`jankscripten` Guide

May 5, 2021

Contents

1 Introduction

`jankscripten` is a compiler and runtime system that targets WebAssembly and supports multiple front-end languages:

- JankyScript is a language with JavaScript syntax. It supports higher-order functions and a standard library that is similar to JavaScript. The compiler translates JankyScript to NOTWASM by performing type inference and closure conversion.
- NOTWASM is an explicitly-typed lower-level language that does not support higher-order functions. However, it does support:
  - Garbage collection: at the moment, a simple mark-and-sweep collector with a stop-and-copy region for floating point numbers.
  - String values: Worth mentioning, since WebAssembly does not support strings natively.
  - Any-typed values (a.k.a. type dynamic): a failed downcast crashes the program with an unrecoverable error (i.e., a WebAssembly trap).
  - Monotonic dynamic objects with prototype inheritance: The fields of these objects are of type Any, and new fields may be added dynamically. However, these objects *do not support field deletion*. Field lookup in dynamic objects is optimized using inline caching.
  - Hash tables and arrays: these store Any-typed values.
  - Explicit closures: Although NOTWASM does not support higher-order functions, it has a representation for closures (i.e., code + environment pointer) which is designed to fit into Any-typed values.
  - ML-style mutable references

JankyScript is *not JavaScript*. It makes several simplifying assumptions, which it inherits from NOTWASM. However, as long as you're working with "sane" JavaScript, e.g., JavaScript generated by a compiler from a saner programming language, you can use JankyScript. With more effort, you can instead use NOTWASM as an intermediate language for a compiler that targets WebAssembly.

2 Prerequisites and Building

Prerequites `jankscripten` is written in Rust, and uses the Rust WebAssembly toolchain. It depends on Rust packages that link to *libssl* on Linux. Finally, the test suite relies on Node. Follow these steps to install the prerequisites:

1. Install the Rust toolchain.
2. Install Node. We require Node 11 or higher.
3. Install the Rust WebAssembly toolchain:

   ```
   rustup target add wasm32-unknown-unknown
   ```

4. Install the Wasm-Bindgen CLI, to allow Rust unit tests to run in WebAssembly:

   ```
   cargo install wasm-bindgen-cli
   ```

5. On Ubuntu Linux, install *libssl* and *pkg-config*:

   ```
   sudo apt-get install libssl-dev pkg-config
   ```

Building

1. Build the `jankscripten` compiler:

   ```
   cargo build
   ```

2. Build the `jankscripten` runtime[1]:

   ```
   (cd runtime && cargo build)
   ```

3. Build the integration testing tool:

   ```
   (cd integration_tests && npm install)
   ```

Testing

```
cargo test
(cd runtime && cargo test) # Runs tests using WebAssembly
(cd integration_tests && npx jest)
```

3 Running

To compile `filename.ext` to WebAssembly:

```
./bin/jankscripten compile filename.ext
```

NOTE: The supported extensions are .js and .notwasm.
To run a compiled WebAssembly program with the jankscripten runtime:

```
./bin/run-node filename.wasm
```

[1]This is a separate step because it targets WebAssembly and not native code

4 Debugging

To debug or profile a compiled WebAssembly program:

```
node --inspect-brk bin/run FILENAME.wasm
```

The Chrome debugger uses source maps correctly to show the original Rust code. You can use Visual Studio Code or Edge, but source maps do not appear to work correctly. See the Node Debugging Guide for more information.