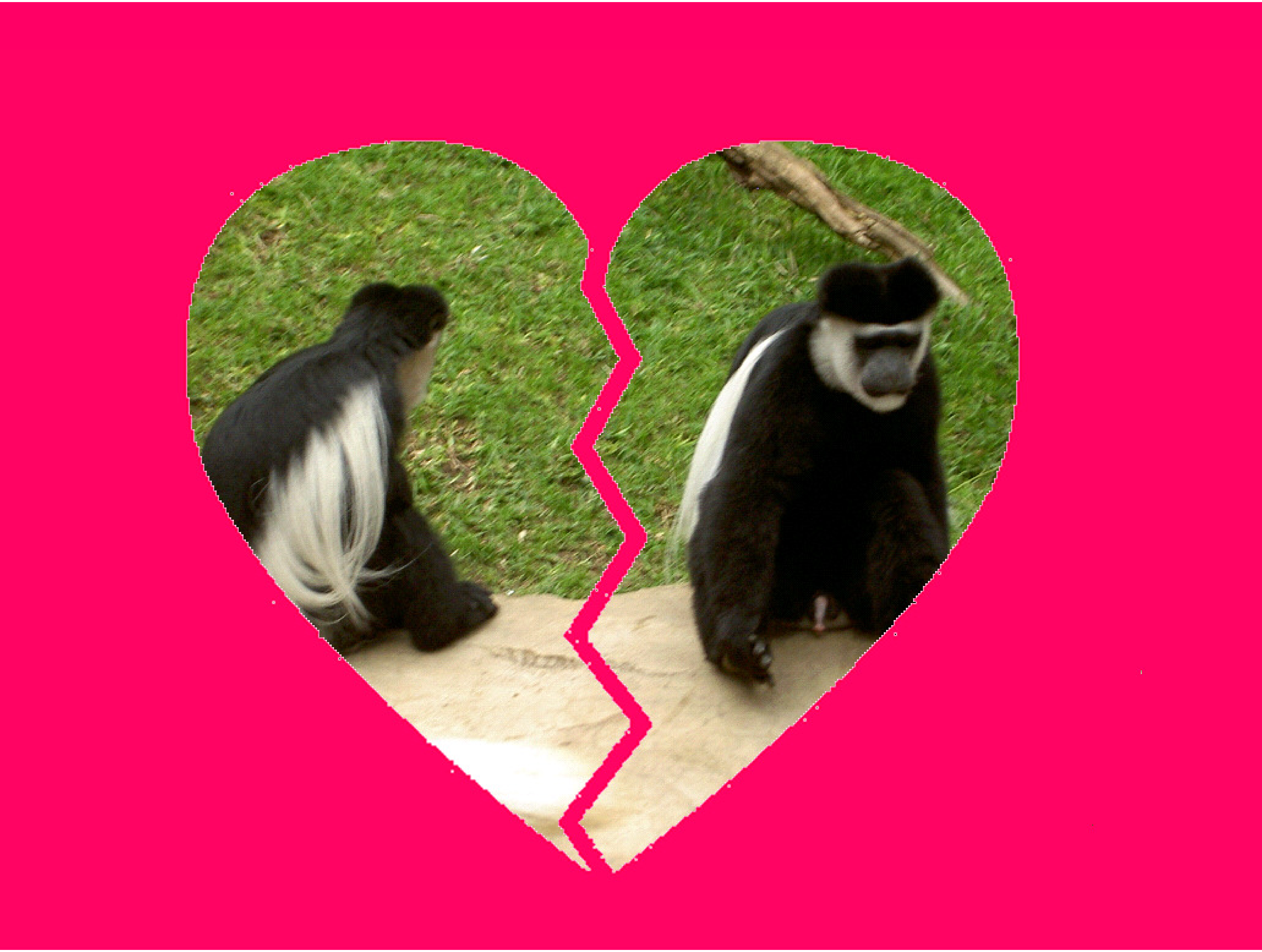
Sorry, you're not my type

Jonathan Worthington

YAPC::EU::2006



Type Systems

Type Theory?!

