

A (Not So Gentle) Introduction To Systems Programming In ATS

Aditya Siram

September 29, 2017

Outline

- Is an ML (not standard)
 - ADTS, pattern-matching etc.
- FP fully supported (TCO)
- Exactly the same performance/memory predictability
 - Decompiles to C
 - No optimizations (except TCO)
 - GCC does the rest
- Exactly the same control of C
 - Pointer arithmetic
 - malloc/free
 - stack allocation
- Completely verified at compile time
 - type system has zero overhead

- Top of The Benchmarks Game
- Now taken down in 2015
 - No idea why!

- Linear logic to manage resources
 - Prove it exists, consume proof, repeat
 - file handles, sockets, anything

```
fun bar
  ...
  =
  let
    val (awesome_proof | fd) = open_file("some_file.txt")
    ~~~~~
    val contents = read_file (awesome_proof | fd)
    ~~~~~
    ...
  in
    ...
  end
```

- Especially memory
 - Prove pointer is initialized, dereference, repeat
 - Type checked pointer arithmetic

- Refinement types

```
fun foo
{
    (i : int ) ...
}
```


- Refinement types

```
fun foo
{
    (i : int n) ...
}
```

- Refinement types

```
fun foo
  {n:int
    (i : int n) ...
  }
```

- Refinement types

```
fun foo
  {n:int | n > 0}
  (i : int n) ...
}
```

- Refinement types

```
fun foo
  {n:int | n > 0 && n < 10}
  (i : int n) ...
```

- Very Difficult
- Intersects
 - refinement types
 - linear logic
 - proofs
 - C
- Research!
 - Funded by the NSF
- No easy story, or newcomer "onboarding"
- Tiny community
- Sparse docs

- Easiest way to get started is C interop
- A generic swap in C
 - Yes, I realize 'size_t' is bad!

```
void swap (void* p1, void* p2, size_t size) {  
    char* buffer = (char*)malloc(sizeof(char)*size);  
    memcpy(buffer, p1, size);  
    memcpy(p1, p2, size);  
    memcpy(p2, buffer, size);  
    free(buffer);  
}
```

- A slightly non-standard swap

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
void swap(void *i, void *j, size_t size) {  
    ...  
}  
%}
```

- A slightly non-standard swap

```
%{  
    #include <stdio.h>  
    #include <stdlib.h>  
    void swap(void *i, void *j, size_t size) {  
        ...  
    }  
%}  
  
extern fun swap (i:ptr, j:ptr, s:size_t): void = "ext#swap"
```


- A slightly non-standard swap

```
%{  
    #include <stdio.h>  
    #include <stdlib.h>  
    void swap(void *i, void *j, size_t size) {  
        ...  
    }  
%}  
  
extern fun swap (i:ptr, j:ptr, s:size_t) : void = "ext#swap"  
extern fun malloc(s:size_t):ptr = "ext#malloc"
```

- Runner

```
implement main0 () =  
  let  
    val i = malloc(sizeof<int>)  
    val j = malloc(sizeof<double>)  
    val _ = swap(i,j,sizeof<double>)  
  in  
    ()  
  end
```

- Runner

```
implement main0 () =  
  let  
    val i = malloc(sizeof<int>) // all good  
  
  in  
  
  end
```

- Runner

```
implement main0 () =  
  let  
    val i = malloc(sizeof<int>)  
    val j = malloc(sizeof<double>) // uh oh!  
  
  in  
  
  end
```

- Runner

```
implement main0 () =  
  let  
    val i = malloc(sizeof<int>)  
    val j = malloc(sizeof<double>)  
    val _ = swap(i,j,sizeof<double>) // oh noes!  
  in  
  
  end
```

- Runner

```
implement main0 () =  
  let  
    val i = malloc(sizeof<int>)  
    val j = malloc(sizeof<double>)  
    val _ = swap(i,j,sizeof<double>)  
  in  
    () // free as in leak  
  end
```

- Can totally mimic C
- Including the bugs
- Gradual migration

- Safe swap

