On Understanding Types, Data Abstraction, and Polymorphism

Excerpted from: Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, 17(4):471–522, 1985.

Kinds of Polymorphism

- Monomorphic languages:
  - All functions and procedures have unique type.
  - All values and variables of one and only type.
  - Comparable to Pascal or C type systems.
- Polymorphic languages:
  - Values and variables may have more than one type.
  - Polymorphic functions admit operands of more than one type.
- Universal polymorphism:
  - Function works uniformly on range of types.
  - Parametric and inclusion polymorphism.
- Ad-hoc polymorphism:
  - Function works on several unrelated types.
  - Overloading and coercion.

Universal Polymorphism

- Parametric polymorphism:
  - Actual type is a function of type parameters.
  - Each application of polymorphic function substitutes the type parameters.
  - Generic functions:
    - "Same" work is done for arguments of many types.
    - Length function over lists.
- Inclusion polymorphism:
  - Value belongs to several types related by inclusion relation.
  - Object-oriented type systems.

Ad-hoc Polymorphism

- Overloading
  - Same name denotes different functions.
  - Context decides which function is denoted by particular occurence of a name.
  - Preprocessing may eliminate overloading by giving different names to different functions.
- Coercion
  - Type conversions convert an argument to a type expected by a function.
  - May be provided statically at compile time.
  - May be determined dynamically by run-time tests.
- Only apparent polymorphism
  - Operators/functions only have one type.
  - Only syntax "pretends" polymorphism.

Overloading and Coercion

- Distinction may be blurred:

```
3   + 4
3.0 + 4
3   + 4.0
3.0 + 4.0
```

- Different explanations possible:
  - + has four overloaded meanings.
  - + has two overloaded meanings (integer and real addition) and integers may be coerced to reals.
  - + is real addition and integers are always coerced to reals.
- Overloading and/or coercion or both!

Preview of Fun

- Language based on lambda-calculus
  - Basis is first-order typed lambda-calculus.
  - Enriched by second-order features for modeling polymorphism and object-oriented languages.
- First-order types
  - Bool, Int, Real, String.
- Various forms of type quantifiers

$$
\begin{array}{rcl}
T & ::= & \cdots \mid S \\
S & ::= & \forall X.T \mid \exists X.T \mid \forall X \subseteq T.T \mid \exists X \subseteq T.T
\end{array}
$$

- Modeling of advanced type systems:
  - Universal quantification: parameterized types.
  - Existential quantifiers: abstract data types.
  - Bounded quantification: type inheritance.

The Typed Lambda-Calculus

- Syntactic extension of untyped lambda-calculus
  - Every variable must be explicitly typed when introduced
  - Result types can be deduced from function body.
- Examples
  - `value succ = fun(x:Int) x+1`
  - `value twice = fun(f: Int -> Int) fun(y:Int) f(f(y))`
- Type declarations:
  - `type IntPair = Int x Int`
  - `type IntFun = Int -> Int`
- Type annotations/assertions:
  - `(3, 4): IntPair`
  - `value intPair: IntPair = (3, 4)`
- Local variables
  - `let a = 3 in a+1`
  - `let a: Int = 3 in a+1`

Universal Quantification

- Simply typed lambda-calculus describes monomorphic functions.
  - Not sufficient to describe functions that behave the same way for argumentes of different types.
- Introduce types as parameters:
  - Type abstraction `all[a] ...`
  - Type application `x[T]`

```
value id = all[a] fun(x:a) x
id[Int](3)


id : forall a. a -> a
id[Int] : Int -> Int
```

- May omit type information:

```
value id = fun(x) x
id(3)
```

  - Type inference (type reconstruction) reintroduces `all[a]`, `a`, and `[Int]`

Examples for polymorphic types

```
type GenericId = forall a. a -> a
id: GenericId
-- examples
value inst = fun(f: forall a. a -> a) (f[Int], f[Bool])
value intid: Int -> Int = fst(inst(id))
value boolid: Bool -> Bool = snd(inst(id))
```

