Smart Contracts

Philip Wadler
University of Edinburgh
IOHK, Lisbon
Wednesday 17 January 2018


LAMBDA LX-56-EST


LAMBDA LX-56-EST

(Thanks to Bruce Milligan)

Plutus,
Plutus Core,
and IELE

Plutus

```
factorial : Integer -> Integer
factorial n =
  if n < 1
  then 1
  else n * factorial (n - 1)
```

Plutus Core

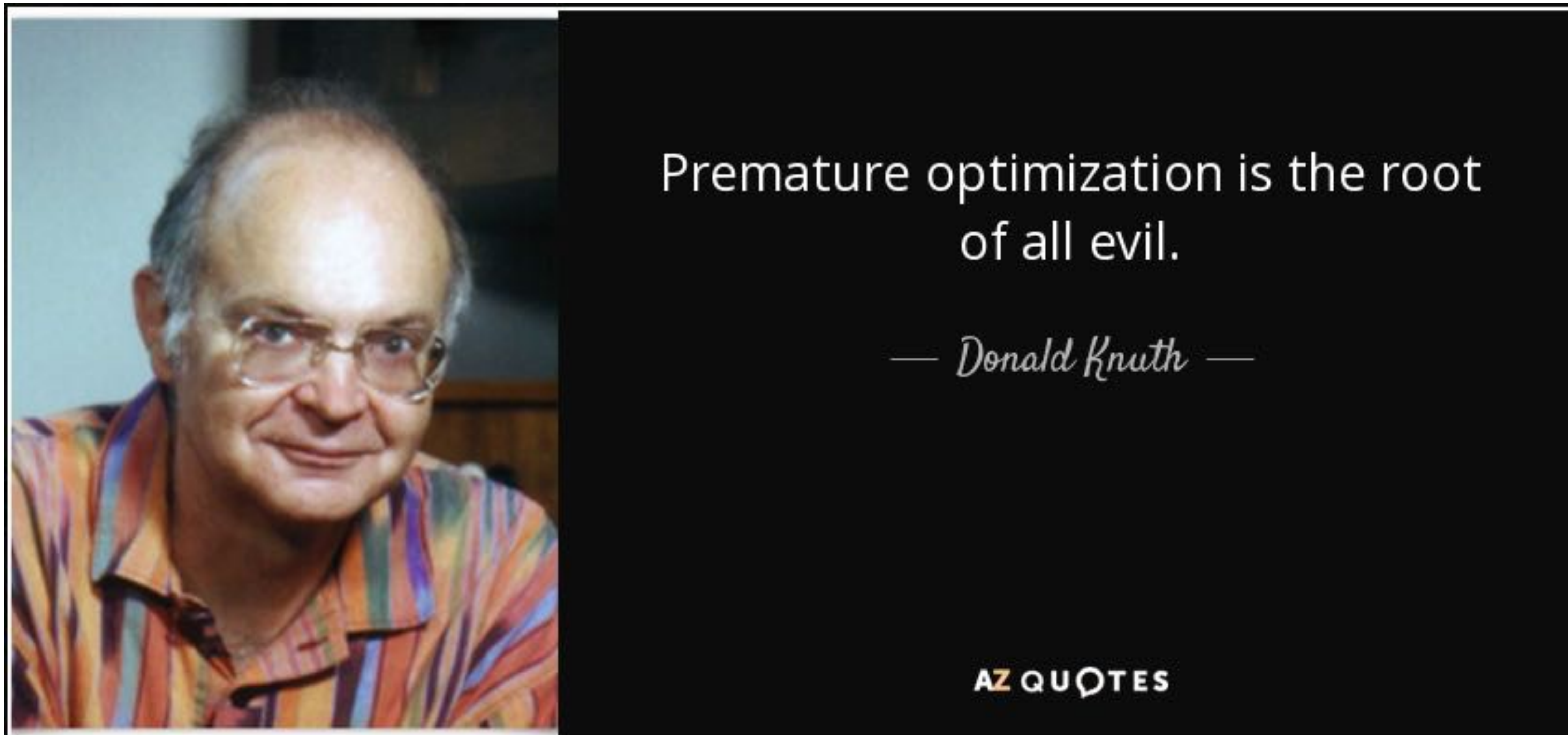
```
(declare factorial (fun (integer) (integer)))
(define factorial (lambda n
  (case [lessThanInteger n 1]
    (Prelude.True () 1)
    (Prelude.False ()
      [multiplyInteger n
        [factorial [subtractInteger n 1]]]))))
```

