Sophia

aeternity smart contract programming language

General Idea

- Functional
- Meta Language family
- Contract definitions as modules
- Encapsulated mutable state
- Strong&strict typing
- ADT/record-like types
- Parametric polymorphism
- Eager evaluation
- Runs on FATE virtual machine

Program Structure

- Contract definitions
  - state - mutable part of the runtime. Can be modified only by stateful functions/entrypoints
  - entrypoints - functions callable from the outside
  - functions - for internal use only. Can be stateful
  - type definitions - non-recursive ADTs, records, aliases
  - only one per project allowed
- Namespaces
  - no state - just libraries
  - functions/private functions
- Contract declarations
  - only type definitions and entrypoint type declarations
  - used to support external calls

Program Overview

```
namespace IntegerAdder =

  function add(a : int, b : int) : int = a + b

contract C =

  type state = int  // type of mutable state

  entrypoint init() = 0  // initialization of state

  stateful entrypoint incr() =

    put(IntegerAdder.add(state, 1))  // entrypoint altering state

  entrypoint get() = state  // entrypoint returning state
```

Entrypoints, functions

Entrypoint:

- can be stateful
- can be payable
- callable from the outside
- no polymorphism
- must be first order

Function:

- can be stateful
- internal
- polymorphic
- higher order

Both:

- recursive
- atomic
- no loops, variables

Functional Constructions

```
function factorial(n : int) : int =
  factorial_(n, 1)


private function factorial_(n : int, acc : int) : int =
  if(n < 2) acc
  else factorial_(n - 1, n * acc)  // tailtailtailtail
```

Imperative Constructions

```
stateful entrypoint get_data(d : data) =
  require(valid_data(d), “invalid data!”)
  if(d.funny)
    haha()
  if(cool(d))
    register(d)
  else fix_and_register(d)
  d.id
```

Type System

- Known from ML-like languages (like OCaml for example)
- Non-recursive parametric polymorphism
- Full type inference
- No local polymorphism nor local recursion
- Types: ADTs, records, type aliases, contracts, type variables, type applications

Type System — builtins

- int (arbitrary size. no floats!)
- bool
- string
- unit
- tuples
- list (of kind Type->Type)
- oracles / oracle queries
- addresses (contracts, accounts)
- bytes / bits
- functions
- cryptographic utils

Type System – polymorphism

```
contract C =
  function identity(x : ‘a) : ‘a = x
  function bad_recursion(f : ‘x => ‘x) : ‘x =
    f(bad_recursion(f))
  entrypoint test() : int =
    bad_recursion(identity) + 2136
```

Type System – type alias

```
contract C =
  type my_int = int  // my_int means literally int
  type endomorphism(‘a) = ‘a => ‘a
  type intomorphism = endomorphism(int) // specialization
  entrypoint f(a : int, b : my_int) : int = a + b
  function partial_add(x : int) : endomorphism(my_int) =
    (y : int) => x + y
```

Type System – algebraic data type

Union/product types:

```
contract C =
  datatype my_bool = Tru | Fal  // either Tru or Fal
  datatype union(‘a, ‘b) = Left(‘a) | Right(‘b)
  datatype product(‘a, ‘b) = Prod(‘a, ‘b)
  function safe_div(a : int, b : int) : union(string, int) =
    if(b == 0) Left(“division by zero!”) else Right(a / b)
  function proj_left(Prod(x, _) : product(‘a, ‘b)) : ‘b = x
```

Type System – record

```
contract C =
  record vector = {x : int, y : int}
  record comonadic_continuation(‘s) =
    { st : ‘s, cont : ‘s => ‘s }
  function is_zero(v : vector) : bool = v.x == 0 && v.y == 0
  function run_cont(con : comonadic_continuation(‘a)) : ‘a =
    con.cont(con.st)
```

Type System – contract

```
contract Stalkee =
  entrypoint stalk : () => unit
contract Stalker =
  record state = { getStalkee : Stalkee }
  entrypoint init(s : Stalkee) = state{getStalkee = s}
  entrypoint run() : unit = state.getStalkee.stalk()
```

Type System – inference

Type annotations are not required - the compiler will guess (but still verify) the typing.

```
function try_me(n) =  // : int => int
  let b = n > 0  // : bool
  if(b) n + 1 else List.head([]) // : int
```

Type System – tuples, functions

```
(3, (x, y) => x) : int * ((‘a, ‘b) => ‘a)


let x = (1, 2, 3)

let (q, _, _) = x
```

Type System – limitations

- No higher kinded types
- No higher rank types
- No local polymorphism
- No ad-hoc polymorphism
- No recursive polymorphism
- No contract inheritance
- No contract polymorphism
- All types are comparable by equality or linear order

Pattern Matching

```
function f(x : int) : unit = switch(x)
  | 0 => abort(“zero bad”)
  | _ => ()
// datatype option(‘a) = None | Some(‘a)
function g(x : option(int)) : int = switch(x)
  | None => 0
  | Some(n) => n
```

Pattern Matching – lists

```
function filter(p : ‘a => bool, l : list(‘a)) : list(‘a) =
  switch(l)
    | [] => []
    | h::t =>
      let rest = filter(p, t)
      if(p(h)) h::rest else rest
```

List Comprehensions

```
let l = [k | a <- [1,2,3], b <- [2,3,4], c <- [3,4,5]
        , if(a >= b && b >= c)
        , let f(x) = x + a ^ 2
        , let k = f(b) * f(c)
        ]
function filter(p, l) = [x | x <- l, if(p(x))]
```

Maps

```
function test() =
  // map keys cannot be polymorphic, functional, nor maps
  let m = {["key1"] = 1, ["key2"] = 2}
  let one = m[“key1”]
```

Oracles

Sophia supports builtin oracles - they can be registered, queried and answered by the interface provided by the standard library.

Oracles provide an interface between a contract and a real-world centralized entity. They are used for example in the Superhero URL solving mechanism.

Builtin Interface

(this is the point where I refer to the stdlib docs)

In short: contracts can query the contracts’ properties, chain, calldata, perform oracle actions, interact with AENS etc.

The standard library features operations on lists, optionals, bitsets, bytestrings, strings, hashes, fractional numers, functional utilities and more.

Exceptions

Builtin abort : string -> unit function allows to throw an uncatchable error that breaks the computation and reverts the state. The gas is still charged.

Optional types are worth of considering if the errors are supposed to be caught.

Splitting to files

```
include “File.aes”
```

Searches file in current path or in standard library. Solves import cycles and diamond problem.

Questions?

There must be some