Polymorphic Functions

- First version of polymorphic `twice`:

```
value twice1 = all[t] fun(f: forall a. a -> a)
                          fun(x: t) f[t](f[t](x))

twice1[Int](id)(3)      -- legal.
twice1[Int](succ)       -- illegal!
```

- Second version of polymorphic twice:

```
value twice2 = all[t] fun(f: t -> t) fun(x: t) f(f(x))

twice2[Int](succ)        -- legal.
twice2[Int](id[Int])(3) -- legal.
```

- Both versions different in nature of f:
  - In `twice1`, `f` is a polymorphic function of type `forall a: a -> a`.
  - In `twice2`, `f` is a monomorphic function of type `t -> t` (for some instantiation of `t`)

Rules for Universal Quantification

Introduction

$$\frac{\Gamma, \alpha \vdash M : \tau \quad \alpha \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau}$$

Elimination

$$\frac{\Gamma \vdash M : \forall\alpha.\tau \quad \Gamma \vdash \tau'}{\Gamma \vdash M : \tau[\tau'/\alpha]}$$

Formation $\Gamma \vdash \tau$

τ can be legally build from variables in Γ

$$\frac{}{\Gamma, \alpha, \Gamma' \vdash \alpha} \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \tau'}{\Gamma \vdash \tau \to \tau'} \qquad \frac{\Gamma, \alpha \vdash \tau \quad \alpha \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \forall\alpha.\tau}$$

Parametric Types

- Type definitions with similar structure:

```
type BoolPair = Bool x Bool
type IntPair = Int x Int
```

- Use parametric definition:

```
type Pair[T] = T x T
type PairOfBool = Pair[Bool]
type PairOfInt = Pair[Int]
```

- Type operators are not types:

```
type A[T] = T -> T
type B = forall T. T -> T
```

  - Different notions!

Recursive Definitions

- Recursively defined type operators:

```
rec type List[Item] =
    [nil: Unit
    ,cons: {head: Item, tail: List[Item]} ]
```

- Constructing values of recursive types:

```
value nil: forall Item.List[Item] =
               all[Item]. [nil = ()]
value intNil: List[Int] = nil[Int]
value cons:
   forall Item. (Item x List[Intem]) -> List[Item] =
      all[Item].
         fun(h Item, t: List[Item])
            [cons = {head = h, tail = t}]
```

Existential Quantification

- Existential type quantification:
  - `p: exists a. t(a)`
  - For some type cfta, `p` has type `t(a)`
- Examples:
  - `(3, 4): exists a. a x a`
  - `(3, 4): exists a. a`
  - A value can satisfy different existential types!
- Sample existential types:
  - `type Top = exists a. a` (type of any value)
  - `exists a. exists b. a x b` (type of any pair)
- Particularly useful: "existential packaging"
  - `x: exists a. a x (a -> Int)`
  - `(snd(x))(fst(x))`
  - `(3, succ)` has this type
  - `([1,2,3], length)` has this type

Information Hiding

- Abstract types:
  - Unknown representation type.
  - Packaged with operations that may be applied to representation.
- Another example:

```
x: exists a. {const: a, op: a -> Int}
x.op(x.const)
```

- Restrict use of abstract types:
  - Simplify type checking.
  - `value p: exists a. a x (a -> Int)`
    `        = pack[a = Int in a x (a -> Int)](3, succ)`
  - Value `p` is a *package*
  - Type `a x (a -> Int)` is the *interface*.
  - Binding `a=Int` is the type *representation*.
- General form:
  - pack [a = typerep in interface](contents)

Use of Packages

- Package must be opened before use:
  - ```
    value p = pack[a = Int in a x (a -> Int)]
                  (3, succ)
    open p as x in (snd(x))(fst(x))
    ```
  - ```
    value p = pack[a = Int in {arg: a, op: a -> Int}]
                  {arg = 3, op = succ}
    open p as x in x.op(x.arg)
    ```
- Reference to hidden type: `open p as x[b] in ...fun(y:b) (snd(x))(y) ...`

