Towards a conceptual history of programming languages : Types

Simone Martini

Collegium - Institut d'études avancées de Lyon 2018-2019
and
Dipartimento di Informatica – Scienza e Ingegneria, Bologna
and
INRIA FoCUS – Sophia / Bologna

séminaire LSC, LIP – October 4, 2018

COLLEGIUM
DE LYON
Université de Lyon

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

The big project

Reflect and trace the interaction of mathematical logic and programming (languages), identifying some of the driving forces of this process.

First episode: Types
HaPOC 2015, Pisa: from 1955 to 1970 (circa)
Cie 2016, Paris: from 1965 to 1975 (circa)

Towards a *Conceptual History of Programming Languages*
Collegium - Lyon Institute for Advanced Studies, 2018-2019

Why types?

Modern programming languages:

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

- Types are a crucial building block of these abstractions
- And they are a mathematical logic concept, aren't they?

Why types?

Modern programming languages:

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

- Types are a crucial building block of these abstractions
- And they are a mathematical logic concept, aren't they?

We today conflate:

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

One of the goals:
separate them and identify when they arrive in the PL literature

We today conflate:

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

One of the goals:
separate them and identify when they arrive in the PL literature

Framing it in a larger context

The quest for a “Mathematical Theory of Computation”

How does mathematical logic fit into this theory?

And for what purposes?

Framing it in a larger context

The quest for a "Mathematical Theory of Computation"

How does mathematical logic fit into this theory?

And for what purposes?

Prehistory

1947 “‘‘

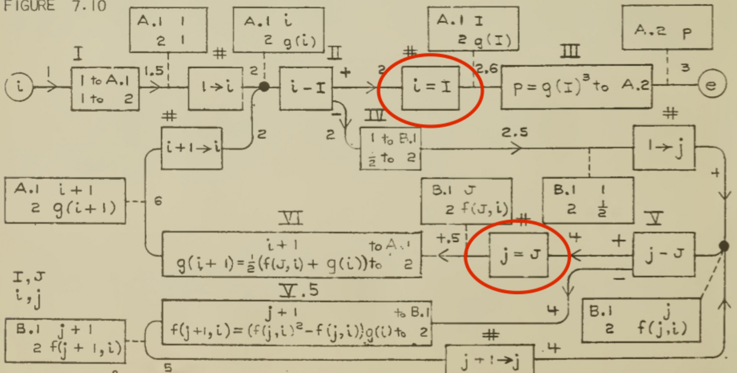
Goldstine and von Neumann

[. . .] coding [. . .] has to be viewed as a logical problem and one that represents a new branch of formal logics.

Hermann Goldstine and John von Neumann
Planning and Coding of problems for an Electronic Computing Instrument
Report on the mathematical and logical aspects of an electronic computing instrument,
Part II, Volume 1-3, April 1947. Institute of Advanced Studies.

Boxes in flow diagrams

FIGURE 7.10



$p = g(I)^3$,

$g(1) = 1$, $g(i+1) = \frac{1}{2}\,(f(J,i) + g(i))$,

$f(1,i) = \frac{1}{2}$, $f(j+1,i) = (f(j,i)^2 - f(j,i))\,g(i)$.

Goldstine and von Neumann, 2

Boxes in flow diagrams

- operation boxes
- substitution boxes
- assertion boxes

The contents of an assertion box are one or more relations.

An assertion box [. . .] indicates only that certain relations are automatically fulfilled whenever [the control reaches that point]

Free and bound variables, etc.

