jankscripten Guide

July 12, 2021

Contents

Chapter 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.

1.1 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, and it relies on the Z3 SMT Solver. Finally, the test suite relies on Node. Follow these steps to install the prerequisites:

1. Install the Rust toolchain. **We require Rust 1.51.0 (released March 2021).** More recent versions of Rust changed the ABI of the WebAssembly backend. We will fix this soon. After installing Rust, run the following commands:

   ```
   rustup toolchain add 1.51.0
   rustup default 1.51.0
   ```

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
   ```

6. Install the Z3 library. On Ubuntu Linux:

   ```
   sudo apt-get install libz3-dev
   ```

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)
```

1.2 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

1.3 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.

Chapter 2

NotWasm Guide

NotWasm is a somewhat low-level language that compiles to WebAssembly. It supports several data structures that WebAssembly does not natively support. It also supports garbage collection (presently, a simple mark-and-sweep collector). NotWasm programs are linked to a runtime system, which is written in Rust and compiled to WebAssembly.

NotWasm programs use the following concrete syntax for types:

Result Types			
R	:=	T	Result of type T
	|	`void`	No result
Types			
T	:=	`any`	Unknown type (occupies 64-bits)
	|	`i32`	A signed, 32-bit integer
	|	`f64`	A 64-bit float
	|	`bool`	A boolean
	|	`str`	A pointer to an immutable string
	|	`Array`	A pointer to an array of any-typed values
	|	`DynObject`	A pointer to an object with a dynamic set of any-typed fields
	|	`HT`	A pointer to a hash table with string keys and any-typed values
	|	`(` *T* `, . . . ,` *T*`) ->` *R*	A pointer to a function
	|	`clos (` *T* `, . . . ,` *T*`) ->` *R*	A pointer to a closure
	|	`Ref(`T`)`	A pointer to a heap-allocated value of type *T*
	|	`env`	A pointer to an environment

NotWasm programs use a psuedo-ANF: *atoms* do no not alter control-flow or allocate memory, *expressions* may allocate memory but do not alter control-flow, and *statements* function bodies that may alter the control-flow of the program.

Primitives Programs may reference primitive operations that are defined in the NotWasm runtime system. These operations are imported at the top of `stdlib.notwasm`, which is in the root of the repository.

Atoms Atoms use the following concrete syntax:

bop	:=	`+`	Integer addition
	|	`-`	Integer subtraction
	|	`*`	Integer multiplication
	|	`>`	Integer greater-than comparison
	|	`<`	Integer less-than comparison
	|	`>=`	Integer greater-than-or-equal-to comparison
	|	`<=`	Integer less-than-or-equal-to comparison
	|	`==`	Integer equal-to comparison
	|	`===`	Pointer equality comparison
	|	`+.`	Floating-point addition
	|	`-.`	Floating-point subtraction
	|	`*.`	Floating-point multiplication
	|	`/.`	Floating-point division
b	:=	**`true`** | **`false`**	Boolean literal
n	:=	...	Integer literals
f	:=	...	Floating point literals
s	:=	`"..."`	String literals
a	:=	b	
	|	n	
	|	f	
	|	s	
	|	**`null`**	The null value (has type **`any`**)
	|	x	Bound identifier
	|	`@` $prim_a$ `(` $a_1 \dots a_n$ `)`	Application of a primitive function that does not allocate memory
	|	`*`a:**`T`**	Pointer dereference
	|	a_1`.`s	Read a field of a **`DynObject`**
	|	a_1 *bop* a_2	Apply a binary operator that does not allocate memory

Expresssions Expressions have the following concrete syntax:

e	:=	a	Atom
	|	`!` $prim_e$ `(` $x_1 \dots x_n$ `)`	Application of a primitive function that may allocate memory
	|	x_f `(` $x_1 \dots x_n$ `)`	Function application
	|	x_f`!(` x_e`,`$x_1 \dots x_n$ `)`	Closure application
	|	x`.`str `=` e	Update a field of a DynObject

Statements Statements have the following concrete syntax:

blk	:=	`{` $s_1 \dots s_n$ `}`	
s	:=	**`var`** x`:`T`=` e`;`	Declare a variable of type T
	|	x `=` e`;`	Assign a value to a variable
	|	**`if`** `(`a `)` *blk* **`else`** *blk*	Conditional
	|	**`loop`** *blk*	Loop
	|	**`return`** a`;`	Return a value
	|	**`break`** l`;`	Break out of a block
	|	`*`x`=` e`;`	Update a local variable

Programs A program is a list of global variables, followed by a list of functions. Global variables have the following concrete syntax:

`var` x`:`T`=` a`;`

Functions have the following concrete syntax:

`function` x_f `(` x_1`:`$T_1 \dots x_n$`:`T_n`)` `:`R *blk*

There *must* be a function called `main` that received no arguments and does not return a result.