Rules for Existential Quantification

Introduction

$$\frac{\Gamma \vdash M : \tau[\tau'/\alpha] \quad \alpha \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \mathsf{pack}[\alpha = \tau' \text{ in } \tau](M) : \exists \alpha.\tau}$$

Elimination

$$\frac{\Gamma \vdash M : \exists \alpha.\tau \quad \Gamma, \alpha, x : \tau \vdash N : \tau' \quad \alpha \notin \mathsf{fv}(\tau', \Gamma)}{\Gamma \vdash \mathsf{open}\ M \text{ as } x[\alpha] \text{ in } N}$$

Packages and Abstract Data Types

Modeling of Ada type system:

- Records with function components model Ada packages.
- Existential quantification models Ada type abstraction.

```
type Point = Real x Real
type Point1 =
   {makepoint: (Real x Real) -> Point,
      x_coord: Point x Real,
      y_coord: Point x Real}

value point1: Point1 =
   {makepoint = fun(x:Real, y:Real)(x, y),
      x_coord = fun(p:Point) fst(p),
      y_coord = fun(p:Point) snd(p)}
```

Ada Packages

```
package point1 is
   function makepoint(x: Real, y: Real) return Point;
   function x_coord(P: Point) return Real;
   function y_coord(P: Point) return Real;
end point1;

package body point1 is
   function makepoint(x: Real, y: Real) return Point;
      -- implementation of makepoint
   function x_coord(P: Point) return Real;
      -- implementation of x_coord
   function y_coord(P: Point) return Real;
      -- implementation of y_coord
end point1;
```

Hidden Data Structures

- Ada:

```
package body localpoint is
   point: Point;
   procedure makePoint(x, y: Real); ...
   function x_coord return Real; ...
   function y_coord return Real; ...
end localpoint
```

- Fun:

```
value localpoint =
  let p: Point = ref((0,0)) in
   {makepoint = fun(x: Real, y: Real) p := (x, y),
      x_coord = fun() fst(!p)
      y_coord = fun() snd(!p)}
```

- First-order information hiding: Use let construct to restrict scoping at value level (hide record components).

Hidden Data Types

Second-order information hiding: Use existential quantification to restrict scoping at type level (hide type representation).

```
package point2
   type Point is private;
   function makepoint(x: Real, y: Real) return Point;
   ...
   private
   -- hidden local definition of type Point
end point2;

type Point2WRT[Point] =
      {makepoint: (Real x Real) -> Point,
         ...}

type Point2 =
   exists Point. Point2WRT[Point]

value point2: Point2 = pack[Point = (Real x Real) in
   Point2WRT[Point]] point1
```

Combining Universal and Existential Quantification

- Universal quantification: generic types.
- Existential quantification: abstract data types.
- Combination: parametric data abstractions.

Signature of list and array operations for examples

```
nil: forall a. List[a]
cons: forall a. (a x List[a]) -> List[a]
hd: forall a. List[a] -> a
tl: forall a. List[a] -> List[a]
null: forall a. List[a] -> Bool

array: forall a. Int -> a -> Array[a]
index: forall a. (Array[a] x Int) -> a
update: forall a. (Array[a] x Int x a) -> Unit
```

Concrete Stacks

```
type IntListStack =
  {emptyStack: List[Int],
   push: (Int x List[Int]) -> List[Int]
   pop: List[Int] -> List[Int],
   top:List[Int] -> Int}

value intListStack: IntListStack =
  {emptyStack = nil[Int],
   push = fun(a: Int, s: List[Int]) cons[Int](a,s),
   pop =  fun(s: List[Int]) tl[Int](s)
   top =  fun(s: List[Int]) hd[Int](s)}

type IntArrayStack =
  {emptyStack: (Array[Int] x Int),
   push: (Int x (Array[Int] x Int)) -> (Array[Int] x Int),
   pop:  (Array[Int] x Int) -> (Array[Int] x Int),
   top:  (Array[Int] x Int) -> Int}

value intArrayStack: IntArrayStack =
  {emptyStack = (array[Int] (100) (0), -1) ...}
```

