Do you have a problem?

Write a compiler!

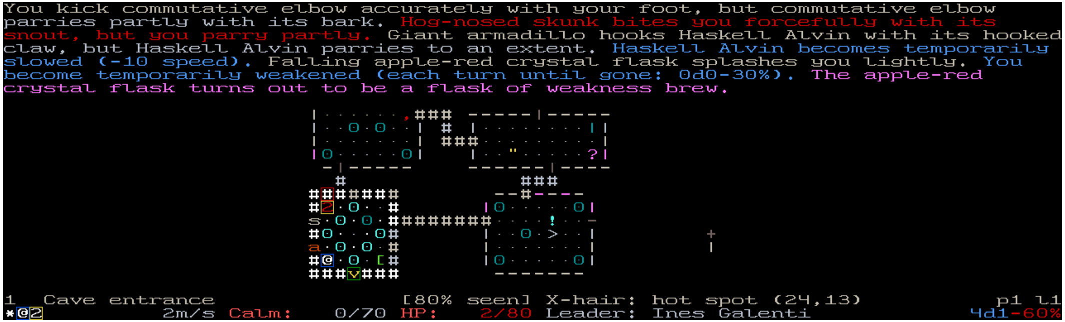
Oleg Grenrus, @phadej

ClojuTRE · Helsinki · 2019-09-26

Well-Typed

The Haskell Consultants

Imagine you are writing a cool game…



```
You kick commutative elbow accurately with your foot, but commutative elbow
parries partly with its bark. Hog-nosed skunk bites you forcefully with its
snout, but you parry partly. Giant armadillo hooks Haskell Alvin with its hooked
claw, but Haskell Alvin parries to an extent. Haskell Alvin becomes temporarily
slowed (-10 speed). Falling apple-red crystal flask splashes you lightly. You
become temporarily weakened (each turn until gone: 0d0-30%). The apple-red
crystal flask turns out to be a flask of weakness brew.
```

```
1  Cave entrance              [80% seen] X-hair: hot spot (24,13)            p1 L1
*@2         2m/s Calm:    0/70 HP:    2/80 Leader: Ines Galenti             4d1-60%
```

Allure of the Stars `http://www.allureofthestars.com/`

Players' comment: So that was a lot of fun,

but *I have no idea what's going on*.

SplitMix is a **fast**, splittable pseudorandom number generator.

Core of a 32-bit variant is like:

```
(defn mix [z0]
  (let [z1 (xor-shift-multiply  z0 16 0x85ebca6b)
        z2 (xor-shift-multiply  z1 13 0xc2b2ae35)
        z3 (xor-shift           z2 16)]
        z3))

(defn xor-shift [x s]
  (bit-xor x (unsigned-bit-shift-right x s)))

(defn xor-shift-multiply [x s m]
  (* (xor-shift x s) m))
```

Well-Typed

- JavaScript is a great platform…
- …but a terrible programming language

For example:

```
0x12345679 * 0x12345679
  = 93281313483490610
```

This is odd, I mean even!

- JavaScript is a great platform…
- …but a terrible programming language

For example:

```
0x123456789 * 0x123456789
  = 93281313483490610
```

This is odd, I mean even!

Well-Typed

Long multiplication with 16-bit "digits":

```
                 1234  5678
×                9abc  def0
───────────────────────────
                 4b4d  2080
           0fda  28c0
           3443  b020
+    0b00  a630
───────────────────────────
     0b00  ea4e  242d  2080
```

Long multiplication with 16-bit "digits":

```
            1234  5678
×           9abc  def0
──────────────────────
            4b4d  2080
      0fda  28c0
      3443  b020
+ 0b00  a630
──────────────────────
  0b00  ea4e  242d  2080
```

SplitMix

```
(defn mix [z0]
  (let [z1 (xor-shift-multiply  z0 16 0x85ebca6b)
        z2 (xor-shift-multiply  z1 13 0xc2b2ae35)
        z3 (xor-shift           z2 16)]
        z3))

(defn xor-shift [x s]
  (bit-xor x (unsigned-bit-shift-right x s)))

(defn xor-shift-multiply [x s m]
  (* (xor-shift x s) m))
```

If we'd use macros to expand the multiplication:

```
(let [z1 (let [z (let [tmp z0] (bit-xor tmp (ubsr tmp 16)))
               x (bit-and (ubsr z 16) 0xffff)
               y (bit-and z 0xffff)
               uv 0x85ebca6b
               u (bit-and (ubsr uv 16) 0xffff)
               v (bit-and uv 0xffff)]
            (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
               (* y v)))
      z2 (let [z (let [tmp z1] (bit-xor tmp (ubsr tmp 13)))
               x (bit-and (ubsr z 16) 0xffff)
               y (bit-and z 0xffff)
               uv 0xc2b2ae35
               u (bit-and (ubsr uv 16) 0xffff)
               v (bit-and uv 0xffff)]
            (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
               (* y v)))
      z3 (let [tmp z2] (bit-xor tmp (ubsr tmp 16)))]
  z3)
```

