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

Aditya Siram

September 29, 2017

Outline

ATS

- 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

ATS

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

ATS

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

ATS

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

ATS

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

ATS

- Refinement types

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

ATS

- Refinement types

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

ATS

- Refinement types

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

ATS

- Refinement types

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

ATS

- Refinement types

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

ATS

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

Swap

- 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);
}
```

Swap

- A slightly non-standard swap

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

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"
```

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"
```

Swap

- Runner

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

Swap

- Runner

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


  in

  end
```

Swap

- Runner

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

  in

  end
```

Swap

- Runner

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

  end
```

Swap

- 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
```

Swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

```
extern fun swap
```

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

Swap

- Safe swap

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

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

Swap

- Safe swap

```
extern fun swap
  {a : t@ype}
  {l1: addr |                }
                                = "ext#swap"
```

Swap

- Safe swap

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

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

Swap

- Safe swap

```
extern fun swap
  {a : t@ype}
  {l1: addr | l1 > null}
  {l2: addr | l2 > null}
                                  = "ext#swap"
```

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

```
extern fun swap
  {a : t@ype}
  {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"
```

Swap

- Safe swap

```
extern fun swap
  {a : t@ype}
  {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"
```

Swap

- Safe swap

```
extern fun swap
  {a : t@ype}
  {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"
```

Swap

- Safe swap

```
extern fun swap
  {a : t@ype}
  {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"
```

Swap

- Safe swap

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

Swap

- Safe swap

```
extern fun malloc
```

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

Swap

- Safe swap

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


   = "ext#malloc"
```

Swap

- Safe swap

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

   = "ext#malloc"
```

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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

Swap

- Safe swap

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




in



end
```

Swap

- Safe swap

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




in



end
```

Swap

- Safe swap

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




in



end
```

Swap

- Safe swap

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



in


end
```

Swap

- Safe swap

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


in


end
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
```

Swap

- 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
  free(pfi2 | i);
  free(pfj2 | j);
end
```

Swap

- 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
  free(pfi2 | i);
  free(pfj2 | j);
  ^^^^^^^^^^^^^^
end
```

Swap

- 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
  free(pfi2 | i);
  ^^^^^^^^^^^^^^
  free(pfj2 | j);
end
```

Swap

- 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                                          ^^^^^^^^^^
  free(pfi2 | i);
  free(pfj2 | j);
end
```

Swap

- 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
  free(pfi2 | i);
  free(pfj2 | j);
end
```

Swap

- 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
  free(pfi2 | i);
  free(pfj2 | j);
end
```

Swap

- Safe swap

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

Swap

- 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

- Viewtype
  - Connects ADTs, linear resources

Viewtype

- Remember 'swap'?

```
extern fun swap
  {a:t@ype}
  {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"
```

Viewtype

- Remember 'swap'?

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

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

Viewtype

- Remember 'swap'?

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

  (a @ l1           | i : ptr l1                    ):

sortdef ...

viewtypedef ...
```

Viewtype

- Remember 'swap'?

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

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

viewtypedef ...
```

Viewtype

- Remember 'swap'?

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

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

viewtypedef                              (a @ l | ptr l)
                                          ^^^^^   ^^^^^
```

Viewtype

- Remember 'swap'?

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

  (a @ l1           | i : ptr l1                              ):

sortdef agz = {l:addr | l > null}
        ^^^
viewtypedef                          [l:agz] (a @ l | ptr l)
                                     ^^^^^^^
```

Viewtype

- Remember 'swap'?

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

  (a @ l1           | i : ptr l1                    ):

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

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

Viewtype

- Remember 'swap'?

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

  (a @ l1           | i : ptr l1                              ):

sortdef agz = {l:addr | l > null}
        ^^^
viewtypedef safe_ptr(a:t@ype) = [l:agz] (a @ l | ptr l)
                                 ^^^^^^^
```

Viewtype

- Remember 'swap'?

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

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

Viewtype

- Remember 'swap'?

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

  (                    i : safe_ptr a                    ):
```

Viewtypes

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

Algebraic datatypes

- Linear lists

```
dataviewtype list_vt
  (a:viewt@ype, 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
```

Algebraic datatypes

- Linear lists

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

Algebraic datatypes

- Linear lists

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

Algebraic datatypes

- Linear lists

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

Algebraic datatypes

- Linear lists

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

Algebraic datatypes

- Linear lists

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

Algebraic datatypes

- Linear lists

```
dataviewtype list_vt
  (a:viewt@ype, 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

```
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  ):                              =
  let
      var res : ptr
      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
```

Factorial

- 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 = ...
```

Factorial

- 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 = ...
```

Factorial

- Inner loop

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

    ) : void = ...
```

Factorial

- Inner loop

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

Factorial

- 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!

