# **Oberon: New dimensions** for design of algorithms for scientific applications

Fyodor V. Tkachov

Leading Scientist
Department of Theoretical Physics
Institute for Nuclear Research, Moscow

How Oberon bridges computing paradigms
How garbage collection is similar to the complex plane
**Benchmark**: Oberon/Component Pascal vs C++ in numerical computations

Concrete experience: BlackBox/Component Pascal
**Conclusions valid for any Oberon**

EPSE-21 — Oberon Day @ CERN — 10 March 2004

# Mathematical views on computations

**Turing machine** (imperative/procedural programming)

row of memory cells + modification of their content

assignments and loops

ASM, Fortran

max efficiency; error prone

**Dijkstra et al.**: programs can be systematically and rigorously *derived*

Restrictions for safety: structured programming [~~GOTO~~], static typing

**Recursive functions** (functional programming)

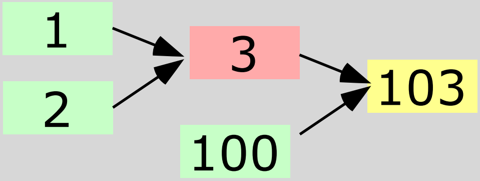
no loops — only functions; no assignments — only parameter substitutions

Lisp (Reduce), ML etc.

Tradition: automatic typing (not specific to rec. fun.’s)

99% of bugs found by compiler; 1% never found; heavyweight, complex

The single most important feature: garbage collection



1+2+100

each intermediate result requires a new memory cell; “garbage” — cells no longer used — must be reclaimed

**Markov’s algorithms**

substitutions on a sequence of symbols

regular expressions; SCHOONSCHIP

# Reality: always synthesis

imperative + best of functional; markovian in libraries

Technological

Run-time efficiency: basis = imperative
Correctness: necessary restrictions + all the wisdom of functional
Compilation efficiency: simplicity and regularity; 1-pass compiler
Portability: minimality

Psychiatric

Readability: we spent more time reading prog's than writing
Readability: 30% of brain visual processing
Non-IT professionals must do tons of programming: give us learn-able PL!
Interface design: simple, streamlined interfaced felt by users more powerful
**We err**: robustness w.r.t. typos; static typing

Spare us **Bug-tris** — enough garbage is falling on our heads daily
Give us **sharp knives** with **non-slippery handles**

Social

We communicate: reduce communication errors
Reduce unnecessary specialization ("caste of IT gurus"??)
Evolution: features impossible to remove
Teaching: new employees; kids

Simplicity here = minimalism + regularity + lucidity

# Oberon: an astounding solution!

See lectures by J.Gutknecht
and N.Wirth

Language Report: just about 20 pages!
The language is so small that it is hard to believe it is complete. **But it is!**

**Such simplicity is deceptive**
Recall how uneconomical beginners’ programs are.

Pascal, Modula-2, Oberon — 3rd iteration — Turing already for the 1st

METICULOUS DESIGN → SIMPLICITY → RELIABILITY + POWER

recall A.Hoare ...

