Proposal for Language Extension: Floating Point Numbers and Operations

Haoran Tang
Zhen Xu

1 Concrete Syntax Impact

We will add the concrete syntax of float number:

```
<expr>: ...
      | <expr> <. <expr>
      | <expr> >. <expr>
      | <expr> <=. <expr>
      | <expr> >=. <expr>
      | <expr> +. <expr>
      | <expr> -. <expr>
      | <expr> *. <expr>
      | isfloat(<expr>)
```

2 Examples

```
1, Floating-point addition
The expression
let a = 3.5 in
let b = 2.1 in
print(a +. b)
will output:
5.6
5.6
Note that the result may not necessarily be 5.6 but it should be a float near 5.6
```

3 Abstract Syntax and Semantics Impact

3.1 AST Modifications

We will add a Float type in Exp and the corresponding Prim opertions in Prim.

```
pub enum Exp<Ann> {
    ...
    Float(f32, Ann),
}

pub enum Prim<Ann> {
    ...
    FAdd,
    FSub,
    FMul,
    FLt,
    FGt,
    FLe,
    FGe,
```

```
    isFloat,
}
```

3.2 Semantic

The semantics of float numbers are mainly straightforward. Note that we don't allow arithmetic operation between float number and integer number.

We will use the following representation for float numbers: Float number will be represented with first 32 bits in IEEE754 format and store in one ward of the heap. A Float number value is stored in variables as the address of the word of that float number, but add 101 to the value to act as a tag. For example, if 1.125 is stored in address 0x0adadad0, the float number value would be 0x0adadad5. The value stored in the address would be 3F900000 00000000, which the first 32bit should be IEEE format of the float number.

With this change, we extend the set of tag bits to the following:

Bit Pattern	Value Type
`0xWWWWWWWW[bbb0]`	Number
`0xFFFFFFF[1111]`	True
`0x7FFFFFF[1111]`	False
`0xWWWWWWWW[b001]`	Array
`0xWWWWWWWW[b011]`	Closure
`0xWWWWWWWW[b101]`	Float Numbers

Table 1: Bit Patterns and Corresponding Value Types

And we will raise errors at runtime in the following cases:

1, +., -., *., will raise an error "arithmetic expected a float number" if the arguments are not both float numbers.

2, < ., > ., <= ., >= ., == ., will raise an error "comparison expected a float number" if the arguments are not both float numbers

3, +., −., ∗., will raise an error "overflow" if the result overflows. There are three different sitution for overflow. First is that when the float is statically too large, it will be readed as inf and when the float is statically too small, it will be readed as 0.0. Second is that when you try to add up two large float which is still in the range of float, the result will be inf. Third is that when add/mul on two inf, the result is still inf and when two inf minus each other, the result is NaN.

We add one new primitive operations, isFloat. This operation has the effect type of Any -¿ Bool, and will return true if the argument is a float.

4 Transformation of Existing Features

- **Print:** The behavior of printing integer numbers, booleans, arrays, closures are as usual. And we will just add print for float numbers. For example, print(3.4) will print float number 3.4 in command line.
- **Equality:** The behavior of == is still the same for integer numbers, booleans, arrays, closures. We allow float numbers in both sides of == and the only case == will evaluate to true is the both sides' float number are the same, other cases will all evaluate to false.

5 Code Generation

Most part is just the same as integer and their operations, while we still need different approaches.

When we load float point numbers, we use instruction fld to load the float numbers for heap address. When we define float point numbers, we store them into heap.

When performing operations, we load the first argument and perform operation according to the specific operation with second argument. And store the float number back to the heap.

Note that the value of RAX will be set to the float number's value according to the semantics sections. (The float operation will set RAX to the result's value)

6 Testing

We include all our past tests for Egg-eater (assignment6). And we create some new unit tests for float numbers and some comprehensive tests in examples repo with automated tests for running them in example.rs. Note that, the operation of float number is not accurate. For example, 0.1 +. 0.2 may not equal to 0.3. So, as long as our float operation end up in a certain range, say ¡0.301 and ¿0.299, we view the operation as correct.

7 More examples

```
2, Floating-point subtraction
The expression
let c = 5.5 in
let d = 1.5 in
c -. d
will output:
4.0

3, Floating-point multiplication
let e = 4.0 in
let f = 2.5 in
e *. f
will output:
10.0

4, Floating-point comparison (greater than)
let g = 7.2;
let h = 3.3;
print(g >. h);
will output: true

5, Floating-point comparison (less than)
let i = 1.1;
let j = 2.2;
print(i <. j);
will output: true

6, Floating-point comparison (greater than or equal to)
let k = 5.5;
let l = 5.5;
print(k >=. l);
will output: true

7, Floating-point comparison (less than or equal to)
let m = 3.3;
let n = 4.4;
print(m <=. n);
will output: true

8, Check if a value is a float
let q = 9.9;
print(isfloat(q));
will output: true
```