Type equivalence

When is one type equal to another?

- implemented in PL/0 with `Type::same` function

"Obvious" for atomic types like `int`, `string`, user-defined types

What about type constructors like arrays?

```
var a1   :array[10] of int;
var a2,a3:array[10] of int;
var a4   :array[20] of int;
var a5   :array[10] of bool;
```


Structural vs. name equivalence

Structural equivalence:

two types are equal if they have same structure

- if atomic types, then obvious
- if type constructors:
  - same constructor
  - recursively, equivalent arguments to constructor
- implement with recursive implementation of `same`

Name equivalence:

two types are equal if they came from the same textual occurrence of type constructor

- implement with pointer equality of `Type` instances
- special case: type synonyms don't define new types

```
var c1, c2:array[2] of real; # complex numbers
var p1, p2:array[2] of real; # points
```


Implementing structural type equivalence

Problem: want to dispatch on **two** arguments, not just receiver

Using **multi-methods**, if C++ had them:

Global function declarations:

```
virtual bool same(Type* t1, Type* t2) {
   return false; }

virtual bool same(IntType* t1, IntType* t2) {
   return true; }

virtual bool same(ProcType* t1, ProcType* t2) {
   return same(t1->args, t2->args); }
```

Some typecheck code:

```
Type *t1, *t2;
...
if (! same(t1, t2)) { ... }
```

Different than static overloading in C++

In C++, using **double dispatching**:

```
class Type {
   virtual bool same(Type* t2) = 0;

   virtual bool sameAsInteger(IntegerType* t1) {
      return false; }
   virtual bool sameAsProc(ProcType* t1) {
      return false; }
};

class IntegerType: public Type {
   bool same(Type* t2) {
      return t2->sameAsInteger(this); }

   bool sameAsInteger(IntegerType* t1) {
      return true; }
}

class ProcType: public Type {
   bool same(Type* t2) {
      return t2->sameAsProc(this); }

   bool sameAsProc(ProcType* t1) {
      return args->same(t1->args); }
};
```

Type conversions and coercions

In C, can **explicitly convert**
an object of type `float` to one of type `int`

- can represent as unary operator
- typecheck, codegen normally

In C, can **implicitly coerce**
an object of type `int` to one of type `float`

- system must insert unary conversion operators as part of type checking

Type casts

In C and Java,
can explicitly **cast** an object of one type to another

- sometimes cast means a conversion
  (casts between numeric types)
- sometimes cast means just a change of static type
  w/o doing any computation
  (casts between pointer types
  or pointer and numeric types)

In C: safety of casts not checked

- allows writing low-level code that's type-unsafe
- more often used to work around limitations in
  C's static type system

In Java: downcasts from superclass to subclass include
run-time type check to preserve type safety

- Java's main form of dynamic type checking