, our `Value` type must fit into a single machine word:

```
#[repr(transparent)]
pub struct Value(u64);
```

We use a tagged pointer[19] scheme to achieve these constraints with minimal overhead. A rather large tag size of four bits allowing for 16 tags in order to represent the most common Scheme values without an extra level of indirection.

```
pub enum ValueType {
    Null = 0,
    Boolean = 1,
    Character = 2,
    Number = 3,
    String = 4,
    Symbol = 5,
    Vector = 6,
    ByteVector = 7,
    Syntax = 8,
    Closure = 9,
    Record = 10,
    RecordType = 11,
    Pair = 12,
    SchemeCompatible = 13,
    HashMap = 14,
}
```

As a small optimization, an extra "Undefined" value, the result of reading a variable that is unbound, is defined as a Pair with a null pointer, or a value of 12 in this case.

This large number of tags allows us to map all of the primitive values in the serde data model[13] into a single `Value` without an extra level of indirection. As a result, any Rust type that implements Serialize can automatically be converted into a `Value`. This represents a large number of types in the Rust ecosystem, as a lot of crates implement Serde for their data types compulsorily.

We also utilize Rust's dynamic dispatch to allow for the storing of any type that implements the `SchemeCompatible` and `Trace` traits. Using this trait we can create new Scheme values that present themselves as sealed records.

```
trait SchemeCompatible: Trace + Any {
  fn record_type(&self) -> Arc<RecordType>;

  fn eqv(&self, rhs: &Value) -> bool;
}
```

Objects that implement this trait can be stored as a `Gc<dyn SchemeCompatible>`, which can then be converted into a `Value`. Values, in turn, can be converted back to a `Gc<dyn SchemeCompatible>`, which can then be downcast to a concrete type.

5 Evaluation

Evaluation of Scheme code in Rust is achieved through two primary objects: the `Closure` and `Application` struct.

As the name implies, the `Closure` struct represents a Scheme closure, which includes a captured environment and some code:

```
struct Closure {
  env:     Box<[Gc<Value>]>,
  func:    FuncPtr,
  runtime: Gc<Runtime>,
```

```
  // some fields omitted
}
```

The env field points to an array of variables that can be accessed from the function. It includes all non-local variables that are captured as part of reifying a function definition into a closure.

The function pointer type `FuncPtr` is the heart of the bridge between asynchronous Rust and Scheme code. It is sum type of three different pointer types:

```
pub enum FuncPtr {
  Continuation(ContinuationPtr),
  User(UserPtr),
  Bridge(BridgePtr),
}
```

The `ContinuationPtr` and `UserPtr` types are JIT compiled Scheme functions and the `BridgePtr` type is a Rust function that has been cast to a function pointer. Therefore all Scheme code is treated synchronously but can switch over to async Rust whenever it needs to await the result of a future. This is achieved by a trampoline; each call to a function returns an application to another function:

```
pub struct Application {
  op:   Option<Gc<Closure>>,
  args: Vec<Value>,
  // some fields omitted
}
```

However, in the case of Rust bridge functions - which can be potentially async - the function returns the *future* of an application. That means the body of trampoline loop is async:

```
impl Application {
  pub async fn(mut self) ->
    Result<Vec<Value>, Exception>
  {
    while let Application {
       op: Some(op),
       args,
    } = self
    {
      self = op.apply(&args).await?;
    }
    Ok(self.args)
  }
}
```

This allows for the JIT compiled `ContinuationPtr` and `UserPtrs` and for the Rust bridge functions to seamlessly interact with each other. The synchronous JIT compiled function pass their continuation to the async Rust code to evaluate the async result after it is awaited in the CPS trampoline.

6 Registering Bridge Functions

Using the inventory[21] crate we can create a global registry of Rust functions which can be imported into Scheme code. Functions with the signature of `BridgePtr` can be registered manually, but a procedural macro is provided to convert functions of many different signatures:

```
#[bridge(
  name = "read-file-to-string",
  lib = "(tokio)"
)]
async fn read_file(file: &Value) ->
  Result<Vec<Value>, Condition>
{
  let file = file.to_string();
  let contents =
    read_to_string(&file)
    .await
    .map_err(|_| Condition::Error)?;
  Ok(vec![Value::from(contents)])
}
```

Every function that registers with this inventory will be available automatically defined in their respective library in any `Registry` object created, GoukiScheme's notion of a package registry.

7 JIT Compilation

After the macro expansion phase, Scheme code is converted into a simple, reduced CPS mid level IR using the algorithm described in Compiling with Continuations[3]. After optimization the CPS IR is converted to LLVM SSA[15] for JIT compilation. LLVM was chosen due to its relative familiarity and large set of supported platforms, but the relative simplicity of the CPS mid level IR does not rule out the adding of additional backends such as Cranelift[2] which may have different performance characteristics and trade-offs.