```
extern fun swap (i:ptr, j:ptr, s:size_t) : void = "ext#swap"
```


- Safe swap

```
extern fun swap
```

```
: void = "ext#swap"
```

- Safe swap

```
extern fun swap                                :      = "ext#swap"
```

- Safe swap

```
extern fun swap
```

```
= "ext#swap"
```

- Safe swap

```
extern fun swap  
  {a : t@type}
```

= "ext#swap"

- Safe swap

```
extern fun swap  
  {a : t@type}  
  {l1: addr |
```

```
}
```

= "ext#swap"

- Safe swap

```
extern fun swap  
  {a : t@type}  
  {l1: addr | l1 > null}
```

= "ext#swap"

- Safe swap

```
extern fun swap
```

```
{a : t@type}
```

```
{l1: addr | l1 > null}
```

```
{l2: addr | l2 > null}
```

= "ext#swap"

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (
    i : ptr l1
    = "ext#swap"
  ):
```


- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (
    i : ptr l1, j : ptr l2
    = "ext#swap"
  ):
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (
    i : ptr l1, j : ptr l2, s: sizeof_t a):
    = "ext#swap"
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (
    | i : ptr l1, j : ptr l2, s: sizeof_t a):
    = "ext#swap"
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1          | i : ptr l1, j : ptr l2, s: sizeof_t a):
    = "ext#swap"
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1 , a @ l2 | i : ptr l1, j : ptr l2, s: sizeof_t a):
    = "ext#swap"
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1 , a @ l2 | i : ptr l1, j : ptr l2, s: sizeof_t a):
    (                                     ) = "ext#swap"
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1 , a @ l2 | i : ptr l1, j : ptr l2, s: sizeof_t a):
    (
      void) = "ext#swap"
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1 , a @ l2 | i : ptr l1, j : ptr l2, s: sizeof_t a):
    (                               | void) = "ext#swap"
```


- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1 , a @ l2 | i : ptr l1, j : ptr l2, s: sizeof_t a):
    (a @ l1          | void) = "ext#swap"
```

- Safe swap

```
extern fun swap
  {a : t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1 , a @ l2 | i : ptr l1, j : ptr l2, s: sizeof_t a):
    (a @ l1, a @ l2 | void) = "ext#swap"
```

- Safe swap

```
extern fun malloc(s:size_t):ptr = "ext#malloc"
```

- Safe swap

```
extern fun malloc
```

```
= "ext#malloc"
```

- Safe swap

```
extern fun malloc  
    {a:t@ype}
```

```
= "ext#malloc"
```

- Safe swap

```
extern fun malloc
  {a:t@ype}
  (s:sizeof_t a):

= "ext#malloc"
```

- Safe swap

```
extern fun malloc
  {a:t@type}
  (s:sizeof_t a):
    (ptr 1)
= "ext#malloc"
```

- Safe swap

```
extern fun malloc
  {a:t@type}
  (s:sizeof_t a):
    (a? @ 1 | ptr 1)
= "ext#malloc"
```


- Safe swap

```
extern fun malloc
  {a:t@type}
  (s:sizeof_t a):
  [
    ] (a? @ 1 | ptr 1)
= "ext#malloc"
```

- Safe swap

```
extern fun malloc
  {a:t@ype}
  (s:sizeof_t a):
  [l:addr          ] (a? @ 1 | ptr 1)
= "ext#malloc"
```

- Safe swap

```
extern fun malloc
  {a:t@ype}
  (s:sizeof_t a):
  [l:addr | l > null] (a? @ l | ptr l)
= "ext#malloc"
```

- Safe swap

```
implement main0 () = let  
  val (      i) = malloc (sizeof<int>)
```

```
in
```

```
end
```