**Oberon = *Elementa* of programming languages**

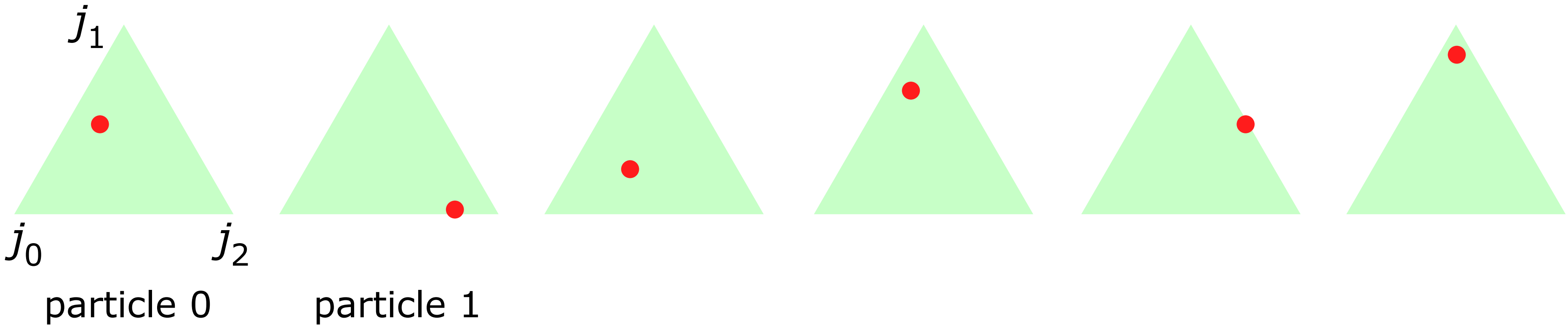
Great designs last

Let’s not be blinded by short-term considerations!

# Case study: development of Optimal Jet Finder (2000)

Solution to a 25 yr old problem.
Leading experts in ‘98: jet definition of that type “unfeasible”:
minimization in >$10^3$ dimensions.

E.g. find optimal distribution of 140 particles into 6 clusters (jets).
Optimal = minimum of a known function (“shape observable”).