Interacting with the LLVM JIT compiler to produce an object that can easily be interacted with in Rust (i.e. a `Closure` object described earlier) requires some care. When a `Closure` goes out of scope, we need a way to free the memory allocated for its function. LLVM's JIT compiler requires that we only have one handle to this memory per thread, which we call the `Context`[20]. In order to provide as safe interface that satisfies this guarantee, we create a handle to a separate thread that fully owns the JIT executor, called the `Runtime`:

```
struct Runtime {
  comp_tasks_tx: Sender<CompilationTask>
}
```

Creating a new `Runtime` spawns a new thread which handles all of our compilation tasks. When the Runtime goes out of scope, its sender is dropped and the compilation task exits, freeing all of the JIT compiled functions.

Once we have a context for which we can JIT compile functions, we need to send those functions back to the caller, via function pointers. In Rust function pointers are of the static lifetime, which is to say they have the lifetime of the entire program. This is obviously incorrect in this instance as the lifetime of `Runtime` is not guaranteed to last the entire program - a user could create a `Runtime`, create a `Closure` from that runtime and summarily drop the given `Runtime` while continuing to use the `Closure`. Therefore, we need to add a pointer from the Closure back to the Runtime from which it was created:

```
struct Closure {
  // ...
  runtime: Gc<Runtime>,
}
```

This is ensures that if the `Runtime` is dropped before the `Closure`, there will still be a live reference preventing the function pointers from being deallocated.

8 Putting It All Together

Now that we have all of the requisite pieces, we can construct a Rust program that compiles and executes async Scheme code at runtime:

```
let rt = Runtime::new();
let registry = Registry::new(&runtime).await;
let env = Environment::from(
  registry.import("(base)")
);
let value = rt.eval(
  &env,
  // Sleep for 10 seconds and return a string
  "(begin (await (sleep 10)) \"hello world\")"
).await.unwrap();
println!("{value}");
```

9 Conclusion

GoukiScheme provides a purely safe, easy-to-use interface for interacting with async Scheme code from within async Rust and vice-versa. It does this without sacrificing performance by implementing a novel JIT compilation and evaluation scheme that takes advantage of an async trampoline and eschewing typical tracing garbage collectors for a concurrent cycle collection scheme.

References

[1] 2023. *Is Rust compile time really that slow?* https://users.rust-lang.org/t/is-rust-compile-time-really-that-slow/102863/6

[2] Bytecode Alliance. 2025. *Cranelift.* https://cranelift.dev/

[3] Andrew W. Appel. 2007. *Compiling with Continuations.* Cambridge University Press, USA.

[4] 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.

[5] Cliff L. Biffle. 2023. *Writing a basic 'async' debugger.* https://cliffle.com/blog/lildb/#why-can-t-i-have-a-stack-trace

[6] Kevin G. Cassidy. 1985. *The Feasibiliy of Automatic Storage Reclamation with Concurrent Program Execution in a LISP Environment.* Master's thesis. Naval Postgraduate School.

[7] Tokio Contributors. 2025. *Tokio documentation.* https://docs.rs/tokio/latest/tokio/

[8] The Rust Project Developers. 2025. *Arc.* https://doc.rust-lang.org/std/sync/struct.Arc.html

[9] The Rust Project Developers. 2025. *Future.* https://doc.rust-lang.org/std/future/trait.Future.html

[10] The Rust Project Developers. 2025. *Implementations - The Rust reference.* https://doc.rust-lang.org/reference/items/implementations.html#r-items.impl.trait.orphan-rule.intro

[11] The Rust Project Developers. 2025. *Procedural Macros - The Rust reference.* https://doc.rust-lang.org/reference/procedural-macros.html#derive-macros

[12] Dion. 2022-01-30. *Async Rust vs RTOS showdown!* https://tweedegolf.nl/en/blog/65/async-rust-vs-rtos-showdown

[13] David Tolnay et al. 2023. *Serde data model.* https://serde.rs/data-model.html

[14] 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/

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

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

[17] Matthew Paras. 2020. *An embeddable and extensible scheme dialect built in Rust.* https://github.com/mattwparas/steel

[18] Matthew Plant. 2023. *Embedded Scheme for the Async Rust Ecosystem.* https://github.com/maplant/scheme-rs

[19] Peter A. Steenkiste. 1991. *Tags and run-time type checking.* The MIT Press. 3–24 pages.

[20] LLVM Team. 2025. *llvm::LLVMContext Class Reference.* https://llvm.org/doxygen/classllvm_1_1LLVMContext.html#details From the documentation: LLVMContext itself provides no locking guarantees, so you should be careful to have one context per thread.

[21] David Tolnay. 2025. *Typed distributed plugin registration.* https://github.com/dtolnay/inventory

[22] Yoshua Wuyts. 2021. *Async Cancellation I.* https://blog.yoshuawuyts.com/async-cancellation-1