Generic Element Types

```
type GenericListStack =
   forall Item.
     {emptyStack: List[Item],
      push: (Item x List[Item]) -> List[Item]
      pop:  List[Item] -> List[Item],
      top:  List[Item] -> Item}

value genericListStack: GenericListStack =
   all[Item]
     {emptyStack = nil[Item],
      push = fun(a: Item, s: List[Item]) cons[Item](a,s),
      pop  = fun(s: List[Item]) tl[Item](s)
      top  = fun(s: List[Item]) hd[Item](s)}

type GenericArrayStack =
   ...

value genericArrayStack: GenericArrayStack =
   ...
```

Hiding the Representation

```
type GenericStack =
   forall Item. exists Stack. GenericStackWRT[Item][Stack]

type GenericStackWRT[Item][Stack] =
  {emptyStack: Stack,
   push: (Item x Stack) -> Stack
   pop: Stack -> Stack,
   top: Stack -> Item}

value listStackPackage: GenericStack =
   all[Item]
      pack[Stack = List[Item]
         in GenericStackWRT[Item][Stack]]
      genericListStack[Item]
value useStack =
   fun(stackPackage: GenericStack)
      open stackPackage[Int] as p[stackRep]
      in p.top(p.push(3, p.emptystack))

useStack(listStackPackage)
```

Extra: Abstracting over Type Constructors

Extension of Fun: can use the abstracted stack at different type instances

```
type GenericStack2 =
   exists Stack. forall Item. GenericStackWRT2[Item][Stack]

type GenericStackWRT2[Item][Stack] =
  {emptyStack: Stack[Item],
   push: (Item x Stack[Item]) -> Stack[Item]
   pop: Stack[Item] -> Stack[Item],
   top: Stack[Item] -> Item}

value listStackPackage2: GenericStack2 =
   pack[Stack = List in GenericStackWRT2[Item][Stack]]
      genericListStack

value useStack =
   fun(stackPackage: GenericStack2)
      open stackPackage as p[SCon] in
      let pi : SCon[Int] = p[Int]
          pb : SCon[Bool] = p[Bool]
      in (pi.top(pi.push(3, pi.emptystack)),
          pb.top(pb.push(true, pb.emptystack)))

useStack(listStackPackage2)
```

Quantification and Modules

- Modules
  - Abstract data type packaged with operators.
  - Can import other (known) modules.
  - Can be parameterized with (unknown) modules.
- Parametric modules
  - Functions over existential types.

Example: Module with two Implementations

```
type PointWRT[PointRep] =
  {mkPoint: (Real x Real) -> PointRep,
   x-coord: PointRep -> Real,
   y-coord: PointRep -> Real}

type Point = exists PointRep. PointWRT[PointRep]

value cartesianPointOps =
  {mkpoint = fun(x: Real, y: Real) (x,y),
   x-coord = fun(p: Real x Real) fst(p),
   y-coord = fun(p: Real x Real) snd(p)}

value cartesianPointPackage: Point =
   pack[PointRep = Real x Real
      in PointWRT[PointRep]]
   (cartesianPointOps)

value polarPointPackage: Point =
   pack[PointRep = Real x Real in PointWRT[PointRep]]
  {mkpoint = fun(x: Real, y: Real) ...,
   x-coord = fun(p: Real x Real)   ...,
   y-coord = fun(p: Real x Real)   ...}
```

Parametric Modules

```
type ExtendedPointWRT[PointRep] =
   PointWRT[PointRep] &
   {add: (PointRep x PointRep) -> PointRep}

type ExtendedPoint =
   exists PointRep. ExtendedPointWRT[PointRep]

value extendPointPackage =
   fun(pointPackage: Point)
   open pointPackage as p[PointRep] in
      pack[PointRep' = PointRep in ExtendedPointWRT[PointRep']]
      p & {add = fun(a: PointRep, b: PointRep)
                     p.mkpoint(p.x-coord(a)+p.x-coord(b),
                               p.y-coord(a)+p.x-coord(b))}

value extendedCartesianPointPackage =
   extendPointPackage(cartesianPointPackage)
```

