G6021 Comparative Programming

Part 4 - Types

Typed λ-calculus

- There are many variants of the λ-calculus (applied λ-calculi).
- Here we briefly outline the simply typed λ-calculus;
- and a minimal functional programming language.

Definition:

- Types: type variables: σ, τ,... and function types: $\sigma \to \tau$
- Typed terms: terms will now be annotated with types, which we write as: $t : \sigma$ (term t has type σ)
  - $x : \sigma$
  - If $M : \tau$ and $x : \sigma$, then $\lambda x.M : \sigma \to \tau$
  - If $M : \sigma \to \tau$ and $N : \sigma$ then $MN : \tau$

Does this look familiar?

Exercise: What are the differences between “pure” and “typed” λ-calculi?

Using Implications

Modus Ponens:

From

- *P* implies *Q*

- and *P*

We can conclude

Q is true

Should be called Implication Elimination - **ImpElim**

but Greeks got there first.

Exercises

1. P **implies** (Q **implies** R) ⊢ (P **and** Q) **implies** R
2. (P **and** Q) **implies** R ⊢ P **implies** (Q **implies** R)

Properties

- Can every λ-term be given a type?
- Strong Normalisation?
- Confluence?

Exercise (for "fun")

Consider the linear λ-calculus: each variable can occur exactly once: i.e. $\lambda x.x$ is linear, but $\lambda x.xx$ is not. Now answer the above questions again.

Extending the λ-calculus (PCF)

- PCF: Programming language for Computable Functions.
- A very simple functional programming language derived from the λ-calculus.
- Types: $\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid \sigma \to \tau$
- Typed terms: Same as the typed λ-calculus, with the addition of constants:
  - $n : \texttt{int}$ for $n = 0, 1, 2, \ldots$
  - true, false : $\texttt{bool}$
  - succ, pred : $\texttt{int} \to \texttt{int}$
  - iszero : $\texttt{int} \to \texttt{bool}$
  - for each type σ, $\text{cond}_\sigma : \texttt{bool} \to \sigma \to \sigma \to \sigma$,
  - for each type σ, $\text{fix}_\sigma : (\sigma \to \sigma) \to \sigma$

Exercise: Write these in Haskell.

Examples

- succ 3 : `int`
- $\mathtt{cond}_{\mathtt{int}}(\mathtt{iszero}(\mathtt{pred}(\mathtt{pred}\ 2)))1 : \mathtt{int} \to \mathtt{int}$

and *factorial*

```
f x = if x==0 then 1 else x * f(x-1)
```

which we can code as

$$\mathtt{fix}_{\mathtt{int}\to\mathtt{int}}\left(\lambda f^{\mathtt{int}\to\mathtt{int}}.\lambda x^{\mathtt{int}}.\mathtt{cond}_{\mathtt{int}}(\mathtt{iszero}\ x)\ 1\ (\mathtt{mult}\ x\ (f\ (\mathtt{pred}\ x)))\right)$$

Exercise: Define mult.

Where did that come from?

Here are several snap-shots of the transformation from Haskell to PCF:

```
f x = if x==0 then 1 else x*f(x-1)
f = \x -> if x==0 then 1 else x*f(x-1)
f = \x -> cond (x==0) 1 (x*f(x-1))
f = \x -> cond (iszero x) 1 (x*f(pred x))
```

What next? PCF does not have recursion... Abstract that out:

```
F = \f -> \x ->  cond (iszero x) 1 (x * f(pred x))
```

`F` is not recursive! But it does not compute factorial...

Exercise: What are these:

```
F (\x -> x)
F succ
F pred
```

Fixpoints

```
F = \f -> \x -> cond (iszero x) 1 (x*f(pred x))
```

None of the above give the factorial function. What we need is a function `fact`, because:

```
F fact = fact
```

We don't have such a function... But we do have a way of finding it!

