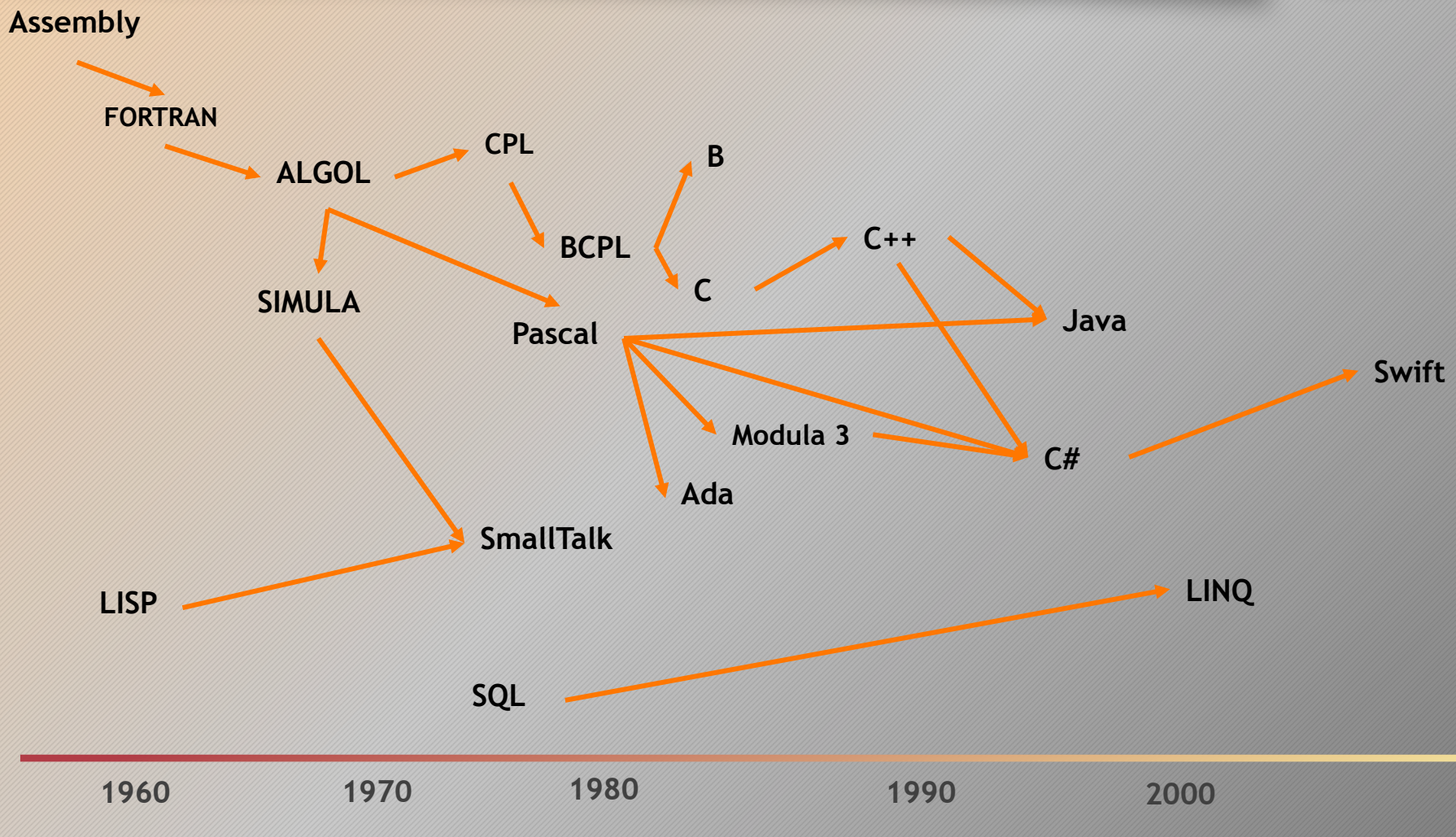
Native Language Integrated Queries with CppLINQ in C++

Vassil Vassilev

PH-SFT, CERN

Programming Language Savannah



Programming Language Savannah

- 0, 1-st, 2-nd... N-th generation of PL
- Multiple language paradigms
- Expressing ideas from different domains are more/less complex depending on the language.