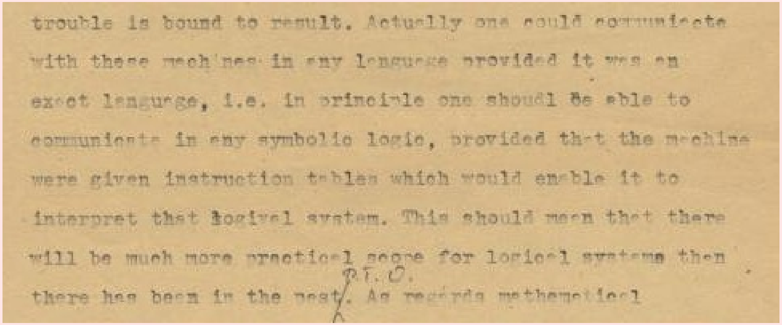
Goldstine and von Neumann

Logic as the discipline to prove assertions

Lecture on Automatic Computing Engine

London Mathematical Soc., 20 Feb 1947. Typewritten notes, in Turing Archive, AMT/C/32

High-level languages



trouble is bound to result. Actually one could communicate with these machines in any language provided it was an exact language, i.e. in principle one should be able to communicate in any symbolic logic, provided that the machine were given instruction tables which would enable it to interpret that logical system. This should mean that there will be much more practical scope for logical systems than there has been in the past. As regards mathematical

P.T.O.

Turing

Lecture on Automatic Computing Engine

London Mathematical Soc., 20 Feb 1947. Typewritten notes, in Turing Archive, AMT/C/32

High-level languages

In principle one should be able to communicate [with these machines] in any symbolic logic [. . .].

This would mean that there will be much more practical scope for logical systems than there has been in the past.

Turing

Logic as the discipline of formal languages

They both see a bright future. . .

Goldstine and von Neumann:

A logical problem [. . .] that represents a new branch of formal logics.

Turing:

There will be much more practical scope for logical systems.

Turing, again: 1949

The programmer should make assertions about the various states that the machine can reach.

The checker has to verify that [these assertions] *agree with the claims that are made for the routine as a whole.*

Finally the checker has to verify that the process comes to an end.

A.M. Turing. Checking a large routine. Paper read on 24 June 1949 at the inaugural conference of the EDSAC computer at the Mathematical Laboratory, Cambridge.
Discussed by Morris and Jones, Annals of the History of Computing, Vol. 6, Apr. 1984.

Types: 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]

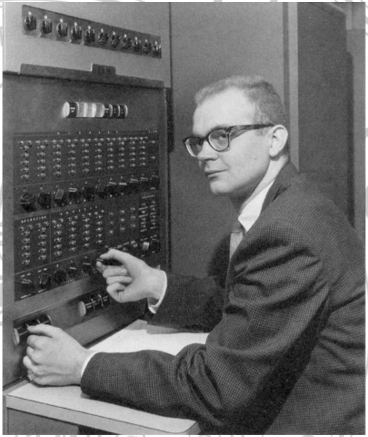
Mathematical theory of programs
Theorems in the style of: "well-typed expressions do not go wrong"

G.W. Patterson's review on JSL 22(01), 1957, 102-103
No known subsequent impact

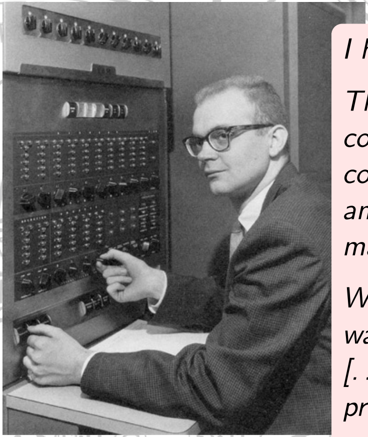
However,

Programming in the fifties (and later. . .) was a different story. . .

Knuth’s recollection, circa 1962



Knuth's recollection, circa 1962



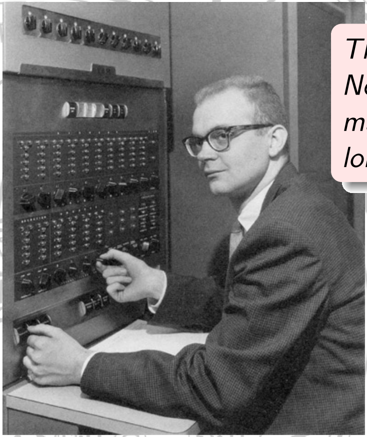
I had never heard of "computer science"

The accepted methodology for program construction was [. . .]: People would write code and make test runs, then find bugs and make patches, then find more bugs and make more patches, and so on.

