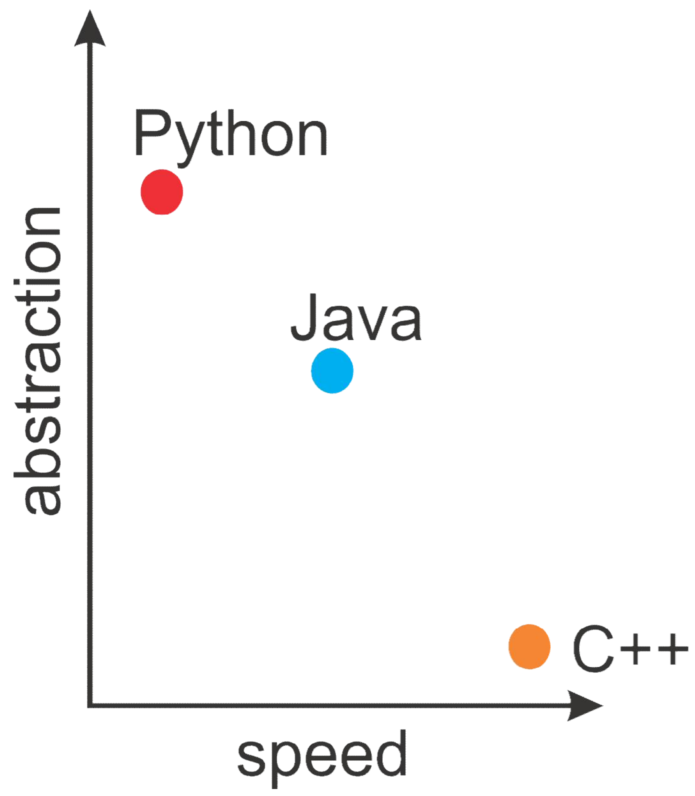
c++ and python – modern programming techniques

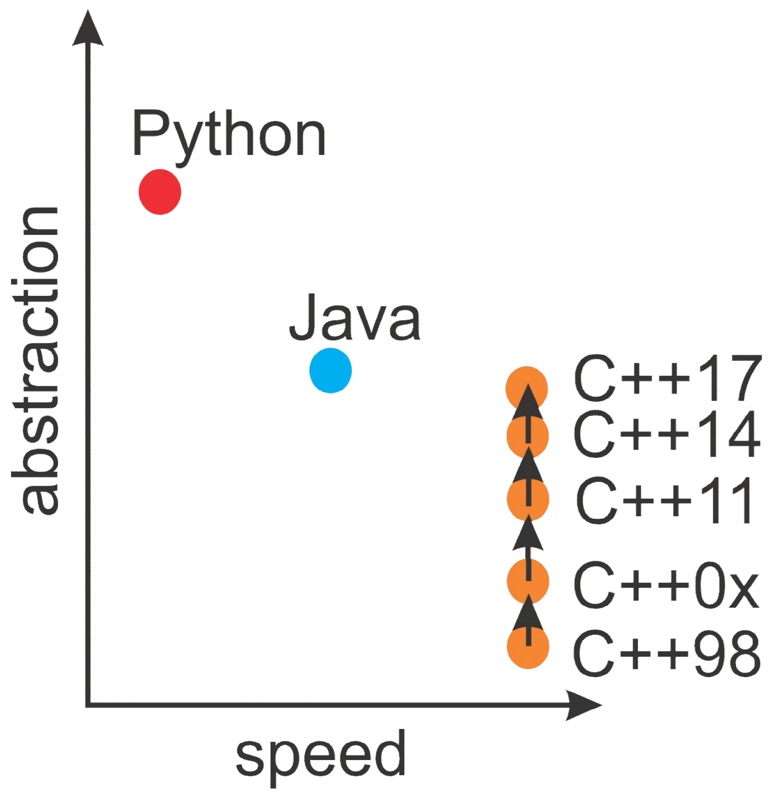
Jakša Vučičević

IPB, Tuesday, February 28th, 2017

Outline

- Introduction to Python, Java and C++
  - programming paradigms
  - compilation model
- Template programming vs. class hierarchies
- Various examples (C++ vs. python)
  - auto typing
  - templates
  - meta-programming (functors, partial evaluation, lambdas)
  - Variadic templates
  - generic DMFT loop
- TRIQS library
  - c++2py
  - numpy.array and triqs::array: linear algebra made easy
  - HDF5 data storage made easy
- Take away messages

