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

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

- Parameterized on a view type, linear types!

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

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!

Manual Memory Management

- 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!

```
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;
}
```

- *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 Memory Management

- 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™...

- 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

```
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!")
```

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

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 translate to type constraints 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) =
```