

A Tase Of ATS

Aditya Siram

June 24, 2019

Outline

- An ML with ADTs, pattern matching, tail calls
- Can be exactly as good the C equivalent
 - Control over memory
 - Performance
- And type safe.

- Compiles to *predictable C*
 - Allows C idioms
 - malloc/free, pointers, stack control
- No compiler optimizations except TCO
 - Recursion is well supported
 - Almost no ...

- Linear/refinement types, proof level language
- Not just for memory!
- File handles, network handles
- Any resource!

- Extremely difficult
 - Syntax
 - Errors
 - Not mature!
- Mine it for the ideas!
 - Language designers, steal!
 - Users, demand!

- Previously spoke about ATS at a high level
- But I want to get into the more interesting features
- Not going to hold back!
 - Wall of code
 - Explain what it's doing
 - But not every bit of syntax

- What are linear types?
- Use once!
- Pass it to a function, consumed.

- Reading from a file
- Linear types to track file descriptors
- C FFI

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
    var f = lam@(s:string):void => println! s  
  in (  
    fwithline(a,f);  
    fclose(a);  
    fclose(b)  
  )  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
  
  in (  
  
  )  
end
```

- Tracked by the linear type system

```
datatype FileHandle = FileHandle of ()
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
    var f = lam@(s:string):void => println! s  
  in (  
    fwithline(a,f);  
    fclose(a);  
    fclose(b)  
  )  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
  
  in (  
  
    fclose(a);  
  
  )  
end
```

```
implement main0(argc,argv) =  
  let  
  
    val b = fopen("test.txt","r")  
  
  in (  
  
    fclose(b)  
  )  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
  
  in (  
  
  )  
end
```



```
fun fopen(path:string,mode:string): FileHandle =  
  let  
    extern castfn toFileHandle(p:ptr0):<> FileHandle  
  in  
    toFileHandle($extfcall(ptr0,"fopen",path,mode))  
end
```

```
fun fopen(path:string,mode:string): FileHandle =  
  let  
  
  in  
      ($extfcall(ptr0,"fopen",path,mode))  
  end
```

```
fun fopen(path:string,mode:string): FileHandle =  
  let  
      toFileHandle(p:ptr0):<> FileHandle  
  in  
      toFileHandle($extfcall(ptr0,"fopen",path,mode))  
end
```

```
fun fopen(path:string,mode:string): FileHandle =  
  let  
    extern castfn toFileHandle(p:ptr0):<> FileHandle  
  in  
    toFileHandle($extfcall(ptr0,"fopen",path,mode))  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
  
  in (  
  
  )  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
  
  in (  
  
  )  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
    var f = lam@(s:string):void => println! s  
  in (      |  
          +----- stack allocated closure!  
  
    )  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
    var f = lam@(s:string):void => println! s  
  in (  
    fwithline(a,f);  
  
  )  
end
```



```
fun fwithline(  
  fh: !FileHandle,  
  f: &(string) -> void  
) : void =  
  let
```

```
  in
```

```
end
```

```

fun fwithline(
  fh: !FileHandle,
  f: &(string) -> void
):void =
  let

```

```

    val _ = $extfcall(int,"getline",
in
    ,
    ,
    )

end

```

```

fun fwithline(
  fh: !FileHandle,
  f: &(string) -<clo1> void
):void =
let
  var len = i2sz(0)
  val lenP = addr@len

  val _ = $extfcall(int,"getline",
                    ,lenP,
                    )
in

end

```

```

fun fwithline(
  fh: !FileHandle,
  f: &(string) -<clo1> void
):void =
let
  var len = i2sz(0)
  val lenP = addr@len
  var buffer = the_null_ptr
  val bufferP = addr@buffer

  val _ = $extfcall(int,"getline",bufferP,lenP,
in

end

```

```

fun fwithline(
  fh: !FileHandle,
  f: &(string) -<clo1> void
):void =
let
  var len = i2sz(0)
  val lenP = addr@len
  var buffer = the_null_ptr
  val bufferP = addr@buffer
                toPtr{1:addr}(f: !FileHandle):<> ptr0
  val _ = $extfcall(int,"getline",bufferP,lenP,toPtr(fh))
in

