Comparative Programming Languages  
CM20318

Russell Bradford

2023/24

### 1. Types

#### Higher Kinded Types

More exotically, some languages (e.g., Haskell, Scala) have *higher kind* (sometimes: *higher kinded*) types

These are maps from types to types

For example, vectors: we can have vectors of integers vector<int>, vectors of booleans vector<bool> and so on

Thus vector is “really” a map from types to types:  
“vector: T -> vector<T>”

Often called a *type constructor* as it makes a new type out of the input type(s)

### 2. Types

#### Higher Kinded Types

Many languages have some basic inbuilt type constructors, e.g., in C we have the type constructor struct:

struct intlist {  
 int first;  
 int \*rest;  
}

This example makes a new type struct intlist from existing types int and int\*

Many languages support making new types using class or defclass or similar

### 3. Types

#### Higher Kinded Types

And making vector types using [] is common:

int v[10];

Which uses a type “vector of integer” int[] derived from int

### 4. Types

#### Higher Kinded Types

And that takes a type and returns a pointer type:

int \*p;

This is type “pointer to integer” in C

There are languages that have things like enum, union, (,) (that makes pairs), and so on that make new types

We might say that [] is a type constructor (like vector above)

And the same for struct and class and (,) or whatever

### 5. Types

#### Higher Kinded Types

But these type constructors are built into the language and are part of the language syntax: we can’t make new types in ways other than the ones the language gives us

The type constructors have special syntax, like [] and struct, defined as part of the language standard

But a few languages, e.g., Haskell, allow us to make other higher kinded types

### 6. Types

#### A contrived example

These higher kinds of type arise when we pursue abstractions in our code

For example, we might want to write code to sum the values in a vector of ints:  
fn sumvecint(v: Vec<int>) -> int ...

Then a vector of doubles:  
fn sumvecdouble(v: Vec<double>) -> double ...

Then other types. We recognise the same code is being written many times so we abstract and write a “single” function that covers all these cases:  
fn sumvec<T>(v: Vec<T>) -> T ...

### 7. Types

#### A contrived example

Thanks to things like monomorphization, or parametric polymorphism, or whatever, this is easy for a compiler to work with

Then we want to sum a List<T>

Then a Tree<T>

And so on. So we want to abstract  
fn sum<C<T>>(v: C<T>) -> T ...

This is harder, as C is a higher kinded type, and (unlike vectors, above) we don’t really have enough information to write a single function that works on all types C<T>

### 8. Types

#### A contrived example

More helpful would be if C were limited in some way, e.g., “C is a collection type” and perhaps we know all collection types have map method that iterates through each value in the collection one by one

We are thus lead to need to be able to consider *classes* (sets/collections) of types

Some languages support this concept, e.g., Haskell, and some only partially, e.g., Scala

**Exercise** Read further on higher kinded types and type classes

### 9. Types

#### Higher Kinded Types

It is very common to see basic structures, classes, unions, vectors, pairs and so on as type constructors in languages, but they are a built-in part of the language

But proper support where you can define and use your own higher kinded types, is rare because

* they are hard for the compiler to understand and generate good code for
* they are hard for the programmer to understand and use appropriately

So many language designers do not include them (and many designers don’t even know that higher kinded types exist in the first place!)

### 10. Types

#### Higher Rank Types

Another higher type, quite different and separate from higher kinds are *higher rank* types: the type of a function that takes a polymorphic function as argument. The function pair here:

fn pair<F>(x: i32, y: f64, f: F) where F: ... -> (i32, f64)   
{  
 (f(x), f(y))  
}

where F is polymorphic A -> A

The polymorphic function f within the scope of the body is used on two different types: firstly on i32, then on f64

### 11. Types

#### Higher Rank Types

When called on concrete values, the two calls to f will be monomorphized in different ways

A reasonable use, but also not often supported in languages as they make type inference much harder

So languages tend to require a name (like f) have the same concrete type (e.g., int or char or whatever) wherever it appears in a given scope

For such languages f would have to monomorphize to the same type in both places in the above example

### 12. Types

#### Higher Rank Types

**Exercise** Investigate Haskell’s support for higher rank types

**Advanced Exercise** Read about *early* vs. *late* binding for types

### 13. Types

#### Dependent Types

Next, there are *dependent types*, where a type depends on a value (Agda, Coq, Idris, C++ kind of, etc.)