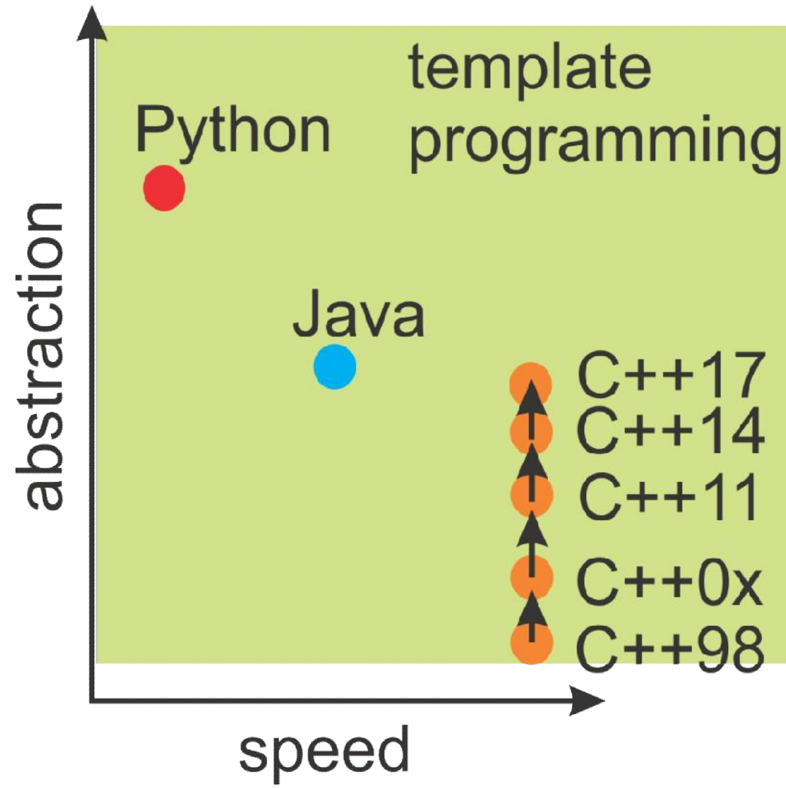

Python
Java
C++
abstraction
speed



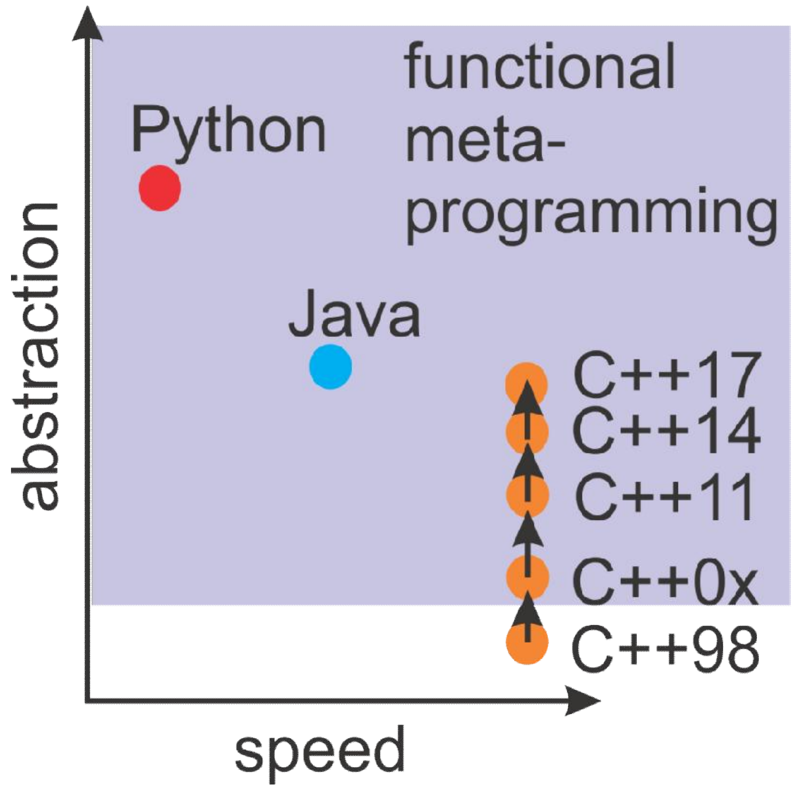
Python
Java
C++17
C++14
C++11
C++0x
C++98
abstraction
speed

