Summer Internship 2016

Introduction to Types

Zach Schmidt

June 28, 2016

IMAGETREND®

A *type* is a classification identifying one of various types of data that determines:

- the possible values for that type
- the operations that can be done on values of that type
- the meaning of the data
- the way values of that type can be stored

Intuitively, we can view this as a *has-a/is-a* relationship:

- A value has-a type
- int/string/float/etc. is-a type
- A type has-a kind
- A kind has-a sort

$\left.\begin{array}{l}\text{A type has-a kind}\\ \text{A kind has-a sort}\end{array}\right\}$ Only of academic interest (currently)

The simply typed lambda calculus (STLC) is the easiest way to see types in action

Let B be the set of all types T (that is to say, T is a type iff $T \in B$). We can now express the production rules inductively as:

$$\tau ::= \tau \to \tau \mid T$$
$$e ::= x \mid \lambda x : \tau . e \mid e\ e \mid c$$

where x is some variable of type τ and c is a constant

This looks gross, but it's really simply – the application $(\lambda y.y)x$ becomes x (i.e. you just replace all occurrences of the variable y inside the abstraction with x)

We can see now that the type of the abstraction $(\lambda x.x)$ is $\tau \to \tau$

Given anything of type τ, this abstraction returns something of type τ (the identity function!)

Recap: How many functions are there with type $\tau \to \tau$?

JavaScript is the perfect substitute for lambda calculus, and makes it easy to see how types (could) work

Let τ be the set of all JavaScript types $\{\texttt{null}, \texttt{undefined}, \texttt{boolean}, \texttt{string}, \texttt{number}, \texttt{object}\}$

We can now see (hopefully) that the 'type' of the 'function' $x => x$ is $\tau \to \tau$

Given anything of type τ, this function returns something of type τ (it's the identity function!)

Recap: How many functions are there with type $T \to T$, where $T \in \tau$?

IMAGETREND®

Recall *Modus Ponens*: If all Tuesdays are sunny and today is Tuesday, it is sunny

More formally, this can be written:

$$\frac{A \to B \qquad A}{B}$$

What if we have some variable a of type A and some function $A \to B$?

Well... we can conclude that after an application of the function we will be guaranteed to have something of type B... we now have *modus ponens*!

It turns out that the *Curry-Howard Isomorphism* says that there is a direct relationship between computer programs and mathematical proofs

This isomorphism blurs the lines between mathematical logic and computer science

The ability to view a program as a mathematical proof allows one to *literally* prove their programs will work as expected

This can be contrasted with the (infinitely) less formal method of QA by various inputs

I can hear the argument *"this is cute and all, but it has no purpose in the 'real world' "*

If you like your nukes to only explode when you want them to, you'd better verify the program!

The driverless control system for the Paris metro was proven correct using an automated theorem prover[1]

Many DOE[2]/DARPA[3]/NASA[4] projects need to be formally verified to meet certification standards

[1] `http://www.prover.com/company/press/view/?id=47`
[2] `http://energy.gov/sites/prod/files/2013/06/f1/O-425-1D_ssm.pdf`
[3] `https://dl.acm.org/citation.cfm?id=581775`
[4] `https://www.cs.utexas.edu/users/ObjectCheck/pubs/FASE2001.pdf`

Typed vs. Untyped

Typed	The language distinguishes types
Untyped	Everything is simply a stream of bits (assembly)

Static vs. Dynamic

Static	The language enforces types before run-time (typically at compile time)
Dynamic	The type is determined at run-time

Weak vs. Strong

Weak	Allows one type to be treated as another (e.g. a string can be seen as a number)
Strong	Prevents the above; an attempt to perform an operation on the wrong type raises an error

C# 3.0 included the ability to let the compiler provide the (explicit) type during compilation.

This is just *syntactic sugar*, and has no effect on performance.

Disclaimer

Implicit typing will *never* introduce an error by itself, but it makes it *exceedingly easy* to have a typo result in undefined behavior!

When looking at the release history of C#, implicit typing appeared at the same time as query expressions, lambda expressions, and anonymous types.

The first two can be seen as special cases of the third:

```
var empEnumerable = from emp in employees
                       where emp.age > 50
                       select emp;

var filterEnumerable = empEnumerable.Where(e => e.age > 50);

var employee = new { name = "testName", age = 30 }
```

Due to the lack of *true* type inference, C# very quickly gives up when writing many lambda expressions.

For example, the most basic of all lambda expressions cannot be implicitly typed:

```
//Error due to unknown RHS
var id = x => x;
```

Return types from functions – provides looser coupling, relies on perfectly named functions:

```
//Nobody knows what type this is
var returnVar = getStuff();

//Loose coupling achieved
var employee = getEmployeeById();
```

Additionally, anonymous implicitly typed (primitive) lists should be avoided, for reasons that will be made clear on the next slide.

When the variable is a primitive, never (!!) use an implicit type:

- `var myFloat = 100;`
- `var saleAmount = 100.0;`
- `var enterpriseGUID = ‘ ’;`

JavaScript is a typed[5], dynamically, weakly[6] typed language

The above means that JavaScript is (perhaps) the best golfing language around... where else could you do the following (?!):

```
var a = "30";
var b = "40";
var c = a- -b;
```

Recap: What is `c`?

