# Assignment 5: Dynamism

**Due: 11/13/25 at 11:59pm** (NO EXTENSIONS!)
**Submission link:** https://www.gradescope.com/courses/1107485/assignments/7080759

In this assignment, you will implement techniques for dynamic analysis of Rice programs. The main task is be to implement a symbolic executor for Rice, and a bonus task is to implement type specialization for structs.

## 5a: Symbolic Execution (100 points)

### Motivation

The goal of symbolic execution is to use SMT solvers to identify inputs which cause an assertion failure. For example, if you write this function in Rice:

```
#[symex]
fn example(x: [int]) {
  if x[0] > 0 {
    assert(x[1] - x[0] < 10)
  }
}
```

Then running symbolic execution should produce an error like this:

```
$ rice example.rice symex
  Error:   × detected assertion failure by symbolic execution
    ╭─[example.rice:4:5]
  3 │     if x[0] > 0 {
  4 │       assert(x[1] - x[0] < 10)
    ·      ──────────┬──────────
    ·                ╰── failing assertion
  5 │     }
    ╰────
  help: x = (store ((as const (Array Int Int)) 7730) 0 7720)
```

Whereas running on a function like this:

```
#[symex]
fn example(x: [int]) {
  if x[0] > 0 {
    assert(x[0] + 1 > 1)
  }
}
```

Then symbolic execution should indicate no issues:

```
$ rice example.rice symex
No assertion failures detected.
```

### Specification

Symbolic execution should be implemented as a new ‘symex‘ subcommand for the Rice compiler, which you can add to the `Command` enum in `main.rs`. When running in symbolic execution mode, the

compiler should lower the input program to bytecode, and then symbolically execute all functions marked as `#[symex]`. For each such marked function, if there exists an input that would cause an assertion failure when executed, then the Rice compiler should emit an error and exit.

Recall from lecture that a symbolic executor is an interpreter which accumulates symbolic rather than concrete values. While a concrete interpreter tracks a single program configuration, a symbolic executor tracks multiple program configurations as they branch off at conditional jumps. Your solution will have a similar look and feel to the interpreter in `rt/mod.rs`, except it will use symbolic values instead of Wasm values, and you will have to implement your own heap instead of relying on the Wasmtime garbage collector.

Formally, the symbolic executor should operate over the domain of symbolic expressions supported by the Z3 solver, which we model as follows:

$$
\begin{aligned}
&\mathsf{SymVar}\ \alpha \\
&\mathsf{SymExpr}\ s ::= c \mid \alpha \mid s_1 \oplus s_2 \mid (s_1, \ldots, s_n) \mid s.i \mid \mathsf{update\text{-}field}(s_1, i, s_2) \\
&\qquad\qquad \mid [s] \mid s_1[s_2] \mid \mathsf{update\text{-}index}(s_1, s_2, s_3) \\
&\mathsf{SymBool}\ \phi \in \mathsf{SymExpr}
\end{aligned}
$$

Symbolic variables are "fresh" variables which could be any value of a given type. Symbolic expressions, like normal expressions, can be constants, symbolic variables, or binary operations. The last line indicates that $\phi$ is an alias for $s$, but used specifically to represent symbolic expressions of boolean type.

Symbolic expressions can also be tuples and arrays, but in a different manner than Rice tuples and arrays. Tuples are constructed and read as in Rice, but updated in an effect-free way via the update-field operation. Arrays are constructed in an length-agnostic manner, so $[s]$ means "the constant array where every index maps to $s$". Indexing works as usual, and update-index is an effect-free update analogous to tuples. Note in Z3 parlance, tuples are datatypes and arrays are arrays (not sequences).

You will implement symbolic execution for the entirety of the Rice language with the exception of closures, structs, and interfaces. Formally, that includes the following bytecode:

$$
\begin{aligned}
\mathsf{Place}\ p &::= x \mid p.i \mid p_1[p_2] \\
\mathsf{Rvalue}\ rv &::= c \mid p \mid p_1 \oplus p_2 \mid f(p^*) \mid \mathsf{alloc}(p^*) \\
\mathsf{Instruction}\ \mathcal{I} &::= p = rv \mid \mathsf{goto}\ i \mid \mathsf{if}\ p\ \mathsf{goto}\ i_1\ \mathsf{else}\ i_2 \mid \mathsf{return}\ p \\
\mathsf{Function}\ g &::= \mathsf{fn}\ f(x^*)\ \{\ \mathcal{I}^*\ \} \\
\mathsf{Program}\ P &\in \mathsf{Var} \rightharpoonup \mathsf{Function}
\end{aligned}
$$

A function is symbolically executed within a stack frame that contains the function, its locals, and a program counter:

$$
\begin{aligned}
&\mathsf{Locals}\ \Sigma \in \mathsf{Var} \rightharpoonup \mathsf{SymExpr} \\
&\mathsf{Frame}\ F ::= \{\mathsf{func}\ g; \mathsf{locals}\ \Sigma; \mathsf{pc}\ n\}
\end{aligned}
$$

Note that the locals are symbolic expressions rather than concrete values.

