Atypical Types

v 1.0

Mark Jason Dominus

23 October 2008

Slides online at:

http://pic.blog.plove

Good AFTERNOON.

I am Mark Dominus.

Thank you for inviting me to NASHVILLE.

It is a real honor to be speaking here at OOPSLA.

Shameful confession

In the programming community, we see a lot of holy wars.

Some of these are merely matters of personal preference.

They go on forever.

For example, should one use `vi` or `emacs`?

It can be easy to forget that other arguments are eventually resolved.

For example, structured programming, or `goto`?

This one is finished now.

The bodies of the `goto` supporters are buried pretty deep.

Before that, there was a holy war about high-level languages vs. assembly language.

I caught the tail end of it when I began programming in the 1970's.

"High-level languages are inefficient," said the assembly language proponents.

And they were right.

They lost anyway.

Manual memory allocation vs. automatic garbage collection.

I didn’t expect to see this resolved as soon as it was.

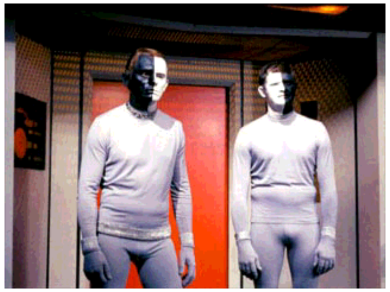
But the advent of Java ended *that* discussison.

Right or wrong, garbage collection has won.

One of these discussions that is still going on concerns strong vs. weak type systems.

C and Pascal programmers used to argue a lot about this in the 1980’s.

Which is kind of funny, since C and Pascal have almost exactly the same type system.



In 1999 ago I gave a talk on this topic.

1999 title: "Strong Typing Doesn’t Have to Suck."

(It was an audience of Perl programmers.)

For Perl programmers, any kind of automatic check is a hard sell.

Perl’s motto is "Enough rope".

I said the question was still open.

In 1999, there was no well-known static type system that did not suck.

(I discussed SML, an academic research language.)

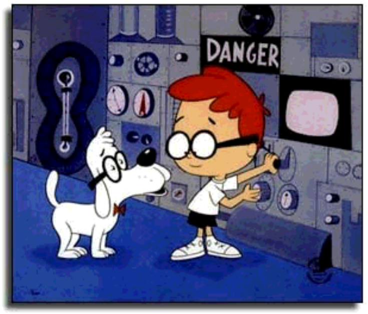
At the time, Java’s type system was a craptastic throwback to the 1970’s.

In 2008, I think Java 5.0 is a persuasive argument in favor of static typing.

Let’s look at the history a bit.

Why Types?

Sherman, set the WABAC machine for 1955!



* I think this idea first appeared in COBOL
* It's a pretty good idea anyway

Early Type Systems: FORTRAN

(This is Fortran 77, but early Fortran was similar.)

- `INTEGER`
  - `INTEGER*2`, `INTEGER*4`, `INTEGER*8`
- `LOGICAL` (Fortran jargon for ‘boolean’)
  - `LOGICAL*1` (synonym: `BYTE`), `LOGICAL*2`, `LOGICAL*4`, `LOGICAL*8`
- `REAL`
  - `REAL*4`, `REAL*8` (synonym: `DOUBLE PRECISION`), `REAL*16`
- `COMPLEX`
  - `COMPLEX*8`, `COMPLEX*16` (synonym: `DOUBLE COMPLEX`), `COMPLEX*32`

Now if you write:

```
INTEGER I
REAL R,S

R = I + S
```

then the compiler can automatically generate the correct instructions

- **Static** type checking

Early Type Systems: FORTRAN

- Side note: Declaration is optional, defaults to:
  - `INTEGER` for variables that begin with `I`, `J`, `K`, `L`, `M`, `N`
  - `REAL` for other variables
- Array types also:

```
INTEGER A(10)
```

- Functions have types:

```
FUNCTION F(X)
INTEGER F, X
F = X+1
RETURN

N = F(37)
```

- **Static** type checking
- *Expressions* have types, determined at *compile time*