For example IntVec<n> as a type of vectors of int of fixed length n

Here, IntVec<3> is a different type to IntVec<7>

After all, it doesn’t make sense to pass a vector of length 3 to a function that expects a vector of length 7

**Advanced Exercise** But what about passing a vector of length 7 to a function that expects a vector of length 3?

### 14. Types

#### Dependent Types

This would allow a compiler to typecheck code like

// pair of vectors -> vector of pairs  
fn pairvec<N: int>(v: DoubleVec<N>, w: DoubleVec<N>) -> ...  
...  
let pv = pairvec(a, b);

The compiler could check that the two vectors a and b have been declared with the same length

And possibly optimise the generated code as it then doesn’t need length checks at runtime, for example

### 15. Types

#### Dependent Types

It may be that the lengths of a and b can only be determined at runtime

In that case, the compiler could examine the code and try to *prove* that the lengths of a and b must be the same, and then determine N

If it couldn’t prove that, it would raise an error and refuse to compile the code: a type error

### 16. Types

#### Dependent Types

Again, this is a little more familiar than you might realise

Many languages have fixed length vectors, like int v[3];

But this is the limit of dependent types in most languages

**Exercise** Find out how much type checking C and Java, etc., do on these kinds of types

### 17. Types

#### Dependent Types

More generally, the type can depend on any property

* Vectors of even length
* Vectors whose elements are in increasing order
* Odd integers that are bounded by
* Pairs of integers where the first element is negative and the second positive
* And so on

### 18. Types

#### Dependent Types

Dependent types are very hard (actually can be undecidable) to support in full generality, as they can require the compiler to run arbitrary code to check the type

E.g., vectors of prime number length

**Exercise** Read about how C++ uses templates to support a form of dependent types

**Exercise** Read about how Rust supports a very simple kind of dependent types

**Exercise** Think about mixing higher kind and dependent types, e.g., Vec<T,n>

### 19. Types

Such advanced types (higher kinds and ranks, dependent) are mostly only seen in more experimental or research-driven languages (Agda, Idris, some support in Haskell)

They are harder for the programmer to understand, and some have theoretical problems, such as undecidable type inference

Simple fixed cases (like vector or structure constructors) are widespread, but more programmatic use of higher level types is still in the future for general-purpose languages

### 20. Types

Though we are starting to see moderate support in languages like C++ (templates), F# and Scala

And there is a history of “experimental” features eventually finding their way into mainstream languages (e.g., classes, lambdas, iterators)

### 21. Types Conclusion

**Exercise** Read about *sum* and *product* types (see union and struct in C and C++; or enum and struct in Rust)

**Exercise** Function types can be constructed in some languages (using notation like lambda or -> or Fn). Read about these

**Exercise** Then find out about covariance and contravariance with subtypes

**Exercise** Read about *algebraic data types*

**Exercise** Typechecking is hard: how do we know when two types are equal? Read about *nominal* typing and *structural* typing

### 22. Types Conclusion

**Exercise** Advanced. Read about *substructural* types including *linear*, *affine*, and *relevant* types

**Exercise** Advanced. Read about the Curry-Howard Correspondence

### 23. Types Conclusion

These days types are considered to be an essential part of a language

And so appear in many different kinds of ways

They are intended to reduce the opportunity for errors, or find errors more quickly, or enable code to be more expressive, or enable a compiler to produce better code

Even in early untyped languages there was a recommendation that the intended type of a value be reflected in the name of a variable

iAge, fSalary. See *Hungarian notation*, and read about the IMPLICIT statement in Fortran

### 24. Types Conclusion

There are many places to check for errors

* compile time: mostly type errors
* run time: e.g., division by 0, null pointers, buffer overruns (accessing beyond the ends of a vector)

Rust has pointers, but its type system is so strong it can avoid null pointers at compile time and so can avoid this kind of run time error

Haskell has no (explicit) pointers, and avoids these errors, too

### 25. Types Conclusion

Java has no explicit pointers, but still manages to get null pointer exceptions

Some languages have a “non-zero value” subtype that helps with the division by 0 question, but in general compile-time checks for things like division by zero are quite hard

### 26. Types Conclusion

There are other places for errors we often forget about

* link time, load time: making sure libraries are present, consistent and correctly called
* coding time: getting it right in the first place

### 27. Types Conclusion

“Strong types are for weak minds”  
Anon.

### 28. Variables

Next: a word on variables

As always, everyone thinks they know how variables behave, but they don’t

You may expect variables to vary, e.g.,

x = 1.0; print(x); x = 2.0; print(x); ...

But this makes code hard to analyse for correctness

In mathematics an here is the same as an everywhere else in your proof

In a program the value of x can be different every time we look at it

Thus making code difficult to check using mathematical means

### 29. Variables

The problem is that *variables vary*

And this has knock-on effects, like making optimisation of code hard for compilers: it can be hard for the compiler to prove that some supposedly optimised code behaves in the same way as the original code

In the early days of programming this wasn’t even a question, as the name “variable” indicates

But it was later found that unrestricted varying can cause the above kind of difficulties

### 30. Variables

Some languages provide a way of indicating a value of a variable never changes

For example, the keyword const in C and C++:  
   
const int x = 42; … x = 23; // compiler raises an error

And the keyword final in Java

The compiler can spot if the programmer accidentally tries to modify a value they shouldn’t

And good compilers can often produce better code if they know a variable does not change or a method cannot be overridden

### 31. Constants

As a tiny example, consider

y = x + 1;  
 ...  
 z = 2\*x;

If the compiler knows that x cannot change between the two uses, it can load the value of x once and reuse it

Otherwise, it would have to reload x on each mention, so slower code