# Types in Programming Languages between Modelling, Abstraction, and Correctness

*Simone Martini*

Dipartimento di Informatica – Scienza e Ingegneria
*Alma mater studiorum* • Università di Bologna
and
INRIA Sophia / Bologna

CiE 2016
Paris, June 28, 2016

Trace the evolution of types in programming languages, identifying some of the driving forces of this process, (in dialogue with mathematical logic.)

First episode:
HaPOC 2015, Pisa
From 1955 to 1970 (circa)

# Why types?

Modern programming languages:

- control flow specification: small fraction
- abstraction mechanisms to model application domains.

## We today conflate:

- Types as an implementation (representation) issue
- Types as an abstraction mechanism
- Types as a classification mechanism (from mathematical logic)

Prehistory

1949

# H.B. Curry, 1949

Types for memory words:

- containing instructions: *orders*
- containing data: *quantities*

*Memoranda of Naval Ordnance Laboratory*

*[see De Mol, Carlé, and Bullyinck, JLC 2015]*

*Mathematical theory of programs*

*Theorems in the "well-typed expressions do not go wrong" style*

*G.W. Patterson's review on JSL 22(01), 1957, 102-103*

*No known subsequent impact*

# Types as an implementation issue

# 1950s and 1960s

- Type based distinctions for compilation: always present
- "Type" as a technical term: Algol 58
- (Almost) stable since Algol 60

- Mode
  - in Algol 68, *d'après* early Fortran usage
  - "types (or modes)", still in Reynolds 1975

# 1950s and 1960s

- Type based distinctions for compilation: always present
- "Type" as a technical term: Algol 58
- (Almost) stable since Algol 60

- Mode
  - in Algol 68, *d'après* early Fortran usage
  - "types (or modes)", still in Reynolds 1975

## The word: “type”

*The use of ‘type,’ as in ‘x is of type* **real**, *’ was analogous to that employed in logic.*

*Both programming language design and logic dipped into the English language and came up with the same word for roughly the same purpose.*

*[A. Perlis, The American side of the development of Algol, 1981]*

# OT Intermezzo: Perlis on the Algol Report

> *Nicely organized, tantalizingly incomplete, slightly ambiguous, difficult to read, consistent in format, and brief, it was a perfect canvas for a language that possessed those same properties.*
>
> *Like the Bible, it was meant not merely to be read, but to be interpreted.*
>
> *[A. Perlis, The American side of the development of Algol, 1981]*

# OT Intermezzo: Perlis on the Algol Report

*Nicely organized, tantalizingly incomplete, slightly ambiguous, difficult to read, consistent in format, and brief, it was a perfect canvas for a language that possessed those same properties.*

*Like the Bible, it was meant not merely to be read, but to be interpreted.*

*[A. Perlis, The American side of the development of Algol, 1981]*

These types:

- guide the compiler
- are a reasonable abstraction over implementation details (*with many exceptions*)

However:

- no provision for other data, but integer, real, Boolean

# Types as an abstraction mechanism

# The needs:

1. from simple to structured values
2. a general modelling tool
3. user definable "extensions"
4. robust abstractions over the representation

# The arrival point

*Type structure is a syntactic discipline
for enforcing levels of abstraction* *[John Reynolds, 1983]*

The two champions:

- Tony Hoare
- Ole-Johan Dahl and Kristen Nygaard

The two champions:

- Records
- Objects

The two champions for data abstraction:

- Abstract data encapsulation
- Procedural encapsulation

# Hoare: records and references

AB21.3.6     <u>RECORD HANDLING</u>

C. A. R. Hoare

Entia non sunt multiplicanda praeter necessitatem -

William of Occam.

1964:

- ordered collection of named *fields*: record classes
- typed references (like pointers, but no operations)
- non stack-based, dynamically allocated structures

# Dahl and Nygaard: objects *ante litteram*

around 1962:

- record class: *activity*;
- record: *process*;
- record field: *local variable of a process*

- a “process” encapsulates both data objects and their operators: a *closure*

# They both make into languages

- Algol W, circa 1970 (and then Pascal, and then . . . )
- Simula 67

Never seen as rivals (on the contrary: many collaborations)

Are the records to have immediate impact

Records beat Objects

1-0

## They both make into languages

- Algol W, circa 1970 (and then Pascal, and then . . . )
- Simula 67

Never seen as rivals (on the contrary: many collaborations)

Are the records to have immediate impact

Records beat Objects

1-0

# Simula 67

## Classes/Objects

- Record class $\rightarrow$ Object class $\rightarrow$ Class
- "declared quantity (class)" vs
  "its dynamic offspring (objects)"

## Subclasses

- Hoare 1966, Villard-de-Lans Summer School:
  - record subclasses
  - dot notation
  - $\Rightarrow$ pure data abstraction
- Simula 67: Prefixing (subclassing)
  - code of the subclass is "permanently glued together" the code of the superclass
  - data and operations