What we need is the *fixpoint* of `F`, which we write as `fix F`.

```
fact = fix F
```

So to compute the factorial of a number, say 3, we write:

```
fix F 3
```

Example

Here are some snap-shots of the reduction of `fix F 3` to demonstrate how the computation works:


```
fix F 3 -> F (fix F) 3
-> (\x -> cond (iszero x) 1 (x*(fix F (pred x)))) 3
-> cond (iszero 3) 1 (3*(fix F (pred 3)))
-> cond False 1 (3*(fix F 2))
-> 3*(fix F 2)
...
```

What's New?

$$
\begin{array}{lcl}
\text{true} & = & \lambda xy.x \\
\text{false} & = & \lambda xy.y \\
n & = & \lambda fx.f^n\ x \\
\text{succ} & = & \lambda abc.b(abc) \\
\text{pred} & = & \lambda z.\text{fix}\ H\ z\ S\ I\ \text{false} \\
\text{cond} & = & \lambda xyz.xyz \\
\text{iszero} & = & \lambda n.n\ S\ I\ \text{true} \\
\text{fix} & = & (\lambda xy.y(xxy))(\lambda xy.y(xxy))
\end{array}
$$

where $I = \lambda x.x$, H and S are defined as:

$$
\begin{array}{lcl}
H & = & \lambda hx.\text{iszero}\ x\ 0\ (\text{succ}(h(x\ \text{false}))) \\
S & = & \lambda xy.y\ \text{false}\ x
\end{array}
$$

Operational Semantics of PCF

$$(\lambda x^{\sigma}.M)N \to M\{x \mapsto N\}$$

$$\mathrm{fix}_{\sigma} M \to M(\mathrm{fix}_{\sigma} M)$$

$$\mathrm{cond}_{\sigma}\ \mathrm{true}\ M\ N \to M$$

$$\mathrm{cond}_{\sigma}\ \mathrm{false}\ M\ N \to N$$

$$\mathrm{succ}\ n \to n+1$$

$$\mathrm{pred}\ (n+1) \to n \qquad \mathrm{pred}\ 0 \to 0$$

$$\mathrm{iszero}\ 0 \to \mathrm{true} \qquad \mathrm{iszero}\ (n+1) \to \mathrm{false}$$

Rules

$$\frac{M \to M'}{\text{cond}_\sigma\, M N_1 N_2 \to \text{cond}_\sigma\, M' N_1 N_2}$$

$$\frac{M \to M'}{\text{succ } M \to \text{succ } M'} \qquad \frac{M \to M'}{\text{pred } M \to \text{pred } M'}$$

$$\frac{M \to M'}{\text{iszero } M \to \text{iszero } M'} \qquad \frac{M \to M'}{MN \to M'N}$$

Remark that:

- The configurations are just terms.
- Computation = evaluation = reduction:

  $M \to N$ means M reduces (evaluates) to N.
- A final value is an irreducible (fully evaluated) term.

Observations

Note that we do not have "reductions in every context". Specifically, we do not have:

$$\frac{N \to N'}{MN \to MN'} \qquad \frac{N \to N'}{\lambda x.N \to \lambda x.N'}$$

Strategies

Which strategy is being used here?

How can we change it to another strategy?

Properties

- Termination?
- Subject Reduction: If $M : \sigma$ and $M \to M'$ then $M' : \sigma$
  If M terminates, then:
  - if M : `int` then $M \to^* n$
  - if M : `bool` then either $M \to^*$ true or $M \to^*$ false

  Otherwise: non-terminating (but still preserves the type)

Summary

1. λ-calculus (pure, typed)
2. PCF: simple functional language

These languages are very primitive (as far as the programmer is concerned)
However they provide the basis of the *functional paradigm*
Many languages based on this:

- Standard ML, CAML
- Haskell
- Clean
- Lisp, Scheme, ...

