# Rusty Runtimes

Aditya Siram

September 17, 2016

# Outline

# Overview

- Rust is a systems programming language
- Less used for language development
- Explores Rust for language implementation
- And as a compilation target!

# About Me

- ML/Lisp background
- Long time user, first time implementor
- Very little low level knowledge
- Mostly managed runtimes

# Rust

- Lot of ML influence
  - Pattern matching
  - Emphasis in immutability
- Easy to learn.
- Mature metaprogramming.
- But still mostly imperative

# KLambda

- Lisp-ish

```
(defun adder (X) (+ 1 X))
```

- Scheme-ish

```
(defun length (list accum)
   (cond ((= () list) accum)
   (true (length (tl list) (+ accum 1)))))
```

- TCO'ed

# KLambda

- Curried!

```
(let F (map (lambda X (+ 1 X)))
   (F (cons 1 (cons 2 (cons 3 ())))))
```

- Tiny
- Has a spec!

# Types

- Base types

```
#[derive(Debug, Clone)]
pub enum KlToken {
    Symbol(String),
    Number(KlNumber),
    String(String),
    Cons(Vec<KlToken>),
    Recur(Vec<KlToken>)
}
```

- Numbers

```
#[derive(Debug, Clone)]
pub enum KlNumber {
    Float(f64),
    Int(i64),
}
```

# Parsing

- Nom.
- Very macro heavy!

# Parsing a string

- Top level

```
named!(klstring<KlToken>,
 chain!(
    char!(’\"’) ~
    contents:  many0!(klstringinnards) ~
    char!(’\"’),
    || KlToken::String(make_quoted_string(contents))
 )
);
```

- Innards

```
named!(klstringinnards< &[u8] >,
       escaped!(none_of!("\"\\"), ’\\’, one_of!("\"n\\"))
       );
```

# Debugging

- Rust macro debugging is nice!

```
named!(klstring<KlToken>,
  chain!(
      ...
      ...
      char!("hello"),
      || ...
));
```

- Error

```
   --> src/main.rs:442:1
    |
442 | named!(klstring<KlToken>,
    | ^
    |
```

# Parsing Symbols

- Parsing symbols

```
named!(klsymbol<KlToken>,
  chain!(
  initial: one_of!(CHARACTERS) ~
  remainder: many0!(
    alt_complete!(
        one_of!(DIGITS) |
        one_of!(CHARACTERS)
    )
  ),
  || {
      let mut res : Vec <char> = vec![initial];
      res.extend(remainder);
      KlToken::Symbol(res.into_iter().collect())
  })
);
```

# Writing A Macro

- s-expression

```
(func b 1 2 3)
```

- Parser

```
named!(klsexp<KlToken>,
  chain!(
    char!(’(’) ~
    inner: many0_until!(char!(’)’), klsexpinnards) ~
    char!(’)’),
    || {
        KlToken::Cons(inner)
    }
  )
);
```

# Writing A Macro

```
macro_rules! many0_until (
 ($input:expr, $stopmac:ident!( $($args:tt)* ), $submac:ident!(
    {
      let mut res = Vec::new();
      let mut input = $input;
      let mut loop_result = Ok(());

      while input.input_len() != 0 {
        match $stopmac!(input, $($args)*) {
          IResult::Error(_) => {
            match $submac!(input, $($args2)*) {
              IResult::Error(_) => {
                  break;
              },
              IResult::Incomplete(Needed::Unknown) => {
              ...
```

# KLambda Types

```
#[derive(Clone,Debug)]
pub enum KlElement {
    Symbol(String),
    Number(KlNumber),
    String(String),
    Cons(Vec<Rc<KlElement>>),
    Closure(KlClosure),
    Vector(Rc<UniqueVector>),
    Stream(Rc<KlStream>),
    Recur(Vec<Rc<KlElement>>)
}
```

# Closures

```
#[derive(Clone)]
pub enum KlClosure {
    FeedMe(Rc<Fn(Rc<KlElement>) -> KlClosure>),
    Thunk(Rc<Fn() -> Rc<KlElement>>),
    Done(Result<Option<Rc<KlElement>>,Rc<KlError>>)
}
```

# Example

- Turning a string into a symbol

```
pub fn intern() -> KlClosure {
 FeedMe(
  Rc::new(
   | string | {
    match &*string {
        &KlElement::String(ref s) => {
            Done(Ok(Some(Rc::new(Symbol(s.clone())))))
        },
        _ => Done(make_error("..."))
    }}))}
```

# Example

- Pos

```
pub fn pos() -> KlClosure {
 FeedMe(
  Rc::new(| string | {
    FeedMe(
     Rc::new(move | number | {
       let string = string.clone();
        match &*string {
          &KlElement::String(ref s) => {
               ...
          },
           ...
       },
       _ => ...
```

# Example

- And

```
pub fn and () -> KlClosure {
   ...
    | a_thunk | {
    ...
        move | b_thunk | {
           let forced = force_thunk(a_thunk.clone())
           match &*forced {
              ...
              _ => {
                       let forced = force_thunk(b_thunk)
                       match &*forced {
                          ...
                          _ => true
                       }
   ...
```

# Stored in a Function Table

- Global mutable function table

```
thread_local!(
static FUNCTION_TABLE: RefCell<HashMap<String, KlClosure>>
     RefCell::new(HashMap::new())
)
```

- Bootstrapping

```
pub fn fill_function_table() {
 FUNCTION_TABLE.with(| function_table | {
     let mut map = function_table.borrow_mut();
     map.insert("pos" , pos());
     map.insert("and" , and());
     ...
```