- Safe swap

```
implement main0 () = let  
  val (    | i) = malloc (sizeof<int>)
```

```
in
```

```
end
```

- Safe swap

```
implement main0 () = let  
  val (pfi | i) = malloc (sizeof<int>)
```

```
in
```

```
end
```

- Safe swap

```
implement main0 () = let  
  val (pfi | i) = malloc (sizeof<int>)  
  val (pfj | j) = malloc (sizeof<int>)
```

```
in
```

```
end
```

- Safe swap

```
implement main0 () = let  
  val (pfi | i) = malloc (sizeof<int>)  
  val (pfj | j) = malloc (sizeof<int>)  
  val          = ptr_set(      i, 1)
```

in

end

- Safe swap

```
implement main0 () = let  
  val (pfi | i) = malloc (sizeof<int>)  
  val (pfj | j) = malloc (sizeof<int>)  
  val          = ptr_set(pfi | i, 1)
```

in

end

- Safe swap

```
implement main0 () = let  
  val (pfi | i) = malloc (sizeof<int>)  
  val (pfj | j) = malloc (sizeof<int>)  
  val (      ()) = ptr_set(pfi | i, 1)
```

in

end

- Safe swap

```
implement main0 () = let  
  val (pfi | i) = malloc (sizeof<int>)  
  val (pfj | j) = malloc (sizeof<int>)  
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
```

in

end

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
```

```
in
```

```
end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val          = swap(          i, j, sizeof<int>)
in

end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val          = swap(          | i, j, sizeof<int>)
in

end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val          = swap(pfi1          | i, j, sizeof<int>)
in

end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val
      = swap(pfi1, pfj2 | i, j, sizeof<int>)
in

end
```


- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val (      ()) = swap(pfi1, pfj2 | i, j, sizeof<int>)
in

end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val (pfi2      | ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in

end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in

end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj1 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj2 | i, j, sizeof<int>)
in
  free(pfi2 | i);

end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj2 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in
  free(pfi2 | i);
  free(pfj2 | j);
end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj2 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in
  free(pfi2 | i);
  free(pfj2 | j);
  ~~~~~
end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj2 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in
  free(pfi2 | i);
  ~~~~~
  free(pfj2 | j);
end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj2 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in
  free(pfi2 | i);
  free(pfj2 | j);
end
```


- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
                        ~~~~~~
  val (pfj2 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in
  free(pfi2 | i);
  free(pfj2 | j);
end
```

- Safe swap

```
implement main0 () = let
  val (pfi | i) = malloc (sizeof<int>)
  val (pfj | j) = malloc (sizeof<int>)
  val (pfi1 | ()) = ptr_set(pfi | i, 1)
  val (pfj2 | ()) = ptr_set(pfj | j, 2)
  val (pfi2,pfj2| ()) = swap(pfi1, pfj1 | i, j, sizeof<int>)
in
  free(pfi2 | i);
  free(pfj2 | j);
end
```

- Safe swap

```
implement main0 () = let
    (pfi    ) =
    (pfj    ) =
    (pfi1    ) =      (pfi      )
    (pfj2    ) =      (pfj      )
    (pfi2,pfj2    ) =      (pfi1, pfj1      )
in
    (pfi2    );
    (pfj2    );
end
```

- Idiomatic swap

```
fun {...}  
  swap  
  {...}  
  (...) : void =  
let  
  val tmp = !p1  
in  
  !p1 := !p2;  
  !p2 := tmp  
end
```

Step back

- Step back.
- Overwhelmed?
 - I am!
- Breathe ...

Factorial

- Recursion
 - First class support!
- Allows typechecker to prove by induction!

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }
    .<n>.
    (acc : double, i : int (n)) : double =
  case- i of
  | 1 => acc
  | i when i > 1 => loop(acc * i, i - 1)

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
```

```
  let
```

```
    fun loop
```

```
  in
```

```
    loop(1.0, i)
```

```
end
```