An abstract configuration then consists of a program, a stack of frames, a global heap, and a path condition:

$$\begin{aligned} \mathsf{Heap}\ H &\in \mathbb{N} \rightharpoonup \mathsf{SymExpr} \\ \mathsf{Stack}\ S &::= F^* \\ \mathsf{AbsConfig}\ \mathbb{C} &::= \{\mathsf{prog}\ P; \mathsf{stack}\ S; \mathsf{heap}\ H; \mathsf{path}\ \phi\} \end{aligned}$$

The heap is represented as a mapping from pointers (natural numbers) to symbolic expressions. The path condition represents the set of accumulated facts deduced from conditional jumps.

The rules for implementing symbolic execution are described on the final page of this handout. It describes a few key judgments:

- $\mathbb{C} \overset{\mathcal{I}}{\hookrightarrow} \mathbb{C}'$: the abstract configuration $\mathbb{C}$ steps to a new configuration $\mathbb{C}'$ for the instruction $\mathcal{I}$. Similar to the WebAssembly evaluation rules, only elements of the configuration that are relevant to a given rule are listed on each side of the arrow. All other elements are assumed to be unchanged.
- $\Sigma; H \vdash rv \Downarrow s \Rightarrow H'$: the rvalue $rv$ evalutes to the symbolic expression $s$ in the context of locals $\Sigma$ and heap $H$, producing an updated heap $H'$. Note that when $rv$ is guaranteed not to be $\mathsf{alloc}$, we use the shorthand $\Sigma; H \vdash rv \Downarrow s$ because $H$ cannot be updated.
- $\Sigma; H \vdash p = s \Rightarrow \Sigma'; H'$: the symbolic expression $s$ is assigned to the place $p$ in the context of locals $\Sigma$ and heap $H$, producing updated locals $\Sigma'$ and an updated heap $H'$.

A few final notes on cases you do or don't need to handle:

- You do not need to handle the case of abstract functions, i.e., symbolically executing a function that takes as input a function of unknown value.
- You do need to handle the case of abstract allocations, i.e., symbolically executing a function that takes as input an array or tuple. You can make the simplifying assumption that all input allocations do not alias, i.e., each input allocation takes on a fresh abstract value.
- You only need to handle assertion failures, and not other kinds of program crashes such as array out-of-bounds accesses.
- You need to handle the possibility of infinite loops. Each configuration should track the number of steps it has executed so far. Your `symex` subcommand should take a `--max-steps <N>` flag such that if a configuration exceeds $N$ steps, then it is discarded by the runtime.
- You do not need to do anything clever in terms of prioritizing which configuration to run. The reference solution simply uses a queue.

## Implementation Tips

You will first need to install Z3. I recommend using version 4.14.1 or newer. I encountered issues using older versions such as the one distributed in the `libz3-dev` Debian package. I recommend the following:

- **Mac**: `brew install z3`

- **Linux or Windows**: Download the Github release and install it at an appropriate spot on your system, like `/usr/local`. Note that the release confusingly puts shared libraries in `bin` so you may need to move `libz3.so` into `lib`.
- **NixOS**: good luck and godspeed!

Next, you will need to add the Z3 bindings to your source. That's as easy as `cargo add z3`, then check out the Z3 binding documentation. Depending on where you installed Z3, you might need to communicate to Rust where to find either the Z3 header files or Z3 dynamic library. See the Cargo documentation for how to put this information into your build script. Note that you should **not** use the `.cargo/config.toml` solution described in the Z3 crate documentation because it may not work with the autograder.

