HacktorScript In ART

blaine garst 2024-02-03 15:00

FOSDEM24

HacktorScript In ART

continuing FOSDEM22 Garst, *Unhackable…End In Sight*

blaine garst 2024-02-03

Brace Yourself!

NEWS FLASH NEWS FLASH NEWS FLASH

As of 8 days ago, 8 days ago, 8 days ago

Everything in FOSDEM22 and MORE MORE MORE has a schedule!

COMMERCIAL SPACE needs UNHACKABLE !!! FUNDING, $$$, ahead!!!!

Chips used in Space are same as chips in BeagleBoard-Fire ($150)

ART running on this chip for over a year (RISC-V) PolarFire Icicle ($1500)

[arm,x86,xtensa32] x [linux,Apple darwin FreeBSD, RTOS] x [32,64]

RaspBerry Pi 4, in space, responded to web query before my eyes, $100

Planet Earth Sociey signed LOI to purchase Abacus-semi samples

Business plan NEEDS OPEN SOURCE EVERYTHING FOR “IDENTITY”!!!!

Home router nodes configured similar to Discord/NOSTR running ART for years

Abstract (as of December23)

As the Internet-of-Things moves into space the need for absolute security becomes paramount. Using inexpensive encrypted secure boot RISC-V devices and software minimalism we build first in the home for fun and then commercialize for space and ground based applications. Bug-free modules of actor components compete for efficiency in a distributed matrix of algorithms.

We present an overview of the multi-core memory safe language called Hactorscript and its widely ported Actor RunTime (ART) above minimal POSIX. With no threads, stacks, locks, or loops, the cores directly compete for work on lockless (“MPMC”) queues. These queues can be fed at “interrupt” levels.

Finite-state-machines are nearly direct Actor specifications (as is TLA) and form the first set of composable bug-free modules.

Brace Yourself!

HacktorScript Concepts (FOSDEM22)

Actors are a (so old is) new kind of mutable object

- An Actor manages message induced mutations to its (wff) DAG state
    - where all changes are atomic via a single available core
    - where computation is energy bounded (memory, cpu, messaging)
    - where pure functions and closures help compute changes
    - where the _STATE_ is completely private
- *Mutable objects* are bad ideas, subclasses break superclass assertions, and concurrent access via locking are expensive nightmares.

Actor Operations (Wikipedia)

as extended, According to Garst

An actor, upon receipt of a message, can, do 0 or more of these “things”

1 create a finite set of new actors (cpu, storage)

2 send a finite number of messages to actors (cpu, storage, network)

3 change behavior before receipt of next message (state and/or code)

And, atomically, all “things” either, succeed transparently

Or, fail with runtime sending message to module “reply” customer actor.

Additionally (Garst), an actor can directly

4 invoke runtime to send success/fail to module reply/customer actor

System Vision

bottom up rewrite of everything as layers of actor subsystems

- Provably correct FSM abstract Actors are composed into **modules**
- Abstract Drag-and-drop modules carry “constraint hair”
- **Modules** bind to concrete implementations via hair to form **subsystems**
- Subsystems are themselves each an actor, and thus also a FSM.
    - subsystem manager actor sends first message to new subsystem
    - given finite resources, runs to completion, forming a success/fail message
    - the a (reply/onward/customer) actor, sent at start, is sent the result message

HacktorScript Philosophy

Less is MORE

HacktorScript is the universal High Level AST result from parsers

- Parsers interpret syntax into ASTs, generally, then bytecodes/llvm/…
- Vision is that HactorScript is the *lingua franca* replacement underbelly
  - Traditional Imperative languages: C, C++, Java, Python, Ruby, …
  - Functional languages: Haskell, ML, Scheme, …
  - Logic languages: Planner, PROLOG

ART runs on **cores**, not threads, runtime handles cpu concurrency

Towards Bug-free/proof via these insights

Cores, not threads, therefore,
no locks, no deadlocks
no stack*, no loops (no stack overflows, infinite loops..,)
finite resources per Actor message implies no Halting Machine

Lexing replaces Parsing (no bugs due to ambiguities)

code in HactorScript is a DAG of array, set, dictionary, list, pair **atoms**, and rich value atoms (immutable objects)

a Hacktor is simply a pair (DAG _of_Code _STATE_)
STATE is always well-formed in proof sense
trivial to prove adding a photo to a library is a **wff => wff** + photo

Actors can be trivially constructed from Finite State Machines

HacktorScript simple example

its pretty dirt simple, like Myamoto Musashi minimal brushstrokes

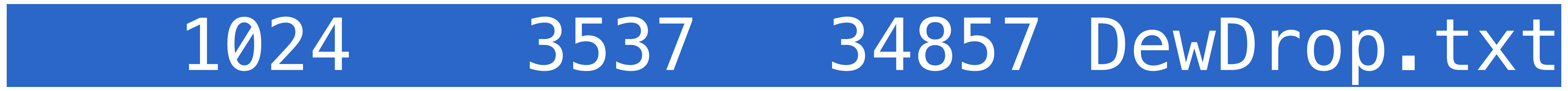
```
LET
  $constvar1 10
  $doubleit    FUNCTION ( $it ) OPER $it + $it OPER; FUNCTION;

   [ SEND $println CALL $doubleit $constvar1 CALL; SEND; ]
LET;
```

Brace Yourself!