simplest generic programming

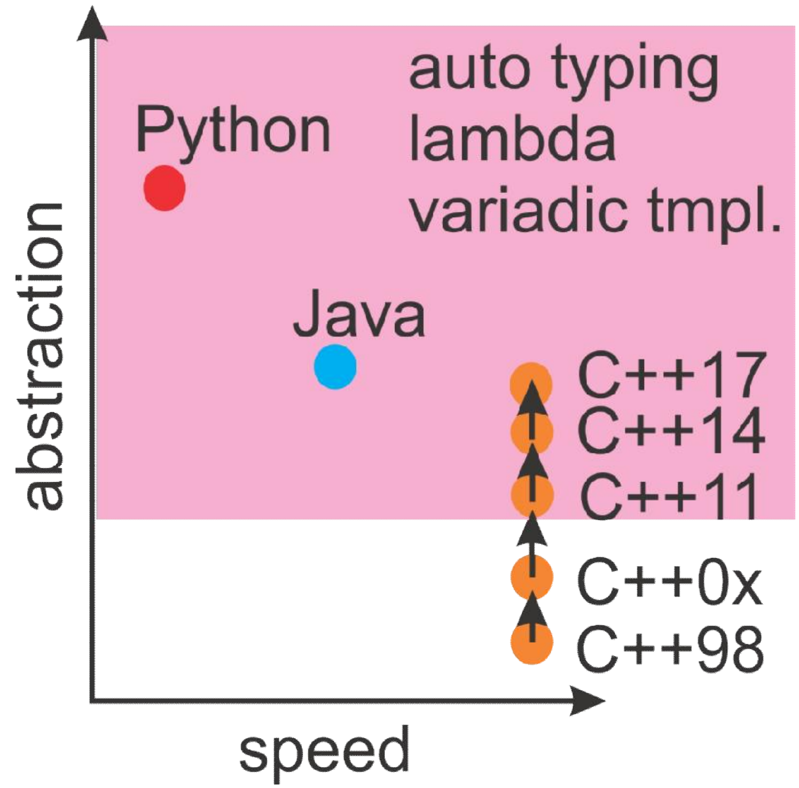
types-to-be-specified-later



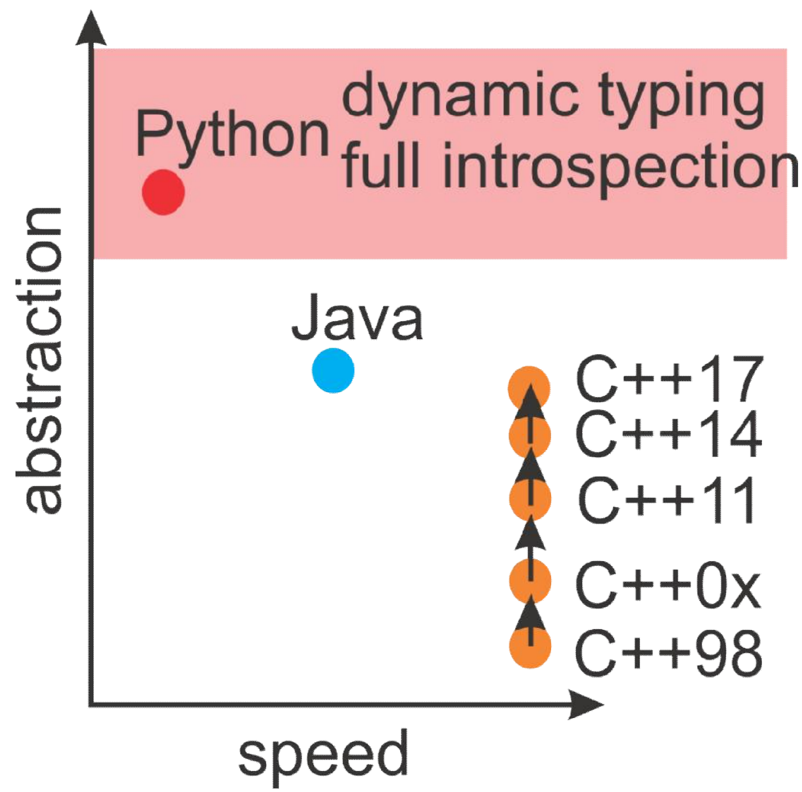
std library support for function objects
partial evaluation

