Why Linear Types Are The Future Of Systems Programming

Aditya Siram

February 9, 2021

Outline

Introduction

Introduction

- ATS programming language
  - ML
  - linear types
  - refinement types
  - dependant types
  - As fast as C! (“blazing fast”)
- Lots of typelevel madness
  - No optimizations
- Hongwei Xi
  - Boston University

Introduction

- Very hard!
  - Research language
  - hbox overfull with ideas
  - Tons of accidental complexity
  - Keywords everywhere ...
  - Zero docs
- And that's OK!
  - Our job to make usable things

Introduction

- Goals
    - Not evangelism!
    - Not adoption!
    - Be dissatisfied
    - Inspire your next language

Introduction

- Very difficult to present
  - Linear/dependant/refinement types, ML, C all converge
- Concrete motivating examples
  - High level handwaving
- Assuming comfort with ML like langs and basic C
- Start by taste of the ML & C side
- It'll get fairly advanced

Option Datatype

- First from the ML side

Option datatype

- A linear `Option` (explanations come later ...)

```
datavtype Option_vt (a:vt@ype, bool) =
  | Some_vt(a, true) of (a)
  | None_vt(a, false)
```

Option datatype

- probably more familiar (`_vt` for viewtype)

```
datavtype Option_vt                           =
  | Some_vt           of (a)
  | None_vt
```

Option datatype

- Indexed on a type-level `bool`, dependent types!

```
datavtype Option_vt                         =
  | Some_vt(a, true) of (a)
  | None_vt(a, false)
```

Option datatype

- Sort level `bool`

```
datavtype Option_vt                        =
  | Some_vt(a, true) of (a)
  | ...         ^^^^
```

Option datatype

- Parameterized on a view type, linear types!

```
datavtype Option_vt (a:vt@ype, bool) =
  | ...                  ^^^^^^
  | ...
```

Option datatype

- All ADTs in ATS are GADTs

```
datavtype Option_vt (a:vt@ype, bool) =
  | Some_vt(a, true) of (a)
  | None_vt(a, false)
```

Array datatype

- A linear C array

```
absvtype arrayptr (a:vt@ype, l:addr, n:int) = ptr(l)
vtypedef arrayptr (a:vt@ype, n:int) =
  [l:addr] arrayptr(a, l, n)
```

Array datatype

- Just a pointer to some address, that's it

```
                          l:addr       = ptr(l)
vtypedef arrayptr                        ^^^^^^^
...
```

Array datatype

- Parameterized on a linear viewtype & size (should be `size_t`)

```
...
vtypedef arrayptr (a:vt@ype, n:int) =
...                  ^^^^^^^^^^^^^^^^
```

Array datatype

- Returns an `arrayptr` to an *existential* (unknown) address type

```
                              l:addr         = ptr(l)
vtypedef arrayptr                   =
  [l:addr]
```

Array datatype

- Don't worry if this isn't clear
- Just a taste . . .
- Tons type level concepts to learn!
- we'll get to some later . . .

Manual Memory Management

- Now from the C side!

- What resources are leaked?

```c
int main(int argc, char** argv) {
  int* i = (int*)malloc(sizeof(int));
  *i = 10;
  FILE* fp = fopen("test.txt","r");
  return 0;
}
```

Manual Memory Management

- Memory!

```c
int main(int argc, char** argv) {
  int* i = (int*)malloc(sizeof(int)); // <--- LEAK!!
  *i = 10;
  FILE* fp = fopen("test.txt","r");
  return 0;
}
```

Manual Memory Management

- File descriptor

```
int main(int argc, char** argv) {
  int* i = (int*)malloc(sizeof(int)); // <--- LEAK!!
  *i = 10;
  FILE* fp = fopen("test.txt","r"); // <-- LEAK!!
  return 0;
}
```

Manual Memory Management

- *Equivalent* ATS program

```
implement main0 () = let
  val (pf | i) = malloc (sizeof<int>)
  val (pfset | ()) = ptr_set(pf | i, 10)
  val (pfFile | fp) = fopen("test.txt", "r")
in
  free(pfset | i);
  fclose(pfFile | fp);
end
```