Type systems and Type Reconstruction

Type systems have become one of the most important theoretical developments in programming languages

Here we will examine several key issues:

- Type reconstruction (and unification)
- Polymorphic types
- Overloading
- Intersection types
- (System F)

Proof Systems

We write $\Gamma \vdash M : A$ to mean that term M has type A using the context Γ

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Using these rules we can build derivations of typed terms

Examples

$$\dfrac{\dfrac{\dfrac{}{x : A, y : B \vdash x : A}}{x : A \vdash \lambda y.x : B \to A}}{\vdash \lambda x.\lambda y.x : A \to B \to A}$$

$$\dfrac{\dfrac{\dfrac{}{f : A \to B, x : A \vdash f : A \to B} \quad \dfrac{}{f : A \to B, x : A \vdash x : A}}{f : A \to B, x : A \vdash fx : B}}{f : A \to B \vdash \lambda x.fx : A \to B}}{\vdash \lambda f.\lambda x.fx : (A \to B) \to A \to B}$$

Type Reconstruction

- Given a term M, can we find its type?

The proof system suggests an algorithm:

- If M is a variable, then look up the type in the context
- If $M = \lambda x.M'$ is an abstraction, find the type of M' in the context extended with $x : A$, then calculate the type of the result
- If M is an application, find the type of the function, then the argument, then calculate the type of the result

But how do we make the types fit?
E.g. $M : A \to B$ and $N : C$. Can we give a type for MN?
(Can we make A and C the "same" type?)

Polymorphism

A second question: can a term (program) have several types?

Example: $P = \lambda x.1 : A \to \text{int}$
Are both P true and P 3 possible?

It seems reasonable, but at what moment does type A become either bool or int?

Polymorphism

Polymorphism is a mechanism which allows us to write functions which can process objects of different types. It is a very powerful programing technique.

Examples:

```
len [] = 0
len (_:t) = 1+len t
len :: (Num t1) => [t] -> t1

len [1,2,3]
3

len "G6021"
5
```

Another Example

```
map f [] = []
map f (h:t) = (f h): map f t
map :: (a -> b) -> [a] -> [b]

map (\x -> x+1) [2,3,4]
[3,4,5]

map (\x -> "X") [\x -> x, \x -> x+1]
["X","X"]
```


- `t`, `t1`, `a`, `b` are type variables
- `len :: [t] -> Int` means that `len` has type $\forall t.[t] \rightarrow$ `Int`. I.e. forall types t.
- t is called a *generic* type

Generalisation and Specialisation

The final machinery that we need are ways of introducing (and eliminating) the $\forall$.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall\alpha.A}\ (GEN) \qquad \frac{\Gamma \vdash M : \forall\alpha.A}{\Gamma \vdash M : [B/\alpha]A}\ (SPEC)$$

Note: $\alpha \notin FV(\Gamma)$ for the GEN rule

Reconstructing Polymorphic types

We give an algorithm which will find "the most general type" of a term

- If M has a type, then we will find it, otherwise fail

Exercise: what could "most general type" mean?

Machinery:

- Substitution (of types)
- Unification
- Type reconstruction

Unification

There is an algorithm $\mathcal{U}$, which given a pair of types either returns a substitution V or fails; further:

- If $\mathcal{U}(\tau, \tau') = V$ then $V\tau = V\tau'$ (we say V *unifies* τ and τ').
- If S unifies τ and τ' then $\mathcal{U}(\tau, \tau')$ returns some V and there is another substitution R such that $S = RV$ (most general unifier).