Early Type Systems: Lisp

- **Dynamic** type checking
- *Values*, not expressions, are tagged with types
- Operations generate type errors at *run time*

```
(+ 1 2)
        3

(+ 1 2.0)
        3.0

(+ 1 "eels")
        Error in +: "eels" is not a number.
```


Static Typing in ALGOL-based languages

- ALGOL (1960), Pascal (1968), C (1971)
- These are all very similar
- Attempt to extend type system beyond scalars
- `array of` *type*
- `pointer to` *type* (‘reference’ in ALGOL)
- `set of` *type* (Pascal only)
- `record of` *types* (`struct` in C)
- `function returning` *type*
- And arbitrary compositions of these operations:

```
/* This is why we love C */
int *((*murgatroyd[17])(void *));
```

Typing: Hard to Get Right

- Goal: Compile-time checking of program soundness
- Pitfalls
  - False negative: Ignore real errors
  - False positive: Report spurious errors

Pascal Examples

```
var     s : array [1..10] of character;
s := 'hello';                      { You wish }

{----Thank you sir and may I have another! ----------}

type string = array [1..40] of character;
procedure error (c: string)
begin
   write('ERROR: ');
   write(c);
   writeln('');
end;

error('File not found');  { In your dreams }
error('File not found                   ');  { You have to d
error('Please just kill me Mr. Wirth         ');
```

Wirth agrees that this was a bad move.

And almost every commercial implementation of Pascal fixed this problem.

Not all these fixes were mututally compatible.

Typing: Hard to Get Right

Pascal is pretty much dead, so let's have a...

C Example

```
#include <stdio.h>

int main(void)
{
  unsigned char *c;
  float f = 10;

  for (c = (char *)&f;
       c < sizeof(float) + (char *)&f;
       c++) {
    printf("%u ",  *c);
  }
  putchar('\n');

  return 0;
}
```

```
float.c: In function `main':
float.c:9: warning: comparison of distinct pointer types lacks a
```

- The warning is spurious

C Example

- The whole program was one giant type violation
  - But the compiler didn't care

Typing in Pascal and C is a Failure

Many spurious errors

- So programmers ignore them

Proliferation of type-defeating features:

- Casts (C) `(char *)(&f)`
- Automatic conversions (C)

```
int i;
i = 1.42857;          /* Silently truncated to 1 */
```

- Variadic functions (C)
- Union types (C and Pascal both)

```
var u: case tag: integer of
         0: (intval: integer);
         1: (realval: real);
         2: (stringval: array [1..20] of character);
         3: (boolval: boolean);
       end;
    r : real;

u.intval = 1428457;
r = u.realval;          { Gack }
```

- Abuse of the separate compilation facility (Pascal)

This proliferation is a sure sign of failure

Coping With Failure

- Static typing, as implemented in C and Pascal, was not very technically successful
- Solution 1: Give up
  - Lisp
  - APL
  - AWK
  - Perl (*really* give up: `+(8/2)."".0.0.0`)

Hey, that worked pretty well!

- Solution 2: Try to do better
  - Haskell (and its precursors ISWIM, Miranda, ML, etc.)
  - Closely related: Java 5

This has *also* worked pretty well.

1999 vs. Today

- In 1999, the Haskell type system was a hard sell
- Haskell was worked on by a bunch of funny-looking ivory-tower types:

Philip Wadler
(University of Edinburgh)

Martin Odersky
(EPFL)

1999 vs. Today

Philip Wadler

Martin Odersky

- But these guys designed the Java 5 "generics" feature
- Which is directly derived from their experience with Haskell and related languages
  - Which they also designed
- The rest of this talk is about Haskell

Static Typing that Works

We saw that typing in Pascal and C failed for several reasons:

- Too fine-grained (`character[12]` vs. `character[13]`)
- Spurious warnings ⇒ ignored warnings
- Too easy to violate (unions, casts)
- Too coarse-grained (`structs`)
- Inconvenient to use (explicit types everywhere)

These problems are surmountable!

