1 Challenges for Type Systems

At the ECOOP'91 W5 Workshop on "Types, Inheritance, and Assignments", convened on Tuesday 16th July 1991 in Geneva, Switzerland, a session on "Challenges for Type Systems" was held.

Most type systems are quite hard to compare on a formal basis, since their premises are often radically different. It was felt that a shortcut to understanding the strengths and weaknesses of various systems might be to compare the way some specific programming tasks are achieved under the various approaches. Workshop participants were invited to submit sample problems by electronic mail before the workshop, and to display solutions during this session.

Andrew Black opened the session by summarising the problems received by mail. (Not all submitted problems were included, partly because of time constraints, and partly because some of the problems did not exercise a languages' type system so much as its other features.)

The challenge problems are summarised below, roughly in the form in which they were presented at the workshop.

1.1 Polymorphic *min*

This problem was originally posed by Craig Chambers in the comp.object newsgroup.

Setting

There are two kinds of numbers, integers and floats; the operation $<$ is defined on all four combinations of numeric arguments. There are also various kinds of ordered collections (sets, lists, etc). $<$ is defined over all homogeneous collections of elements that themselves understand $<$; it performs lexicographic ordering of two collections. Note that $<$ is not defined between numbers and collections.

Challenge

Type the obvious polymorphic *min* function that returns the lesser of its two arguments. The same function should be usable to compute the minimum of two numbers, or of two collections of numbers, or of two collections of collections of numbers, etc.

1.2 Implementation Independence

This problem was submitted by Ranier Fischbach.

Setting

There are two implementations of the natural numbers, called *Nat1* and *Nat2*. *Nat1* and *Nat2* Objects have identical operation sets, and the signatures of the corresponding operations are isomorphic. That is, + on a *Nat1* object takes a *Nat1* argument and returns a *Nat1* object, while + on a *Nat2* object takes a *Nat2* argument and returns a *Nat2* object.

Challenge

$x + y$ should be allowed as a expression. What is the type of this expression.?

1.3 Subtyping with Contravariant Operations

Situation

In Eiffel, it is possible to define the types *comparable* (an abstract type characterising objects that can totally ordered), *intset* (set of integers), and *vec2* (vector of length 2) such that

1. *comparable*, *intset* and *vec2* all have a < operation with signature

   < (other: **like current**): **Boolean**

2. *intset* and *vec2* are both subtypes of *comparable*

The following code fragment is then type correct:

```
c1, c2 : comparable
iset : intset
vector : vec2
iset.create
vector.create (6.0, 9.2)
c1 := iset
c2 := vector
if c1 < c2 then
    io.putstring("that's strange")
else
    io.putstring("that's stranger")
end
```

Unfortunately, this code causes the < operation of an *intset* to be invoked with a *vec2* argument, a combination that ought to be prohibited.

Challenge

How is this problem avoided in your type system?

1.4 Homogeneous Collections

This is the obvious follow-up to the previous challenge.

Setting

The language allows one to write parameterised constructors such as *set_of*$[t]$ and *ordered_array_of*$[u]$. In general, it is necessary to limit the types that may be presented as arguments to such constructors. For these examples, we require that t possesses an equality $=$ operation for arguments of type t, and that u possesses a comparison operation $<$ for arguments of type u.

Challenge

1. Give the type of the constructor operation *ordered_array_of*, including the constraint that its argument must possess the appropriate $<$ operation.
2. Show how the implementation of the *sort* operation on an object of type *ordered_array_of*$[u]$ is type-checked.
3. Explain how a use of a constructor, e.g., $a := ordered\_array\_of[char]$, is type checked.
4. Can elements be recovered without losing type information, e.g., is $c : char := a.subscript[1]$ type-correct?

1.5 Trading

This problem is due to Andrew Watson. A trader is in essence no more than a type-safe software directory. Rather than searching from public software services by name, a trader searches by *interface*, i.e., by type.

Setting

A *Trader* accepts *export* invocations from servises that wish publicise themselves, and *import* invocations from clients that wish to use public services. The type *TraderType* has two operations:

op export(x: type, a: x) $\rightarrow$ ()
op import(x: type) $\rightarrow$ **oneof** { r: x, apology}

Challenge

Type-check both an implementation of a trader and uses of import and export. Uses of a trader should be checkable entirely statically. The trader implementation must compare types at run time, but the type checker should still be able to guarantee statically that an *import* invocation that returns a result r (rather than an *apology*), does indeed return an object of the requested type.

1.6 Self Application

Applying a function x to the argument x itself is a standard problem in functional languages. Andrew Black phrased the following version of this problem for languages in which objects rather than functions are the citizens of interest.

Setting

Consider the object w defined as follows.

```
const w : c =
    object q
        operation selfApp (x: a) → (r: b)
            r := x.apply(x)
        end selfApp
    end q
```

This defines w to have type c, and to be an object with the single operation *selfApp* that takes an argument x of type a and returns a result r of type b. x must have an operation called *apply* to which x itself is a valid argument, and which returns a result that can be assigned to r.

Challenge

Write down the types a, b and c, and explain how the above code fragment can be type-checked in your language.