$$\frac{}{\Gamma \vdash n : int} \text{ (provided } n \text{ is an integer)}$$

$$\frac{}{\Gamma \vdash true : bool}$$

$$\frac{}{\Gamma \vdash false : bool}$$

$$\frac{}{\Gamma \vdash x : T} \text{ (provided } \Gamma(x) = T\text{)}$$

$$\frac{\Gamma \vdash t_1 : int \quad \Gamma \vdash t_2 : int}{\Gamma \vdash t_1 + t_2 : int}$$

$$\frac{\Gamma \vdash t_1 : bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : T}$$

An overview of type systems

What Is A Type?

- TMTOWTDI (There's More Than One Way To Define It)
- A common definition: a type classifies a value (e.g. 42 is an integer, "monkey" is a string…)
- Another definition: a type defines the representation of and set of operations that can be performed on a value.

What Is A Type System?

- Real programs consist of terms that compute values.
  - "`29 + 13`"
- A type system classifies a term in a program according to the type of values that it will compute.
  - "`29 + 13`" will have type "`integer`"
- Vary greatly between languages.

Why Type Systems Are Good

- The biggest win is that we can ensure that our programs cannot perform certain bad operations.
- For example, most high level languages only allow a reference to be used in a de-reference operations.
- Not the case in all languages; in C can create a pointer from any integer => programs can segfault.

Why Type Systems Are Good

- Perl 5's type system only allows references to be de-referenced; you get a runtime "type error" if you try to de-reference an integer (with strict on).

```
$ cat test.pl
#!/usr/bin/perl
use strict;
my $bar = 0xdeadbeef;
print $$bar;
$ perl test.pl
Can't use string ("3735928559") as a SCALAR ref while
"strict refs" in use at test.pl line 4.
```

Why Type Systems Are Good

- Compare that with what C’s type system lets you do.

```
int main()
{
    int x = 0xdeadbeef;
    int* p = (int*)x; /* int becomes int pointer! */
    int y = *p; /* Dereference...KABOOM! */
    return 0;
}
```

- This program will produce a segfault when you run it.

- Perl is **type safe**, while C is not.

Why Type Systems Are Good

- Types also provide optimisation hints.
- In Perl 6, you can (optionally) specify types.

```
my int $x = 42;
my int @array;
```

- The lowercase “int” type allows the compiler to use a native integer to represent the value => JITed code fast!
- Allows for more compact arrays.

A Quote From The Perl 6 Design Docs

Perl 6 has an optional type system that helps you write safer code that performs better. The compiler is free to infer what type information it can from the types you supply, but will not complain about missing type information unless you ask it to.

Type Systems Can Be A Pain Too

- The choice of type system greatly affects how a language feels to work in.
- Let's compare writing a simple program in Perl 5 and C# 1.0.
- The user can enter a number of integers, followed by a blank line. The average is then computed.

The Perl Implementation

```
my @values;
while (my $num = <>) {
    chop $num;
    if ($num) {
        push @values, $num;
    } else {
        last;
    }
}
my $total = 0;
foreach my $v (@values) {
    $total += $v;
}
my $average = $total / @values;
print "Average: $average\n";
```

The C# Implementation

```
ArrayList values = new ArrayList();
bool finished = false;
while (!finished) {
    String s = Console.ReadLine();
    if (s != "") {
        double value = Double.Parse(s);
        values.Add(value);
    } else {
        finished = true;
    }
}
double total = 0;
for (int i = 0; i < values.Count; i++)
    total += (double) values[i];
double average = total / (double) values.Count;
Console.WriteLine("Average: " + average + "\n");
```

Type Annotations

```
ArrayList values = new ArrayList();
bool finished = false;
while (!finished) {
    String s = Console.ReadLine();
    if (s != "") {
        double value = Double.Parse(s);
        values.Add(value);
    } else {
        finished = true;
    }
}
double total = 0;
for (int i = 0; i < values.Count; i++)
    total += (double) values[i];
double average = total / (double) values.Count;
Console.WriteLine("Average: " + average + "\n");
```

Need to write explicit type annotations => more to type

Type System Not Expressive Enough

- The most annoying problem is that we have to cast a value when removing it from a collection.

```
total += (double) values[i];
```

- Type system not expressive enough for us to indicate that the collection will only ever contain doubles.
- C# 2.0 added generics to resolve this => much nicer language to work in.