end

```

```

fun fwithline(
  fh: !FileHandle,
  f: &(string) -<clo1> void
):void =
let
  var len = i2sz(0)
  val lenP = addr@len
  var buffer = the_null_ptr
  val bufferP = addr@buffer
  extern castfn toPtr{1:addr}(f: !FileHandle):<> ptr0
  val _ = $extfcall(int,"getline",bufferP,lenP,toPtr(fh))
in

end

```

```

fun fwithline(
  fh: !FileHandle,
  f: &(string) -<clo1> void
):void =
  let

    var buffer = the_null_ptr

  in
    f (
      (buffer))
  end

```

```
fun fwithline(  
  fh: !FileHandle,  
  f: &(string) -> void  
):void =  
  let  
  
    var buffer = the_null_ptr  
  
  in  
    f ($UN.castvwtp0{string}(buffer))  
end
```



```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
    var f = lam@(s:string):void => println! s  
  in (  
    fwithline(a,f);  
  
  )  
end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
    var f = lam@(s:string):void => println! s  
  in (  
    fwithline(a,f);  
    fclose(a);  
  
  )  
end
```

```
fun fclose(f:FileHandle):void =  
  let  
    extern castfn fromFH(f:FileHandle):<> ptr0  
  in  
    $extfcall(void,"fclose",fromFH(f))  
  end
```

```
implement main0(argc,argv) =  
  let  
    val a = fopen("test.txt","r")  
    val b = fopen("test.txt","r")  
    var f = lam@(s:string):void => println! s  
  in (  
    fwithline(a,f);  
    fclose(a);  
    fclose(b)  
  )  
end
```

```
fun fwithline(  
    fh: !FileHandle,  
  
    ):void =  
  
fun fclose(f: FileHandle):void =
```

- Not just for memory (any resource can be linearly tracked!)
- C FFI is very easy & encouraged
- Strong roots in C

- Building an linearly tracked array from scratch
- Datatypes + linear views
- Proofs!
 - Interleaved with term level code

```
datatype arr(a:vtflt,addr,int) =  
  | {l:addr}  
    arr_nil(a,l,0) of ()  
  | {l:addr}{n:nat}  
    arr_cons(a,l,n+1) of (a,arr(a,l+sizeof(a),n))
```



```

datatype arr(a:vtflt,addr,int) =
|
  arr_nil(      ) of ( )
|
  arr_cons(      ) of (

```

```
datatype arr(a:vtflt,addr,int) =  
  | {l:addr}  
    arr_nil(a,l,0) of ()  
  |  
    arr_cons(      ) of (      )
```

```
datatype arr(a:vtflt,addr,int) =  
  | {l:addr}  
    arr_nil(a,l,0) of ()  
  | {l:addr}{n:nat}  
    arr_cons(      ) of (      )
```

```
datatype arr(a:vtflt,addr,int) =  
  | {l:addr}  
    arr_nil(a,l,0) of ()  
  | {l:addr}{n:nat}  
    arr_cons(a,l,n+1) of (a,arr(
```

```
datatype arr(a:vtflt,addr,int) =  
  | {l:addr}  
    arr_nil(a,l,0) of ()  
  | {l:addr}{n:nat}  
    arr_cons(a,l,n+1) of (a,arr(a,          ))
```

```
datatype arr(a:vtflt,addr,int) =  
  | {l:addr}  
    arr_nil(a,l,0) of ()  
  | {l:addr}{n:nat}  
    arr_cons(a,l,n+1) of (a,arr(a,l+sizeof(a)  ))
```

```
datatype arr(a:vtflt,addr,int) =  
  | {l:addr}  
    arr_nil(a,l,0) of ()  
  | {l:addr}{n:nat}  
    arr_cons(a,l,n+1) of (a,arr(a,l+sizeof(a),n))
```

