A Survey of Object-Oriented Languages

Problems in Object-Oriented Language Design

C. Schaffert

Rapporteur: L. Mancini

A Survey of Object-Oriented Languages

Craig Schaffert

DEC

Cambridge Research Lab

Object-Oriented Programming is not a new idea

Evolving since late 60's

Combines & purifies existing practice

Language Characteristics

- support for data abstraction
  - new types appear built-in
  - " " protected
- subtyping
  - organizing large numbers of ty
  - generic calls
- inheritance
  - implementation technique
  - code sharing
- execution autonomy
  - threads of control

Major Language Types

Variables & Values

variable = storage

value = bits

assignment basic

Black Box Objects

everything is an object

objects only manipulated via operations

(Storage Management)

Experimental (Prototyping)

learn by programming

resulting program is secondary

Production

program is the goal

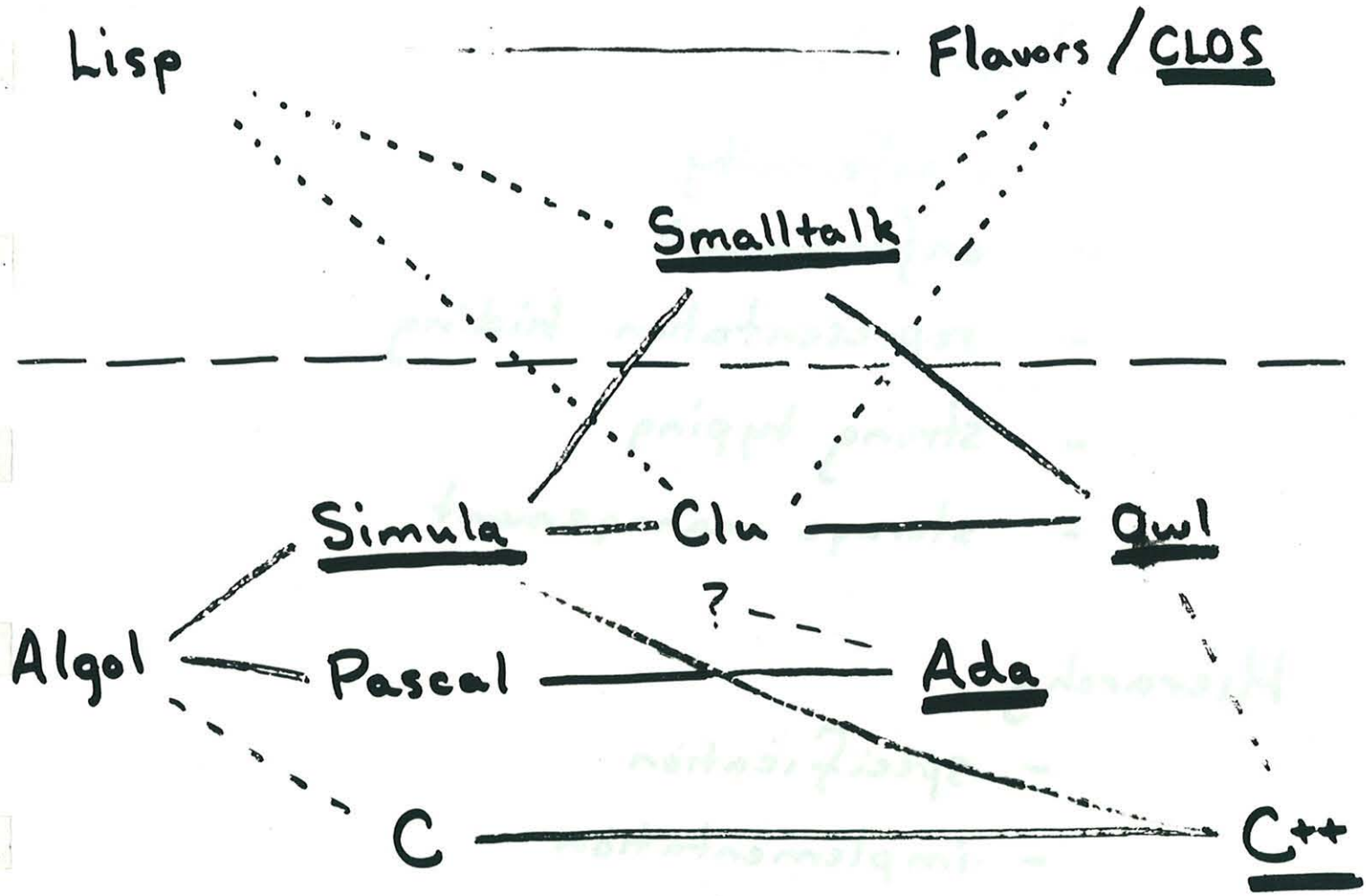
long life

---

coding speed & flexibility

vs.