Zooming in...

```
(let [z (let [tmp z0] (bit-xor tmp (ubsr tmp 16)))
      x (bit-and (ubsr z 16) 0xffff)
      y (bit-and z 0xffff)
      uv 0x85ebca6b
      u (bit-and (ubsr uv 16) 0xffff)
      v (bit-and uv 0xffff)]
   (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
      (* y v)))
```

Well-Typed

Zooming in…

```
(let [z (let [tmp z0] (bit-xor tmp (ubsr tmp 16)))
      x (bit-and (ubsr z 16) 0xffff)
      y (bit-and z 0xffff)
      uv 0x85ebca6b
      u (bit-and (ubsr uv 16) 0xffff)
      v (bit-and uv 0xffff)]
  (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
     (* y v)))
```

Zooming in…

```
 (let [z (let [tmp z0] (bit-xor tmp (ubsr tmp 16)))
       x (bit-and (ubsr z 16) 0xffff)
       y (bit-and z 0xffff)
       uv 0x85ebca6b
       u (bit-and (ubsr uv 16) 0xffff)     ;; high bits
       v (bit-and uv 0xffff)]              ;; low bits
   (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
      (* y v)))
```

Let’s rather write a compiler

We are interested in **a very very tiny subset** of CLOJURE

```
(defn mix32 [z0]

  (let [z1 (l/xor-shift-multiply z0 16 0x85ebca6b)
        z2 (l/xor-shift-multiply z1 13 0xc2b2ae35)
        z3 (l/xor-shift          z2 16)]
    z3))
```

We are interested in **a very very tiny subset** of CLOJURE

```
(defn mix32 [z0]
  (magic
  (let [z1 (l/xor-shift-multiply z0 16 0x85ebca6b)
        z2 (l/xor-shift-multiply z1 13 0xc2b2ae35)
        z3 (l/xor-shift          z2 16)]
    z3)))
```

`magic` speeds up execution by 10 percent with Lumo

What is this magic?

```clojure
(defmacro magic
  [form]
  (->> form
       (from-clojure environment)
       (rewrite-once expand-mult)
       (optimise)
       (to-clojure)))
```

```
(defmacro magic
  [form]
  (->> form
       (from-clojure environment)
       (rewrite-once expand-mult)
       (optimise)
       (to-clojure)))
```

Convert to and from **internal representation**

```clojure
(defmacro magic
  [form]
  (->> form
       (from-clojure environment)
       (rewrite-once expand-mult)
       (optimise)
       (to-clojure)))
```

Rewrite * to produce **correct results**

```clojure
(defmacro magic
  [form]
  (->> form
       (from-clojure environment)
       (rewrite-once expand-mult)
       (optimise)
       (to-clojure)))
```

Make it **fast**

We use nested vectors for internal representation:

```
(+ (bit-xor 1 3) unknown)
```

is represented as

```
[:add [:xor [:lit 1] [:lit 3]] [:gbl "unknown"]]
```

$\Rightarrow$ CODE is DATA

Well-Typed

- …so CODE is DATA
- But how to represent (local) **variables**?
- A solution is **de Bruijn** indices: no names, no problems.

```
(let [x 42 y 29]          (let [a 42 b 29]
  (+ x y))                  (+ a b))
```

$\Longrightarrow$

```
[:let [:lit 42]
  [:let [:lit 29]
    [:add [:var 1] [:var 0]]]]
```

. . . and names are metadata

```
^{:name x}[:let [:lit 42]
  ^{:name y}[:let [:lit 29]
    [:add [:var 1] [:var 0]]]]
```

Optimizations

Recall our code snippet

```
(let [z (let [tmp z0] (bit-xor tmp (ubsr tmp 16)))
      x (bit-and (ubsr z 16) 0xffff)
      y (bit-and z 0xffff)
      uv 0x85ebca6b
      u (bit-and (ubsr uv 16) 0xffff)
      v (bit-and uv 0xffff)]
  (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
     (* y v)))
```

Well-Typed

Let us keep it super simple.

- Given a set of **simple rewrite rules**
- **Find** a match $\longrightarrow$ apply the rewrite
- Continue until nothing matches

Inlining is the most powerful optimisation: it makes opportunities for others optimisations to happen. We rewrite **`let`** to “no let”.

```
(let [x expression]                (let [x 1
  body)                                  y 2]
                                     (+ x y))
```

$\Longrightarrow$

```
body'                             (+ 1 2)
```

where `body'` is `body` where `x` is replaced by `expression`

We don't inline `(fibonacci 100)` it's expensive to compute...

```
(let [x (fibonacci 100)]
  (+ x x))
```

but if there's already a value, it's cheap to inline:

```
(let [x 354224848179261915075]
  (+ x x))
```

$\Longrightarrow$

```
(+ 354224848179261915075 354224848179261915075)
```

Well-Typed

When inlining is a **valid** rewrite?