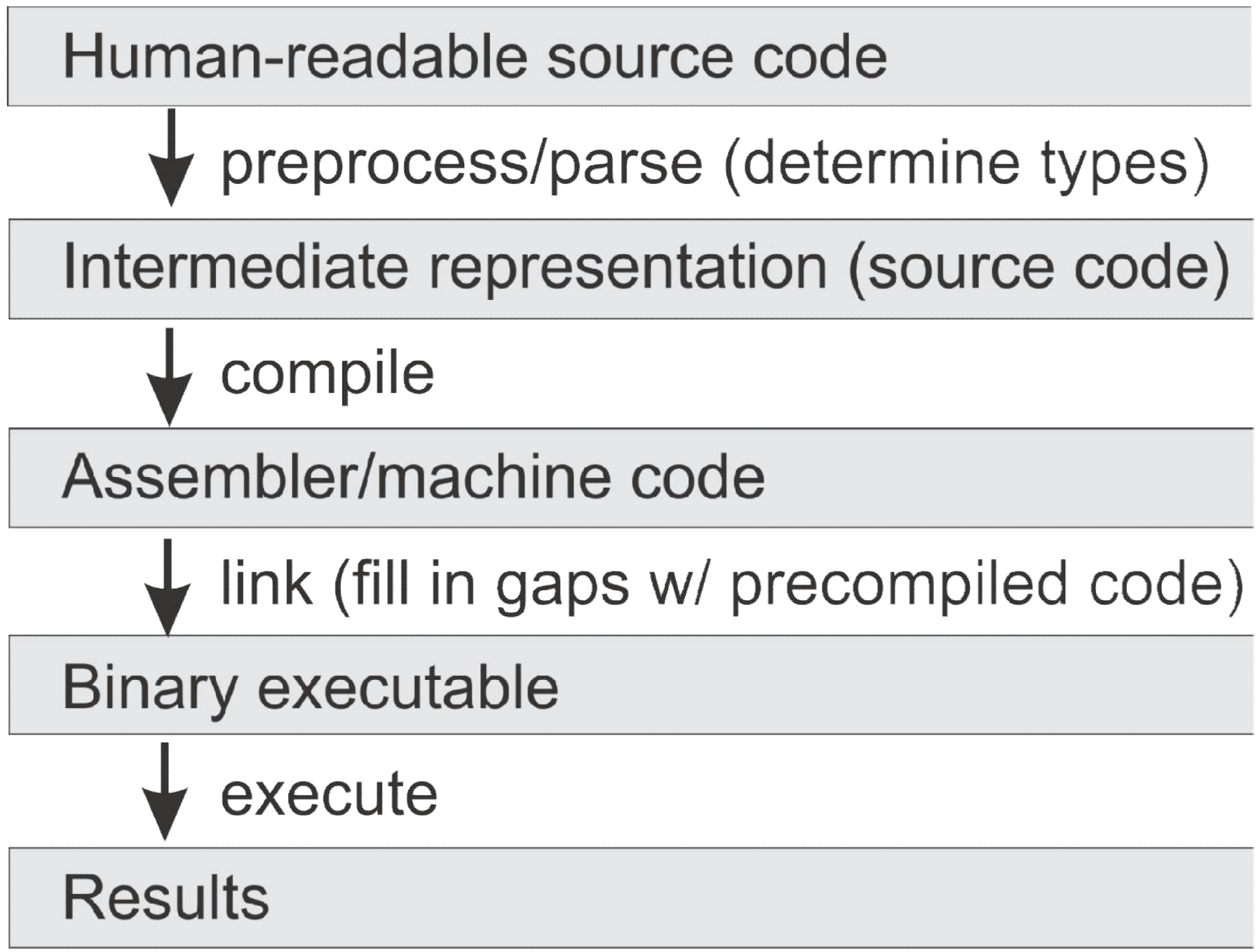


almost full generic programming – auto typing, type “copying”, variadic templates
full meta-programming – passing around code instead of objects
lambda functions = scope wormholes



