Dynamic Typing in D

Adam D. Ruppe

adam.ruppe@gmail.com

(Shiver me timbers! There be slides this year!)

File written: May 29, 2015, 9:38 AM, MDT

What are types?

In the beginning, Computer created the registers and the RAM.

And the memory was without types, yea, even `void*`.

```
mov EAX, 65;
// Does EAX hold cast(char) 'A'?
// Or cast(int) 65?
// Or cast(MyStruct*) 0x41?
// Nobody knows. And the hardware doesn't care.
```

And Computer said, Let there be types: and there were types.

And Computer saw the types, that they were good: and Computer divided the compile-time from the run-time.

(Genesis 1:1-4)

Types of Typing

(A Great Apostasy)

not to scale

Static (Compile-time checked)

C

C++, Haskell

Weak (Implicitly Coerced)

Strong (Mismatches as errors)

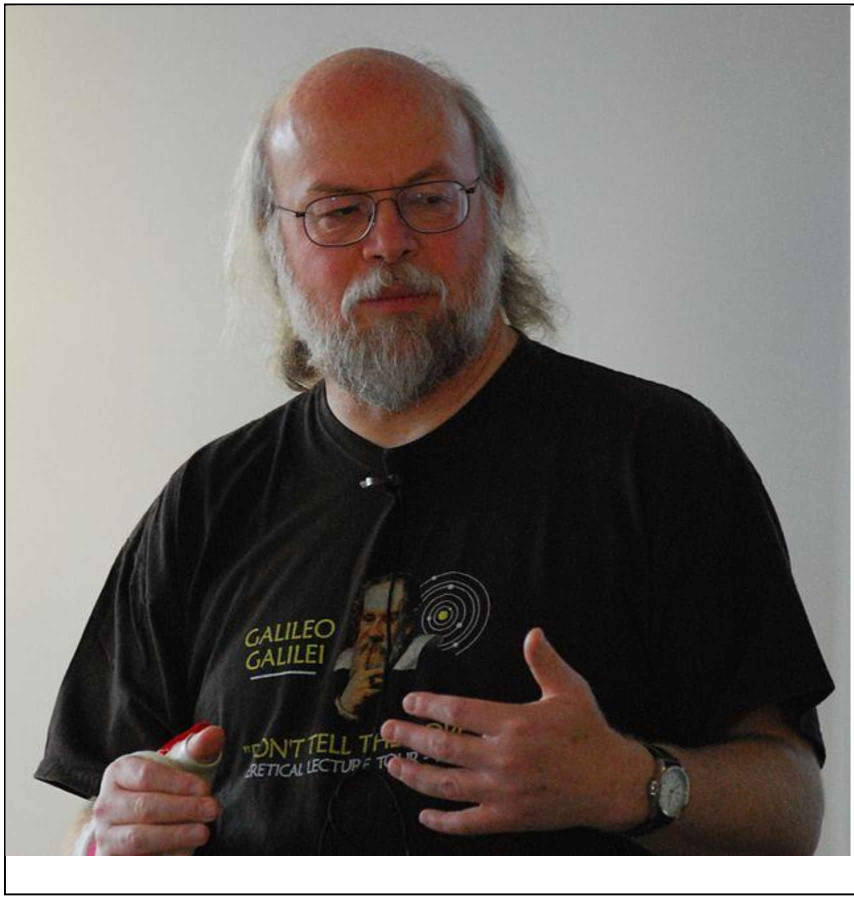
Javascript, PHP

Ruby, Python, Java (sort of)

Dynamic (runtime tagged)

Raw memory is untyped and also not quite coerced; it is reinterpreted which is a bit different.

LOL GENERIC PROGRAMMING CONCEPTS ABOVE


GALILEO
GALILEI

Er... wrong James...

Restored Typing

- Static types on variables
- Strongly checked at compile time
- Inferred types (aka `auto`)
- Templated types and functions
- `static assert` for more checks
- Ongoing revelation of the fullness of Computer's plan of happy programming

Implicitly coerced, run-time-tagged typing in D

Adam D. Ruppe

Is this D?!

```
// this is valid D code!
                var a = 10;
                var b = "20";
                var c = a + b;
                var d = json!q{ "foo": { "bar": 10.2 } };
                writeln(d.foo); // {"bar":10.2}
                d.foo.bar = (var a) => a ~ b;
                writeln(d.foo.bar()("hello! "));
```

What does D need with a dynamic type?

- External APIs (database, web)
- Runtime `interfaces`
- Prototyping
- Interacting with ~~apostate~~ scripting languages
- Showing off because we can (learn the language, use the language)
- **the same true language that does kernel can do this too**

Fullness of the type system

- std.variant : Variant, Algebriac
- jsvar
- Plenty in-between

Basic technique: tagged union

```
struct MyType {
        enum Holding {Int, String}
        Holding type;

        union {
                int Int;
                string String;
        }
}
```

std.variant: open-ended tagged union

```
TypeInfo type;
union {
        void* data;
        ubyte[MAX_SIZE] small_type_optimization;
}
```

COFFEE

Sugary treats

- Cookies
- Cake
- Pie
- Brownies
- Chocolate

They all go well with milk!