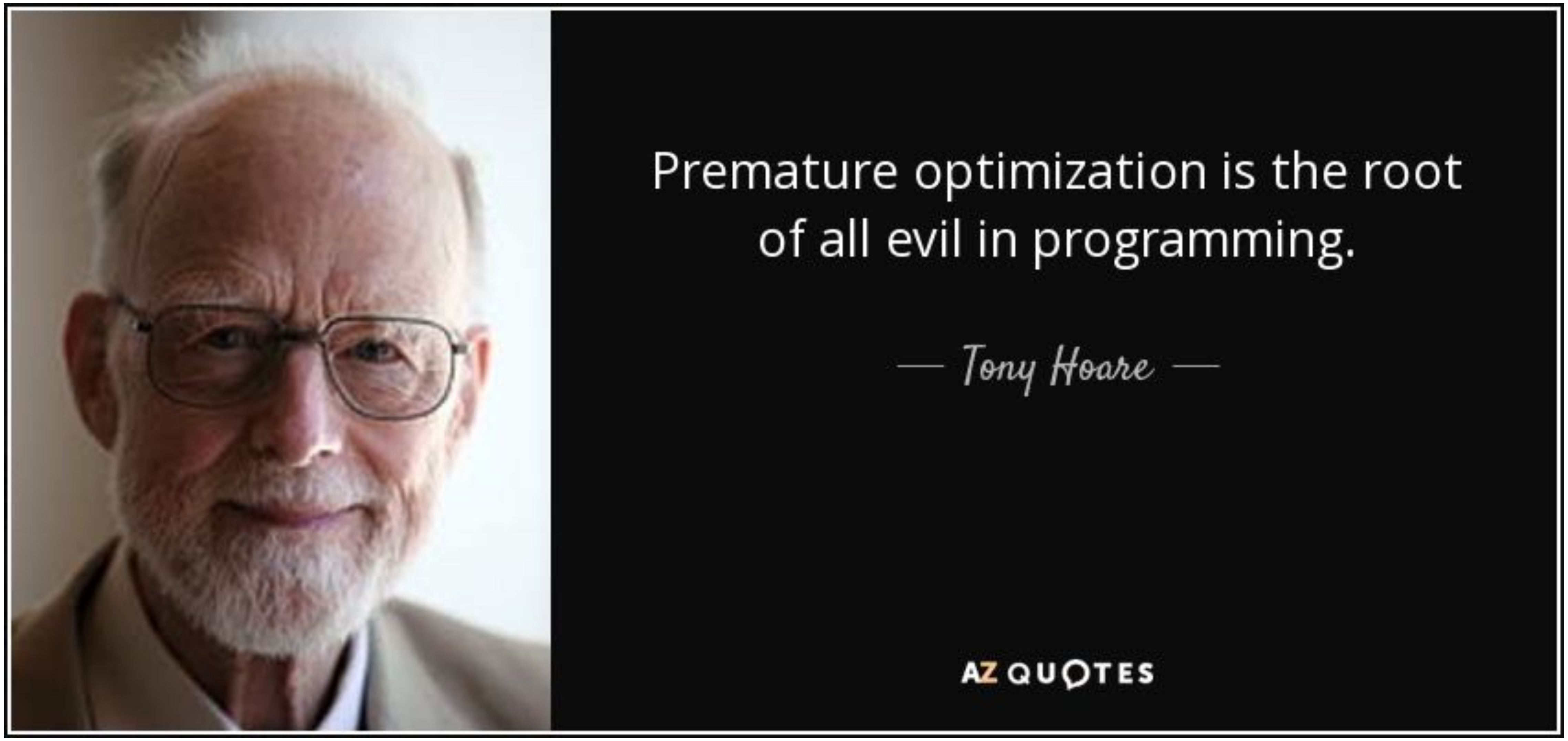
IELE

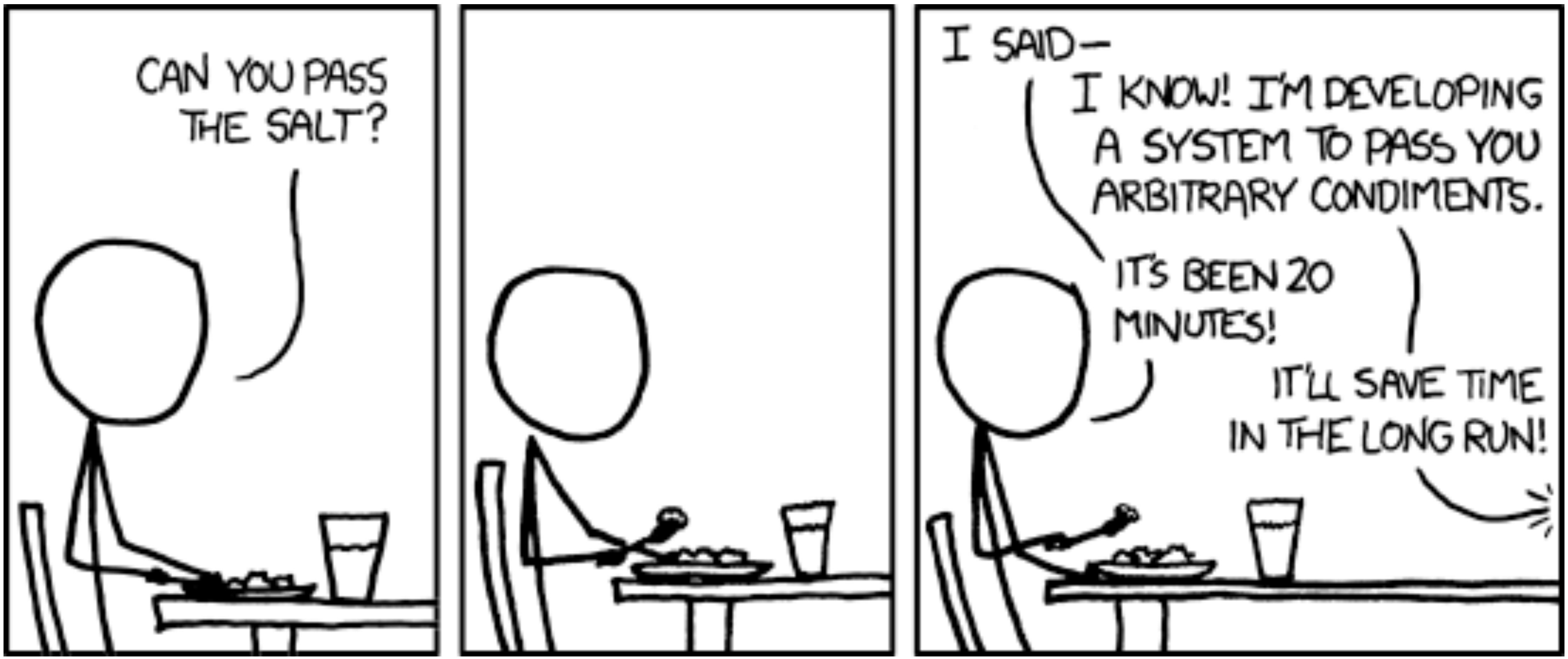
```
contract Factorial {
  define public @factorial(%n) {
    // ensure that %n is larger than or equal to 0.
    %lt = cmp lt %n, 0
    br %lt, throw
    %result = 1
  condition:
    %cond = cmp le %n, 0
    br %cond, after_loop
  loop_body:
    %result = mul %result, %n
    %n      = sub %n, 1
    br condition
  after_loop:
    ret %result
  throw:
    call @iele.invalid()
  }
}
```

Premature optimisation

Most of the time in a smart contract will be spent executing cryptographic primitives


Premature optimization is the root of all evil.
— Donald Knuth —
AZ QUOTES


Premature optimization is the root of all evil in programming.
— Tony Hoare —
AZ QUOTES


CAN YOU PASS THE SALT?
I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

Comparative resources

IELE

8 —> 19

(Grigore Rosu)

Plutus

1.2

(Darryl McAdams)

Three issues

1. Unbounded integers

A cool idea

- Erlang has unbounded integers.

- Say one deploys a successful phone switch that runs for a long time. Counter passes word size.

- Previously: overflow!

- Now: no problem!

Uh oh!

How do we allocate gas cost?