When inlining is a **valid** rewrite?

Consider

```
(let [foo (do-foo)
      bar (do-bar)
      _   (do-quux)
  (+ bar foo))
```

$\Longrightarrow$

```
(+ (do-bar) (do-foo))
```

Constant folding is a trivial rewrite

- For every primitive operation ($+$, $\times$, ...),
- if the arguments are constants i.e. literals,
- perform the computation at compile-time

Example:

```
(+ 1 2)
```

$\Longrightarrow$

```
3
```

Well-Typed

When **constant folding** is a valid rewrite?

When **constant folding** is a valid rewrite?

- That’s a trick question.
- It depends on primitives, whether they can be evaluated at the compile time.
- But what about non-primitive application:

```
(def precalculated (precalculate 100))
```

Not only optimisations are conceptually simple, their implementation is simple too:

```
(defn constant-fold [[key & args :as e]]
  (if (and (not (or (= key :lit) (= key :gbl) (= key :var)))
           (all? is-lit? args))
    [:lit (evaluate  e)]))
```

With constant folding and inlining **this**

```
(let [z1 (let [z (let [tmp z0] (bit-xor tmp (ubsr tmp 16)))
               x (bit-and (ubsr z 16) 0xffff)
               y (bit-and z 0xffff)
               uv 0x85ebca6b
               u (bit-and (ubsr uv 16) 0xffff)
               v (bit-and uv 0xffff)]
           (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
              (* y v)))
      z2 (let [z (let [tmp z1] (bit-xor tmp (ubsr tmp 13)))
               x (bit-and (ubsr z 16) 0xffff)
               y (bit-and z 0xffff)
               uv 0xc2b2ae35
               u (bit-and (ubsr uv 16) 0xffff)
               v (bit-and uv 0xffff)]
           (+ (bsl (bit-and (+ (* x v) (* y u)) 0xffff) 16)
              (* y v)))
      z3 (let [tmp z2] (bit-xor tmp (ubsr tmp 16)))]
  z3)
```

Well-Typed

With constant folding and inlining our example **becomes**

```
(let
  [z1
   (let [z (bit-xor z0 (ubsr z0 16))
         y (bit-and z 0xffff)]
     (+ (bsl (bit-and (+ (* (bit-and (ubsr z 16) 0xffff) 0xca6b)
                         (* y 0x85eb)) 0xffff) 16)
        (* y 0xca6b)))
   z2
   (let [z (bit-xor z1 (ubsr z1 0xd))
         y (bit-and z 0xffff)]
     (+ (bsl (bit-and (+ (* (bit-and (ubsr z 16) 0xffff) 0xae35)
                         (* y 0xc2b2)) 0xffff) 16)
        (* y 0xae35)))]
  (bit-xor z2 (ubsr z2 16)))
```

Well-Typed

Let-from-let optimization:

```
(let [x (let [y y-value]
          x-value)]
  body)
```

$\Longrightarrow$

```
(let [y y-value
      x x-value]
  body)
```

Let-floating: moving bindings to give faster programs by Simon Peyton Jones, Will Partain, André Santos; ICFP '96

Well-Typed

With three optimisations we get quite pretty

```
(let [z1 (bit-xor z0 (ubsr z0 16))
      z2 (bit-and z1 0xffff)
      z3
      (+ (bsl (bit-and (+ (* (bit-and (ubsr z1 16) 0xffff) 0xca6b)
                          (* z2 0x85eb)) 0xffff) 16)
         (* z2 0xca6b))
      z4 (bit-xor z3 (ubsr z3 13))
      z5 (bit-and z4 0xffff)
      z6
      (+ (bsl (bit-and (+ (* (bit-and (ubsr z4 16) 0xffff) 0xae35)
                          (* z5 0xc2b2)) 0xffff) 16)
         (* z5 0xae35))]
  (bit-xor z6 (ubsr z6 16)))
```

Well-Typed

Conclusion

- Implementing small (domain specific) languages is **fun**
- Not only numerics, but also HTTP routing, authorization rules, CI-scripts. . .
- Make them typed, lazy, pure, total, dependent. . .
- ordinary programs transform DATA to DATA, compilers CODE to CODE. . .

Extras

Define a function, and call it a day!?

	runtime	**relative** to GHC
GHC	1470ms	$1\times$
GHCJS	158000ms	$107\times$
GHCJS + OPT	14000ms	$10\times$
LUMO	4240ms	$2.9\times$
LUMO + MACRO	3760ms	$2.6\times$
LUMO + OPT	3370ms	$2.3\times$

Well-Typed

More optimisatons: CSE

Common subexpression elimination. A simple variant, is consider only existing bindings

```
(let [x expression]
  (... expression ...))
```

$\Longrightarrow$

```
(let [x expression]
    (... x ...))
```

Such situations arise, when inlining does its job. E.g. the same `expression` could be bound to different variables.