A Circle Package

```
type CircleWRT2[CircleRep, PointRep] =
  {pointPackage: PointWRT[PointRep],
   mkcircle: (PointRep x Real) -> CircleRep,
   center: CircleRep -> PointRep, ...}

type CircleWRT1[PointRep] =
   exists CircleRep. CircleWRT2[CircleRep, PointRep]

type Circle =
   exists PointRep. CircleWRT1[PointRep]

type CircleModule =
   forall PointRep.
   PointWRT[PointRep] -> CircleWRT1[PointRep]

value circleModule: CircleModule =
   all[PointRep]
      fun(p: PointWRT[PointRep])
         pack[CircleRep = PointRep x Real
            in CircleWRT2[CircleRep,PointRep]]
        {pointPackage = p,
         mkcircle = fun(m: PointRep, r: Real)(m, r) ...}

value cartesianCirclePackage =
   open CartesianPointPackage as p[Rep] in
      pack[PointRep = Rep in CircleWRT1[PointRep]]
         circleModule[Rep](p)

open cartesianCirclePackage as c0[PointRep] in
open c0 as c[CircleRep] in
  ...c.mkcircle(c.pointPackage.mkpoint(3, 4), 5) ...
```

A Rectangle Package

```
type RectWRT2[RectRep, PointRep] =
  {pointPackage: PointWRT[PointRep],
   mkrect: (PointRep x PointRep) -> RectRep, ...}

type RectWRT1[PointRep] =
   exists RectRep. RectWRT2[RectRep, PointRep]

type Rect =
   exists PointRep. RectWRT1[PointRep]

type RectModule = forall PointRep.
   PointWRT[PointRep] -> RectWRT1[PointRep]

value rectModule: RectModule =
   all[PointRep]
      fun(p: PointWRT[PointRep])
         pack[PointRep' = PointRep
            in RectWRT1[PointRep']]
        {pointPackage = p,
         mkrect = fun(tl: PointRep, br: PointRep) ...}
```

A Figures Package

```
type FiguresWRT3[RectRep, CircleRep, PointRep] -
  {circlePackage: CircleWRT[CircleRep, PointRep],
   rectPackage: RectWRT[RectRep, PointRep],
   boundingRect: CircleRep -> RectRep}

type FiguresWRT1[PointRep] =
   exists RectRep. exists CircleRep.
      FiguresWRT3[RectRep, CircleRep, PointRep]

type Figures =
   exists PointRep. FiguresWRT1[PointRep]

type FiguresModule = forall PointRep.
   PointWRT[PointRep] -> FiguresWRT1[PointRep]

value figuresModule: FIguresModule =
   all[PointRep]
      fun(p: PointWRT[PointRep])
         pack[PointRep' = PointRep
            in FiguresWRT1[PointRep]]
         open circleModule[PointRep](p) as c[CircleRep] in
            open rectModule[PointRep](p) as r[RectRep] in
              {circlePackage = c, ...}
```

Bounded Quantification

Subtyping

- Type `A` is *subtype* in type `B` when all values of A may also be considered values of B.
- Subtyping on subranges, records, variants, function, universally and existentially quantified types.

Subtyping Records and Variants

Subtyping records

- Width subtyping: $R_1 \leq R_2$ iff R_1 has more fields than R_2
- Depth subtyping: $R_1 \leq R_2$ iff, for all fields of R_2, the type of the field in R_1 is a subtype of the corresponding field in R_2.
- Example: $\{a : \texttt{int}, b : \texttt{int}\} <: \{a : \texttt{double}\}$

Subtyping variants

- Width subtyping: $V_1 \leq V_2$ iff V_1 has fewer fields than V_2
- Depth subtyping: $V_1 \leq V_2$ iff, for all fields of V_1, the type of the field in V_1 is a subtype of the corresponding field in V_2.
- Example: $[a : \texttt{int}] <: [a : \texttt{double}, b : \texttt{int}]$