understandability & evolution


Lisp
Flavors / CLOS
Smalltalk
Simula
Clu
Owl
?
Algol
Pascal
Ada
C
C++

Goals

Data Abstraction

- uniformity
- enforcement
- representation hiding
- strong typing
- storage management

Hierarchy

- specification
- implementation

Abstractions beyond A.D.T.

- threads
- exceptions
- operations as objects
- generics

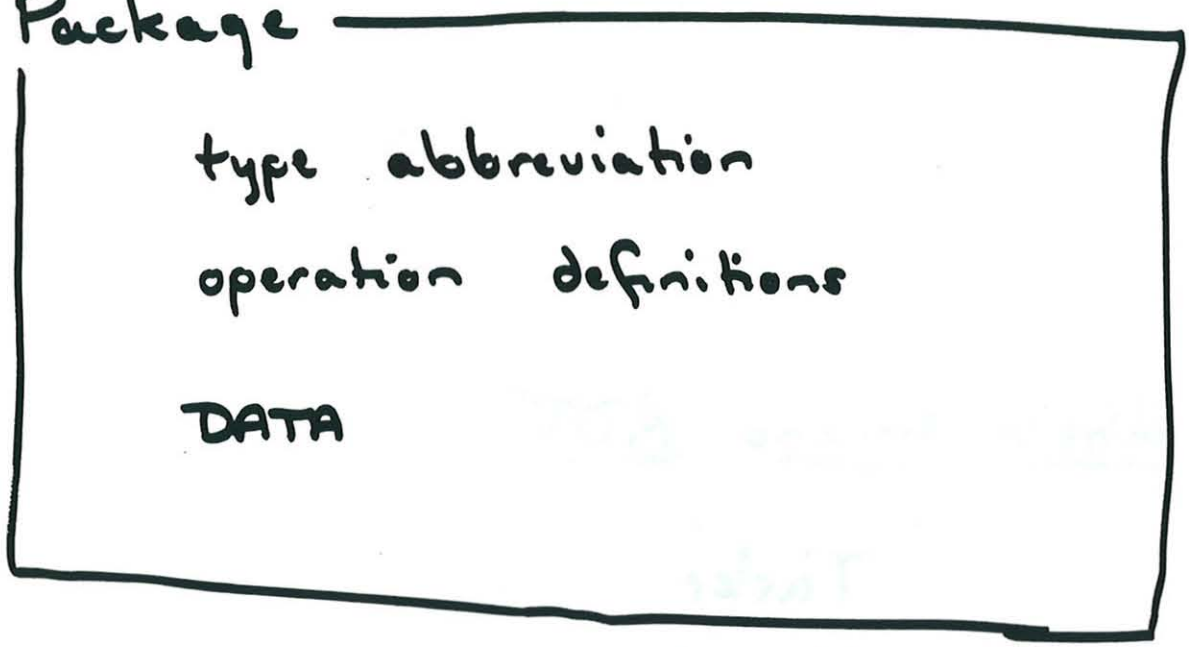
Environment

Library

ADA

not really object oriented

general purpose
high performance
conventional programming style



Enforces abstraction
Strong typing

Weak representation hiding
- assignment

Explicit storage management

No hierarchy

Overloading (static)

Abstr beyond ADT

Tasks

Exceptions

No operation objects

Weak generics

Prog Envn: active work

Libraries: informal, small

C++

compatability with C

general purpose
high performance

Data Abstraction

Separate type & operation defs

Enferces abstraction

Strong typing
- easily defeated

Better representation hiding

Explicit storage management
- destroy ops

Single inheritance hierarchy

Overloading (generic calls)

Abstr beyond ADT

No threads

Exceptions (soon?)

Operation objects

Generics (soon?)