Manual Memory Management

- "Client-facing" code, analogous, safe, this is why ATS is "fast"

```
implement main0 () = let
  val (      i) = malloc (sizeof<int>)
  val (         ()) = ptr_set(     i, 10)
  val (          fp) = fopen("test.txt", "r")
in
  free(         i);
  fclose(         fp);
end
```

Manual Memory Management

- `malloc` *produces* a linear proof `pf`, *consumed* by `ptr_set`

```
implement main0 () = let
  val (pf | i) = malloc (sizeof<int>)
  val (      | ()) = ptr_set(pf | i, 10)
  val (          fp) = fopen("test.txt", "r")
in
  free(        i);
  fclose(          fp);
end
```

Manual Memory Management

- `ptr_set` *produces* a proof `pfset`

```
implement main0 () = let
  val (pf | i) = malloc (sizeof<int>)
  val (pfset | ()) = ptr_set(pf | i, 10)
  val (        | fp) = fopen("test.txt", "r")
in
  free(        i);
  fclose(          fp);
end
```

Manual Memory Management

- `fopen` produces a proof of the file descriptor `pfFile`

```
implement main0 () = let
  val (pf | i) = malloc (sizeof<int>)
  val (pfset | ()) = ptr_set(pf | i, 10)
  val (pfFile | fp) = fopen("test.txt", "r")
in
  free(pfset | i);
  fclose(pfFile | fp);
end
```

Manual Memory Management

- What happens when `free` and `fopen` are deleted?

```
implement main0 () = let
  val (pf | i) = malloc (sizeof<int>)
  val (pfset | ()) = ptr_set(pf | i, 10)
  val (pfFile | fp) = fopen("test.txt", "r")
in


end
```

Manual Memory Management

- `pfset` is left unconsumed

```
implement main0 () = let
  val (pf | i) = malloc (sizeof<int>)
  val (pfset <---
  val (pfFile | fp) = fopen("test.txt", "r")
in


end
```

Manual Memory Management

- `pfFile` is left unconsumed

```
implement main0 () = let
  val (pf | i) = malloc (sizeof<int>)
  val (pfset <---
  val (pfFile <---
in



end
```

Manual Manual Management

- Consumed by `free`

```
implement main0 () = let
  ...
  val (pfset <----

in
  free(pfset | i); <---

end
```

Manual Memory Management

- Consumed by `fclose`, and that's it!

```
implement main0 () = let



  val (pfFile <---
in

  fclose(pfFile | fp); <--
end
```

Manual Memory Management

- Linear types == generalized resource tracking!
- Free to write your all your code this way!
  - safe from buffer overflows & pointer bugs
  - … there's sugar for implicitly passing proofs around
- Reuse decades of design sensibilities (safely!)
- But you're not benefitting from Functional Programming™…

Dependant & Refinement Types

- First “big” example
    - Read a number from the user between 1 and 10
    - Allocate an array of that length
    - Fill it
    - Print it to console
    - Exit
- Doesn’t seem like it but it’s a LOT

Dependant & Refinement Types

- Overall structure, types simpliifed
- Not too far from a functional program

```
fun read_input():Option_vt(a) = ...
fun make_array (len:int n): arrayptr = ...
implement main0() = begin
    println! ("Length of array? (1-10):");
    case+ read_input<int>() of
    | ~None_vt() => println! ("Not a number!")
    | ~Some_vt(len) =>
        if (len >= 1) * (len <= 10) then
          make_array(len)
        else println! ("Bad number!")
```

Dependant & Refinement Types

- Simplified `make_array` type signature

```
fun make_array (len:int n): arrayptr = ...
...
...
...
```

Dependant & Refinement Types

- Real `make_array` type signature

```
fun make_array
  {n:int| n >= 1; n <= 10}
  (len:int n): [l:addr] arrayptr(int,l,n) =
  ...
```

Dependant & Refinement Types

- `len` is indexed with a refined int *sort*, `n`.

```
fun make_array
  {n:int| n >= 1; n <= 10} <-- refines it
  (len:int n): [l:addr] arrayptr(int,l,n) =
       ^^^^^
```