Syntax sugar makes it usable

- Operator overloading
- Constructors
- `opDispatch`
- Parenthesis-less function calling
- `q{ string literals }`
- Templates and `single_template!arg`
- CT Reflection
- `var` isn't a keyword :)

Operator Overloading

```
public var opBinary(string op, T)(T t) {
        var n;
        if(payloadType() == Type.Object) {
                var* operator = this._payload._object._peekMember("opBinary", tr
                if(operator !is null && operator._type == Type.Function) {
                        return operator.call(this, op, t);
                }
        }
        return _op!(n, this, op, T)(t);
}

public var opBinaryRight(string op, T)(T s) {
        return var(s).opBinary!op(this);
}
```

Implementation of operators

- Check types and act based on them
- Check for type families with std.traits - isIntegral, isFloatingPoint, etc.
- Strong Open-endedness can be done with generated functions for given type

```
Variant opBinary(string op)(T rhs) { return Variant(mixin("this.get!T" ~ op ~ "rhs));
```

- `std.conv.to` rox for conversions in weakly typed dynamics
- string mixins help generate those functions

CT Reflection handles advanced cases

```
} else static if(isCallable!T) {
        this._type = Type.Function;
        static if(is(T == typeof(this._payload._function))) {
                this._payload._function = t;
        } else
        this._payload._function = delegate var(var _this, var[] args) {
                var ret;

                ParameterTypeTuple!T fargs;
                foreach(idx, a; fargs) {
                        if(idx == args.length)
                                break;
                        cast(Unqual!(typeof(a))) fargs[idx] = args[idx].get!(typ
                }

                static if(is(ReturnType!t == void)) {
                        t(fargs);
                } else {
                        ret = t(fargs);
                }
```

More reflection

```
// and also wrapped native classes, automatically
WrappedNativeObject wrapNativeObject(Class)(Class obj) if(is(Class == class)) {
        return new class WrappedNativeObject {
                override Object getObject() {
                        return obj;
                }

                this() {
                        wrappedType = typeid(obj);
                        // wrap the other methods
                        // and wrap members as scriptable properties

                        foreach(memberName; __traits(allMembers, Class)) {
                                static if(is(typeof(__traits(getMember, obj, mem
                                static if(is(typeof(__traits(getMember, obj, mem
                                        static if(is(type == function)) {
                                                _properties[memberName] = &__tra
                                        } else {
                                                // if it has a type but is not a
                                                _properties[memberName] = new Pr
                                                        () => var(__traits(getMe
```

opDispatch

You can convert foo.bar to foo["bar"] to punt it to runtime

```
var[string] properties;
var opDispatch(string member)() { return properties[member]; }
```

dangers of delegates in structs and using a static nested function to capture specific variables

```
        else static if(isDelegate!T) {
                // making a local copy because otherwise the delegate might refer to a s
                auto func = this._payload._function;

                // the static helper lets me pass specific variables to the closure
                static T helper(typeof(func) func) {
                        return delegate ReturnType!T (ParameterTypeTuple!T args) {
                                var[] arr;
                                foreach(arg; args)
                                        arr ~= var(arg);
                                var ret = func(var(null), arr);
                                static if(is(ReturnType!T == void))
                                        return;
                                else
                                        return ret.get!(ReturnType!T);
                        };
                }

                return helper(func);
```

Bonus Technique!!!

```
ref var thing() { return *( new var(null) ); }
```

This is garbage. But it works!

See also

- delegate pattern matching
- TypeTuple CT/RT bridge

Contrast my usage of reflection with the protocol generation use - this is kinda needed here, can't be reasonably done ahead of time. We take a compile time hit, but it enables new stuff.

Static types are great for generation; none of this dynamic niceness would be really possible without it! Also rox for form generation etc btw.

```
        class CastExpression : Expression {
                string type;
                Expression e1;

                override string toString() {
                        return "cast(" ~ type ~ ") " ~ e1.toString();
                }

                override InterpretResult interpret(PrototypeObject sc) {
                        var n = e1.interpret(sc).value;
                        foreach(possibleType; CtList!("int", "long", "float", "double",
                                if(type == possibleType)
                                        n = mixin("cast(" ~ possibleType ~ ") n");
                        }

                        return InterpretResult(n, sc);
                }
        }
<
```

What's missing

- Implicit constructors for func calls
- Implicit casts back to static types
- Multiple alias this(?)
- @property on the edge case of returning delegate

Implicit construction

Regular struct cons is explicit: `SName(some_arg)`.

```
void func(var a) { }

func(null); // can this implicitly make func(var(null)?)
func(10); // func(var(10)) implicitly?
```

C++ can do this. D sucks.

Useful outside dynamic types: what about library array replacements taking null? BigInt taking int?

Will it mess up overloading?

Is this wise?

Use this sparingly, so saith the Computer. Even laziness isn't a good justification here!

Implicit construction today

```
void func(var a) {}
dycall!func(null); // dycall template wraps args
```

Doable, but not quite a drop-in replacement for language built-ins

d rox

Implicit casts back

```
var v = 10;
int a = v;
```

C++ can do this. D sucks.

no d rox

```
var v = 10;
auto a = v.get!int;
```

Whereas we are supposed to use this sparingly, I think this is nice. auto rox enough, explicit movement back is good.

@property needs to work

```
Callable prop() {}
prop(); // should call Callable
```

Please don't blab able optional parens, this is all I care about, leave the rest the same.

Let's use this.

```
var globals = var.emptyObject;
globals.loadJsonFile = delegate var(string name) {
        import std.file;
        return var.fromJson(readText(name));
};
globals.saveJsonFile = delegate var(string name, var obj) {
        import std.file;
        write(name, obj.toJson());
        return obj;
};

// wrapping my http2.d was easy too!
globals["get"] = delegate var(string path) {
        auto request = client.navigateTo(Uri(path));
        request.send();
        return var(request.waitForCompletion());
};



writeln(interpret(line, globals));
```

d rox

ask me stuff

adam.ruppe@gmail.com