# Example

From Dahl's recollection:

- Queuable ("Link"):
  next/precedessor in queue
- Car
  subclass of Queuable
- Truck and Bus
  both subclasses of Car

## A further emerging need

### Correctness of programs

- Floyd, Assigning meanings to programs, 1967
- Hoare, An axiomatic basis for computer programming, 1969
- Burstall, Proving properties of programs by structural induction, 1969
- McCarthy and Painter, Correctness of a compiler for arithmetic expressions, 1967

# A further emerging need

## Correctness of programs

- Floyd, Assigning meanings to programs, 1967
- Hoare, An axiomatic basis for computer programming, 1969
- Burstall, Proving properties of programs by structural induction, 1969
- McCarthy and Painter, Correctness of a compiler for arithmetic expressions, 1967

## The Algol research program

Mark Priestly, A Science of Operations, Springer 2011

Algol 60 was not particularly successful in practical terms.
However. . .

- A coherent and comprehensive research programme
- Algol 60 report: a paradigmatic (à la Kuhn) achievement
- First theoretical framework for studying:
  - the design of programming languages,
  - the process of software development.

Several attempts
towards general mechanisms for data definition

# Extensible languages

## Explicit definitions

- Galler and Perlis, A proposal for definitions in ALGOL, CACM 10, 1967
- Schuman and Jorrand, Definition mechanisms in extensible programming languages.
  Proc. AFIPS, Vol. 37, 1970.

# Standard Abstract Operations

## Representation and representation independence

- Levels of systems, each represented on the other:
- $\rho$ represents $D$ onto $D'$
- $\pi$ procedure on $D$ data
- The correspondence guarantees that representation and implementation commute.

[Mealy, Another look at data. Proc. AFIPS, Vol. 31, 1967.]

we require that the following diagram be *commutative:*

$$
\begin{tikzcd}
D \arrow[r, "\rho"] \arrow[d, "\pi"'] & D' \arrow[d, "\pi'"] \\
\overline{D} \arrow[r, "\rho"] & \overline{D}'
\end{tikzcd}
$$

# Standard Abstract Operations, 2

*The programmer should be able to construct his program in terms of the logical processing required without regard to either the representation of data or the method of accessing and updating. This concept we call "Dataless programming".*

*[Balzer, Dataless programming. Proc. AFIPS, Vol. 31, 1967.]*

Abstract procedures to handle representation:
**create**, **access**, **modify**, and **destroy** abstract data collections.

# Information hiding

**Parnas 1972**

- a stable interface towards the rest of the program
- to protect those design choices which are bound to change
- a general design methodology, which applies to types, modules, packages, etc.

**From the programming language community:**

information hiding enforced by linguistic abstractions,
and not merely by a design methodology.

# Information hiding

**Parnas 1972**

- a stable interface towards the rest of the program
- to protect those design choices which are bound to change
- a general design methodology, which applies to types, modules, packages, etc.

**From the programming language community:**

information hiding enforced by linguistic abstractions,
and not merely by a design methodology.

# Towards ADTs

## Morris, 1973 and Reynolds, 1974

*The meaning of a syntactically-valid program in a "type-correct" language should never depend upon the particular representation used to implement its primitive types.*

*The main thesis of [Morris 1973] is that this property of representation independence should hold for user-defined types as well as primitive types.*

*[Reynolds, 1974]*

A ready-made abstraction mechanism:

procedures
(closures)

# Procedural encapsulation

*Procedures, particularly procedures which can return procedures as their result, are the proper mechanism for modularizing both programs and data.*

*Procedural encapsulation: representing system components in terms of one or more procedures such that interactions among components are limited to procedure calls.*

*Similar to classes in SIMULA 67. Unlike SIMULA, however, the local variables [. . . ] are not made accessible outside the procedure in which they are defined.* *[Zilles, 1973]*

## Moral, 1

**From our perspective, post festam:**

Simula's classes,

extended with a visibility mechanism protecting local variables from outside access,

provide a good encapsulation abstraction.

No need of a separate abstraction mechanisms:
use *closures*: code + environment

*But this is not what happened back then...*

# Moral, 1

From our perspective, post festam:

Simula's classes,

extended with a visibility mechanism protecting local variables from outside access,

provide a good encapsulation abstraction.

No need of a separate abstraction mechanisms:
use *closures*: code + environment

*But this is not what happened back then...*

# Abstract Data Types

## Liskov and Zilles, 1974 ff

- Public part:

  | | |
  |---|---|
  | name | `complex` |
  | operations | `create, add, get-x, get-y, equal` |

- Private part:

  representation for type
  implementation of operations

- Inside the private part: representation is accessible
- Outside the private part: representation is inaccessible

# A CLU cluster

```
complex = cluster is create, add, get-x, get-y, equal

rep = struct[x, y: real]

create = proc (x, y: real) returns (cvt)
         return(rep$[x: x, y:y])
         end create

add = proc (a, b: cvt) returns (cvt)
      return(rep$[x: a.x + b.x, y: a.y + b.y])
      end add
...
end complex
```

[CLU Reference Manual, LNCS 114, 1981]

# Data encapsulation

A specific abstraction mechanism enforces information hiding, and then guarantees representation independence.

