On Understanding Types, Data Abstraction, and Polymorphism

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

From Untyped to Typed Universes

- Untyped universes.
  - Only one type for all objects:
  - Bit strings in computer memory,
  - S-expressions in pure LISP,
  - λ-expressions in the λ-calculus,
  - Sets in set theory.
- Organization of objects:
  - Classification of usage/behavior,
  - Characters, integers,
  - Lists, pairs,
  - Functions, programs.
- Type distinction still shallow:
  - Easy to violate type distinctions:
  - Boolean or of character and machine operation?

Static and Strong Typing

- Types impose constraints to enforce correctness.
  - Protection of underlying (untyped) representation.
  - Constrains interaction of objects.
- *Static* type structure of programs.
  - Little or no type information given explicity.
  - Types associated to constants/function symbols.
  - *Type inference system* infers types of expressions by static analysis.
- *Strong* typing system.
  - Expressions are guaranteed to be type consistent.
  - Type itself may be statically unknown.
  - Introduce *run time* type checking.

Kinds of Polymorphism

- *Monomorphic* languages:
  - All functions and procedures have unique type.
  - All values and variables of one and only type.
  - Pascal-like type systems.
- *Polymorphic* languages:
  - Values and variables may have more than one type.
  - Polymorphic functions have operatnds of more than one type.
  - Polymorphic types have operations applicable to 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.
  - *Overlading* and *coercion*.

Universal Polymorphism

- *Parametric* polymorphism:
  - Uniformity achieved by *type parameters*.
  - Determines argument type for each application of polymorphic function.
  - ML-like type systems.
- *Inclusion* polymorphism:
  - Object may belong to several types.
  - Types related by inclusion relation.
  - Object-oriented type systems.
- *Generic* functions:
  - “Same” work is done for arguments of many types.
  - Length function over lists.

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!

For history of type evolution see [Cardelli and Wegner, 1985].

Preview of Fun

- λ-calculus based language
  - Basis is first-order typed λ-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*
  - Type ::= ... | Quantified Type
  - Quantified Type ::=
    $\forall$A. Type |
    $\exists$A. Type |
    $\forall$A $\subseteq$Type. Type | $\exists$A $\subseteq$Type. Type
- Modeling of advanced type systems:
  - *Universal* quantification: parameterized types.
  - *Existential* quantifiers: abstract data types.
  - *Bounded* quantification: type inheritance.

The Untyped λ-Calculus

- Expressions:
  - e ::= x
  - e ::= fun(x) e
  - e ::= e(e)
- Introduction of names
  - value id = fun(x) x
  - value succ = fun(x) x+1
  - value twice = fun(f) fun(y) f(f(y))

The Typed λ-Calculus

- Extension of Untyped λ-Calculus
  - Every variable must be explicitly typed when introduced as typed variable
  - Result types can be deduced from function body.
- Examples
  - value succ = fun(x:Int) x+1
  - value twice = fun(f: Int $\rightarrow$ Int) fun(y:Int) f(f(y))
- Type declarations:
  - type IntPair = Int $\times$ Int
  - type IntFun = Int $\rightarrow$ 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

Basic and Structured Types

- Basic types:
  - Unit (trivial type, only element ())
  - Bool (with if-then-else)
  - Int (with arithmetic and comparison)
  - Real (with arithmetic and comparison)
  - String (with infix concatenation ^)
- Type constructors:
  - $\rightarrow$ (function space)
  - $\times$ (Cartesian product)
  - record types (labeled Cartesian products)
  - variant types (labeled disjoint sums)
- Example:
  - value p: Int $\times$ Bool = 3, true
    fst(p), snd(p)

Record Types

- Example:
  - type ARec = {a: Int, b: Bool}
    value r: ARec = {a = 3, b = true}
    r.b
- Functions as components:
  - type FunRec = {f1: Int → Int, f2: Real → Real}
    value funRec: FunRec = {f1 = succ, f2 = sin}
- Concatenation of record types:
  - type NewRec = FunRec & {f3: Bool → Bool}
- Private data structures:
  - value counter =
    ```
        let count = ref(0) in
        { inc = fun(n:Int) count := count+n
          total = fun() count
        }
    ```
  - counter.inc(3), counter.total()