Notation

Types

- We usually specify that a term has a type by placing a colon between the two.

$$42 : int$$

$$1 + 5 : int$$

$$true : bool$$

- Notation exists for more complex types; I’ll only detail functional types.

Functional Types

- Functional types (that is, types of functions) use an arrow notation
  - The type of the arguments go to the left of the arrow.
  - The type of the return value goes to the right of the arrow.

$$sub\ double\ (int\ \$x)\ \{\ 2\ *\ \$x\ \}\ :\ int \rightarrow int$$

$$sub\ iszero\ (int\ \$x)\ \{\ \$x == 0\ \}\ :\ int \rightarrow bool$$

Type Environments

- A type environment, often written Γ (uppercase Greek letter gamma), maps names (of variables in languages that have them) to types.
- For example, the following type environment tells us the types of the scalars $x and $b.

$$\Gamma = \{ \ \$x \rightarrow \ int, \ \$b \rightarrow \ bool \ \}$$

Type Environments

- The type environment Γ on the last slide allows us to determine the following typing:

$$2 * \$x : int$$

- Formally we write this as follows:

$$\Gamma \vdash 2 * \$x : int$$

- Which we read as "gamma proves that 2 * $x has type int".

Typing Rules

- We describe a type system formally using a group of inductive typing rules.
- Inductive means…
  - The type of a term in a program is defined in terms of its sub-terms.
  - There are a set of terms that do not break down any further, known as base cases.

Typing Rules

- Here are some typing rules for base cases. The line above them indicates that they do not depend on any other rules to determine the type.

$$\frac{}{\Gamma \vdash n : int} \text{ (provided } n \text{ is an integer)}$$

$$\frac{}{\Gamma \vdash true : bool}$$

$$\frac{}{\Gamma \vdash false : bool}$$

$$\frac{}{\Gamma \vdash x : T} \text{ (provided } \Gamma(x) = T\text{)}$$

Typing Rules

- Addition may have the following typing rule:

$$\frac{\Gamma \vdash t_1 : int \quad \Gamma \vdash t_2 : int}{\Gamma \vdash t_1 + t_2 : int}$$

- You can read this as "we can prove that $t_1 + t_2$ has type int provided that t_1 has type int and t_2 has type int".
- The conditions above the line must be true for the what is below the line to be.

Typing Rules

- The typing rule for "if" is a little more complex; we introduce a type variable T:

$$\frac{\Gamma \vdash t_1 : bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : T}$$

- This specifies that the condition of the if statement must be a boolean and the branches of the if must have the same type (not true of all languages!)

Terminology

Type Checking

- Given a type environment, a term and the type that we believe the term to have, type checking verifies that the term does indeed have that type.

Given a type environment Γ, a term t and a type T, show that $\Gamma \vdash t : T$

- This is the process by which C# would decide to reject the following program:

```
int x = 5;
int y = 13;
string z = x + y;  /* x + y doesn’t have type string */
```

Type Inference

- Given a type environment and a term, type inference finds the type that the term has, if it does indeed have one.

Given a type environment Γ and a term t, find a type T such that $\Gamma \vdash t : T$

- Often seen in functional languages (ML, Haskell).
- Computationally harder than type checking; type inference problem is undecidable for some type systems!

Static vs. Dynamic Typing

- The distinction being made is when type checking takes place.
- Statically typed languages will type check the entire program at compile time.
- Dynamically typed languages usually require values to carry their types around with them and perform a check at runtime when a value is used.