The Haskell Programming Language

- Extremely expressive and fine-grained type system
- Many fascinating and powerful features that I will not discuss today
- Originally a research language
- Solves the type problems of C and Pascal

Types in Haskell

Scalars

```
17          Integer
17.3        Float
'x'         Char
True        Bool
```

Types in Haskell

Tuples

```
(17, 'x')                      (Integer, Char)
(12.5, 13.5, 9)                (Float, Float, Int)
(True, False, True)            (Bool, Bool, Bool)
```


Types in Haskell

Lists

```
[True, False, True]              [Bool]
[True, False, True, False]       [Bool]
[1,2,3,4,5]                      [Integer]
['O', 'O', 'P', 'S', 'L', 'A']   [Char]
"OOPSLA"                         [Char]
```

- `String` is accepted as a synonym for `[Char]`
- Types like `[Integer]` this should remind you of Java types like `List<Integer>` etc.
- Just as Java has `List<List<Integer>>`, Haskell has `[[Integer]]`

```
[ [1,2,3], [4,6], [0,233] ]      [[Integer]]
[ "I", "like", "pie" ]           [[Char]]
[17, "foo"]                      Error
```

Types in Haskell

Polymorphism

```
[]                                 [a]
[ [1,2,3], [], [] ]                [[Integer]]
[ ['p', 'i', 'e'], [], [] ]        [[Char]]

([], [])                           ([a], [b])
```

(Better examples coming up shortly.)

Types in Haskell

Type composition

```
[ (True, [1, 2, 3]),
  (False, []),
  (False, [4, 5])
]                                [ (Bool, [Integer]) ]
```

Types in Haskell

Function types

```
not         Bool -> Bool
words       String -> [String]
unwords     [String] -> String

length      [a] -> Int
reverse     [a] -> [a]

head        [a] -> a
tail        [a] -> [a]
:           a -> [a] -> [a]
```

- `:` is the "cons" operation
  - `[1,2,3]` is shorthand for `1:2:3:[]`

Overloading

- *Type classes* are something like object classes in Java
- Several different types might be instances of the same class
  - This means they must support some basic set of operations
- For example, any type *t* might be an instance of the `Show` class
  - If so, there must be a function `show` of type *t* -> `String`
  - The Haskell standard library makes all the standard types instances of `Show`
  - So for example:

```
show 137        yields    "137"
show True       yields    "True"
show "Foo"      yields    "\"Foo\""
```

- If you define your own type, you can define a `show` method
  - And you can declare your type to be an instance of `Show`
- Notation:

```
Show Integer        ("Integer is an instance of Show")
Show Bool           ("Bool is an instance of Show")
Show [Char]         ("[Char] is an instance of Show")
```

Overloading

- The `show` function itself has this type:

```
(Show a) => a -> String
```

- That is, it takes an argument of type `a` and returns a `String`
  - But only if `a` is an instance of `Show`
  - The `(Show a)` is called a *context*
- The `show` function for `Bool` has type `Bool -> String`

Overloading

- Numeric operations are similarly overloaded
- The type of `+` is

```
(Num a) =>  a -> a -> a
```

- So you can add two `Integer` arguments and get another `Integer`
- Add two `Float` arguments and get another `Float`
- Define your own `Vector` type
  - Declare that it's an instance of `Num`
  - Define `+` (and `*`, etc.) operations on it
  - And then add two `Vector` arguments and get another `Vector`
  - But if you mess up and add a `Vector` to an `Integer` you'll get a compile-time error

Overloaded constants

- Constants like 163 are taken to be shorthand for

```
fromInteger 163
```

- Where `fromInteger` has type

```
(Num a) => Integer -> a
```

- So you can use "`163`" as a constant of any numeric type
  - As long as that type defines an appropriate `fromInteger` function

Overloaded constants

- In particular, this works:

```
163 + 13.5
```

- 163 gets the same type as 13.5 here
  - An appropriate value is manufactured by an appropriate version of `fromInteger`
- No nonsense like this:

```
double fahrenheit = 98.6;
double celsius1 = 5/9 * (fahrenheit - 32);
double celsius2 = (fahrenheit - 32) * 5/9;

printf("%.1f\n%.1f\n", celsius1, celsius2);

/* This is why we love C */

        0.0
        37.0
```

- A constant like `163` actually has this type:

```
(Num a) => a
```

- "Any type *a*, as long as it's an instance of `Num`."