Prog. Envn: none

Libraries: under discussion

Simula

Simulation
general purpose
performance less important

Data Abstraction

Unified type & operation defs
Enforces abstraction
Strong typing
Good representation hiding
- all pointers
Auto. storage management

Single inheritance hierarchy

Overloading (generic calls)

Abstr beyond ADT

Coroutines

No Exceptions

No operation objects

No generics

Prog. Envn : none

Libraries : small ?

SMALLTALK

experimental programming
performance less important

Data Abstraction

Unified definitions

Does NOT enforce abstraction

No compile-time typing

Excellent representation hiding

Auto storage management

Single inheritance hierarchy

- recent extension to multi

Overloading

Abstr beyond ADT

Control Structure Abstraction

- blocks

Coroutines

No exceptions

Operation objects ?

No need for generics

Progr Envn: excellent

Libraries: extensive

Flavors & CLOS

compatability with Lisp

Data Abstraction

Separate type & operation defs

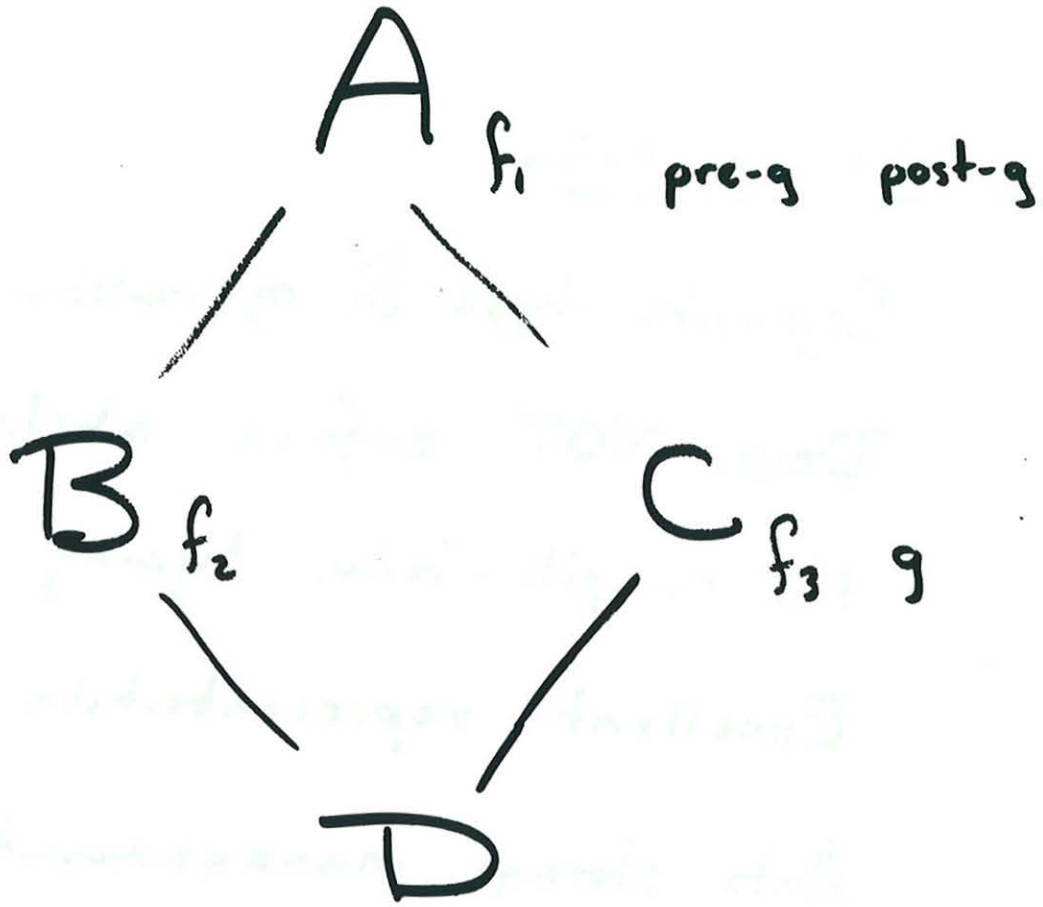
Does NOT enforce abstractions

No compile-time typing

Excellent representation hiding

Auto storage management

Very elaborate multi-inheritance hierarchy



D has f_2

pre-g ; g ; post-g

Abstr beyond ADT

No threads (Scheme)

No exceptions

Operation objects

No need for generics

Can re-define inheritance rules.