One-word integers
Addition: constant
Multiplication: constant

Unbounded integers
Addition: maximum of logarithm of values
Multiplication: sum of logarithm of values

RAML

Resource-Aware ML
(Jan Hoffman and others, www.raml.co)

Does a good job with one-word integers

Struggles to analyse multiword integers

RAML: one-word integers

```
let iplus n m = let () = Raml.tick 1.0 in n+m
type nat = Z | S of nat
let sumorial n =
    let rec sumo n a =
      match n with
      | Z -> 0
      | S n' -> iplus a (sumo n' (iplus a 1))
    in
    sumo n 1

Resource Aware ML, Version 1.3.2, January 2017
== sumorial :

  Simplified bound:
     9.00 + 26.00*M
   where
     M is the number of S-nodes of the argument
```

RAML: multiword integers

```
type bigint = int list
let of_int n = [n]
let add b c = …
let sumorial n =
    let rec sumo n a =
      match n with
      | Z -> of_int 0
      | S n' -> add a (sumo n' (add a (of_int 1)))
    in
    sumo n (of_int 1)

Resource Aware ML, Version 1.3.2, January 2017
Analyzing function sumorial ...

 Simplified bound:
     21.00 + 125.33*M + 80.00*M^2 + 26.67*M^3
   where
     M is the number of S-nodes of the argument
```

RAML: multiword integers

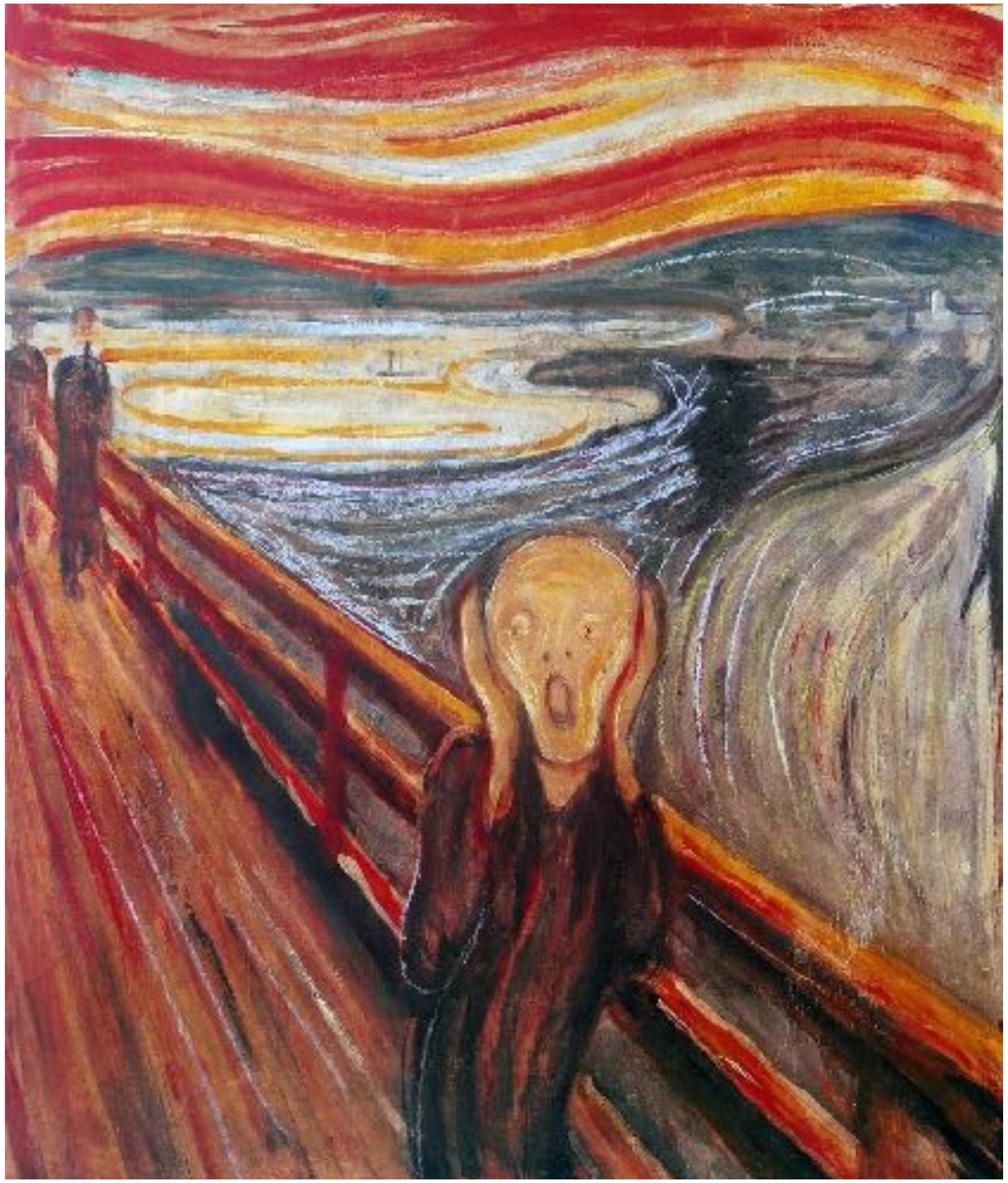
```
type bigint = int list
let of_int n = [n]
let add b c = …
let mult b c = …
let factorial n =
    let rec fact n a =
      match n with
      | Z -> of_int 1
      | S n' -> mult a (fact n' (add a (of_int 1)))
    in
    fact n (of_int 1)
```

```
Resource Aware ML, Version 1.3.2, January 2017
Analyzing function factorial …

  A bound for factorial could not be derived.
  The linear program is infeasible.
```