- Array pointer at *some* address

```
fun make_array
  {n:int| n >= 1; n <= 10}
  (len:int n): [l:addr] arrayptr(int,l,n) =
               ^^^^^^^^^
```

Dependant & Refinement Types

- Length between 1 & 10!

```
fun make_array
  {n:int| n >= 1; n <= 10}
  (len:int n): [l:addr] arrayptr(int,l,n) =
                                       ^^^
```

Dependant & Refinement Types

- ... being called here

```
implement main0() =
    ...
    case+ ... of
    | ...
    | ...
        if (len >= 1) * (len <= 10) then
          make_array(len)
          ^^^^^^^^^^^^^^^
```

Dependant & Refinement Types

- how does it know `{n:int| n >= 1; n <= 10}`?!!

```
implement main0() =
    ...
    case+ ... of
    | ...
    | ...
        if (len >= 1) * (len <= 10) then
          make_array(len)
          ^^^^^^^^^^^^^^^
```

Dependant & Refinement Types

- It statically understands runtime checks!

```
implement main0() =
    ...
    case+ ... of
    | ...
    | ...
        if (len >= 1) * (len <= 10) then
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
          ...
```

Dependant & Refinement Types

- Runtime checks discharge proofs at compile time.

```
implement main0() =
    ...
    case+ ... of
    | ...
    | ...
        if (len >= 1) * (len <= 10) then
           ^^^^^^^^^^^^^^^^^^^^^^^^^
          ...
```

Dependant & Refinement Types

- Now anything in `make_array`'s call graph inherits the refinement

```
fun make_array
  {n:int| n >= 1; n <= 10}
  ^^^^^^^^^^^^^^^^^^^^^^^^
  (len:int n): [l:addr] arrayptr(int,l,n) =
```

Dependant & Refinement Types

- Reading user input is actually the most interesting bit
  - It interleaves basic theorem, dependent types & runtime checks!
  - The interleaving is unique to ATS to my knowledge ...

- The old `read_input`:

```
fun read_input():Option_vt(a) = ...
```

- The actual `read_input` type signature:

```
fun {a:t@ype} read_input():Option_vt(a) =
    ^^^^^^^^^                ^^^^^^^^^^^^
```

Dependant & Refinement Types

- The body:

```
let
  var result: a?
  val success = fileref_load<a> (stdin_ref,result)
in
if success then
  let prval () = opt_unsome(result)
  in Some_vt(result) end
else
  let prval () = opt_unnone(result)
  in None_vt end
end
```

Dependant & Refinement Types

- Make a *stack* variable!

```
let
  var result: a? <---


in
if success then




else




end
```

Dependant & Refinement Types

- Fill it with user input

```
let
  var result: a?
  val success = fileref_load<a> (stdin_ref,result)
in                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
if success then



else



end
```

Dependant & Refinement Types

- Stuff it into a `Some`:

```
let
  var result: a?
  val success = fileref_load<a> (stdin_ref,result)
in                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
if success then
  let prval () = opt_unsome(result)
  in Some_vt(result) end
else



end
```

Dependant & Refinement Types

- Hold up! `result` is of type a?, uninitialized

```
let
  var result: a? <----


in
if success then


  in Some_vt(result) end
else



end
```

Dependant & Refinement Types

- ...and `Option_vt(a)` needs a *not* `a?`

```
let
  var result: a? <----


in
if success then


  in Some_vt(result) <----
else




end
```

Dependant & Refinement Types

- The magic is happening with proof functions

```
let
  var result: a?
  val success = fileref_load <---
in
if success then  <---
  let prval () = opt_unsome(result) <---
  in Some_vt(result) end
else



end
```

Dependant & Refinement Types

- Interleave a proof level function, erased at runtime!

```
let
  var result: a?
  val success = fileref_load
in
if success then
  let prval () = opt_unsome(result)
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
else



end
```

Dependant & Refinement Types

- Step back and look at `fileref_load`

```
let
  var result: a?
  val success = fileref_load  <---
in
if success then



else



end
```

- The *scary* type of `fileref_load`:

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
 ...
```

Dependant & Refinement Types

- Takes a reference to `stdin`:

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
^^^^^^^^
```

Dependant & Refinement Types

- A reference (l-value) to an uninitialized stack variable:

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
          ^^^^
```

Dependant & Refinement Types

- And returns a `bool` *indexed* with `bool`!

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
                                    ^^^^^^^^^^^^^^^^^^
```

Dependant & Refinement Types

- success == `true` indexed with a static `true`.