Prog Envn: excellent

Libraries: small

TRELLIS / OWL

general purpose
good performance
merge Smalltalk & Clu

Data Abstraction

Unified definitions
Enforces abstraction
Strong typing
Excellent representation hiding
Auto storage management

Simple multi-inheritance hierarchy

- spec based
- subtype compatability
- visibility control

Abstr beyond ADT

Iterators

Activities

Exceptions

Operation objects

Generics

- with constraints & conditional subtyping

Progr Envn: very good

Libraries: extensive

DISCUSSION

Dr. Kay stressed that the significance of Simula INNER and VIRTUAL constructs is often underestimated. He said it had been very hard to leave out from Smalltalk such powerful ideas, but that this had been dictated by the design goal of simplicity.

Dr. Schaffert concurred, and added that INNER can be seen as a precursor to Flavors' MIXIN.

Professor Nygaard observed that a major feature of an object-oriented programming language is whether it is endowed with metaclasses, as Smalltalk and CLOS are.

In answer Dr. Schaffert recalled from his lecture that there are two language communities, one that wants maximum flexibility within a system so as to essentially develop new languages within that system, the other which maintains that the purpose of types is to provide a program structure that can be dealt with statically.

Dr. Kay said that he happened to dislike blocks in Smalltalk-80. Blocks were not in the previous versions and were basically introduced under the influence of LISP people, to achieve generality and flexibility. However, blocks violate a lot of safety features: an internal block may be used as a value and this can give access to the interior of an object, which contradicts one of the reasons for using objects in the first place.

Dr. Schaffert said that indeed an essential part of language design is the balance between values and structure, i.e. between the flexibility to create new styles, and the provision of a pattern which programmers can think in terms of and then rely on.

Professor Atkinson enquired whether also the older object-oriented languages have a top of the class hierarchy. Dr. Schaffert answered that this was the case, with the exception that some languages, like Lisp-Flavors or C++, have an object-oriented part and a conventional built-in part which does not fit in entirely with the former.

Mr. Kerr observed that serious implications stem from the view that an object-oriented style can be pursued even in a non object-oriented language. In particular, in such an attempt, one is bound to find a point where, for lack of language support, he has to compromise the structure that is inherent in the object-oriented style. As a result, the object-oriented structure of the software fails to come out explicitly and there is a blurring of the boundaries between the structural components of the model. For instance, it turns out that generic types cannot actually be implemented as generic code. Moreover, the protection and security given by strong typing have to be jeopardized in order to achieve generality. In summary, the resulting code is not a suitable candidate for reusable software.

Dr. Haynes went back to the dichotomy between experimental programming and strong typing, to ask whether these approaches could be combined into a single system.

Answering, Dr. Schaffert recalled that typing essentially allows the programmer to state an intention whose violations can be detected by the system. Thus, if a programming environment allows intentions to be changed and rechecked quickly, it will be adequate for experimental programming: what is needed for experimental programming is fast change.

Professor Randell commented on the fact that, as stated in the lecture, type checking can be regarded as a sort of microverification. He thought that some of the relevant problems could be ascribed to the passage of time between when programs are verified and when they exist and run.

Doctor Schaffert thought this distinction between various states of a program to be very productive.

Professor Lee said he found the issues raised about inheritance and multiple inheritance to be very interesting, but expressed concern about the danger of ending up with two kinds of programmers: the programmer who constructs the type hierarchy, and that who is just a user of instances of types in the hierarchy.

In answer, Dr. Schaffert said he thought it productive to consider those two programmers to be different people even though all of them were the same person. Actually, when taking up the other role, the programmer ought to forget about the previous one, to avoid introducing too much coupling in the program because of the information he remembers. This can be summarized with the phrase "compartmentalization of knowledge".

Professor McDermid noted that the many problems with multiple inheritance may arise from an attempt to achieve too many goals with the same mechanism. Dr. Schaffert said this was possible.

With reference to the issues raised in the talk, Professor Nygaard discussed the clauses EXCLUDING and EXCLUDED IN in the Beta language.

Dr. Schaffert aptly concluded that much had been said about the problems of the object-oriented approach, but these were largely outnumbered by the benefits.