Domain-specific languages (DSLs)

Depending on the domain:

- Science
  Mathlab, R, ROOT's C++ extensions
- Engineering
  Verilog, VHDL, LabView
- Parallelism
  OpenMP, OpenACC
- Computer Graphics
  Shading Languages
- Relation algebra (Relational Databases)
  SQL, LINQ, Datalog

Mixing DSLs in imperative languages

- Usually as strings passed to a db driver (SQL)
  Lack of type safety
- Through #pragma-s (OpenMP…)
  Lack of good diagnostics
- C#'s Language Integrated Queries (LINQ)
  Incorporating type safety and language-native diagnostics

Language Integrated Queries

- Native support of SQL-like syntax
- Operate on collections using relational algebra operations.

Iterators

- How many collection iterations are done in my code? How many of them include relational algebra? Set union, set difference, projection, selection, ...
- Iteration fundamental to the design on frameworks collections and a cornerstone for STL.
- No wonder why C++11 provides iteration simplifiers such as range-based iteration.
- C++ & STL are LINQ-friendly

Implementations

- cpplinq
- boolinq – boost-based implementation of
- MS Reactive Components
- ...

Implementations use operator overloading and or trampoline/meta objects.

Usage

#include LINQ header file

```
#include ”cpplinq/linq.hpp”

// Returns true if the parameter is a prime number.
bool is_prime(int);
long sum_primes() {
  auto xs = int_range(0, 100); // boost::counting_iterator
  using namespace cpplinq;
  return from(xs).where(is_prime).sum();
}
```

Enables LINQ

Sums all prime numbers in [0, 100)

Usage


```
#include ”cpplinq/linq.hpp”

// Returns true if the parameter is a prime number.
bool is_prime(int);
using namespace cpplinq;
long sum_squared_primes() {
  auto xs = int_range(0, 100); // boost::counting_iterator
  return from(xs).where(is_prime).
                  select([](int x){return x*x}).sum();
}

std::vector<int> gimme_some_primes (int howMany) {
  auto xs = int_range(0, -1); // 0 to MAX(int)
  return from(xs).where(is_prime).take(howMany).to_vector();
}
```

Field of HEP

Why Marry the ROOT and .NET Worlds?

- I am much more productive in languages like python and C# than I am in C++. Many of the functional features (lambda's, Language Integrated Query (LINQ), etc.) make these languages much more concise in expression problems. In addition, the recent activity around functional languages (OCaml, etc., F# on the .NET platform) are bringing a new way of expressing problems in very concise declarative forms rather than our usual imperative forms.
- .NET (and the Java platform as well) use a runtime and allow trivial sharing of libraries between languages: making the library available for one language (C#) generally means it is available for all others (F#, etc.). .NET was chosen
- ROOT is the de facto data storage and histogramming, and fitting, etc., package for the field of HEP. This project is the infrastructure for an attempt to marry these two worlds.
- Initial motivation was driven by a number of small projects at the D
the Tevatron accelerat
wrappers that I wrote v
Passing ROOT objects t
ROOT objects was diffi
ing minimal use of
avoid the extra work. I
of python files used in
they were slow.
- The pyROOT and Rub

Only @ 1 CHEP Event

CRD

Python: nicer syntax ...