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
                                    ^^^^^^^^^^^^^^^^^^
```

Dependant & Refinement Types

- failure == `false` indexed with a static `false`.

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
                                    ^^^^^^^^^^^^^^^^^^
```

Dependant & Refinement Types

- The linear proof is in-place transformed . . .

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
             ^^^^
```

Dependant & Refinement Types

- … into a tuple of an initialized a and static bool

```
(FILEref, &a? >> opt(a, b)) -<fun1> #[b:bool] bool(b)
                  ^^^^^^^^^^^
```

Dependant & Refinement Types

- Back to the example!

```
let
  var result: a?
  val success = fileref_load
in
if success then



else



end
```

Dependant & Refinement Types

- `success` is a `bool` indexed with a `bool`

```
let
  var result: a?
  val success = fileref_load  <---
in
if success then



else



end
```

Dependant & Refinement Types

- `result` is a now `(a,true|false)`

```
let
  var result: a? <---
  val success = fileref_load
in
if success then



else



end
```

Dependant & Refinement Types

- Now `result` is `(a,true)`!

```
let
  var result: a?
  val success = fileref_load
in
if success then <---



else



end
```

Dependant & Refinement Types

- Now look at the *proof function* `opt_unsome`

```
let
  var result: a?
  val success = fileref_load
in
if success then
  let prval () = opt_unsome(result) <---

else



end
```

Dependant & Refinement Types

- The scary proof function:

```
praxi opt_unsome{a:vt@ype}
  (x: opt(a, true) >> a):<prf> void
...
```

Dependant & Refinement Types

- It's a "proof axiom" (`praxi`)

```
praxi opt_unsome{a:vt@ype}
^^^^^
...
```

- ... essentially a proof level assertion!

```
praxi opt_unsome{a:vt@ype}
^^^^^
...
```

Dependant & Refinement Types

- In-place transforms a `opt(a,true)` into a!

```
praxi opt_unsome{a:vt@ype}
  (x: opt(a, true) >> a):<prf> void
      ^^^^^^^^^^^^^^^^^^
```

Dependant & Refinement Types

- So now `result` is a not `a?` !

```
let
  var result: a?
  val success = fileref_load
in
if success then
  let prval () = opt_unsome(result)
  in Some_vt(result) end <---
else

end
```

Dependant & Refinement Types

- `opt_unnone` does something similar!

```
let
  var result: a?
  val success = fileref_load<a> (stdin_ref,result)
in
if success then



else
  let prval () = opt_unnone(result) <--
  in None_vt end
end
```

Dependant & Refinement Types

- Everything after `fileref_load` is purely mechanical

```
let
  var result: a?
  val success = fileref_load
in
if success then                        <--
  let prval () = opt_unsome(result) <--
  in Some_vt(result) end               <--
else                                   <--
  let prval () = opt_unnone(result) <--
  in None_vt end                       <--
end
```

Dependant & Refinement Types

- Could all be synthesized!

```
let
  var result: a?
  val success = fileref_load
in
if success then                           <--
  let prval () = opt_unsome(result) <--
  in Some_vt(result) end                  <--
else                                      <--
  let prval () = opt_unnone(result) <--
  in None_vt end                          <--
end
```

Dependant & Refinement Types

- Taking stock ...
- Dependent types are cool
- Interleaved proof functions are a game changer
- And! ...

Dependant & Refinement Types

- Back to runtime checks!

```
fun read_input ... =
 let
    ...
 in
 if success then <---
 else ...
```

Dependant & Refinement Types

- Back to runtime checks!