Variant Types and Recursion

- Example:
  - type AVar = [a: Int, b: String]
    value v1: AVar = [a = 3]
    value v2: AVar = [b = "abcd"]
  - value f = fun(x: AVar)
      case x of
        [a = anInt] "int"
        [b = aString] "string" ^ aString
        otherwise "error"
- Recursive function definitions
  - rec value fact =
      fun(n: Int) if n=0 then 1 else n*fact(n-1)
- Recursive type definitions
  - rec type IntList =
      [nil: Unit
        cons: {head: Int, tail: IntList}]

Universal Quantification

- Typed λ-calculus describes monomorphic functions.
  - Unsufficient to describe functions that behave the same way for argumentes of different types.
- Introduce types as parameters:
  - value id = all[a] fun(x:a) x
    id[Int](3)
- May omit type information:
  - value id = fun(x:a) x
    id(3)
  - Type-checker reintroduces all[a] and [Int]
- Polymorphic types:
  - type GenericId = $\forall$a. a $\rightarrow$ a
    id: GenericId
  - value inst = fun(f: $\forall$a. a $\rightarrow$ a)(f[Int], f[Bool])
  - value intid: Int $\rightarrow$ Int = fst(inst(id))
    value boolid: Bool $\rightarrow$ Bool = snd(inst(id))

Polymorphic Functions

- First version of polymorphic twice:
  - value twice1 = all[t] fun(f: ∀a: a → a)
    fun(x: t) f[t](f[t](x))
  - twice1[Int](id)(3) is legal.
  - twice1[Int](succ) is illegal!
- Second version of polymorphic twice:
  - value twice2 = all[t] fun(f: t → t) fun(x: t)
    f(f(x))
  - twice2[Int](succ) is legal.
  - twice2[Int](id[Int])(3) is legal.
- Both versions different in nature of f:
  - In twice1, f is a polymorphic function of type ∀a: a → a.
  - In twice2, f is a monomorphic function of type $t \to t$ (for some instantiation of t)

Parametric Types

- Type definitions with similar structure:
  - type BoolPair = Bool × Bool
  - type IntPair = Int × Int
- Use *parametric* definition:
  - type Pair[T] = T × T
  - type PairOfBool = Pair[Bool]
  - type PairOfInt = Pair[Int]
- Type operators are *not* types:
  - type A[T] = T → T
  - type B = ∀T. T → T
  - Different notions!

Recursive Definitions

- Recursively defined type operators:
  - rec type List[Item] =
    [nil: Unit
    cons: {head: Item, tail: List[Item]}]
  - value nil: ∀Item.List[Item] = all[Item]. [nil = ()]
    value intNil: List[Int] = nil[Int]
  - value cons:
    ∀Item. (Item × List[Intem]) → List[Item] =
    all[Item].
    fun(h Item, t: List[Item])
    [cons = {head = h, tail = t}]

Cons can be only used to build homogeneous lists!

Existential Quantification

- Existential type quantification:
  - p: $\exists$a. t(a)
  - For some type a, p has type t(a)
- Examples:
  - (3, 4): $\exists$a. a $\times$ a
  - (3, 4): $\exists$a. a
  - Constant can satisfy different existential types!
- Sample existential types:
  - type Top = $\exists$a. a (type of any value)
  - $\exists$a. $\exists$b. a $\times$ b (type of any pair)
- Particularly useful:
  - x: $\exists$a. a $\times$ (a $\rightarrow$ Int)
  - (snd(x))(fst(x))

Information Hiding

- *Abstract* types:
  - Unknown type representation.
  - Packaged with operations that may be applied to type.
- Another example:
  - x: $\exists$a. {const: a, op: a $\rightarrow$ Int}
  - x.op(x.const)
- Restrict use of abstract types:
  - Simplify type checking.
  - value p: $\exists$a. a $\times$ (a $\rightarrow$ Int)
    = pack[a = Int in a $\times$ (a $\rightarrow$ Int)](3, succ)
  - Value p is a *package*
  - Type a $\times$ (a $\rightarrow$ 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 $\times$ (a $\to$ Int)](3, succ)
    open p as x in (snd(x))(fst(x))
  - value p = pack[a = Int in
    {arg: a, op: a $\to$ Int}](3, 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) ...