Copyright © 1999,2008 Mark Dominus

Overloading

- Early versions of this type system had problems with equality
- What's the type of `==`?
- Something like `a -> a -> Bool`
  - *Except* that `a` must not be a function type
- Haskell solves this problem:
  - `(Eq a) => a -> a -> Bool`
  - And function types are not instances of `Eq`
- Similarly, ordered types should be declared instances of `Ord`
  - Values can be compared with <, >, etc.

Big Deal?

One big deal is that you do *not* need to declare types!

Let's consider everyone's favorite example:

```
int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

In Haskell, that looks almost the same:

```
fact 0 = 1
fact n = n * fact(n-1)
```

Hey, where did the `int`s go?

Type inference

The compiler says to itself:

```
fact 0 = 1
fact n = n * fact(n-1)
```

"`0` has type `(Num a) => a`."

Type inference

fact :: (Num *a*) => *a* -> *b*

"`0` has type `(Num a) => a`."

```
fact 0 = 1
fact n = n * fact(n-1)
```

"So *n* must have that type too."

Type inference

fact :: (Num *a*) => *a* -> *b*
n :: (Num *a*) => *a*

"`0` has type `(Num a) => a`."

"So *n* must have that type too."

```
fact 0 = 1
fact n = n * fact(n-1)
```

"`n-1` checks out okay."

Type inference

fact :: (Num *a*) => *a* -> *b*
n :: (Num *a*) => *a*

"*n* has type `(Num a) => a`."

```
fact 0 = 1
fact n = n * fact(n-1)
```

"`*` requires two arguments of the same type, both instances of `Num`."

"So `fact` must return `(Num a) => a` also."

Type inference

fact :: (Num *a*) => *a* -> *a*
n :: (Num *a*) => *a*

"`fact` must return `(Num a) => a` also."

```
fact 0 = 1
fact n = n * fact(n-1)
```

"The return value of `1` is consistent with that."

Type inference

fact :: (Num *a*) => *a* -> *a*
n :: (Num *a*) => *a*

```
fact 0 = 1
fact n = n * fact(n-1)
```

"Okay, everything checks out!"

- And if you ask it, it will *tell you* the type of `fact`:

  ```
  fact :: (Num a) => a -> a
  ```

- If you ask for the factorial of an `Integer`, you get back an `Integer`
- If you ask for the factorial of a `Float`, you get back a `Float`
- If you ask for the factorial of a `String`, you get a compile-time error
  - Because `String` is not an instance of `Num`

Haskell types are always correct

```
fact :: (Num a) => a -> a
```

- Ask the compiler to tell you the type of some function
- Is it what you expect?
  - Yes? Okay then!
  - If not, your program almost certainly has a bug.
    - A *real* bug, not a nonsense string-the-wrong-length bug

Haskell types are always correct

- When there's a type error, you do not have to groan and pull out a bunch of casts
  - Or figure out to trick the compiler into accepting it anyway
  - Instead, you stop and ask yourself "What did I screw up this time?"
  - And when you figure it out, you say "Whew! Good thing I caught that."

Type Inference Example 2

```
sumof []    = 0
sumof (h:t) = h + sumof t
```


Type Inference

```
sumof []    = 0
sumof (h:t) = h + sumof t
```

"The argument is `[]`."

"That's some kind of list, say `[a]`."

"And let's say that the return type is `b` for now."

Type Inference

sumof :: [*a*] -> *b*

"The argument has type `[a]`."

```
sumof []     = 0
sumof (h:t)  = h + sumof t
```

"`h:t` is also a list, so that's okay."

"`h` must have type *a* and `t` must have type `[a]`."

```
h :: a
t :: [a]
```


Type Inference

sumof :: [*a*] -> *b*
h :: *a*
t :: [*a*]

"`h` must have type *a* and `t` must have type `[a]`."