```
implement main0() =
    ...
    case+ ... of
    | ... =>
        if (len >= 1) * (len <= 10) then
            ^^^^^^^^^^^^^^^^^^^^^^^^
```

Proof functions

- Manipulating proof terms as 1st class citizens is a game-changer
- Can statically avoid data races!
  - Given a proof of an array of length `l` and static index `i`
  - Statically split it into two proofs!
  - Give each thread a sub-proof
  - Can't access other thread's array elements!
- Emulate slices!

Proof Functions

- Proof function type signature:

```
prfun split
  {a:t@ype}
  {l:addr}{n,i:nat | i <= n}
(
  pfarr: array_v (a, l, n)
) : ( array_v (a, l, i),
      array_v (a, l+i*sizeof(a), n-i)
    )
...
```

Proof Functions

- `prfun` == proof level function

```
prfun split



...
```

Proof Functions

- Takes *proof* arguments of an array, static natural `i`

```
prfun split
  {a:t@ype}
  {l:addr}{n,i:nat | i <= n}
(
  pfarr: array_v (a, l, n)
) :



...
```

Proof Functions

- Returns *two* proofs

```
prfun split



(


) : ( array_v (a, l, i), <--
      array_v (a, l+i*sizeof(a), n-i) <--
    )
...
```

Proof Functions

- Proof of an array at `l` of length `i`

```
prfun split



(


) : ( array_v (a, l, i), <--

    )
...
```

Proof Functions

- Proof of the second section of the array!

```
prfun split



(


) : (
      array_v (a, l+i*sizeof(a), n-i) <--
    )
...
```

Proof Functions

- The body

```
sif i > 0 then let
  prval (pf1, pf2arr) = array_v_uncons pfarr
  prval (pf1res1, pf1res2) =
    split{..}{n-1,i-1} (pf2arr)
in
  (array_v_cons (pf1, pf1res1), pf1res2)
end else let
  prval EQINT () = eqint_make{i,0}((*void*))
in
  (array_v_nil (), pfarr)
end
```

Proof Functions

- There a corresponding `sif` , “static” if

```
sif i > 0 then let




in

end else let

in

end
```

Proof Functions

- Grab *proof* of the head and tail of the array

```
sif i > 0 then let
  prval (pf1, pf2arr) = array_v_uncons pfarr <--



in


end else let


in


end
```

Proof Functions

- `array_v_uncons` is a `praxi` just like `opt_unsome`!

```
praxi array_v_uncons :
{a:vt0p}{l:addr}{n:int | n > 0}
array_v (a, l, n)
  -<prf> (a @ l, array_v (a, l+sizeof(a), n-1))
```

Proof Functions

- Recurse with the proof of the tail and updated static counters

```
sif i > 0 then let
  prval (pf1, pf2arr) = ...
  prval (pf1res1, pf1res2) =
    split{..}{n-1,i-1} (pf2arr)
in  ^^^^^^^^^^^^^^^^^^^^^^^^^^^

end else let

in

end
```

Proof Functions

- Put the two sections back together!

```
sif i > 0 then let
  prval (pf1, pf2arr) = ...
  prval (pf1res1, pf1res2) =

in
  (array_v_cons (pf1, pf1res1), pf1res2) <--
end else let

in

end
```

Proof Functions

- Otherwise the first section is proof of an empty array

```
sif i > 0 then let




in


end else let


in
  (array_v_nil (), pfarr)
end
```

Proof Functions

- In a function `prval` the proofs and work in parallel!
- `thread1` and `thread2` *can not* stomp on each other!
- That's it!

```
...
prval(pf1,pf2) = split(pfarr)
thread1(pf1 | ...);
thread2(pf2 | ...);
...
```

Taking stock

- Tip of the iceberg!
- Proof functions means very customizable type environments
- Dependant types means much easier domain modeling
    - Skeptical "simple sum types" are sufficient
- Linear types means bullet-proof resource tracking

Taking stock

- All these are great ideas!
    - ATS is a great POC!
- Steadily peels back the veil
    - eg. every language designers knowns proof terms
    - but keeps them internal!
    - ATS shows we're ready for them
- *The* engineering problem is UX/DX