Pen 言語の進捗報告

@raviqqe

目次

- 進捗報告
  - 非同期 I/O の実装
- 今後の予定
  - Rust 側の FFI 強化
  - 並列・並行計算プリミティブの実装

進捗報告

非同期 I/O の設計

- Pen 言語側のコードは同期・非同期に関わらず同じ
  - Go と同じ設計
  - `async`・`await` キーワードのようなものはない
  - CPS によって実装
- システムパッケージ（メイン関数を実行するパッケージ）が独自に実装
  - I/O 等を非同期にするかは自由

非同期 I/O の実装

- `os-async`（仮）という標準パッケージに実装
  - `os` パッケージの API をすべて一度に実装するのは大変
  - 同期版も残したい（WASI 用）
- Rust の非同期実行の言語機能をそのまま利用
- タスクの実行には tokio のマルチスレッドランタイムを使用

メイン関数の実装

- `tokio::main` 属性でメイン関数をマーク
- `poll_fn` 関数 (まだ unstable)で Pen 側のメイン関数を包む

メイン関数の実装 (続)

```
#[tokio::main]
async fn main() {
    let mut trampoline: (
        ffi::cps::StepFunction<ffi::Number>,
        ffi::cps::ContinuationFunction<ffi::Number>,
    ) = (_pen_os_main, exit);
    let mut stack = ffi::cps::AsyncStack::new(INITIAL_STACK_CAPACITY);

    poll_fn::<(), _>(|context| {
        stack.set_context(context);

        let (step, continue_) = trampoline;
        unsafe { step(&mut stack, continue_) };

        trampoline = stack.resume();

        // We never get ready until the exit function is called.
        Poll::Pending
    })
    .await;

    unreachable!()
}
```

ライブラリ関数の実装

引数なしの場合

例 - 標準入力からの読み出し

```
#[no_mangle]
unsafe extern "C" fn _pen_os_read_stdin(
    stack: &mut ffi::cps::AsyncStack,
    continue_: ffi::cps::ContinuationFunction<ffi::Arc<FfiResult<ffi::ByteString>>>,
) -> ffi::cps::Result {
    let mut future = stack
        .restore()
        .unwrap_or_else(|| Box::pin(utilities::read(stdin())));

    match future.as_mut().poll(stack.context().unwrap()) {
        Poll::Ready(value) => continue_(stack, ffi::Arc::new(value.into())),
        Poll::Pending => {
            stack.suspend(_pen_os_read_stdin, continue_, future);
            ffi::cps::Result::new()
        }
    }
}
```

ライブラリ関数の実装 (続)

引数ありの場合

- 2つ関数を実装
  - Future の初期化用
  - Future の poll 用

例 - 標準出力への書き込み

初期化

```
#[no_mangle]
unsafe extern "C" fn _pen_os_write_stdout(
    stack: &mut ffi::cps::AsyncStack,
    continue_: ffi::cps::ContinuationFunction<ffi::Arc<FfiResult<ffi::Number>>>,
    bytes: ffi::ByteString,
) -> ffi::cps::Result {
    let mut future: WriteStdoutFuture = Box::pin(utilities::write(stdout(), bytes));

    match future.as_mut().poll(stack.context().unwrap()) {
        Poll::Ready(value) => continue_(stack, ffi::Arc::new(value.into())),
        Poll::Pending => {
            stack.suspend(_pen_os_write_stdout_poll, continue_, future);
            ffi::cps::Result::new()
        }
    }
}
```

例 - 標準出力への書き込み

ポーリング

```
unsafe extern "C" fn _pen_os_write_stdout_poll(
    stack: &mut ffi::cps::AsyncStack,
    continue_: ffi::cps::ContinuationFunction<ffi::Arc<FfiResult<ffi::Number>>>,
) -> ffi::cps::Result {
    let mut future: WriteStdoutFuture = stack.restore().unwrap();

    match future.as_mut().poll(stack.context().unwrap()) {
        Poll::Ready(value) => continue_(stack, ffi::Arc::new(value.into())),
        Poll::Pending => {
            stack.suspend(_pen_os_write_stdout_poll, continue_, future);
            ffi::cps::Result::new()
        }
    }
}
```

今後の予定

Rust 側の FFI 強化

- 非同期 API の実装も含め、Rust の FFI にボイラープレートが多い
- Proc マクロ等で自動生成?

並列・並行計算プリミティブの実装

- まだ構文やセマンティクス何も決まってない
  - 関数にするか、新しい構文にするか
  - タスク並列 vs データ並列

まとめ

- 非同期ランタイムの I/O が動いた
- 並列・並行計算プリミティブを早く実装したい