[5]Want to read arguments all day?
`https://stackoverflow.com/questions/964910/is-javascript-an-untyped-language`

[6]This is not strictly true, but implicit casting is rampant

JavaScript *isn't* the worlds best language at preventing run-time errors...

Anyone who has programmed more than 100 lines of JS knows that

Unfortunately, when JS encounters a run-time error, it frequently fails silently[7]

I have tons of ideas for better JavaScript error handling[8], but time is an issue...

[7] `https://opensourcehacker.com/2008/08/19/catching-silent-javascript-exceptions-with-a-function-decorator/`

[8] In the meantime, this is the best source: `https://developer.chrome.com/devtools`

TypeScript is a strict superset of JavaScript
(i.e. every JavaScript file is a valid TypeScript file)

Allows type annotations

Compiles down to normal JavaScript
erases types... becomes default JavaScript runtime type checks

Developed by the same fella who was the lead architect for C#

TypeScript by default allows you to catch 'silly' mistakes

```
var name: string = "bob";
function warnUser(): void {
    alert("This is my warning message");
}
interface ClockInterface {
    currentTime: Date;
}
class Clock implements ClockInterface  {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

TypeScript is intentionally *unsound*, because all types are erased after compilation

Recall that a *sound* type system is one that rejects all "bad" programs

> *"typeful programs may be easier to write and maintain, but their type annotations do not prevent runtime errors"* [9]

[9] `https://www.cs.umd.edu/~aseem/safets.pdf`

```
class Animal {}
class Snake extends Animal {
    slither() {
        alert("Slithering!")
    }
}
class Horse extends Animal {}

var a: Snake[] = [new Snake()];
var b: Animal[] = a;
b[0] = new Horse();
var notASnake: Snake = a[0];
notASnake.slither();
```

```
Uncaught TypeError:  notASnake.slither is not a function
```

IMAGETREND®

This leads to some sad programming practices when using TypeScript

```
function f(s:string):number{
  if(typeof s !== "string")
    return -1;
...
}
```

The programmer must still check that the argument is indeed a `string`

To prevent this additional burden on the programmer that run-time checks impose, Microsoft research created the “Safe TypeScript” package[10]

When the `--safe` flag is added to the TypeScript compiler, it performs two passes over the code:

1. The normal static type checking is performed (vanilla TypeScript)
2. Dynamic type checking is added via runtime checks based on RTTI (runtime type information)... Safe TypeScript!

[10] Available here: `https://github.com/nikswamy/SafeTypeScript`

When the `--safe` flag is used, TypeScript becomes type safe in the following regards[11]:

- copying instances of classes via `checkAndTag()`
- accessing members of classes via `readField()`
- altering members of classes via `writeField()`

[11] There are a host of other benefits/features, I discuss these in a more advanced talk here: `https://goo.gl/QKXqVe`

Using the `--safe` flag *does* incur a slowdown[12] of 15%

It's therefore up to the designer to determine whether they are willing to trade performance for type safety

[12] There is another method of dynamic type checking called RTTI-on-creation which has a slowdown of 1.4 - 3x

Who are types for? The editor? The compiler? You?

If you dig into type theory, this question comes up often – should we use types to increase code readability, allow proofs of correctness, better encapsulate business logic?

I *love* to discuss the philosophical implications of types...

If you were *really* paying attention, you might be asking *"where does first order logic fall into the Curry-Howard Isomorphism?"*

If we want to say $\forall A, A \rightarrow \tau$, we need to be able to vary over all types A... this is polymorphism!

What about the existential quantifier? That's encapsulation! $\exists A, A \rightarrow \tau$ means that there is some type A (that we don't care about) which will produce some τ[13]

[13] `https://www.cs.cornell.edu/courses/cs6110/2013sp/lectures/lec37-sp13.pdf`

Recall the CH-Isomorphism stated that we can treat proofs as programs... but this said nothing of the value

That is, we can say that we know some program will produce an integer, but it does not allow us to say it will produce a *valid* integer

There has been an (academic) effort[14] to produce a variant of JavaScript which is *dependently typed*, which means that we could say a given program will produce an integer between x and y, for example

[14] `http://goto.ucsd.edu/~ravi/research/oopsla12-djs.pdf`

I mentioned that C# doesn't have type inference...

Most languages which have type inference use the Hindley-Milner type system:

> *Among HM's more notable properties is completeness and its ability to deduce the most general type of a given program without the need of any type annotations or other hints supplied by the programmer*

This is typically implemented using *Algorithm W* (but it's usually only found in functional languages)

Discussion regarding formal verification:
`http://c2.com/cgi/wiki?ProofsCantProveTheAbsenceOfBugs`

Free book on languages:
`http://cs.brown.edu/courses/cs173/2012/book/book.pdf`

Really good StackOverflow answer about value/type/kind:
`http://programmers.stackexchange.com/a/255928`

Off the deep end with the Curry-Howard Isomorphism: `http://disi.unitn.it/~bernardi/RSISE11/Papers/curry-howard.pdf`

Intro to Type Theory: `http://www.cs.ru.nl/~herman/onderwijs/provingwithCA/paper-lncs.pdf`

TypeScript Tutorial: `https://www.typescriptlang.org/docs/tutorial.html`