Moreover, V only involves variables in τ and τ'. Example:

$$\mathcal{U}(\alpha \to \alpha, (\beta \to \beta) \to \beta \to \beta) = [\beta \to \beta / \alpha]$$

Disagreement sets

The algorithm for unification is specified in terms of the notion of a *disagreement set*. When unifying pairs of types we will have a disagreement set that is either empty or cardinality 1.

$$\begin{aligned}\mathcal{D}(\tau,\tau') &= \emptyset \text{ (if } \tau=\tau') \\ &= \{(\sigma,\sigma')\}\end{aligned}$$

where σ,σ' are the first two subterms at which τ,τ' disagree, using depth first comparison. Some examples are in order:

$$\begin{aligned}\mathcal{D}(\alpha\to\beta,\alpha\to\beta) &= \emptyset \\ \mathcal{D}(\sigma\to\tau,\alpha\to\beta) &= \{(\sigma,\alpha)\} \\ \mathcal{D}(\texttt{int},\alpha\to\beta) &= \{(\texttt{int},\alpha\to\beta)\} \\ \mathcal{D}((\texttt{int}\to\texttt{int})\to\texttt{int},\alpha\to\beta) &= \{(\texttt{int}\to\texttt{int},\alpha)\}\end{aligned}$$

Unification

$$\mathcal{U}(\tau,\tau') = \texttt{iter}(\texttt{id},\tau,\tau')$$

where

$$\begin{aligned}
\texttt{iter}(V,\tau,\tau') &= V, \text{ if } V\tau = V\tau' \\
&= \texttt{iter}([b/a]V,\tau,\tau'), \text{ if } a \text{ does not occur in } b \\
&= \texttt{iter}([a/b]V,\tau,\tau'), \text{ if } b \text{ does not occur in } a \\
&= \mathit{FAIL}, \text{ otherwise.}
\end{aligned}$$

where $\{(a,b)\} = \mathcal{D}(V\tau, V\tau')$.

Reconstruction of Types

Using unification, and the proof system as a guide, the algorithm is a function which takes a set of assumptions (Γ) and a term to be typed (e), and returns two things: a substitution (T) and the type of the whole term (τ): $\mathcal{T}(\Gamma, e) = (T, \tau)$

1. $\mathcal{T}(\Gamma, x) = (\mathtt{id}, \tau)$ where $\tau = [\beta_1/\alpha_1, \ldots \beta_n/\alpha_n]\sigma$ if $x : \forall\alpha_1 \ldots \forall\alpha_n.\sigma \in \Gamma$. Otherwise, $\tau = \bot$ x
2. $\mathcal{T}(\Gamma, MN) = (USR, U\beta)$ where $(R, \rho) = \mathcal{T}(\Gamma, M)$, $(S, \sigma) = \mathcal{T}(R\Gamma, N)$ and $U = \mathcal{U}(S\rho, \sigma \to \beta)$ (β new)
3. $\mathcal{T}(\Gamma, \lambda x.M) = (R, R\beta \to \tau)$ where $(R, \rho) = \mathcal{T}(\Gamma \cup x : \beta, M)$ (β new)
4. $\mathcal{T}(\Gamma, \text{let } x = M \text{ in } N) = (SR, \tau)$ where $(R, \sigma) = \mathcal{T}(\Gamma, M)$, $(S, \tau) = \mathcal{T}(R\Gamma \cup x : \sigma', N)$, $\sigma' = \forall\alpha_1 \ldots \forall\alpha_n.\sigma$ and $\{\alpha_1, \ldots, \alpha_n\} = FV(\sigma) - FV(\Gamma)$

Reconstruction of types in functional languages

We can add a number of extra rules for the built-in types. For example, something like this:

$$\frac{}{\Gamma \vdash n :: Integer} \qquad \frac{}{\Gamma \vdash True :: Bool} \qquad \frac{}{\Gamma \vdash False :: Bool}$$

$$\frac{\Gamma \vdash P :: Bool \qquad \Gamma \vdash Q :: Bool}{\Gamma \vdash P \&\& Q :: Bool} \qquad \frac{\Gamma \vdash P :: Int \qquad \Gamma \vdash Q :: Int}{\Gamma \vdash P + Q :: Int}$$

$$\frac{}{\Gamma \vdash [] :: [a]} \qquad \frac{\Gamma \vdash h :: a \qquad \Gamma \vdash t :: [a]}{\Gamma \vdash h : t :: [a]}$$

Type reconstruction can be extended in a straightforward way.

