Yizhao's Egg-Eater Extension

Grammer

Syntax

The following new syntax are added to the egg-eater syntax:

```
<expr> :=
  ...
  | (tuple-get <expr> <expr>)
  | (tuple-set! <expr> <expr> <expr>)
  // removed (index <expr> <expr>)

<op1> := ... | istuple
<op2> := ... | ==
```

Semantics

- `tuple-get` is the renamed `index`.
- `(tuple-set! t i x)` changes the `i`-th element of tuple `t` to `x`. Similar to `tuple-get`, a runtime error will be issued if:
  - `t` is not a tuple.
  - `i` is not a valid index.
- `==` is the new expression for reference equality (use to be `=`). Now, `=` represents structural equality. The reason I choose this is that `==` looks **more strict** than `=`, and as we know, structural equality must hold if two values are the same reference.
- `istuple` is added analogously to `isnum` and `isbool`.
- For `print`, a tuple value with `1 2 3` will be printed as `(1 2 3)`. Empty tuple is `()`. If we are printing tuple `t`, and we encounter `t` again as some (maybe nested) element, we will print the inner `t` as `(...)`.

Implementation

Handling structural equality

I used a similar approach to the one showed in the lecture. More specifically, I added a function in the Rust runtime (`start.rs`) to compare two snek values. Here's the relevant code. The function recursively compares the elements in `v1` and `v2`, if they are both nonempty tuples.

```
fn snek_structural_eq(default: bool, v1: i64, v2: i64, pending: &mut
Vec<(i64, i64)>) -> bool {
    if v1 == v2 { true }
    else if v1 & 3 == 1 && v2 & 3 == 1 {
        if v1 == 1 && v2 == 1 {
            true
```

```
        } else if v1 == 1 || v2 == 1 {
            false
        } else if pending.contains(&(v1, v2)) { default }
        else {
            let a1 = (v1 - 1) as *const i64;
            let a2 = (v2 - 1) as *const i64;
            let l1 = unsafe { *a1 } >> 1;
            let l2 = unsafe { *a2 } >> 1;
            if l1 != l2 { false }
            else {
                pending.push((v1, v2));
                for i in 1..l1 as usize + 1 {
                    if unsafe { !snek_structural_eq(default, *a1.add(i),
*a2.add(i), pending) } { return false; }
                }
                pending.pop();
                true
            }
        }
    } else { false }
}
```

In egg-eater, when I need to compile structural equality expressions, I call this function. Since the calling convention I used is different from Rust's, I also added a function to the compiler to call runtime functions with two arguments. Here's the code. Basically, `RDI` and `RSI` are saved on stack, and the arguments are passed through `RDI` and `RSI`. I also need to handle the 16-byte alignment.

```
fn compile_external_call_2(a1: Val, a2: Val, n: &str, c: &Context, _mc:
&mut MutContext, instrs: &mut Vec<Instr>) {
    if !c.aligned { instrs.push(Instr::Sub(Val::Reg(Reg::RSP),
Val::Imm32(8))); }
    instrs.push(Instr::Push(Val::Reg(Reg::RDI)));
    instrs.push(Instr::Push(Val::Reg(Reg::RSI)));
    instrs.push(Instr::Mov(Val::Reg(Reg::RDI), a1));
    instrs.push(Instr::Mov(Val::Reg(Reg::RSI), a2));
    instrs.push(Instr::Call(n.to_string()));
    instrs.push(Instr::Pop(Val::Reg(Reg::RSI)));
    instrs.push(Instr::Pop(Val::Reg(Reg::RDI)));
    if !c.aligned { instrs.push(Instr::Add(Val::Reg(Reg::RSP),
Val::Imm32(8))); }
}
```

Handling cycles

Similar to the approach in the lecture, `snek_structural_eq` uses a mutable argument `pending` to detect cycles. If we are going to compare two values that are already in the `pending` list, then we find a cycle. This is when the `default` argument comes into play. If `default` is `false`, then the function will reject any comparison with cyclic logic (which should make some sense I guess? But we don't want this here). If `default` is set to `true`, then the behavior should be what we discussed in the lecture. So I wrapped `snek_structural_eq` with `default` set to `true`, making the following function.

```
#[export_name = "\x01snek_structural_eq_true"]
fn snek_structural_eq_true(v1: i64, v2: i64) -> i64 {
    let mut pending = Vec::<(i64, i64)>::new();
    if snek_structural_eq(true, v1, v2, &mut pending) { 7 } else { 3 }
}
```

This is what is actually called by egg-eater.