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 × Real
type Point1 =
  {makepoint: (Real × Real) → Point,
    x_coord: Point × Real,
    y_coord: Point × 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;
```

Package specification and body are part of value specification!

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 Point2 =
  ∃Point.
    {makepoint: (Real × Real) → Point,
      ...}
type Point2WRT[Point] =
    {makepoint: (Real × Real) → Point,
      ...}
value point2: Point2 = pack[Point = (Real × Real) in
  Point2WRT[Point]] point1
```

Combining Universal and Existential Quantification

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

nil: $\forall$a. List[a]
cons: $\forall$a. (a $\times$ List[a]) $\rightarrow$ List[a]
hd: $\forall$a. List[a] $\rightarrow$ a
tl: $\forall$a. List[a] $\rightarrow$ List[a]
Null: $\forall$a. List[a] $\rightarrow$ Bool

array: $\forall$a. Int $\rightarrow$ Array[a]
index: $\forall$a. (Array[a] $\times$ Int) $\rightarrow$ a
update: $\forall$a. (Array[a] $\times$ Int $\times$ a) $\rightarrow$ Unit

Concrete Stacks

type IntListStack =
  {emptyStack: List[Int],
   push: (Int $\times$ List[Int]) $\rightarrow$ List[Int]
   pop: List[Int] $\times$ List[Int],
   top:List[Int] $\rightarrow$ 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] $\times$ Int),
   push: (Int $\times$ (Array[Int] $\times$ Int)) $\rightarrow$ (Array[Int] $\times$ Int),
   pop: (Array[Int] $\times$ Int) $\times$ (Array[Int] $\times$ Int),
   top:(Array[Int] $\times$ Int) $\rightarrow$ Int}

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

Generic Element Types

```
type GenericListStack =
  ∀Item.
    {emptyStack: List[Item],
     push: (Item × 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 =
  ∀Item. ∃Stack. GenericStackWRT[Item][Stack]

type GenericStackWRT[Item][Stack] =
 {emptyStack: List[Item],
  push: (Item × List[Item]) → List[Item]
  pop: List[Item] × List[Item],
  top: List[Item] → 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)
```

Another Example

```
type Point =
  ∃PointRep.
    {mkPoint: (Real × Real) → PointRep,
     x-coord: PointRep → Real,
     y-coord: PointRep → Real}

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

type Point = ∃PointRep. PointWRT[PointRep]
value cartesianPointOps =
  {mkpoint = fun(x: Real, y: Real) (x,y),
   x-coord = fun(p: Real × Real) fst(p),
   y-coord = fun(p: Real × Real) snd(p)}
value cartesianPointPackage: Point =
  pack[PointRep = Real × Real
    in PointWRT[PointRep]]
  cartesianPointOps
```

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.

Parametric Modules

```
type ExtendedPointWRT[PointRep] =
  PointWRT[PointRep] &
  {add: (PointRep × PointRep) → PointRep}

type ExtendedPoint =
  ∃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 × Real) → CircleRep,
   center: CircleRep → PointRep, ...}
type CircleWRT1[PointRep] =
   ∃CircleRep. CircleWRT2[CircleRep, PointRep]
type Circle = ∃PointRep. CircleWRT1[PointRep]
value circleModule: CircleModule =
all[PointRep]
   fun(p: PointWRT[PointRep])
      pack[CircleRep = PointRep × Real
         in CircleWRT2[CircleRep,PointRep]]
     {pointPackage = p,
      mkcircle = fun(m: PointRep, r: Real)(m, r) ...}
value cartesianCirclePackage =
   openCartesianPointPackage as p[Rep] in
      pack[PointRep = Rep in CircleWRT1[PointRep]]
         circleModule[Rep](p)
open cartesian CirclePackage as c[PointRep][CircleRep]
in ...c.mkcircle(c.pointPackage.mkpoint(3, 4), 5) ...
```