We never realized that there might be a way to construct a rigorous proof of validity [. . .] even though I was doing nothing but proofs when I was in a classroom

[D.K. Knuth, Robert W. Floyd, in memoriam. ACM SIGACT News 2003]

Knuth's recollection, circa 1962



The early treatises of Goldstine and von Neumann, which provided a glimpse of mathematical program development, had long been forgotten.

Donald E. Knuth

Born, 1938
Bachelor and Master of science:
    Physics, Mathematics, 1960
PhD Mathematics, 1963
Stanford University, since 1968
The Art of Computer Programming: 1968 - today
Turing Award, 1974 (he was 36. . .)

A Mathematical Theory of Computation

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last.

John McCarthy, MIT 1961; Stanford 1963

From the conclusion of the final version of the paper (1963): A Basis for a Mathematical Theory of Computation. 1961: the Western Joint Computer Conference; 1962: IBM symposium in Blaricum, Netherlands; 1963: in *Computer Programming and Formal Systems*, North Holland.

A Mathematical Theory of Computation

It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last.

John McCarthy, MIT 1961; Stanford 1963

From the conclusion of the final version of the paper (1963): A Basis for a Mathematical Theory of Computation. 1961: the Western Joint Computer Conference; 1962: IBM symposium in Blaricum, Netherlands; 1963: in *Computer Programming and Formal Systems*, North Holland.

John McCarthy

Born, 1927
Died, 2011

Bachelor of science: Mathematics, 1942
PhD Mathematics, 1951
MIT, 1956–1962
Stanford University, since 1963
Turing Award, 1971

Time sharing (MIT,BBN) Artificial intelligence
Lisp

Which matematics for computing?

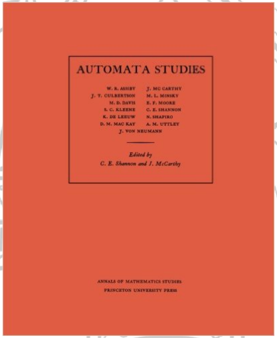
Numerical analysis

Roundoff errors in matrix computation: $Ax = b$

- Turing
- Goldstine & von Neumann: solve $A'Ax = A'b$, for A' transpose of A

Jim Wilkinson (Turing Aw. 1970): backward error analysis

Which matematics for computing?



AUTOMATA STUDIES

W. R. ASHBY
J. T. CULBERTSON
M. D. DAVIS
S. C. KLEENE
K. DE LEEUW
D. M. MAC KAY
J. MC CARTHY
M. L. MINSKY
E. F. MOORE
C. E. SHANNON
N. SHAPIRO
A. M. UTTLEY
J. VON NEUMANN

Edited by
C. E. Shannon and J. McCarthy

ANNALS OF MATHEMATICS STUDIES
PRINCETON UNIVERSITY PRESS

Automata theory

McCulloch and Pitts (1943)

Kleene ("regular events"), Nerode, Myhill, Shepherdson

Automata Studies, Shannon and McCarthy (eds) [Davis, Kleene, Minsky, Moore, etc.] Princeton Univ Press, 1956

Rabin and Scott. Finite Automata and their decision problems. IBM J. 1959

A basis for a Mathematical Theory of Computation

Expected practical Results:

1. To develop a universal programming language

   "Universal" = machine independent and general

2. To define a theory of the equivalence of computation processes

   Define equivalence-preserving transformations: optimization, compilation, etc.

A basis for a Mathematical Theory of Computation

Expected practical Results:

3. To represent algorithms by symbolic expressions in such a way that significant changes in the behavior represented by the algorithms are represented by simple changes in the symbolic expressions.

   Learning algorithms, whose modifiable behavior depends on the value of certain registers.

A basis for a Mathematical Theory of Computation

Expected practical Results:

4. To represent computers as well as computations in a formalism that permits a treatment of the relation between a computation and the computer that carries out the computation.

5. To give a quantitative theory of computation. There might be a quantitative measure of the size of a computation analogous to Shannon's measure of information.