```
val a = arr_cons(string0_copy_vt("a"),  
    arr_cons(string0_copy_vt("b"),  
        arr_cons(string0_copy_vt("c"),  
            arr_cons(string0_copy_vt("d"),  
                arr_nil())))))
```

```
["a","b","c","d"]
```


- Split the array!
- **Prove** it!
- Statically split the array

```
val a = arr_cons(string0_copy_vt("a"),
                 arr_cons(string0_copy_vt("b"),
                           ...
prval (a1,a2) = array_split(a,1)
|_____ type level proof!
|_____ proof value

["a"], ["b","c","d"]
```

- Proofs erased at runtime, zero cost!

```
prfn arr_split
  {a:vtflt}
  {l:addr}
  {n:int}{i:nat | i <= n}
  (pfarr: arr(a,l,n), i:size(n)):
    @(arr(a,l,i), arr(a,l+i*sizeof(a),n-i)) =
split (pfarr) where {
  prfun split
    ...
}
```

```
prfn arr_split
```

```
(pfarr: arr(a,l,n), i:size(n)):  
  @(arr(a,l,i), arr(a,l+i*sizeof(a),n-i)) =  
  
}
```

```
prfn arr_split
```

```
{n:int}{i:nat | i <= n}  
(pfarr: arr(a,l,n), i:size(n)):  
  @(arr(a,l,i), arr(a,l+i*sizeof(a),n-i)) =  
  
}
```

```
prfn arr_split
  {a:vtflt}
  {l:addr}
  {n:int}{i:nat | i <= n}
  (pfarr: arr(a,l,n), i:size(n)):
    @(arr(a,l,i), arr(a,l+i*sizeof(a),n-i)) =

}
```

```
prfn arr_split
  {a:vtflt}
  {l:addr}
  {n:int}{i:nat | i <= n}
  (pfarr: arr(a,l,n), i:size(n)):
    @(arr(a,l,i), arr(a,l+i*sizeof(a),n-i)) =
split (pfarr) where {
  prfun split
    ...
}
```

```
prfun split
...
sif i > 0 then
  let
    prval (pfx, pfxs) = uncons(pfarr)
    prval (pfleft, pfright) = split{l+sizeof(a)}{n-1, i-1}(pfxs)
  in
    (arr_cons (pfx, pfleft), pfright)
  end
else
  let
    prval EQINT () = eqint_make{i, 0}()
  in
    (arr_nil{a}{l}(), pfarr)
  end
```

```

prfun split
  ...
  sif i > 0 then
    let
      prval (pfx, pfxs) = uncons(pfarr)
      -----|
      |
extern praxi uncons
  {a:vtflt}{l:addr}{n:nat | n > 0}
  (arr(a,l,n)):(a,arr(a,l+sizeof(a),n-1))

```



```

prfun split
  ...
  sif i > 0 then
    let
      prval (pfx, pfxs) = uncons(pfarr)
      -----|
      |
extern praxi uncons

(arr(a,l,n)):(a,arr(a,l+sizeof(a),n-1))

```

```

prfun split
  ...
  if i > 0 then
    let
      prval (pfx, pfxs) = uncons(pfarr)
      prval (pflleft, pflright) = split{..}{n-1, i-1}(pfxs)
    in
      (arr_cons (pfx, pflleft), pflright)
    end
  else
    l+sizeof(a)
  end
end

```

```
prfun split
  ...
  sif i > 0 then
```

```
  else
    let
      prval EQINT () = eqint_make{i,0}()
    in
      (arr_nil{a}{l}(), pfarr)
    end
```

```
prfun split
```

• • •

```

sif i > 0 then

```

```
dataprop EQINT(int,int) = {x:int} EQINT(x,x) |
extern prfun eqint_make{x,y:int | x == y}(): EQINT(x, y)
```

else

let

```
prval EQINT () = eqint_make{i,0}()
```

in

```
(arr_nil{a}{l}(), pfarr)
```

end