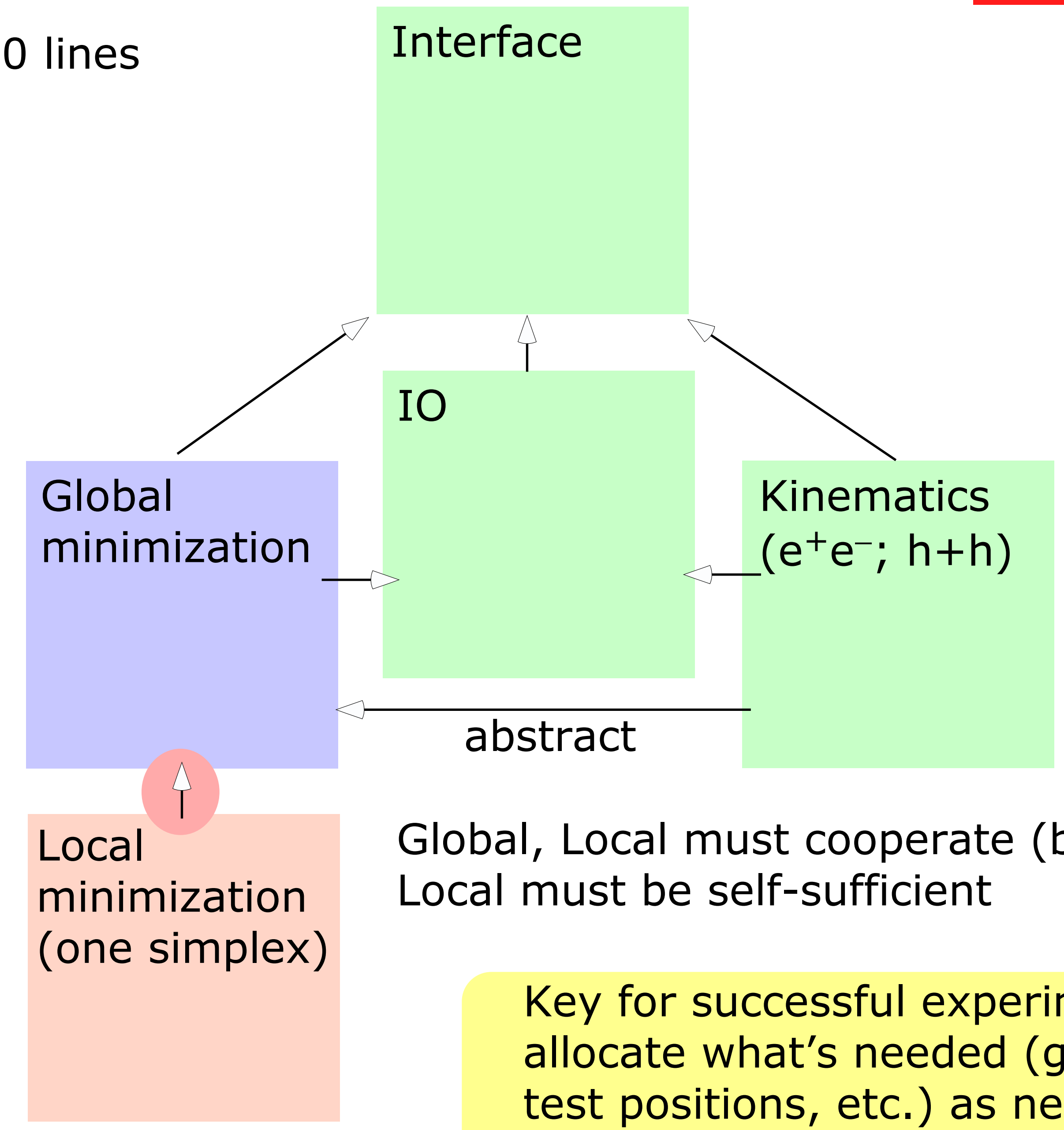


Each configuration of dots = distribution of particles between jets.

Gradient — which? Coordinate-wise? Boundaries??

Performance is critical.

total: ~1200 lines



Interface

IO

Global minimization

Kinematics (e$^+$e$^-$; h+h)

abstract

Local minimization (one simplex)

hardest;
essential to isolate

Global, Local must cooperate (backup!)
Local must be self-sufficient

Key for successful experimentation:
allocate what's needed (gradients, new test positions, etc.) as needed in Local, relying on garbage collection.
When works, map to static memory.

Prejudice against garbage collection (“inefficient”), however:
Garbage collection is here not as bad as e.g. in Lisp
because much more functionality per allocation

**Efficient construction of sophisticated algorithms:**
Find algorithm using dynamic allocation, relying on GC.
Eliminate dynamic allocation.

AI...

(Like we go into the complex plane when doing integrals.)

Nth variation of Pareto’s Law:

**New!**

**80% of dynamic allocation requires 20% effort to get rid of.**

Leave the remaining 20% to the Oberon kernel to take care of!

We have not started to realize the full potential
of numerical algorithms with “intelligence”, i.e. dynamic data structures

Indeed: OJF has been ported to Fortran (2001) and C++ (2004)

To return to such archaic languages after Oberon = **really sad;**
like **walking on a mine field**

First version in Component Pascal < 4 weeks;
first attempt to implement in Fortran from CP ~2 weeks;
ironing out floating point — failed in 3 months with Fortran;
going back to CP: fixed in 3 days.

Particularly telling is comparison with C++ because 1:1 correspondence with the original Component Pascal (quasi-mechanical port in < 1 week):

| **BlackBox** | **MS Visual C++ 6.0** | |
|---|---|---|
| | debug | full speed optimization |
| **3.6**±0.1 sec | **9.3**±0.1 sec | **3.5**±0.1 sec |
| With all safety checks,<br>with all debugging info! | | O(10) slower compilation |

no multiple inheritance...

**Lib** — math library by Robert D. Campbell (BAE Systems, UK)

**Gr** — a toolbox of histogramming and graphical modules to support developing interactive data acquisition (DAQ) and monitoring programs, by Wojtek Skulski (Univ. of Rochester, US).

Several data plotting and fitting tools ... (http://www.zinnamturm.de/)

**Fortran to Oberon compiler** — Douglas G. Danforth (Greenwood Farm Technologies, LLC, US).

Based on the Coco/R compiler tool by H.Mossenbock (ETHZ),
+ a modified scanner based on a parallel string search algorithm adopted from Stanford University.

Variants of FORTRAN and will be addressed as plugin modules of a general translation framework. The output, in like manner, will be a plugin for generating Component Pascal, Oberon-2, and Active Oberon modules. Upon completion of the FORTRAN components effort will be directed toward the C, C++, and C# family.

# Large-scale symbolic manipulation

M.Veltman’s **SCHOONSCHIP**:

interpreted symbolic engine, allowed a compiled Fortran subroutine

Design of **MINCER** (FT, 1982):

a **huge** boost from judiciously exploiting that feature
(a pretty complex computation with integers in static memory)

**BEAR** (Basic Extensible Algebra Resource; FT 1998-present):

A symbolic manipulation framework within a compiled language:
everything’s compiled unless absolutely requires dynamical handling.

40-hrs long calculations — never crashes.
3 times faster than the speed king Form-3 on equivalent algorithm,
“regular” optimization options still not employed.

We have only scratched the surface...