We hope that the reader will not be angry about the contrast between the great expectations of a mathematical theory of computation and the meager results presented in this paper.

Contents

- a class of recursively computable functions
- based on arbitrary domains of data and operations on them
- with conditional expressions

- functionals
- a general theory of datatypes
- recursion induction to prove equivalences

Reflections

- A mathematical theory is the entrance ticket to science
- Successes: eg, deterministic parsing: LL, LR etc.
- Numerical analysis, formal languages, complexity theory, algorithms, . . .
- But only mathematical logic seems to be dreamed as the mathematics of computing

The grand view

Structural engineering

- mathematical physics laws
- empirical knowledge

to understand, predict, and calculate the stability, strength and rigidity of structures for buildings.

McCarthy:

the relationship between computation and mathematical logic will be as fruitful as that between analysis and physics.

The standard model

The standard model is to PL what movement without friction is to mechanics.

There is no true pendulum in nature; there is no unbounded arithmetic inside any laptop.

No two bodies of different masses fall to the ground at the same time from the leaning tower of Pisa.

Yet, you do not understand a single bit of mechanics if you don't abstract away friction, and don't approximate to small oscillations.

The standard model

The standard model is to PL what movement without friction is to mechanics.

There is no true pendulum in nature; there is no unbounded arithmetic inside any laptop.

No two bodies of different masses fall to the ground at the same time from the leaning tower of Pisa.

Yet, you do not understand a single bit of mechanics if you don't abstract away friction, and don't approximate to small oscillations.

Back to types

We today conflate:

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

Goal:
separate them and identify when they arrive in the PL literature

Antefact: mathematical logic

A type is the range of significance of a variable.

[Russell and Whitehead, 1910]

Types forbid certain inferences which would otherwise be valid, but does not permit any which would otherwise be invalid.

[ibidem]

And then...

Leon Chwistek, in 1921

Frank P. Ramsey in 1926

...

Alonzo Church in 1940

...

Antefact: mathematical logic

A type is the range of significance of a variable.

[Russell and Whitehead, 1910]

Types forbid certain inferences which would otherwise be valid, but does not permit any which would otherwise be invalid.

[ibidem]

And then...

Leon Chwistek, in 1921

Frank P. Ramsey in 1926

...

Alonzo Church in 1940

...

Pre-antefact

Frege's *Stufe* (in the *Grundgesetze*)
or *Ordnung* (before the *Grundgesetze*)

usually translated with "level", or "degree"

Part I

The word: “type”

Types in early Fortran?

Two types of constants are permissible: fixed points (restricted to integers) and floating points

32 types of statement

[The FORTRAN automatic coding system, 1956]

Any fixed point (floating point) constant, variable, or subscripted variable is an expression of the same mode.

[ibidem]

Algol 58: types

Type declarations serve to declare certain variables, or functions, to represent quantities of a given class, such as the class of integers or class of Boolean values.

[Perlis and Samelson. Preliminary report: International algebraic language. Commun. ACM 1(12), December 1958.]

No types in the preparatory papers!

A data symbol falls in one of the following classes:
a) Integer b) Boolean c) General

The symbol classification statements are:
INTEGER $(s_1, \ldots, s_n)$
BOOLEAN $(s_1, \ldots, s_n)$

[Backus et al. Proposal for a programming language. ACM Ad Hoc Committee on Languages, 1958.]

Algol 60: maturity

Integers are of type **integer***. All other numbers are of type* **real***.*

*The various "types" (***integer***,* **real***,* **Boolean***) basically denote properties of values.*

[Backus et al. Report on the algorithmic language ALGOL 60. Commun. ACM 3(5), May 1960.]

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 technical term "type":

- appears to be a semantical shift from the generic term
- no role of the "type" from mathematical logic

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 (*contra*: FORTRAN !)