Factorial

- Factorial

```
fun factorial
```

```
  (i : int ) :      =
```

```
  let
```

```
    fun loop
```

```
  in
```

```
    loop(1.0, i)
```

```
end
```

Factorial

- Factorial

```
fun factorial
```

```
    (i : int ) : double =  
    let  
        fun loop
```

```
in  
    loop(1.0, i)  
end
```

Factorial

- Factorial

```
fun factorial
```

```
  (i : int n) : double =  
  let  
    fun loop
```

```
  in  
    loop(1.0, i)  
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }

    (acc : double, i : int (n)) : double =

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }
    .<n>.
    (acc : double, i : int (n)) : double =

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }
    .<n>.
    (acc : double, i : int (n)) : double =
      case- i of

in
  loop(1.0, i)
end
```


Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }
    .<n>.
    (acc : double, i : int (n)) : double =
  case- i of
  | 1 => acc
  |

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }
    .<n>.
    (acc : double, i : int (n)) : double =
  case- i of
  | 1 => acc
  | i

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }
    .<n>.
    (acc : double, i : int (n)) : double =
  case- i of
  | 1 => acc
  | i when i > 1

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
  { n : int | n >= 1 }
  (i : int n) : double =
let
  fun loop
    { n : int | n >= 1 }
    .<n>.
    (acc : double, i : int (n)) : double =
  case- i of
  | 1 => acc
  | i when i > 1 => loop(acc * i, i - 1)

in
  loop(1.0, i)
end
```

Factorial

- Factorial

```
fun factorial
```

```
  let
```

```
    fun loop
```

```
      { n : int | n >= 1 } <---
```

```
      case- i of
```

```
      |
```

```
      | i when i > 1 => loop(acc * i, i - 1)
```

```
      ~~~~~
```

```
  in
```

```
    loop(1.0, i)
```

```
end
```

Factorial

- Factorial

```
fun factorial
```

```
  let
```

```
    fun loop
```

```
      { n : int | n >= 1 } <---
```

```
      case- i of
```

```
      |
```

```
      | i when i > 1 => loop(acc * i, i - 1)
```

```
      ^^^^^
```

```
  in
```

```
    loop(1.0, i)
```

```
end
```

Factorial

- Factorial

```
fun factorial
```

```
  let
```

```
    fun loop
```

```
      .<n>. <---
```

```
      case- i of
```

```
      |
```

```
      | i when i > 1 => loop(acc * i, i + 1)
```

```
      ^^^^^
```

```
  in
```

```
    loop(1.0, i)
```

```
end
```

- Viewtype
 - Connects ADTs, linear resources

- Remember 'swap'?

```
extern fun swap
  {a:t@type}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
  (a @ l1 , a @ l2 | i : ptr l1, j : ptr l2, s: sizeof_t a):
    (a @ l1, a @ l2 | void) = "ext#swap"
```

- Remember 'swap'?

```
extern fun swap
```

```
  {a:t@type}
```

```
  {l1: addr | l1 > null}
```

```
  (a @ l1          | i : ptr l1          ):
```

- Remember 'swap'?

```
extern fun swap
```

```
  {a:t@type}
```

```
  {l1: addr | l1 > null}
```

```
  (a @ l1          | i : ptr l1          ):
```

```
sortdef ...
```

```
viewtypedef ...
```

- Remember 'swap'?

```
extern fun swap
  {a:t@type}
  {l1: addr | l1 > null}
  ~~~~~
  (a @ l1          | i : ptr l1          ):

sortdef agz = {l:addr | l > null}
             ~~~~~

viewtypedef ...
```

- Remember 'swap'?

```
extern fun swap
  {a:t@type}
  {l1: addr | l1 > null}
  ~~~~~
  (a @ l1          | i : ptr l1                                ):
    ~~~~~          ~~~~~
sortdef agz = {l:addr | l > null}
             ~~~~~
viewtypedef                                     (a @ l | ptr l)
                                             ~~~~~
```

- Remember 'swap'?