DewDrop.txt summary of features

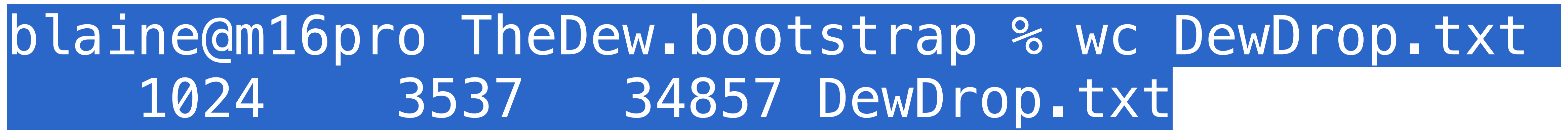
- Registration/login persistence
- Database of hubs, clients, friends
- **Authenticated remote** activation
    - both music and video playing and recording, console print
    - File transfer
- Has its own read-eval-do loop

- Haskell’s FOLDL as subroutine
- Extends its own code live
- invokes GUI in captive clients
- I/O loops to validate passphrase
- EEC crypto key generation, uses
- build system
- **in 1024 lines of HactorScript !!!!**



```
    1024     3537    34857 DewDrop.txt
```

DewDrop.txt



```
blaine@m16pro TheDew.bootstrap % wc DewDrop.txt 
    1024    3537   34857 DewDrop.txt
```

a very few illustrative examples

its beyond dense, it will make your brain hurt and for some your spirit fly

First principle: less is almost always more, strive for minimalism - its elegant

SYNTAX is pretty and damning waste of cpu for 1,5d code

Conceptual syntax of HactorScript LET … LET; expression using

Traditional Syntax style markers,
let, =, ;;, in, .

instead of **concrete** syntax hiding the Abstract

```
let

   bind = FUNCTION ( slot, value )
      NEXT ( _STATE_ +*= slot value ) NEXT;
   FUNCTION; ;;

   <elided>


   setSlots = FUNCTION ( dict )
       NEXT OPER _STATE_ ++ dict OPER; NEXT; ;;
   FUNCTION; ;;

   in

   <elided expression result>
.
```

LEXING is ENOUGH, for now, soon GUI will construct graphs of code in 2d

Actual LET … LET; **lexer** specification

No syntax style markers!, yet lexical **$** marker for parameters needed

we use the Abstract Syntax Tree (AST) directly

```
 LET

   $bind FUNCTION ( $slot $value )
      NEXT OPER _STATE_ +*= $slot $value OPER; NEXT;
   FUNCTION;

   <elided>


   $setSlots FUNCTION ( $dict )
      NEXT OPER _STATE_ ++ $dict  OPER; NEXT;
   FUNCTION;

   <elided expression result>
LET;
```

```
LET


    $bind FUNCTION ( $slot $value )
        NEXT OPER _STATE_ +*= $slot $value OPER; NEXT;
    FUNCTION;

   <elided>

    $setSlots FUNCTION ( $dict )
        NEXT OPER _STATE_ ++ $dict  OPER; NEXT;
    FUNCTION;

    <elided expression result>
LET;
```

The FUNCTION (…) <expr> FUNCTION; expressionin the required array construct **(…)** of two arguments named $slot and $value, eval <expr> as result

The Actor “change state” to value construct NEXT <expr> NEXT;

The ART special symbol _STATE_ representing existing actors state

STATE can be any value, here we assume it is a classic (key->value) mapping dictionary map

The map operator +*= means replaceor add ($slot->$value) mapping

The **object** dispatch **constructor** OPER <object> … OPER: where … represents 0 or more expressions (Kleene *)

The map operator ++ means construct join of dictionary using add/replace where 2nd map overrides values of common keys, and adds mappings with keys not already in target map

Lexer uses capitalized pairs IDEA … IDEA;

container brackets (…), { … }, […], {% … %}, {(% … %)}

Note: +*= and ++ are just **words** and have no special lexical significance

```
FUNCTION ( #sign #tell $machine #do ... )
    NOTE lookup $where in veps table,
       get back pattern-ish #cmd tree/bush
       then verify that our request matches on our end.
       pack and send.
    NOTE;

   NOTE turn it into an signed envelope NOTE;
   LET

      $args OPER _MSG_ tail 4 OPER;
      $host OPER _STATE_ . #host OPER;
      $name OPER _STATE_ . #name OPER;
      $root OPER _STATE_ . #root OPER;
      $prikey OPER $root . #prikey OPER;
      $treeball OPER $args treeball OPER;
      $hashed OPER $treeball sha256hash OPER;
      $signed OPER $prikey sign $hashed OPER;
      $envelope {%
        #name $name
        #host $host
        #signature $signed
        #on $args
      %}

      SEND _SELF_ #send $machine $envelope $sink SEND;
   LET;
   SEND _SELF_ #askforit SEND;
 FUNCTION;
```

The tag #sign means 16 byte utf8-word sequence

The comment expression NOTE … NOTE;

The words **lookup** and **in** and **veps** and **table,** are simply sequences of utf8-byte non-spacing sequences forming a **word**

The special symbol _MSG_ represents (#sign #tell …) from above

The word of length one utf8-sequence of value **.** has no special lexical meaning

A mapping dictionary has lexical markers **{%** to start and **%}** to end.

The Actor send message construct SEND <actor> … SEND;

The special symbol _SELF_ represents the actor within which this ‘function’ is a container element of code behaviors

Let’s have some fun!

When Fun = Learn by doing M:1 with Wizard

For {%+ #heart #money #purpose +%}, one or more pattern

Now: connect to Blaine Garst, Wizard, on LinkedIn

soon: @wizardofcoding on substack.com

I intend to be leading code reviews of TheDew.txt, team lead being recruited

I intend to be leading code review of the ART: C implementation, rewrite

New apps: Music Sharing, Smartest Home Everywhere, Ideals sharing

learn by doing, ground floor, heart first, NeXT style. (see/addtributes on LinkedIn)

Next update: HackFest, Sophia, Bulgaria 2024/11/3-4 (to be confirmed