Question: What about user defined types?

Type checking versus type inference

- Type checking refers to the process of checking that the types declared in a program are compatible with the use of the functions and variables.
- Type inference (or type reconstruction) is the process of inferring types for the elements of the program (where type declarations might be present, optionally).

Other notions of type

- Overloading. `1 + 4, 1.2 + 3.7, "this " + " is a " + "string"`
  Also known as ad hoc polymorphism.
- Intersection types

$$\frac{\Gamma \vdash M : (\sigma_1 \cap \sigma_2)}{\Gamma \vdash M : \sigma_i} \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}$$

- System F: types as terms, dependent types, etc.

Type classes in Haskell

- Polymorphic: same code executed
- Overloaded: different code executed
- Haskell has an intermediate notion: type classes.

```
len :: (Num t1) => [t] -> t1
```

  - `Num` is a typeclass: all things like numbers. So, `len` takes a list of anything (really anything) and produces a number of some kind (but might be Int, Integer, etc.). Saying that the type is in this class groups all these functions together.
  - Another example: `Eq` class defines equality (==) and inequality (/=). All the basic datatypes exported by the Prelude are instances of Eq, and Eq may be derived for any datatype whose constituents are also instances of Eq.

Not to be confused with classes in Java.

Subtypes

- $A <: A$ (reflexivity)
- $A <: B$ and $B <: C$ then $A <: C$ (transitivity)
- $a : A$ and $A <: B$ then $a : B$ (subsumption)

We also add a "top" type $\top$, which is above everything else: $A <: \top$

- Can you give examples of these from Java?
- Objects: A larger type is a subtype of a smaller type

Types of polymorphism

- Parametric polymorphism: operates uniformly across different types.
- Subtype polymorphism: operates through an inclusion relation.
- Ad-hoc polymorphism is another name for overloading and is about the use of the same name for different functions.

Object-Oriented Languages

- Many modern programming languages are based around the object model: Java, Eiffel, C++, Smalltalk, Self, etc.
- Naïve understanding: object = (pointer to a) record
- Basic features: Object creation, Field selection, Field update, and Method invocation

We could study an object calculus which allows us to understand the basic elements of object-oriented programming in the same spirit as the λ-calculus for functional programming.
However, we want to focus now on comparing the paradigms.

Question: Functions vs. objects?

Object Oriented Programming

Objects:

- public data: methods (member functions), public variables
- private data: instance variables, hidden methods

Object-Oriented Program:

- Send messages to objects

Object-Oriented Programming

- Programming methodology: organise concepts into objects and classes
- Concepts: encapsulate data, subtyping (extensions of data-types), inheritance (reuse of implementation)

Four Basic Concepts

- Dynamic Lookup - when a message is sent to an object, the method executed is determined by the object implementation. Different objects can respond differently to the same message. The response is not based on the static property of the variable or pointer.
- Abstraction - implementation details are hidden inside a program unit and exposed via a specific interface. Usually a set of public methods manipulate private data.
- Subtyping - if object A has all the functionality of another object B, we can use A in place of B in contexts expecting A. Subtyping means that the subtype has at least as much functionality as the base type.
- Inheritance - reuse definition of one type of object when defining another object.

Aside: delegation-based languages

Examples:

- Dylan
- Self

In these languages, objects are defined directly from other objects by adding methods and replacing methods (rather then from classes).

Dynamic Lookup

A method is selected dynamically (at run time) according to the implementation of the object that receives the message: Different objects may implement the same operation differently.

Example:
`x.add(y)` means send the message `add(y)` to the object `x`. If `x` is an integer, then we may perform usual addition; if `x` is a string, then concatenation; if `x` is a set, then we add the element `y` to the set, etc. etc. Thus:

```
while(c) {
    ...
    x.add(y)
    ...
}
```

may perform a different operation each time we enter the loop.