BERKELEY LAB


```
// retrieve data for analysis
TFile f = new TFile("data.root");
TTree t = (TTree*)f->Get("events");

// associate variables
Data* d = new Data;
t->SetBranchAddress("data", &d);

Long64_t isum = 0;
Double_t dsum = 0.;

// read and use all data
Long64_t N = t->GetEntriesFast();
for (Long64_t i=0; i<N; i++) {
   t->GetEntry(i);
   isum += d->m_int;
   dsum += d->m_float;
}

// report result
cout << sumi << " " << sumd << endl;
```

```
# retrieve data for analysis
input = TFile("data.root")

# read and use all data
isum, dsum = 0, 0.
for event in input.data:
   isum += input.data.m_int
   dsum += input.data.m_float

# report result
print isum, dsum
```

Python allows boilerplate code to be hidden through hooks in the language

Note: simplistic example chosen to make sure that language overhead fully dominates *rather than I/O or object construction.*

Office of Science U.S. DEPARTMENT OF ENERGY

Simple EventData demo


```
...
struct Particle {
  ROOT::Math::XYZVector  fPosition; // vertex position (Gaus)
  ROOT::Math::PtEtaPhiMVector fVector; // particle vector
  int  fCharge; // particle charge
  short fType; // particle type (0=pho, 1 ele, 2...)
};
struct EventData {
  std::vector<Particle> fParticles; // Poisson distributed mean 40
};
...
```

Simple EventData demo


```
// Read back the data
void ReadEventData(std::vector<Particle>& v) {
   TFile *myFile = TFile::Open("eventdata_s99.root");
   TTreeReader tree("tree", myFile);
   TTreeReaderValue<std::vector< Particle> > particles_value(tree, "fParticles");
   while (myReader.Next()) {
      v.insert(v.end(), particles_value->begin(), particles_value->end());
   }
  ...
}

void PrintElectronCount() {
  std::vector<Particle> ED; ReadEventData(ED);
  auto elCnt = from(ED).
                 where([](const Particle& p) {return p.fType == Electron}).
                count();
  printf("Electron count in ED is:%d\n", elCnt);
}
```

LINQ Operators

- groupby
- select
- select_many
- where
- aggregate
- any
- all
- cast
- contains
- count
- element_at
- empty
- first
- last
- max
- min
- sum
- take
- to_vector
- late_bind
- from
- …

Debugging your code

- Sometimes can become a terrible experience.
- Complex template mechanics at the backstage
- Could be improved by using various techniques

Field of HEP


```
// Calculate the pt (momentum) distribution for all positive and negative
// charged particles and compare the values.
using namespace cpplinq;
double PosPtSum =
   from(from (ED).
        where([](const Particle& p) {return p.fCharge > 0;})).
        select([](const Particle& p) {return p.fVector.Pt();}).
        sum();

double NegPtSum =
   from(from (ED).
        where([](const Particle& p) {return p.fCharge < 0;})).
        select([](const Particle& p) {return p.fVector.Pt();}).
        sum();
printf("Positive +PT Sum=%f, and negative PT Sum=%f\n", PosSum, NegSum);
```

LINQ and ROOT6 TreeReader


```
// Calculate the pt (momentum) distribution for all positive and negative
// charged particles and compare the values.
TFile *myFile = TFile::Open("eventdata_s99.root");
TTreeReader tree("tree", myFile);
TTreeReaderArray<Particle> particles(tree, "fParticle");

using namespace cppling;
double PosPtSum =
  from(from(particles).
            where([] (const Particle& p) {return p.fCharge > 0; } )).
       select([](const Particle& p){ return p.fVector.Pt();}).sum();

double NegPtSum =
  from(from(particles).
            where([] (const Particle& p) {return p.fCharge < 0; } )).
      select([](const Particle& p){ return p.fVector.Pt();}).sum();
 printf("Positive +PT Sum=%f, and negative PT Sum=%f\n", PosSum, NegSum);
```

Computation Abstraction

Going from imperative to declarative paradigm:

- Allows to describe the result of the computations not the computations themselves.
- Opens entire new world of scheduling

Computation Abstraction

Advantages

- We can communicate the ‘expected’ results to the batch system, where the scheduler can be smarter.
- Allows smart compiler to further-optimize the requested operations

Performance

- Based on std::iterator and heavily templated code.
- All exposed to the C++ compiler, thus optimization-friendly
- A lot to explore and improve

Future

- Expression trees
- Late binding
- Add support in ROOT
- Implement cpplinq.remote for PROOF-like systems

Thank you!

Further resources:

https://github.com/vgvassilev/RxCpp