```
extern fun swap
  {a:t@type}
  {l1: addr | l1 > null}

  (a @ l1          | i : ptr l1          ) :

sortdef agz = {l:addr | l > null}
  ~~~

viewtypedef          [l:agz] (a @ l | ptr l)
  ~~~~~
```

- Remember 'swap'?

```
extern fun swap
  {a:t@type}
  {l1: addr | l1 > null}

  (a @ l1          | i : ptr l1          ):

sortdef agz = {l:addr | l > null}

viewtypedef safe_ptr(a:t@type) = [l:agz] (a @ l | ptr l)
```

- Remember 'swap'?

```
extern fun swap
  {a:t@type}
  {l1: addr | l1 > null}

  (a @ l1          | i : ptr l1          ) :

sortdef agz = {l:addr | l > null}
      ^^^

viewtypedef safe_ptr(a:t@type) = [l:agz] (a @ l | ptr l)
      ^^^^^^^
```


- Remember 'swap'?

```
extern fun swap
```

```
  {a:t@type}
```

```
  {l1: addr | l1 > null}
```

```
  (a @ l1          | i : ptr l1          ):
```

- Remember 'swap'?

```
extern fun swap  
  {a:t@type}
```

```
(  
                                i : safe_ptr a                                ): 
```

- Viewtypes are the basic building block
- Can create algebras of linear resources!

- Linear lists

```
dataviewtype list_vt
  (a:viewt@type, int) =
  | list_vt_nil(a, 0) of ()
  | {n:int | n > 0}
    list_vt_cons(a, n) of (a, list_vt(a, n-1))
```

Algebraic datatypes

- Linear lists

```
dataviewtype list_vt  
  (   
    | list_vt_nil  
    |  
      list_vt_cons
```

- Linear lists

```
dataviewtype list_vt  
  (a:viewt@type      ) =  
  | list_vt_nil  
  |  
    list_vt_cons
```

- Linear lists

```
dataviewtype list_vt  
  (a:viewt@type, int) =  
  | list_vt_nil  
  |  
    list_vt_cons
```

- Linear lists

```
dataviewtype list_vt
  (a:viewt@type, int) =
  | list_vt_nil(a, 0) of ()
  |
  list_vt_cons
```


- Linear lists

```
dataviewtype list_vt
(a:viewt@type, int) =
| list_vt_nil(a, 0) of ()
|
  list_vt_cons(a, n)
```

- Linear lists

```
dataviewtype list_vt
(a:viewt@type, int) =
| list_vt_nil(a, 0) of ()
|
  list_vt_cons(a, n) of (a, list_vt(a, n-1))
```

- Linear lists

```
dataviewtype list_vt
  (a:viewt@type, int) =
  | list_vt_nil(a, 0) of ()
  | {n:int | n > 0}
    list_vt_cons(a, n) of (a, list_vt(a, n-1))
```

- Linear lists

```
list_vt_cons(1,  
  list_vt_cons(2,  
    list_vt_nil())) : list_vt(int,2)
```

Factorial

- A factorial that preserves intermediate results in a list

```
factorial(10)
```

```
=> [(10 * 9), (10 * 9 * 8), (10 * 9 * 8 * 7) ...]
```

Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n): list_vt(double, n-1) =
let
  var res : ptr
  fun loop
    ...
  val initial = g0i2f(i) * g0i2f(i-1)
  val () = loop(initial,i-2,res)
in
  res
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
```

```
  let
```

```
    var res : ptr
```

```
    fun loop
```

```
      ...
```

```
  in
```

```
    res
```

```
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
```

```
    (i:int) : int =
```

```
    let
```

```
        var res : int
```

```
        fun loop
```

```
            ...
```

```
    in
```

```
        res
```

```
    end
```


Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n):
    =
let
  var res : ptr
  fun loop
    ...