Dynamic lookup, continued

In functional languages, `x.add(y)` would be written as `add(x,y)`: the meaning of the operation stays the same.

Exercise: does dynamic lookup = overloading ?

Answer: To some extent: however, overloading is a *static* concept: it is the static type information that dictates which code is used.

Dynamic lookup is an important part of Java, C++ and Smalltalk. (It is the default in Java and Smalltalk, in C++ only *virtual* member functions are dynamic).

Abstraction (encapsulation)

- Programmer has a detailed view of program
- User has an abstract view

Encapsulation is a mechanism for separating these two views

SML has a notion of abstraction:

```
abstype Set with
    empty : unit -> Set
    isEmpty : Set -> boolean
    add : int * Set -> Set
    union : Set * Set -> Set
  is ... (* detailed implementation *)
  in ... (* program *) end
```

Abstraction (Haskell example)

```
module Stack (Stack,empty,isEmpty,push,top,pop)
    where
empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a,Stack a)

newtype Stack a = StackImpl [a]
empty = StackImpl []
isEmpty (StackImpl s) = null s
push x (StackImpl s) = StackImpl (x:s)
top (StackImpl s) = head s
pop (StackImpl (s:ss)) = (s,StackImpl ss)
```

Encapsulation

- Guarantee invariants of data structure: only functions of the data type have access to the internal representation of the data
- Limited Reuse: cannot reuse code

Exercise: What is the essential difference between functional style abstraction and OO abstraction?

Object-oriented languages allow encapsulation in an extensible form.

Subtyping and Inheritance

- Interface: The external view of an object; messages accepted by an object; the *type*
- Subtyping: relation between interfaces
- Implementation: internal representation of an object
- Inheritance: relation between implementations

Subtyping

- interface Point: `x`, `y`, `move`
- interface ColouredPoint: `x`, `y`, `move`, `colour`, `changeColour`.

If interface *A* contains all of interface *B*, then *A* objects can also be used as *B* objects.

ColouredPoint interface contains Point: ColouredPoint is a *subtype* of Point

Inheritance

- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

Example:

```
class Point
   float x,y;    Point move(float dx, dy)

class ColouredPoint
   float x,y; colour c;  Point move(float dx, dy)
   Point changeColour(colour newc)
```


- Subtyping: ColouredPoints can be used in place of Points: property used by the client
- Inheritance: ColouredPoints can be implemented by reusing the implementation of Point: property used by the programmer

Multiple Inheritance

- A controversial aspect of Object-oriented programming
- Should we be allowed to build a class by inheriting from more than one base class?

Problems:

- Name clashes: if class *C* inherits from classes *A* and *B*, where *A* and *B* have members of the same name, then we have a name clash.
  solutions:
  - Implicit resolution: arbitrary way defined by the language
  - Explicit resolution: programmer decides
  - Disallow name clashes: programs are not allowed to contain name clashes

Exercise: can you give an example of name clashes using a Java-like syntax?

Case Study: Java

Design Goals

- Portability
- Reliability
- Safety
- Simplicity
- Efficient (secondary)

Almost everything in Java is an object. Does not allow multiple inheritance, statically typed.

- Syntax similar to C++
- Objects: fields, methods
- Dynamic lookup: similar behaviour to other languages, static typing (more efficient than some other languages, e.g. Smalltalk)
- Dynamic linking (slower than C++)

Terminology

- Class, Object as usual
- Field: data member
- Method: data function
- Static members: class fields and methods
- this: self
- Package: set of classes in a shared namespace
- Native method: method written in another language (often C)

Java Encapsulation

Every field belongs to a class, every class is part of some package

Visibility

- Four distinctions: public, private, protected, package
- Method can refer to:
  - private members of class it belongs to
  - non-private members of all classes in the same package
  - protected members of superclasses (in different packages)
  - public members of classes in visible packages

Inheritance

- Similar to other languages (C++, Smalltalk)
- Subclass inherits from superclass: but only single inheritance
- Some additional features:
  - final classes and methods
  - use of super in constructors (subclass constructor *must* call the super constructor - compiler will add it anyway! Note that if the superclass does not have a constructor with same number of arguments, then we get a compilation error!)