Finally, a few implementation tips:

- The type signature for stepping a configuration in the reference codebase is:

  ```
  fn step(mut self) -> Result<SmallVec<[Self; 2]>>;
  ```

  That is, stepping consumes ownership of the configuration, and returns a result which errors upon an assertion failure. In the success case, it returns a small vector which is either just a modified version of `self` or `self` and a clone in the case of a fork.
- The reference compiler uses the Z3 `Solver` type to implement the path condition $\phi$, and the Z3 `Dynamic` type to implement all other symbolic expressions.
- Z3 symbolic constants are convertible to Rust constants. For example, if `x` is a `Dynamic` and an integer constant, then you can do `x.as_int().unwrap().as_u64().unwrap()` to get a `u64` out.

## Grading

Your implementation will be evaluated on a set of test cases which should either pass (there are no failing inputs) or fail (there are failing inputs). The set of test cases will be provided on the course website.

## 5b: Monomorphization (10 points, extra credit)

As an optional portion of this assignment, you will implement a system for adaptively optimizing and JIT compiling a Rice program to eliminate overhead from dynamic dispatch.

### Motivation

Consider a Rice library for vectors with a `Vec` interface, implementations for types like `Vec2`, and generic functions like a vector magnitude:

```
interface Vec {
  fn get(Self, int) -> float;
  fn dims(Self) -> int;
}

struct Vec2(float, float);

impl Vec for Vec2 {
  // ...
}

fn mag(v: @Vec) -> float {
  let i = 0 in
  let n = 0. in
  while i < v.dims() {
    n := n + v.get(i) * v.get(i);
    i := i + 1
  };
  sqrt(n)
}
```

A library client might use these functions in a hot loop to compute the magnitude of many vectors, like so:

```
fn main() {
  let N = 1000000 in
  let pts = [|new Vec2(0., 0.) as @Vec; N|] in
  let i = 0 in
  while i < N {
    pts[i] := new Vec2(i as float, i as float) as @Vec;
    i := i + 1
  };

  i := 0;
  while i < N {
    mag(pts[i]);
    i := i + 1
  }
}
```

Running this program under an optimized build of the reference Rice compiler at `-O1` takes about 13 seconds on my M1 Macbook Pro. If I change this program to eliminate the dynamic dispatch and call a specialized version of `mag` like this:

```
fn mag_vec2(v: Vec2) -> float {
  sqrt(v.0 * v.0 + v.1 * v.1)
}
```

Then the program takes about 4.5 seconds, nearly 3 times faster. Your challenge is to implement an adaptive optimization to generate and patch in functions like `mag_vec2` at runtime.

## Specification

This optimization has three main components:

1. **Type profiling:** at each call site, for each argument which is an interface object, track whether it is consistently called with an object of the same struct type. Note that this requires extending the runtime representation of interface objects to contain some kind of type ID to specify which struct generated the object.
2. **Specialization:** after identifying that a function is consistently called with a particular type, then generate the bytecode for that function with specialized arguments, and run optimizations on the function. Your optimizations must include inlining known functions and optionally unrolling loops with constant bounds.
3. **Patching:** with a specialized version of the function call, the runtime should save this function and remember to invoke it on subsequent calls only if future arguments have the expected type.

## Grading

You should take an example like the `Vec` interface above and demonstrate that your runtime achieves at least a 2x speedup using these optimizations. You can compare the optimizations by, for instance, having one call to `mag` which uses a heterogeneous array of interleaved `Vec2` and `Vec3` types versus a homogeneous array of `Vec3` types.

If you implement this optimization, then leave instructions in your README for how to run your compiler and observe the speedup. Send Will a note indicating that you attempted the extra credit portion of this assignment, and he will inspect your implementation and evaluation.

$\boxed{\mathbb{C} \overset{\mathcal{I}}{\hookrightarrow} \mathbb{C}'}$

$$\frac{\Sigma; H \vdash rv \Downarrow s \Rightarrow H' \qquad \Sigma; H' \vdash p = s \Rightarrow \Sigma'; H''}{(S, \{\Sigma; n\}); H \xhookrightarrow{p=rv} (S, \{\Sigma'; n+1\}); H''} \text{ E-Stmt} \qquad \frac{}{(S, \{n_0\}) \xhookrightarrow{\text{goto } n'} (S, \{n'\})} \text{ E-Goto}$$

$$\frac{\Sigma; H \vdash p \Downarrow s \qquad \phi' = \phi \wedge s \qquad \phi' \text{ SAT}}{(S, \{\Sigma\}); H; n_0 \xhookrightarrow{\text{if } p \text{ goto } n_1 \text{ else } n_2} (S, \{\Sigma\}); H; \phi'; n_1} \text{ E-If-True}$$