My nightmare

- Say one deploys a successful smart contract that runs for a long time. Counter passes word size.
- Previously: overflow exception
- Now: out of gas
- And we've paid for it by making it far harder to analyse gas cost!



2. Abstract data types

Abstract data types in Haskell

```
module Stack(Stack, empty, isempty, push, pop, top) where

newtype Stack = MkStk [Int]

empty :: Stack
empty = MkStk []

isempty :: Stack -> Bool
isempty (MkStk x) = null x

push :: Int -> Stack -> Stack
push a (MkStk x) = MkStk (a:x)

pop :: Stack -> Stack
pop (MkStk (a:x)) = MkStk x

top :: Stack -> Int
top (MkStk (a:x)) = a
```

Abstract data types in Miranda

```
abstype stack
with empty :: stack
     isempty :: stack->bool
     push :: num->stack->stack
     pop :: stack->stack
     top :: stack->num

stack == [num]
empty = []
isempty x = null x
push a x = a:x
pop (a:x) = x
top (a:x) = a
```

Trade offs

Haskell:
More familiar to some of our user base

Miranda:
Easier to read and write

3. Data constructors

Validator and Redeemer

```
validator :: A → comp B

redeemer :: comp A
```

Validator and Redeemer

The validator may create a new abstract type,
which is used by the redeemer

```
validator :: (∀x. A[x] → comp B[x]) → comp C

redeemer :: ∀x. A[x] → comp B[x]
```

Validator and Redeemer

```
validator :: (∀stack.
                stack
                (stack→bool) →
                (num→stack→stack) →
                (stack→num) →
                (stack→stack) →
                comp B[x]) →
              comp C
validator redeemer =
   let answer =
     redeemer stack
              empty
              isEmpty
              push
              pop
              top
   in … do stuff with answer …
```

What about data type declarations?

```
data Nat = Zero | Suc Nat

plus Zero    n  =  n
plus (Suc m) n  =  Suc (plus m n)
```

Constructors used in pattern matching are not just functions.
Needs a whole new model. Not standard.
Uh oh!

Church Encoding

```
abstype nat
   with zero :: nat
        suc :: nat → nat
        ncase :: nat → (∀x. x → (x → x) → x)

nat == (∀x. x → (x → x) → x)
zero x z s = s
suc x z s n = s (n x z s)
ncase n = n

plus :: nat → nat → nat
plus m n = ncase m n suc
```

Scott Encoding

```
abstype nat
   with zero :: nat
        suc :: nat → nat
        ncase :: nat → ∀x. x → (nat → x) → x

nat == ∀x. x → (nat → x) → x
zero z s = z
suc n z s = s n
ncase n = n

plus :: nat → nat → nat
plus m n = ncase m n (λm. suc (plus m n))
```

Plutus Core

Kinds `J,K ::=`

```
*
J → K
```

Types `A,B ::=`

```
X
A → B
∀X.B
μX.B
ρ
```

Terms `L,M,N ::=`

```
x
λx:A.N
L M
ΛX:K.N
L A
μx:A.N
ρ
```

Conclusion

Do you have opinions about programming languages?

We need your help!