A Rectangle Package

```
type RectWRT2[RectRep, PointRep] =
  {pointPackage: PointWRT[PointRep],
   mkrect: (PointRep × PointRep) → RectRep, ...}

type RectWRT1[PointRep] =
   ∃RectRep. RectWRT2[RectRep, PointRep]
type Rect = ∃PointRep. RectWRT1[PointRep]
type RectModule = ∀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] =
  ∃RectRep. ∃CircleRep.
    FigureWRT3[RectRep, CircleRep, PointRep]
type Figures = ∃PointRep. FIgureWRT1[PointRep]
type FiguresModule = ∀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

- Type inclusion:
  - Type A is *included in* (*a subtype of*) type B when all values of A are also values of B.
  - Inclusion relation on subranges, records, variants, function, universally and existentially quantified types.
- Integer subrange type $n \ldots m$
  - $n \ldots m \leq n' \leq m'$ iff $n' \leq n$ and $m \leq m'$
  - 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
  - $s \to t \leq s' \to t'$ iff $s' \leq s$ and $t \leq t'$
  - Function of type 3 ... 7 can be also considered as function of type 4 ... 6 →6 ... 10

Record Types and Inheritance

- Record type:
  - $\{a_1 : t1, \ldots, a_n : t_n, \ldots, a_m : t_m\} \leq \{a_1 : u_1, \ldots, a_n : u_n\}$ iff $t_i \leq u_i$ (for $i \in 1 \ldots$ n)
  - A is subtype of B if A has all the fields of B (possibly more) and the types of the common fields are subtypes.
- Concept of *inheritance* (*subclasses*)
  - Records may have functional components.
  - Class instance is record with functions and local variables.
  - Subclass instance is record with at least those functions and variables.
- Variant types:
  - $[a_1 : t1, \ldots, a_n : t_n]$
    $\leq [a_1 : u_1, \ldots, a_n : u_n, \ldots, a_m : u_m]$ iff $t_i \leq u_i$ (for $i \in$ 1 ... n)

Bounded Quantification 'n Subtyping

- Mix subtyping and polymorphism.
  - value f_0 = fun(x: {one: Int}) x.one
    f_0({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 g_0 = all[a ≤ {one: Int}] fun(x: a) x.one
    g_0[{one:Int, two:Bool}]({one=3, two=true})
- Two forms of inclusion constraints:
  - In f_0, implicit by function parameters.
  - In g_0, explicit by bounded quantification.
- Type expressions:
  - g_0: $\forall a \leq$ {one: Int}. $a \rightarrow$ 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 moveX$_0$ =
  fun(p: Point, dx: Int) p.x := p.x + dx; p
value moveX =
  all[P $\leq$ 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:
  - ∃a ≤t. t'
  - ∃a. t := ∃a ≤Top. t
- *Partially abstract* types:
  - a is abstract.
  - We know a is subtype of t.
  - a is not more abstract than t is.
- Modified packing construct:
  - pack [a ≤ t = t' in t''] e

Points and Tiles

type Tile = $\exists$P. $\exists$T $\leq$ P. TileWRT2[P, T]

type TileWRT2[P, T] =
{mktile: (Int $\times$ Int $\times$ Int $\times$ Int) $\rightarrow$ T,
origin: T $\rightarrow$ P,
hor: T $\rightarrow$ Int,
ver: T $\rightarrow$ Int}

type TileWRT[P] = $\exists$T $\leq$ 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}

Points and Tiles

```
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)..