Factorial

- 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)

Factorial

- 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

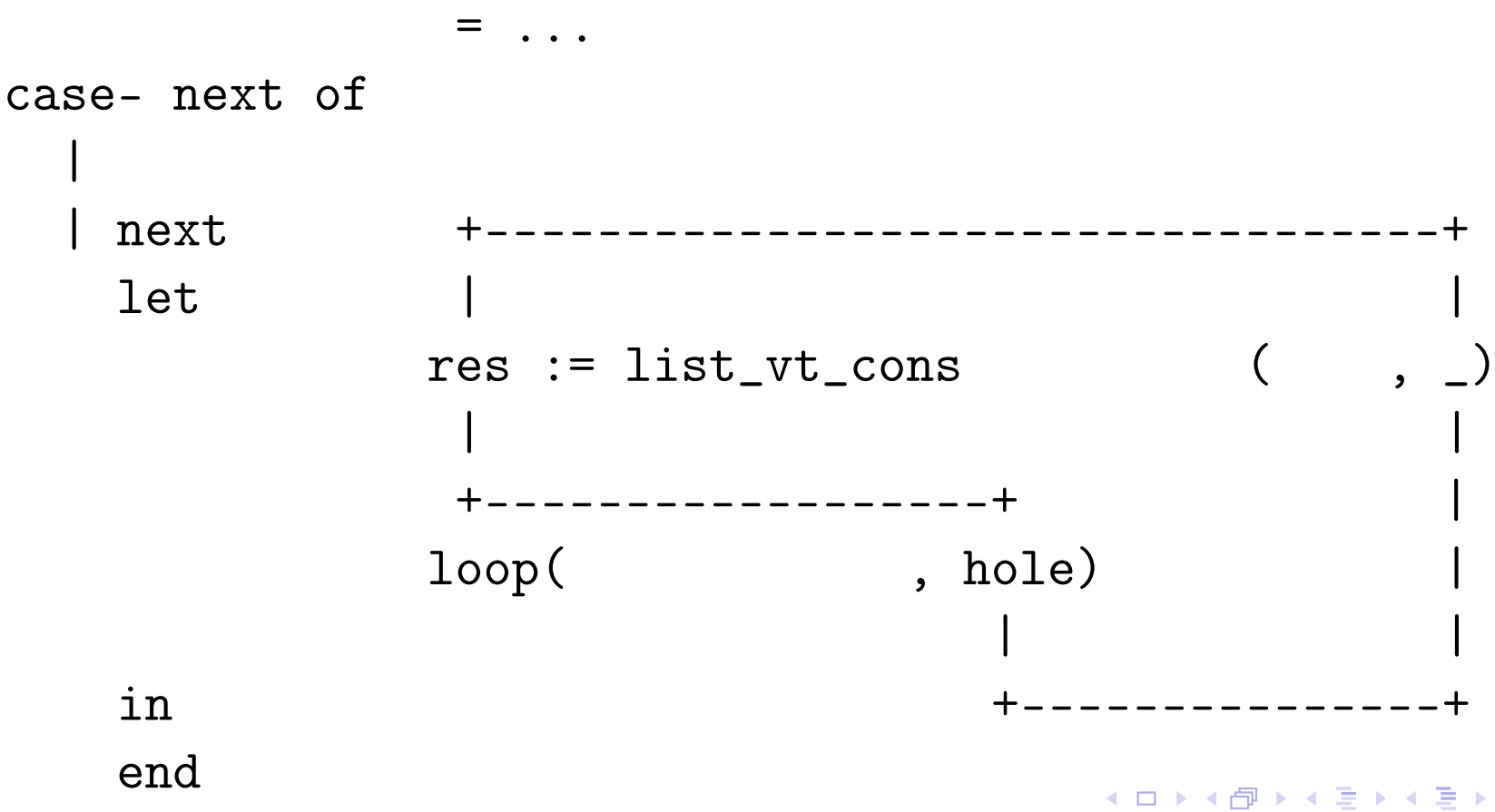
- Inner loop

```
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
```



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())
    |
```

```
    in
    end
```

Sum up

- 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

Sum up

- Smart typechecker/dumb compiler
  - Amazing idea!
  - ATS is a frontend to C
- Haven't even talked about
  - writing your own proofs
  - ATS has a whole proof level language