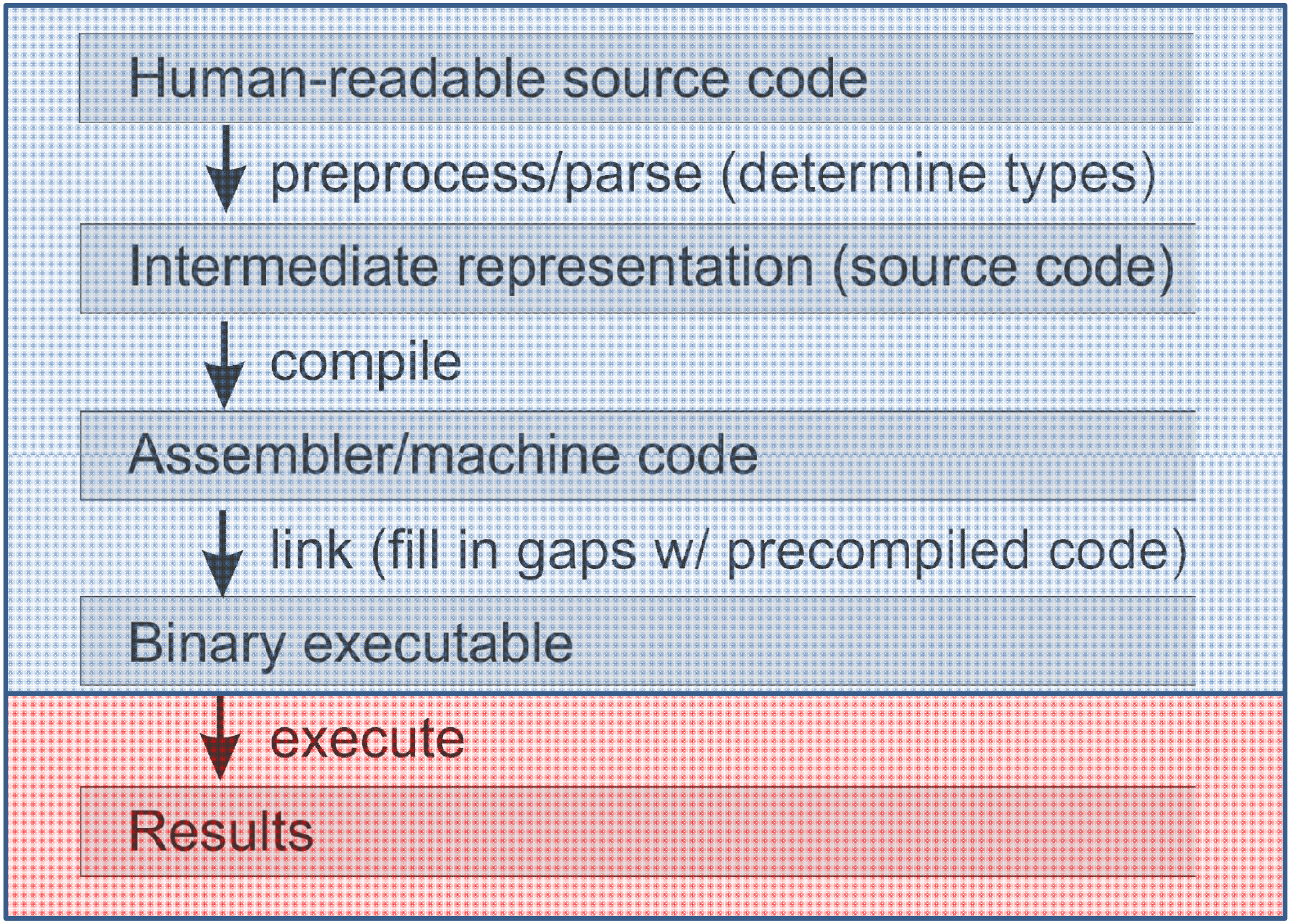
no need for templates – all types automatic!
introspection – all info on all types and scopes available at runtime




