A Tase Of ATS

Aditya Siram

June 24, 2019

Outline

ATS

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

ATS

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

ATS

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

ATS

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

ATS

- 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

ATS

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

ATS

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

ATS

```
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
```

ATS

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


  in (



  )
  end
```

ATS

- Tracked by the linear type system

```
datavtype FileHandle = FileHandle of ()
```

ATS

```
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
```

ATS

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


  in (

    fclose(a);

  )
  end
```

ATS

```
implement main0(argc,argv) =
  let

    val b = fopen("test.txt","r")

  in (


    fclose(b)
  )
  end
```

ATS

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


  in (



  )
  end
```

ATS

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

ATS

```
fun fopen(path:string,mode:string): FileHandle =
  let

  in
                ($extfcall(ptr0,"fopen",path,mode))
  end
```

ATS

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

ATS

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

ATS

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


  in (



  )
  end
```

ATS

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

  in (



  )
  end
```

ATS

```
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
```

ATS

```
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
```

ATS

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




  in

  end
```

ATS

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

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

  end
```

ATS

```
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
```

ATS

```
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
```

ATS

```
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{l:addr}(f: !FileHandle):<> ptr0
    val _ = $extfcall(int,"getline",bufferP,lenP,toPtr(fh))
  in

  end
```

ATS

```
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{l:addr}(f: !FileHandle):<> ptr0
    val _ = $extfcall(int,"getline",bufferP,lenP,toPtr(fh))
  in

  end
```

ATS

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


    var buffer = the_null_ptr



  in
    f (                          (buffer))
  end
```

ATS

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


    var buffer = the_null_ptr



  in
    f ($UN.castvwtp0{string}(buffer))
  end
```

ATS

```
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
```

ATS

```
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
```

ATS

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

ATS

```
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 =
```

ATS

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

ATS

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

ATS

```
datavtype 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))
```

ATS

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

ATS

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

ATS

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

ATS

```
datavtype 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(                    ))
```

ATS

```
datavtype 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,                 ))
```

ATS

```
datavtype 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)  ))
```

ATS

```
datavtype 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))
```

ATS

```
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"]
```

ATS

- 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!

ATS

```
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
        ...
  }
```

ATS

```
prfn arr_split


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


  }
```

ATS

```
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)) =



  }
```

ATS

```
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)) =




  }
```

ATS

```
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
        ...
  }
```

ATS

```
prfun split
  ...
  sif i > 0 then
    let
      prval (pfx,pfxs) = uncons(pfarr)
      prval (pfleft,pfright) = split{..}{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
```

ATS

```
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))
```

ATS

```
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))
```

ATS

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



    end
```

ATS

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

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

ATS

```
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
```