Class Object

In Java, every class extends another class: superclass is *Object* if no other class is named.

Class Object has a number of methods:

- getClass
- toString
- equals
- hashCode
- Clone
- wait, notify, notifyAll (used with concurrency)
- finalize

Types

Primitive types (not objects): int, bool etc.
Reference types: classes, interfaces, arrays
Type conversion:

- Cast is checked at run-time (may raise Exception)
- if $A < B$ and $B\ x$ then can cast x to A.

Subtyping subclass produces subtype; single inheritance implies tree structure of subclasses
However, an interface can have multiple subtypes (multiple subtyping)

Generic Programming

Exercise: Does Java support parametric polymorphism?

- Class Object is the supertype of all types: allows subtype polymorphism
- Early versions of Java did not allow templates (parametric polymorphism)

Note that we can use Object to write generic data-structures (for instance lists), but what are the problems with this?

Templates

We write:

```
class Stack {
  void push(Object o){ ... }
  Object pop() { ... }
...
```

But would like to write:

```
class Stack<A> {
  void push (A a){ ... }
  A pop(){ ... }
...
```

This was considered one of the main shortfalls of Java. Many proposals put forward, but is now “standard”.

Representing types in different paradigms

Different paradigms support different ways of representing structured data:

- Product types
- Disjoint union types
- Other?

In this short case study we will focus on products and unions in three paradigms:

- Functional (Haskell)
- Object-oriented (Java)
- Imperative (C)

Product types

- A *product type* is the simplest way of combining several types.
- Also known as a record in some older languages.
- Example: if a function (procedure, method...) needs to return two values then we can make a product type (pair).

Exercise: Define a type to represent a colour as a name and three numbers (RGB).

Product types in different paradigms

Haskell. Products are built-in:

```
("red",255,0,0) :: ([Char],Int,Int,Int)
```

Note that in Haskell we can give a new name to an old type. Example:

```
type Name = String
type Colour = ([Char],Int,Int,Int)
```

Java. Define a new object.

```
class Pair { int x,y; }
class Colour { String name; int r,g,b; }
```

C. use a 'struct'

```
struct Pair { int x,y; };
```

Exercise: test these in the labs. Build products and use them. Note the difference in the way they are accessed (e.g. how do you destruct a product to access the components).

Disjoint union (sum) types

- A *disjoint union type* is way of representing several *alternative* types.
- Known as a sum type, or just union type, in some languages.
- Example: if we want an array of integers or Booleans, we can define a type IntOrBool and create an array of this type.

Note: can represent these using products. (How?)

Disjoint types in different paradigms

Haskell. Built using data:

```
data Bool = True | False
data Suit = Diamond | Spade | ...
data Option a = Nothing | Something a
```

Once defined, we have a new type and new constructors. Can be used in pattern matching directly.

Java. Define several new objects using subclasses:

```
class Suit {}
class Diamond extends Suit {}
class Spade extends Suit {}
...
```

C. use a "union"

```
union intorchar { int x; char y; };
```

Summary of case study

- Each paradigm supports structured data types.
- Main issues: ease of creation, natural representation, ways in which they are used, etc.
- Exercise: test these in the labs. Build sums and products and use them. Note the difference in the way they are accessed (e.g. how do you know which component of the sum is being represented, etc.).

Summary

- Types play a very important role in programming languages
- Different paradigms use types in different ways
- Overloading is a way of using the same name (less things for the programmer to remember)
- Polymorphism is a way of using the same code for different types
- Inheritance is a way of reusing implementations of other objects. Multiple inheritance is a problem though.