Human-readable source code
preprocess/parse (determine types)
Intermediate representation (source code)
compile
Assembler/machine code
link (fill in gaps w/ precompiled code)
Binary executable
execute
Results

C++ compile time

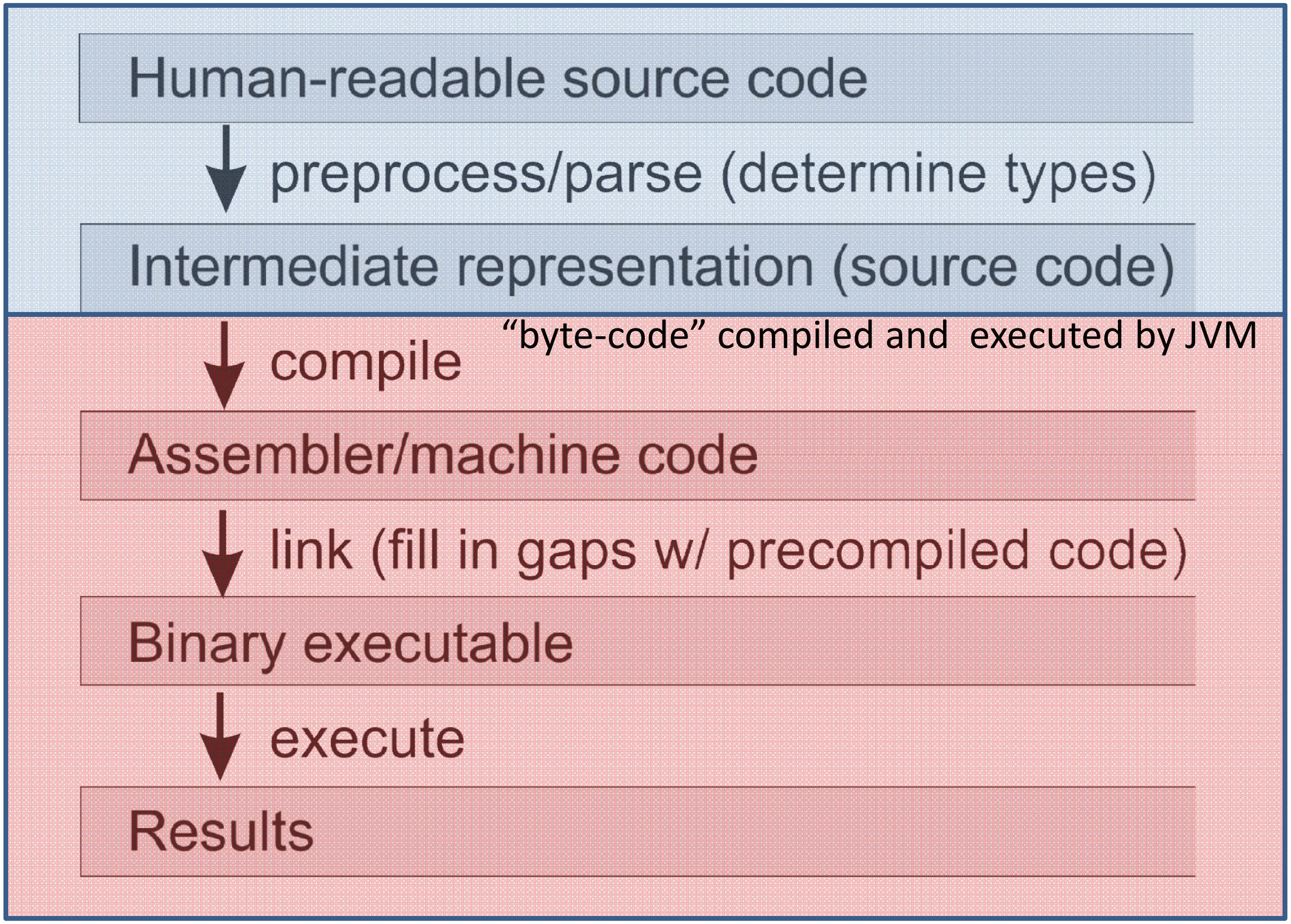