# Lookup

```
pub fn lookup_function(s: &String) -> Option<KlClosure> {
    FUNCTION_TABLE.with(|table|{
      let table = table.borrow();
      let function = table.get(s);
      match function {
        Some(f) => Some((*f).clone()),
        None => None
        ...
```

# Code Generation

- Function calls

```
(cons 1 ())
```

- Rust output

```
match function_apply(String::from("cons"),
                     vec![Rc::new(Number(Int(1))),
                          Rc::new(Cons(vec![]))])
{
  Ok(c) => c.clone(),
  Err(s) => Done(make_error(s.clone().as_str()))
}
```

# Code Generation

- Lets

  ```
  (let X 1 (+ X X))
  ((lambda X (+ X X)) 1)
  ```

# Code Generation

- Lambda

```
match apply_lambda(
  FeedMe(
    Rc::new(move |X| {
       let X_Copy = (*X).clone();
       match function_apply(
          String::from("+"), vec![
             Rc::new(X_Copy.clone()),
             Rc::new(X_Copy.clone())
       ])
       {
         Ok(c) => ..,
         Err(s) => ..
       }
    })),
    Rc::new(KlElement::Number(KlNumber::Int(1))))
  ...
```

# Code Generation

- Lambda

```
match apply_lambda(
  ...
           (move |X| {
       let X_Copy = (*X).clone();
             function_apply(
          String::from("+"), vec![
          (X_Copy.clone()),
                    (X_Copy.clone())
       ])
       {
         Ok(c) => ..,
         Err(s) => ..
       }
    })),
                              KlNumber::Int(1)
  ...
```

# Code Generation

```
(let X 2 (let Y (* X X) X))
((lambda X ((lambda Y X) (* X X))) 2)
```

# Code Generation

```
match lambda_apply(
 FeedMe(Rc...(move |X| {
   let X_Copy = (*X).clone();
   match lambda_apply(
    FeedMe(Rc::new(move |Y| {
      let X = X.clone();
      let X_Copy = (*X).clone();
      let Y_Copy = (*Y).clone();
        KlClosure::Done(Ok(Some(Y_Copy.clone())))
    })),
    match function_apply(String::from("+"), vec![
        Rc::new(X_Copy.clone()),
        Rc::new(X_Copy.clone())])
      {
      ...
    })),
```

# Code Generation

```
match lambda_apply(
             (move |X| {
   let X_Copy = (*X).clone();
   match lambda_apply(
                  (move |Y| {
      let X = X.clone();
      let X_Copy = (*X).clone();
      let Y_Copy = (*Y).clone();


    match function_apply(            ("+"), vec![
               (X_Copy.clone()),
               (X_Copy.clone())])
      {
      ...
      })),
```

# Tail Calls

- Paths to tail calls

  ```
  (defun length (accum list)
      (cond
        ((= list ()) accum)
        (true (length (+ accum 1) (tl list)))))
  ```

- [3 2 1]

# Tail Calls

- Token representation

```
[Cons([Symbol("defun"), Symbol("length"), Cons(...),
  Cons([Symbol("cond"),
    Cons([Cons([Symbol("="), ...]),
    Cons(Symbol("true"), [Symbol("length"),
                           Cons([Symbol("+"), ...]),
                           Cons([Symbol("tl"), ...])]) ...
```

# Tail Calls

- Mark tail calls

```
[Cons([Symbol("defun"), Symbol("length"), Cons(...),
  Cons([Symbol("cond"),
     Cons(...),
     Cons(..., [Recur [Cons([Symbol("+"), ...]),
                        Cons([Symbol("tl"), ...])]]) ...
```

- Token Type

```
#[derive(Debug, Clone)]
pub enum KlToken {
     ...
     Recur(Vec<KlToken>)
}
```

# Tail Calls

- Equivalent Element Type

```
#[derive(Clone,Debug)]
pub enum KlElement {
    ...
    Recur(Vec<Rc<KlElement>>)
}
```

# Tail Calls

- Add trampoline to closure

```
..(move |accum|
 ..(move |list|
    let trampoline = | accum,list| {
        match function_apply("cond", vec![
            Rc::new(
              Recur(vec![
                match function_apply("+", vec![..]) {
                  ...
                },
                match function_apply("tl", vec![..]) {
                  ...
                }
          ...
     }
   ...
```

# Tail Calls

```
...(move |accum|
  ...(move |list|
      let trampoline = |accum,list| {
            ...
      }
      ...
      let mut done= None;
      let mut args = vec![accum.clone(),list.clone()];
      while !done.is_some() {
        let result = trampoline(args[0].clone(),args[1].clone(
        match &*result {
           &Recur(ref v) => args = v.clone(),
           output => done = Some(Done(Ok(Some(result.clone()))
        };
      }
     done.unwrap()
```

# Benchmarks

- Test program

```
(defun build (counter list)
  (cond
     ((= 0 counter) list)
     (true (build (- counter 1)
                  (cons counter list)))))

(build 100000 ())
```

# Benchmarks

- SBCL
  - Runtime: sub second
  - Memory:
    - 23MB initial
    - jumps to 50MB
    - holds . . .
- Lua
  - Runtime: sub second
  - Memory:
    - 2MB initial
    - jumps to 14MB
    - holds . . .

# Benchmarks

- Guile
    - Runtime: 1.5 seconds
    - Memory:
        - 15MB initial
        - jumps to 18MB
        - holds
- Rust Klambda
    - Runtime: 7 mins!!!!
    - Memory:
        - 5.5 MB steady

# Lessons learned

- Needs more static analysis
  - Unnecessary copies
  - Currying is complex!
  - Most function calls are saturated
- Plenty of low hanging fruit
  - Rust inexperience
- Generating Rust is hard but worth it.
  - Type checker catchs errors in output!

# Thanks!

- Questions!