*Abstract types are intended to be very much like the built-in types provided by a programming language.*

*An ADT defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type.* *[Liskov and Zilles, 1974]*

## Semantics of ADTs: Algebras

- An ADT is an abstract algebra, where "abstract" means unique up to isomorphism.
- A representation is a concrete many sorted algebra
- The presentation of an abstract algebra, is *the* initial algebra in a certain class.

$$
\begin{tikzcd}[row sep=large]
\text{data element:} & 110110 \arrow[d, "\text{element of}"] \\
\text{data set:} & \{0,1,10,11,100,...\} \arrow[d, "\text{carrier of}"] \\
\text{concrete algebra:} & B = \langle \texttt{BiNat}, 0, succ \rangle \arrow[d, "\text{element of}"] \\
\text{abstract algebra:} & \mathcal{B} = \text{isomorphism class of } B
\end{tikzcd}
$$

# Initial algebras

- J. Goguen, Some remarks on data structures, *unpublished* notes of ETH course, 1973.
- ADJ, Abstract data types as initial algebras (...), IEEE 1975 ff
- J. Guttag, PhD thesis Toronto, 1975

- Equations would give correctness constraints
- Freeness ensures abstraction
- Freeness allows proofs by (structural) induction

## The seemingly unstoppable march of ADTs

ADTs beat Objects

2-0

## Meanwhile, in the opposite camp

### Smalltalk

- Alan Kay, from 1972
- Simula concept of class and objects
- In a *new metaphor* and design methodology
- To use for “open” systems

*The "official" computer science world started to regard Simula as a possible vehicle for defining abstract data types.*
*To put it mildly, we were quite amazed at this.*

*What Simula had whispered was something much stronger than simply reimplementing a weak and ad hoc idea.*

*You could now replace bindings and assignment with goals.*

*The objects should be presented as sites of higher level behaviors more appropriate for use as dynamic components.*

*[Kay, The early history of Smalltalk, 1993]*

## Someone noticed, though

John Reynolds:

User-defined types and procedural data structures as complementary approaches to data abstraction
in *New Directions in Algorithmic Languages, 1975*

User-defined types = ADTs
Procedural data structures = Procedural encapsulation
(= Objects)

Do not cite Simula
Cites Hoare and Dahl; Balzer's Dataless programming

# ADTs vs Procedural abstraction

## ADTs

- Centralized implementation
- All operations defined together with implementation

## Procedural abstraction

- Decentralized implementation: each value is independent
- Operations are attached to the value they act upon

Procedural approaches: easier to extend !

# ADTs vs Procedural abstraction

## ADTs

- Centralized implementation
- All operations defined together with implementation

## Procedural abstraction

- Decentralized implementation: each value is independent
- Operations are attached to the value they act upon

Procedural approaches: easier to extend !

# ADTs, extension, compatibility

```
type C{
  fun m(c:C){}
}
type D{
  fun m(d:D){modified wrt to C}
  fun op(d:D){}
}
```

Clearly `D<:C` (by "Liskov substitution principle").
Hence for any `d:D`, we have `d:C`.

We process a list `L` of elements of type `C`:

```
L : list(C)
foreach e in L:
   m(e)
```

When `e:D` this breaks abtraction.

## Objects, extension, compatibility

```
class C{
  meth m(c:C){}
}
class D{
  meth m(d:D){modified wrt to C}
  meth op(d:D){}
}
```

We process a list `L` of elements of type `C`:

```
L : list(C)
foreach e in L:
   m(e)
```

Late binding: which `m` is called depends on the actual class of `e`

## Object oriented languages

### The key ingredients

- Abstraction: to pack data and code
- Inheritance: reuse of implementations
- Subtyping: compatibility of interfaces
- Late binding: to reconcile all of them

A research question:

Both Simula and Smalltalk were designed for specific application domains.

How this influenced their characteristics?

## The 80s and 90s

Objects beat ADTs

3-2

C++ (C with classes, 1979, after Simula)
Java (Oak, 1991)
Javascript (Mocha, 1995)

# The 80s and 90s

Objects beat ADTs

3-2

C++ (C with classes, 1979, after Simula)
Java (Oak, 1991)
Javascript (Mocha, 1995)

In programming language design, types:

are proposed as an enabling feature (Voevodsky),
allowing simpler writing of programs,
and better verification of their correctess.

The Algol research program is still at work. . . :-)

# Sipario

The history of computer science is innervated by the continuous tension between formal beauty and technological effectiveness. Types in programming languages are an evident example of this dialectics.

We always exploited what we found useful for the design of more elegant, economical, usable artefacts.