For `print`, similarly, we used a `seen` to detect cycles. If we are printing a value that is already in `seen`, `(...)` is printed. Here's the code.

```
fn snek_str(val: i64, seen: &mut Vec<i64>) -> String {
    if val == 7 { "true".to_string()}
    else if val == 3 { "false".to_string() }
    else if val % 2 == 0 { format!("{}", val >> 1) }
    else if val == 1 { "()".to_string() }
    else if val & 1 == 1 {
        if seen.contains(&val) { "(...)".to_string() }
        else {
            let addr = (val - 1) as *const i64;
            let len = unsafe { *addr } >> 1;
            seen.push(val);
            let s = (1..len as isize + 1).map(|i| snek_str(unsafe
{*addr.offset(i)}, seen)).collect::<Vec<_>>().join(" ");
            seen.pop();
            format!("({})", s)
        }
    } else { format!("Unknown value: {}", val) }
}
```

Tests

Here are the tests added for the extension. For the original tests, see the later section.

`equal.snek`

Code

```
(let (
    (a1 (tuple 1 20))
    (a01 a1)
    (a2 (tuple 1 20))
    (b (tuple 3 40))
    (f11 (tuple a1 a1))
```

```
    (f011 f11)
    (f111 (tuple a1 a1))
    (f12 (tuple a1 a2))
    (f22 (tuple a2 a2))
    (g (tuple a1 b))
)
(block
    (print (tuple (= a1 a01) (== a1 a01)))
    (print (tuple (= a1 a2) (== a1 a2)))
    (print (tuple (= a1 b) (== a1 b)))
    (print (tuple (= f11 f011) (== f11 f011)))
    (print (tuple (= f11 f111) (== f11 f111)))
    (print (tuple (= f11 f12) (== f11 f12)))
    (print (tuple (= f11 f22) (== f11 f22)))
    (print (tuple (= f11 g) (== f11 g)))
    0
))
```

Compilation output

```
$ make tests/equal.run
cargo run -- tests/equal.snek tests/equal.s
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
     Running `target/debug/egg-eater tests/equal.snek tests/equal.s`
nasm -f elf64 tests/equal.s -o tests/equal.o
ar rcs tests/libequal.a tests/equal.o
rustc -L tests/ -lour_code:equal runtime/start.rs -o tests/equal.run
rm tests/equal.s
```

The compilation output for later tests are quite similar, so I will just omit them.

Output

```
(true true)
(true false)
(false false)
(true true)
(true false)
(true false)
(true false)
(false false)
0
```

Explanation

By have the same value, I mean two objects are structurally equal.

- `a1` and `a01` are the same reference.
- `a1` and `a2` are different objects, but are structurally equal.
- `a1` and `b` are different object with different values.
- `f11` and `f011` are the same reference.
- `f11`, `f12` and `f22` have the same value because `a1` and `a2` have the same value.
- `f11` and `g` have different values.

`cycle-print1.snek`

Code

```
(let (
    (a (tuple 1 20))
    (b (tuple 3 40))
    (c (tuple -5))
)
(block
    (print (tuple a b c))
    (tuple-set! a 0 b)
    (print (tuple a b c))
    (tuple-set! b 1 a)
    (print (tuple a b c))
    (tuple-set! a 0 a)
    (print (tuple a b c))
    (tuple-set! c 0 c)
    (print (tuple a b c))
    0
))
```

Output

```
((1 20) (3 40) (-5))
(((3 40) 20) (3 40) (-5))
(((3 (...)) 20) (3 ((...) 20)) (-5))
(((...) 20) (3 ((...) 20)) (-5))
(((...) 20) (3 ((...) 20)) ((...)))
0
```

Explanation

This example shows how nested tuples and cyclic tuples are printed. After `(tuple-set! b 1 a)`, both value are cyclic. After `(print (tuple a b c))`, `c` becomes a simple loop.

`cycle-print2.snek`

Code

```
(let (
    (a (tuple 1 20))
    (b1 (tuple 1 20))
    (b2 (tuple 1 20))
)
(block
    (print (tuple a b1 b2))
    (tuple-set! a 1 a)
    (tuple-set! b1 1 b2)
    (tuple-set! b2 1 b1)
    (print (tuple a b1 b2))
    (tuple-set! b1 0 9000)
    (print (tuple a b1 b2))
    0
))
```

Output

```
((1 20) (1 20) (1 20))
((1 (...)) (1 (1 (...))) (1 (1 (...))))
((1 (...)) (9000 (1 (...))) (1 (9000 (...))))
0
```

Explanation

Notably, at `((1 (...)) (1 (1 (...))) (1 (1 (...))))`, `a` and `b1` (also `b2`) are structurally equal, although they print to different strings. We will examine this later. After modifying one of the fields to `9000`, we can see that `a`, `b1` and `b2` are (subtly) different.