in
  res
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n): list_vt(double, n-1) =
let
  var res : ptr
  fun loop
    ...

in
  res
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n): list_vt(double, n-1) =
let
  var res : ptr
  fun loop
    ...
  val initial = g0i2f(i) * g0i2f(i-1)

in
  res
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n): list_vt(double, n-1) =
let
  var res : ptr
  fun loop
    ...
  val initial = g0i2f(i) * g0i2f(i-1)
  val () = loop(initial,i-2,res)
in
  res
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
```

```
  let
```

```
    var res : ptr
```

```
    fun loop
```

```
      ...
```

```
    val initial = ...
```

```
    val () = loop(initial,i-2,res)
```

```
  in
```

```
    res
```

```
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n):
    =
let
  var res : ptr
  fun loop
    ...
  val initial = ...
  val () = loop(initial,i-2,res)
in
  res
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n): list_vt(double, n-1) =
let
  var res : ptr
  fun loop
    ...
  val initial = ...
  val () = loop(initial,i-2,res)
in
  res
end
```

Factorial

- Factorial with intermediate results

```
fun factorial
  {n:int | n >= 2}
  (i:int n): list_vt(double, n-1) =
let
  var res : ptr
  fun loop
    ...
  val initial = g0i2f(i) * g0i2f(i-1)
  val () = loop(initial,i-2,res)
in
  res
end
```


- Inner loop

```
fun loop
  {n1:int | n1 >= 0 && n1 <= n-2}
  .<n1>.
  (
    seed: double,
    next: int n1,
    res: &ptr? >> list_vt(double, n1+1)
  ) : void = ...
```

Factorial

- Inner loop

```
fun loop
{
    .<  >.
    (

) : void = ...
```

- Inner loop

```
fun loop
{
    .< >.
    (
        next: int n1,
    ) : void = ...
}
```

Factorial

- Inner loop

```
fun loop
  {n1:int | n1 >= 0 && n1 <= n-2}
  .<n1>.
  (

    next: int n1,

  ) : void = ...
```

Factorial

- Inner loop

```
fun loop
  {n1:int | n1 >= 0 && n1 <= n-2}
    ~~~~~~

...
fun factorial {n:int | n >= 2} (i:int n)
  ...
  val () = loop(initial,i-2,res)
```

Factorial

- Inner loop

```
fun loop
  {n1:int | n1 >= 0 && n1 <= n-2}
  .<n1>.
  (

    next: int n1,

  ) : void = ...
```

- Inner loop

```
fun loop
  {n1:int | n1 >= 0 && n1 <= n-2}
  .<n1>.
  (
    seed: double,
    next: int n1,

  ) : void = ...
```

- Inner loop

```
fun loop
  {n1:int | n1 >= 0 && n1 <= n-2}
  .<n1>.
  (
    seed: double,
    next: int n1,
    res: &ptr?
  ) : void = ...
```


- Inner loop

```
fun loop
  {n1:int | n1 >= 0 && n1 <= n-2}
  .<n1>.
  (
    seed: double,
    next: int n1,
    res: &ptr? >> list_vt(double, n1+1)
  ) : void = ...
```

Factorial

- Inner loop

```
fun loop
```

```
    = ...
```

```
case- next of
```

```
  | 0 =>
```

```
  | next when next > 0 =>
```

```
    let
```

```
in
```

```
end
```

Factorial

- Inner loop

```
fun loop ( seed: double,  
          next: int n1,  
          res: &ptr? >> list_vt(double, n1+1)  
        ) : void = ...  
case- next of  
  | 0 =>  
  | next when next > 0 =>  
    let  
  
in  
end
```

Factorial

- Inner loop

```
fun loop ( seed: double,  
          next: int n1,  
          res: &ptr? >> list_vt(double, n1+1)  
        ) : void = ...  
case- next of  
  | 0 => res := list_vt_cons(seed, list_vt_nil())  
  | next when next > 0 =>  
    let  
  