Subrange and Functions

Integer subrange type `n..m`

- `n..m` <: `n'..m'` iff $n' \leq n \wedge m \leq m'$
- ```
  value f = fun(x: 2..5) x+1
  f: 2..5 -> 3..6
  f(3)
  value g = fun(y: 3..4) f(y)
  ```

Function type

- `s -> t <: s' -> t'` iff `s' <: s` and `t <: t'`
- Function of type `3..7 -> 7..9` can be also considered as function of type `4..6 -> 6..10`
  ```

Bounded Quantification and Subtyping

- Mix subtyping and polymorphism.

```
value f0 = fun(x: {one: Int}) x.one
f0({one = 3, two = true})

value f = all[a] fun(x: {one: a}) x.one
f[Int]({one = 3, two = true})
```

- Constraint `all[a <: T] e`

```
value g0 = all[a <: {one: Int}] fun(x: a) x.one
g0[{one:Int, two:Bool}]({one=3, two=true})
```

- Two forms of inclusion constraints:
  - In `f0`, implicit by function parameters.
  - In `g0`, explicit by bounded quantification.
  - Type expressions:

    ```
    g0: forall a <: {one: Int}. a -> Int
    ```
  - Type abstraction:

    ```
    value g = all[b] all[a <: {one: b}] fun(x:a)x:one
    g[Int][({one:Int,two:Bool})]({one=3,...})
    ```

Object Oriented Programming

```
type Point = {x: Int, y: Int}

value moveX0 =
   fun(p: Point, dx: Int) p.x := p.x + dx; p
value moveX =
   all[P <: Point] fun(p:P, dx: Int) p.x := p.x + dx; p

type Tile = {x: Int, y: Int, hor: Int, ver: Int}
moveX[Tile]({x = 0, y = 0, hor - 1, ver = 1}, 1).hor
```

- Result of `moveX` is same as argument type.
- `moveX` can be applied to objects of (yet) unknown type.

Bounded Existential Quantification and Partial Abstraction

- Bounding existential quantifiers:
  - `exists a <:  t.  t'`
  - `exists a.  t` is short for `exists a <:  Top.  t`
- Partially abstract types:
  - `a` is abstract.
  - We know `a` is subtype of `t`.
  - `a` is not more abstract than `t`.
- Modified packing construct:

  `pack [a <= t = t' in t"] e`

Points and Tiles

```
type Tile = exists P. exists T <= P. TileWRT2[P, T]

type TileWRT2[P, T] =
  {mktile: (Int x Int x Int x Int) -> T,
   origin: T -> P,
   hor: T -> Int,
   ver: T -> Int}

type TileWRT[P] = exists T <= P. TileWRT2[P, T]
type Tile = exists P. TileWRT[P]

type PointRep = {x: Int, y: Int}
type TileRep = {x: Int, y: Int, hor: Int, ver: Int}

pack [P = PointRep in TileWRT[P]]

pack [T <= PointRep = TileRep in TileWRT2[P, T]]
  {mktile = fun(x:Int, y: Int, hor: Int, ver: Int)
     {x=x, y-y, hor=hor, ver=ver},
      origin = fun(t: TileRep) t,
      hor = fun(t: TileRep) t.hor,
      ver = fun(t: TileRep) t.ver}

fun(tilePack: Tile)
   open tilePack as t[pointRep][tileRep]
      let f = fun(p: pointRep) ...
      in f(t.tile(0, 0, 1, 1))
```

Summary

- Three main principles
  - Universal type quantification (polymorphism).
  - Existential type quantification (abstraction).
  - Bounded type quantification (subtyping).
- Resulting programs may be statically type-checked.
  - Bottom-construction of types.
  - More sophisticated type inference possible (ML).
- More general type systems.
  - Type-checking typically not decidable any more.
  - Dependent types (Martin-Löf).
  - Calculus of constructions (Coquand and Huet)..