However:

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

Part II

Types as an abstraction mechanism

McCarthy: the “weakness” of Algol

1961:

defining new data spaces in terms of given base spaces and (. . .) defining functions on the new spaces in terms of functions on the base spaces

[John McCarthy. A basis for a mathematical theory of computation, preliminary report. 1961]

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

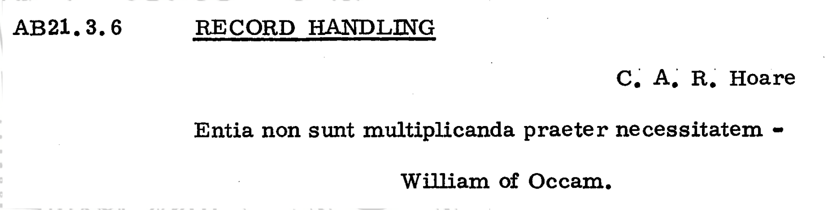
The two champions:

- Records
- Objects

The two champions for data abstraction:

- Abstract data encapsulation
- Procedural encapsulation

Hoare: records and references


AB21.3.6 RECORD HANDLING

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*

Hoare: records and references, 2

With Hoare's paper, types become a general abstraction mechanism:

[Our proposal] *is no arbitrary extension to an existing language, but represents a genuine abstraction of some feature which is fundamental to the art or science of computation.*

[Tony Hoare, 1964]

Modelling tool

In the simulation of complex situations in the real world, it is necessary to construct in the computer analogues of the objects of the real world, so that procedures representing types of even may operate upon them in a realistic fashion.

[Tony Hoare, 1964] (page 46, and, more generally, all Section 4)

Robust abstraction

It was a firm principle of our implementation that the results of any program, even erroneous, should be comprehensible without knowing anything about the machine or its storage layout

[Tony Hoare, 2014, personal communication]

Algol W, circa 1970

Every value is said to be of a certain type.

The following types of structured values are distiguished: array: (. . .), record: (. . .).

[Algol W reference manual, 1972]

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 → Object class → Class
- "declared quantity (class)" vs
  "its dynamic offspring (objects)"

Subclasses

- Hoare 1966, Villard-de-Lans Summer School:
  - record subclasses
  - dot notation
  - ⇒ 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:
- ρ represents D onto D'
- π 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.

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

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

Objects

A research question:

Both Simula and Smalltalk were designed for specific application domains.

How this influenced their characteristics?

What about the interaction with the “computational objects”?
   à la Papert: objects you can
   “get to know [. . .] like the way you get to know a person”

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)

Part III

Types from mathematical logic

- certainly people knew “some logic”: McCarthy, Hoare, Landin, Scott (!), Morris, etc.

  but

- Morris (1968) cites Curry (1958), but not Church (1940)
- Reynolds (1974) rediscovers Girard’s System F (1971)
- Milner (1977-78) rediscovers simple type inference (Hindley, 1969)

Programming languages and proof-theory are talking the same language, but the conflation is anonymous.

- certainly people knew “some logic”:
  McCarthy, Hoare, Landin, Scott (!), Morris, etc.

  but

- Morris (1968) cites Curry (1958), but not Church (1940)
- Reynolds (1974) rediscovers Girard’s System F (1971)
- Milner (1977-78) rediscovers
  simple type inference (Hindley, 1969)

Programming languages and proof-theory are talking the same language, but the conflation is anonymous.

- certainly people knew "some logic":
  McCarthy, Hoare, Landin, Scott (!), Morris, etc.

  but

- Morris (1968) cites Curry (1958), but not Church (1940)
- Reynolds (1974) rediscovers Girard's System F (1971)
- Milner (1977-78) rediscovers
  simple type inference (Hindley, 1969)

Programming languages and proof-theory are talking the same language, but the conflation is anonymous.

Yet, compare:

Types forbid certain inferences which would otherwise be valid, but does not permit any which would otherwise be invalid.

[Russell and Whitehead, 1910]

We shall now introduce a type system which, in effect, singles out a decidable subset of those wfes that are safe; i.e., cannot given rise to ERRORs. This will disqualify certain wfes which do not, in fact, cause ERRORS and thus reduce the expressive power of the language.

[Morris, PhD thesis, 1968]

Yet, compare:

Types forbid certain inferences which would otherwise be valid, but does not permit any which would otherwise be invalid.