$$\frac{\Sigma; H \vdash p \Downarrow s \qquad \phi' = \phi \wedge \neg s \qquad \phi' \text{ SAT}}{(S, \{\Sigma\}); H; \phi; n_0 \xhookrightarrow{\text{if } p \text{ goto } n_1 \text{ else } n_2} (S, \{\Sigma\}); H; \phi'; n_2} \text{ E-If-False}$$

$$\frac{\Sigma_1; H \vdash p_i \Downarrow s_i \qquad P(f) = \mathsf{fn}\ (x^*)\{P\} \qquad \Sigma_2 = \{(x \mapsto s)^*\}}{P; (S, \{\Sigma_1\}); H \xhookrightarrow{p=f(p^*)} P; (S, \{\Sigma_1\}, \{\Sigma_2; P; \mathsf{pc} = 0; \phi = \mathsf{true}\}); H} \text{ E-Call}$$

$$\frac{\Sigma_2; H \vdash p_{\mathsf{ret}} \Downarrow s \qquad g[n] = \text{“}p_{\mathsf{dst}} = f(p^*)\text{”} \qquad \Sigma_1; H \vdash p_{\mathsf{dst}} = s \Rightarrow \Sigma_1'; H'}{(S, \{g; \Sigma_1; n\}, \{\Sigma_2\}); H \xhookrightarrow{\text{return } p_{\mathsf{ret}}} (S, \{g; \Sigma_1'; n+1\}); H'} \text{ E-Return}$$

$\boxed{\Sigma; H \vdash rv \Downarrow s \Rightarrow H'}$

$$\frac{}{\Sigma; H \vdash c \Downarrow c} \text{ E-Const} \qquad \frac{}{\Sigma; H \vdash x \Downarrow \Sigma(x)} \text{ E-Var} \qquad \frac{\Sigma; H \vdash p \Downarrow n \qquad H(n) = s}{\Sigma; H \vdash p.i \Downarrow s.i} \text{ E-Field}$$

$$\frac{\Sigma; H \vdash p_1 \Downarrow n \qquad H(n) = s_1 \qquad \Sigma; H \vdash p_2 \Downarrow s_2}{\Sigma; H \vdash p_1[p_2] \Downarrow s_1[s_2]} \text{ E-Index}$$

$$\frac{\Sigma; H \vdash p_1 \Downarrow s_1 \qquad \Sigma; H \vdash p_2 \Downarrow s_2}{\Sigma; H \vdash p_1 \oplus p_2 \Downarrow s_1 \oplus s_2} \text{ E-Binop}$$

$$\frac{\Sigma; H \vdash p \Downarrow s \qquad n = |H|}{\Sigma; H \vdash \mathsf{alloc}_{\mathsf{array}}(p; \_) \Downarrow n \Rightarrow H[n \mapsto [s]]} \text{ E-Alloc-Array}$$

$$\frac{\Sigma; H \vdash p_i \Downarrow s_i \qquad n = |H|}{\Sigma; H \vdash \mathsf{alloc}_{\mathsf{tuple}}(p_1; \ldots; p_n) \Downarrow n \Rightarrow H[n \mapsto (s_1; \ldots; s_n)]} \text{ E-Alloc-Tuple}$$

$\boxed{\Sigma; H \vdash p = s \Rightarrow \Sigma'; H'}$

$$\frac{}{\Sigma; H \vdash x = s \Rightarrow \Sigma[x \mapsto s]; H} \text{ E-Assign-Var}$$

$$\frac{\Sigma; H \vdash p \Downarrow n \qquad H(n) = s_{\mathsf{tuple}}}{\Sigma; H \vdash p.i = s_{\mathsf{value}} \Rightarrow \Sigma; H[n \mapsto \mathsf{update\text{-}field}(s_{\mathsf{tuple}}, i, s_{\mathsf{value}})]} \text{ E-Assign-Field}$$

$$\frac{\Sigma; H \vdash p_1 \Downarrow n \qquad H(n) = s_{\mathsf{array}} \qquad \Sigma; H \vdash p_2 \Downarrow s_{\mathsf{index}}}{\Sigma; H \vdash p_1[p_2] = s_{\mathsf{value}} \Rightarrow \Sigma; H[n \mapsto \mathsf{update\text{-}index}(s_{\mathsf{array}}, s_{\mathsf{index}}, s_{\mathsf{value}})]} \text{ E-Assign-Index}$$