```
sumof []     = 0
sumof (h:t)  = h + sumof t
```

"We're adding `h` to the return value of `sumof`."

"So the return value must be *a* also."

"And `+` is only defined for instances of `Num`, so *a* is such an instance

"So the return value is really of type (Num *a*) => *a*."

```
sumof :: (Num a) => [a] -> a
```

Type Inference

sumof :: (Num *a*) => [*a*] -> *a*
h :: (Num *a*) => *a*
t :: (Num *a*) => [*a*]

"So the return value is really (Num *a*) => *a*."

```
sumof []    = 0
sumof (h:t) = h + sumof t
```

"That fits with the other return value of 0."

"And everything else checks out okay."

- If you ask, it will say that the type is:

  `sumof :: (Num a) => [a] -> a`

- If we had put `0.0` instead of 0, it would have deduced:

  `sumof :: (Fractional a) => [a] -> a`

- (`Fractional` is a subclass of `Num`)
  - Among other things, it supports division
- If we had put `"Fred"` we would have gotten a type error
  - Because `String` is not an instance of `Num`

Type Inference Example 3

```
map(f, [])  = []
map(f, h:t) = f(h) : map(f, t)
```

Type Inference

```
map(f, []) = []
map(f, h:t) = f(h) : map(f, t)
```

"`f` has some type, say p, and `[]` has some list type, say `[`a`]`."

Type Inference

map :: (p, [a]) -> q
f :: p

"`[]` has some list type, say `[`a`]`."

```
map(f, []) = []
map(f, h:t) = f(h) : map(f, t)
```

"`h` must have type a and `t` must have type `[`a`]`."

Type Inference

map :: (*p*, [*a*]) -> *q*
f :: *p*
h :: *a*
t :: [*a*]

"`h` must have type *a*."

```
map(f, []) = []
map(f, h:t) = f(h) : map(f, t)
```

"`f` is used as a function applied to `h`."

"So `f` must have type `a -> b` for some *b*."

"`f` must take an argument of type *a* and return a result of type *b*."

Type Inference

map :: (*a* -> *b*, [*a*]) -> *q*
f :: *a* -> *b*
h :: *a*
t :: [*a*]

"`f` must take an argument of type *a* and return a result of type *b*."

```
map(f, []) = []
map(f, h:t) = f(h) : map(f, t)
```

"The result of `f` is consed to the result of `map`."

"So `map` must return `[b]`."

Type Inference

map :: (a -> b, [a]) -> [b]
f :: a -> b
h :: a
t :: [a]

"`map` must return `[b]`."

```
map(f, [])  =  []
map(f, h:t) = f(h) : [map(f, t)
```

"That fits with the return value in the other clause."

"Everything else checks out okay."

- If you ask the compiler, it will say that the type is:

```
map :: (a -> b, [a]) -> [b]
```


Type Inference Example 3 Continued

```
map :: (a -> b, [a]) -> [b]
```

Normally `map` is defined as a *curried* function

Instead of this:

```
map(f, []) = []
map(f, h:t) = f(h) : map(f, t)
```

We write this:

```
map f []    = []
map f (h:t) = f(h) : map f t
```

And the type is:

```
map :: (a -> b) -> [a] -> [b]
```

Then for example:

```
length :: [a] -> Integer
map length ["I", "like", "pie"]
    [1, 4, 3]

length_all = map length

length_all :: [[a]] -> [Integer]
length_all ["I", "like", "pie"]
    [1, 4, 3]
```

Life with Haskell

The Haskell type system has a lot of unspectacular successes.

Programming in Haskell is pleasant

- No type declarations—everything is automatic
- You get quite a few type errors (darn!)
- But *every error* indicates a real, *serious* problem
- Not like `lint` or C or Pascal.

A Spectacular Example

Here's a *spectacular* example, due to Andrew R. Koenig.

We will write a merge sort function.

Strategy:

- Split list into two lists
- Sort each list separately
- Merge sorted lists together

We expect the type of this function to be

```
(Ord a) => [a] -> [a]
```

Splitting

```
split [] = ([], [])
split [a] = ([a], [])
split (a:b:rest) = (a:a', b:b')
    where (a', b') = split rest


split :: [t] -> ([t], [t])
```


Merging

```
merge [] ls = ls
merge ls [] = ls
merge (a:as) (b:bs) =
  if a <= b then a : merge as (b:bs)
  else           b : merge (a:as) bs

merge :: (Ord t) => [t] -> [t] -> [t]
```

Merge Sort

```
sort [] = []
sort ls = merge (sort p) (sort q)
  where (p, q) = split ls
```

- If we ask Haskell for the type of `sort`, it says:

```
sort :: (Ord a) => [t] -> [a]
```

Huh??

Huh??