`cycle-print3.snek`

Code

```
(let (
    (a (tuple 1 20))
    (b1 (tuple 1 20))
    (b2 (tuple b1 300))
    (b3 (tuple b1 b2))
)
(block
    (print a)
    (print b1)
    (print b2)
    (print b3)
    (tuple-set! b1 0 b2)
    (tuple-set! b1 1 b3)
    (tuple-set! b2 1 b3)
```

```
    (tuple-set! a 0 a)
    (tuple-set! a 1 a)
    (print a)
    (print b1)
    (print b2)
    (print b3)
    0
))
```

Output

```
(1 20)
(1 20)
((1 20) 300)
((1 20) ((1 20) 300))
((...) (...))
(((...) ((...) (...))) ((...) ((...) (...))))
(((...) ((...) (...))) (((...) (...)) (...)))
((((...) (...)) (...)) (((...) (...)) (...)))
0
```

Explanation

This example is related to `cycle-equal3.snek`. Again, the objects actually have subtly different structures, but they are regarded as equal.

`cycle-equal1.snek`

Code

```
(let (
    (a (tuple 1 20))
    (b (tuple 3 40))
)
(block
    (print (= a b))
    (tuple-set! b 0 1)
    (print (= a b))
    (tuple-set! a 1 a)
    (print (= a b))
    (tuple-set! b 1 b)
    (print (= a b))
    (tuple-set! a 1 b)
    (print (= a b))
    (tuple-set! b 1 a)
    (print (= a b))
    0
))
```

```
```

Output

```
false
false
false
true
true
true
0
```

Explanation

The value of `a` and `b` changes like the follows.

```
0   a:  (1 20)  b:  (3 40)
1   a:  (1 20)  b:  (1 40)
2   a:  (1 a)   b:  (1 40)
3   a:  (1 a)   b:  (1 b)
4   a:  (1 b)   b:  (1 b)
5   a:  (1 b)   b:  (1 a)
```

So `a` and `b` are equal since line `3`.

`cycle-equal2.snek`

Code

```
(let (
    (a (tuple 1 20))
    (b (tuple 1 20))
    (c (tuple 3 40))
)
(block
    (print (tuple (= a b) (= a c) (= b c)))
    (tuple-set! a 1 a)
    (tuple-set! b 1 c)
    (tuple-set! c 1 b)
    (print (tuple (= a b) (= a c) (= b c)))
    (tuple-set! c 0 1)
    (print (tuple (= a b) (= a c) (= b c)))
    0
))
```

Output

```
(true false false)
(false false false)
(true true true)
0
```

Explanation

At the end, `a` is a loop with one node, and `b` and `c` forms a cycle with two two nodes. If we are only accessing the elements in the objects (but not the value of pointers), we are not going to tell the difference between `a` and `b`.

`cycle-equal3.snek`

Code

```
(let (
    (a (tuple 1 20))
    (b1 (tuple 1 20))
    (b2 (tuple b1 300))
    (b3 (tuple b1 b2))
)
(block
    (tuple-set! b1 0 b2)
    (tuple-set! b1 1 b3)
    (tuple-set! b2 1 b3)
    (tuple-set! a 0 a)
    (tuple-set! a 1 a)
    (print (tuple (= a b1) (= a b2) (= a b3)))
    (print (tuple (= b1 b2) (= b1 b3) (= b2 b3)))
    0
))
```

Output

```
(true true true)
(true true true)
0
```

Explanation

This is another example like `cycle-equal2.snek`, but more exaggerated. It is consistent to say all the objects are structurally equal (that is, each object is like `(x y)`, where `x` and `y` are from the set of equal values), which is, in my opinion, not quite intuitive.

Other features and known problems

I added `istuple` for checking whether a value is a tuple. Also, `isbool` in the original egg-eater is not updated and thus buggy after the new tag is added for tuples. Now it is fixed.

There's a known problem. Although `=` never result in a infinite loop and always gives meaningful result, the time complexity can be very bad (up to exponential). Consider the following program:

```
(fun (f n) (if (== n 0)
(tuple)
(let ((t (f (sub1 n)))) (tuple t t))
))
(let (
    (a (f input))
    (b (f input))
)
(= a b)
)
```

Consider `(f n)`, which looks like a full binary tree. However, the left child and right child of each node are the same reference, so `(f n)` takes only `O(n)` space in memory. However, it takes `O(2^n)` time to compare two such value, which is not quite ideal. Although, we can probably justify this complexity by saying `(print (f n))` also takes exponential time. But in fact I believe we can indeed do better, by storing a list of pairs of equal values. Unfortunately, I don't have time to implement this.

Resources used

Again, not much other than lectures and code provided by the instructor.