[Russell and Whitehead, 1910]

We shall now introduce a type system which, in effect, singles out a decidable subset of those wfes that are safe; i.e., cannot given rise to ERRORs. This will disqualify certain wfes which do not, in fact, cause ERRORS and thus reduce the expressive power of the language.

[Morris, PhD thesis, 1968]

“*Denoting*” vs. “*Non denoting*”

becomes

“*Non producing errors*” vs. “*Producing errors*”

Well-typed expressions do not go wrong.

[Milner, 1978]

The formidable middleman:

λ-calculus

The catalist:

Curry-Howard isomorphism, (1969); 1980

The formidable middleman:

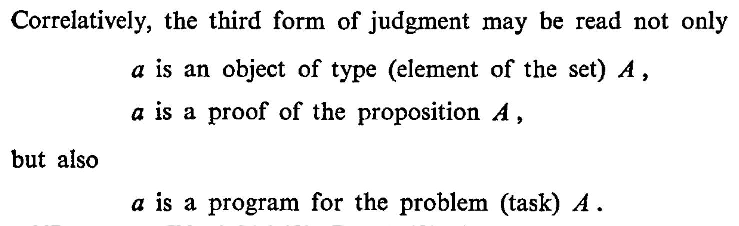
λ-calculus

The catalist:

Curry-Howard isomorphism, (1969); 1980

The explicit recognition:

Per Martin-Löf.
Constructive mathematics and computer programming.
(1979); 1982.



Correlatively, the third form of judgment may be read not only

a is an object of type (element of the set) A,

a is a proof of the proposition A,

but also

a is a program for the problem (task) A.

In foundations of mathematics, types:

- never supposed to be used by the working mathematician
- *in principle* could be used, to avoid paradoxes

In programming languages, types:

- are used everyday, by everyone
- should be made more “expressive”, “flexible”

In foundations of mathematics, types:

are perceived as constraints
(they “forbid” something, as in Russell’s quote).

In programming languages, types:

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

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

Why this is interesting

Our programming languages are also (a huge part of) the metalanguage in which we express the discipline.

“Programming” languages

No scientific discipline exists without first inventing a visual and written language which allows it to break with its confusing past.

[B. Latour, Visualisation and Cognition: Thinking with Eyes and Hands; 1986]

Referring to Dagognet, F.: Tableaux et Langages de la Chimie. Paris : Le Seuil 1969;
and to: Ecriture et Iconographie. Paris : Vrin 1973.

What we call programming languages are both such a founding language and one of the very objects of the discipline.

“Programming” languages

No scientific discipline exists without first inventing a visual and written language which allows it to break with its confusing past.

[B. Latour, Visualisation and Cognition: Thinking with Eyes and Hands; 1986]

Referring to Dagognet, F.: Tableaux et Langages de la Chimie. Paris : Le Seuil 1969;
and to: Ecriture et Iconographie. Paris : Vrin 1973.

What we call programming languages are both such a founding language and one of the very objects of the discipline.

The study of programming languages, and of their "conceptual" history, could become a blueprint for a more general epistemological investigation.

The project for the Collegium de Lyon

IRPHIL
LIP/PLUME

At the Collegium. . .

Next steps?

- Continuation passing transformation (van Wijngaarden, 1964)
- Exceptions handlers (PL/I: resume-based; etc.)
- Pinpoint the impact of the Curry-Howard isomorphism
- . . .

Methodology

History and Philosophy of Computing: HAPOC

Interplay between researchers across disciplines
may add to the maturity of the field of computing in general.

Bring together:
computer scientists, historians, and philosophy scholars.

Narrowing the gap between a technology and a professional history of that technology.

The long-term perspective:
provide a historical recollection that could be useful for the (young) technician of today, and could help her to make better science and technology.

Side paths

- Computational thinking: not just coding
- Programming as interaction (Smalltalk, Logo,. . . , Scratch, etc.)
- Program as inscriptions (Latour)
- Is the “traditional” Mathematical theory of computation still useful?

Sipario

Please, talk to me if interested.

Thank you.