```
sort :: (Ord a) => [t] -> [a]
```

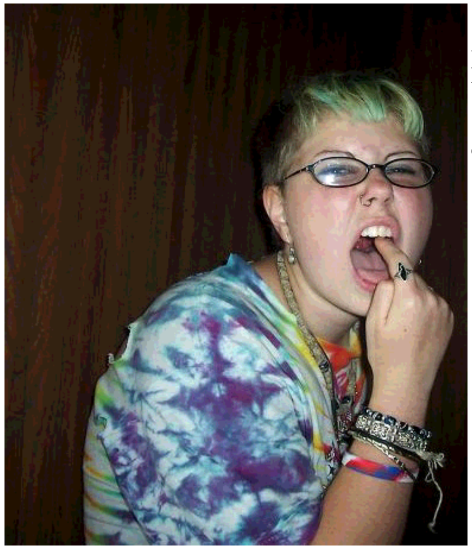
- This says that we could put in *any* kind of list `[t]`
  - It does not even have to be ordered
- And what we get out has nothing to do with what we put in
  - We could put in a list of `Integer` and get out a list of `String`
    - Which is impossible

Huh??

```
sort :: (Ord a) => [t] -> [a]
```

- But this is *impossible*

One way the impossible can occur is if it never can occur



“Go out with you? Sure, when Arnold Schwarzenegger is elected president.”

“But he isn't an American citizen.”

“Right!”

Huh???

```
sort :: (Ord a) => [t] -> [a]
```

“Given a list of numbers, it could return a list of strings.”

“But it can’t possibly return a list of strings.”

“Right!”


```
sort [] = []
sort ls = merge (sort p) (sort q)
  where (p, q) = split ls
```

In fact, this function has a bug.

- It never returns
  - (Except when the input is empty.)
  - (In which case it *does* return a list of type `[a]`)
- *Type checking* found an infinite loop bug!
- At compile time!!
- !!!!!!!!!

Where's the Bug?

```
sort [] = []
sort ls = merge (sort p) (sort q)
  where (p, q) = split ls
```

Suppose the function is trying to sort a one-element list `[x]`

It calls `split` and gets `([x], [])`

Then it tries to recursively sort the two parts

Sorting `[]` is okay.

Sorting `[x]` puts it into an infinite loop

Solution: Add a clause

```
sort [] = []
sort [x] = [x]
sort ls = merge (sort p) (sort q)
  where (p, q) = split ls
```

The type is now:

```
sort :: (Ord a) => [a] -> [a]
```

as we expected it should be.

Summary

Thank you!

They say to allot 3–5 minutes per slide

So I won’t pretend that there will be time for questions

(sorry)

Please email me or catch me in the hallway

http://pic.blog.plov

mjd@plover.com

Thank you!

They say to allot 3–5 minutes per slide

So I won't pretend that there will be time for questions

(sorry)

Please email me or catch me in the hallway

http://pic.blog.plov

mjd@plover.com

Solution: Add a clause

```
sort [] = []
sort [x] = [x]
sort ls = merge (sort p) (sort q)
  where (p, q) = split ls
```

The type is now:

```
sort :: (Ord a) => [a] -> [a]
```

as we expected it should be.

Summary

Thank you!

They say to allot 3–5 minutes per slide

So I won't pretend that there will be time for questions

(sorry)

Please email me or catch me in the hallway

http://pic.blog.plov

mjd@plover.com

Thank you!

They say to allot 3–5 minutes per slide

So I won't pretend that there will be time for questions

(sorry)

Please email me or catch me in the hallway

http://pic.blog.plov

mjd@plover.com