    in  
  end
```

Factorial

- Solve this puzzle in a strict FP language!
- Fold over a list and copy it!
- As efficiently as a while/for loop with initial null
- That's it!

- But!
 - No reversing at the end! (1-pass only)
 - No macros!
 - No continuations!
 - No peep holing!
 - No weird optimization pragmas

Factorial

- MLTon reverses
- OCaml peep holes
- Rust (uses macros or peep holes)

- Until recently these were elegant!
- Now just dissatisfied!
- ATS supports "tail allocation"
 - A principled, safe way of passing along a "hole"

Factorial

- Inner loop

```
fun loop ( seed: double,  
          next: int n1,  
          res: &ptr? >> list_vt(double, n1+1)  
        ) : void = ...  
case- next of  
| 0 => res := list_vt_cons(seed, list_vt_nil())  
| next when next > 0 =>  
  let  
    val () = res := list_vt_cons{..}{n1+1}(seed, _)  
  
in  
end
```

Factorial

- Inner loop

```
fun loop ( seed: double,  
          next: int n1,  
          res: &ptr? >> list_vt(double, n1+1)  
        ) : void = ...  
case- next of  
| 0 =>  
|                                     =>  
  let  
    val () = res := list_vt_cons{..}{n1+1}(seed, _)  
  
in  
end
```

Factorial

- Inner loop

```
fun loop ( seed: double,
          next: int n1,
          res: &ptr? >> list_vt(double, n1+1)
        ) : void = ...
case- next of
| 0 =>
|                                     =>
    let
        val () = res := list_vt_cons{..}{n1+1}(seed, _)
                                     |
                                     an uninitialized hole-----+

in
end
```

Factorial

- Inner loop

```
fun loop ( seed: double,
          next: int n1,
          res: &ptr? >> list_vt(double, n1+1)
        ) : void = ...
case- next of
| 0 =>
|                                     =>
  let
    val () = res := list_vt_cons{..}{n1+1}(seed, _)
    val+list_vt_cons(_,hole) = res
                                |
                                |
                                +-- uninitialized hole --+

  in
  end
```

Factorial

- Inner loop

```
fun loop ( seed: double,  
          next: int n1,  
          res: &ptr? >> list_vt(double, n1+1)  
        ) : void = ...  
case- next of  
| 0 =>  
|      =>  
  let  
    val () = res := list_vt_cons{..}{n1+1}(seed, _)  
    val+list_vt_cons(_,hole) = res  
    val curr = seed * g0i2f(next)  
  
  in  
  end
```

Factorial

- Inner loop

```
fun loop ( seed: double,
          next: int n1,
          res: &ptr? >> list_vt(double, n1+1)
        ) : void = ...
case- next of
| 0 =>
|                                     =>
    let
        val () = res := list_vt_cons{..}{n1+1}(seed, _)
        val+list_vt_cons(_,hole) = res
        val curr = seed * g0i2f(next)
        val () = loop(curr, next-1, hole)
    in                                     to be filled! --+
end
```

Factorial

- Inner loop

```
fun loop
```

```

                                = ...
case- next of
|
| next
  let
    res := list_vt_cons
    |
    +-----+
    loop(      , hole)
    |
    in
    end
                                +-----+
                                |
                                |
                                |
                                |
                                |
                                |
                                +-----+
```

- ATS is rough
 - but contains glimpses of the sys. programming future!
- Linear logic
 - Great idea!
 - Need 1st class access
- Refinement types
 - Great idea!
 - Other languages are coming around to it

- Smart typechecker/dumb compiler
 - Amazing idea!
 - ATS is a frontend to C
- Haven't even talked about
 - ATS has a whole proof level language
 - Pattern matching and all!

The end

- Repo for this talk
 - <https://github.com/deech/ATS-Strange-Loop-Talk>
- Will add more links to resources.
- Thanks!