Static vs. Dynamic Typing Example

- The following program may work fine in a dynamically typed language, but fail to compile under a statically typed one.

```
x = "foo"
if (complex condition that is always true)
    x = 39
y = x + 3
```

- Value always an integer by the time x is used in the add operation; static type check can't determine this.

Hybrid Type Systems

- We'd like the expressiveness of dynamic typing along with the elimination of runtime checks achievable from static typing.
- Hybrid type systems check what they can statically and insert dynamic checks for what they can't decide.

"Static when possible, dynamic when needed."

Sorry, you’re not my type

Duck Typing

- If two objects provide the same interface required for an operation (for example, the same set of methods) then they are interchangeable.
- Works regardless of the class inheritance hierarchy.
- A form of dynamic typing.
- Used heavily in Ruby.

Duck Typing

- Need to be careful: both “firework” and “diagram” implement the “explode” method, but they do different things!

Strong vs. Weak Typing

- A vague definition: "how strictly are type rules enforced?"
- A strongly typed language (e.g. C#) would reject the following program; a weakly typed language (Visual Basic, Perl) would accept it.

```
x = 42;
y = "20"
z = x + y
```

Strong vs. Weak Typing

- Strongly typed languages generally enforce that coercions between types that may cause data loss (such as string to integer) must be written explicitly as casts.
- Weakly typed languages assume the programmer knows what they are doing (not always a good assumption!) and performs a coercion implicitly.

Polymorphism

Polymorphism

- Again, TMTOWTDI (both for D = Define and D = Do).
- One definition: polymorphism occurs when a term or value can be classified as having more than one type.
- Another definition: polymorphism allows the same code to operate on values of different types.

Polymorphism

- Many ways to achieve polymorphism.
- I will quickly look at three of them that feature in Perl 6 in some form.
  - Subclassing
  - Parametric polymorphism (aka generics and parameterized types)
  - Refinement types
- See proceedings for detail++.

Subclassing

- More commonly known as inheritance.
- A key part of object oriented programming.
- A subclass may be used in place of a parent class because it only adds to the behaviours and representation that the parent class has.
- Found in the many OO languages.

Subclassing

- Perl 6 has some nicer syntax for defining a subclass than Perl 5:

```
class Melon is Fruit {
    ...
}
```

- We formalize subclassing by adding a sub-typing rule that looks something like this (we really need to define "isa").

$$\frac{\Gamma \vdash t : S \qquad S\ isa\ T}{\Gamma \vdash t : T}$$

Parametric Polymorphism

- Key idea: a type can take type parameters, just as a function takes function parameters.
- We could define the types "integer list", "string list", etc.
- Parametric polymorphism allows us to give the list the type "α list", where α is a type parameter that we supply when using the list.

Parametric Polymorphism

- For example, we could implement a parametric List type in C# 2.0 that looks something like this:

```
public class List<T>
{
    public void Add(T value)
    {
        ...
    }
    public T Get(int index)
    {
        ...
    }
```

Parametric Polymorphism

- The type parameter is supplied when an instance of the list class is created.

```
List<int> = new List<int>();
```

- Perl 6 provides parametric polymorphism in an interesting way!
- A role (basically a group of methods that are composed into a class) is implicitly parameterized on the type of the invocant.

Refinement Types

- A refinement type is obtained by adding constraints to an existing type.
- For example, the type EvenInt is a refinement of the Int type that only contains even integers.
- In Perl 6, EvenInt would be defined like this:

```
subset EvenInt of Int where { $^n % 2 == 0 }
```

Refinement Types

- Anonymous refinement types in Perl 6 will be very useful!

```
sub Halve (Int $n where { $^n % 2 == 0 }) returns Int
{
    return $n / 2;
}
```

- Can use a more refined type in place of a less refined one, providing yet another path to polymorphic code!

Any questions?