ahead-of-time compilation



C++ run time

Java compile time

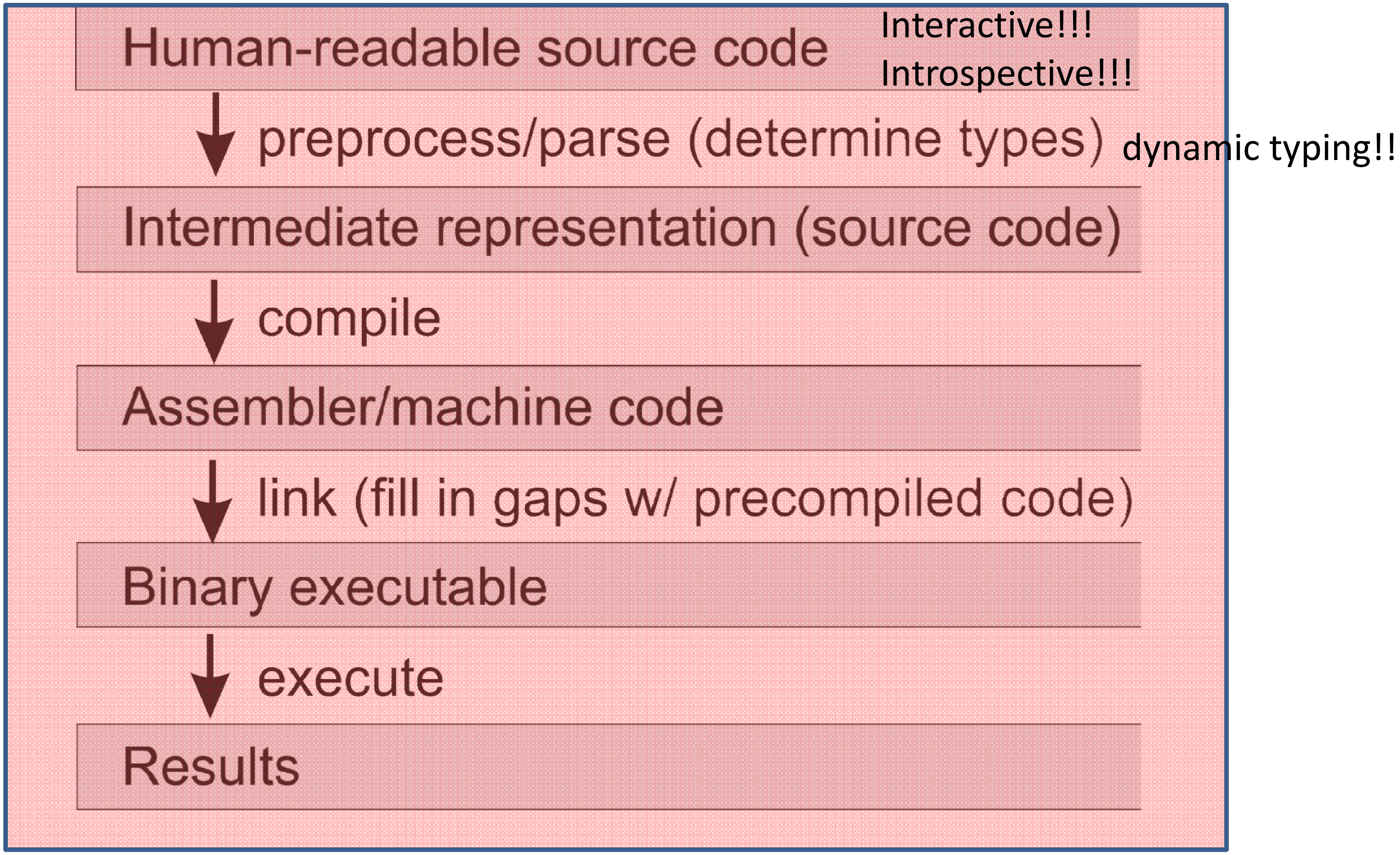
just-in-time compilation



Java run time

Python – no “compile time”

interpretation



Python run time

Generic programming

templating vs. class hierarchies

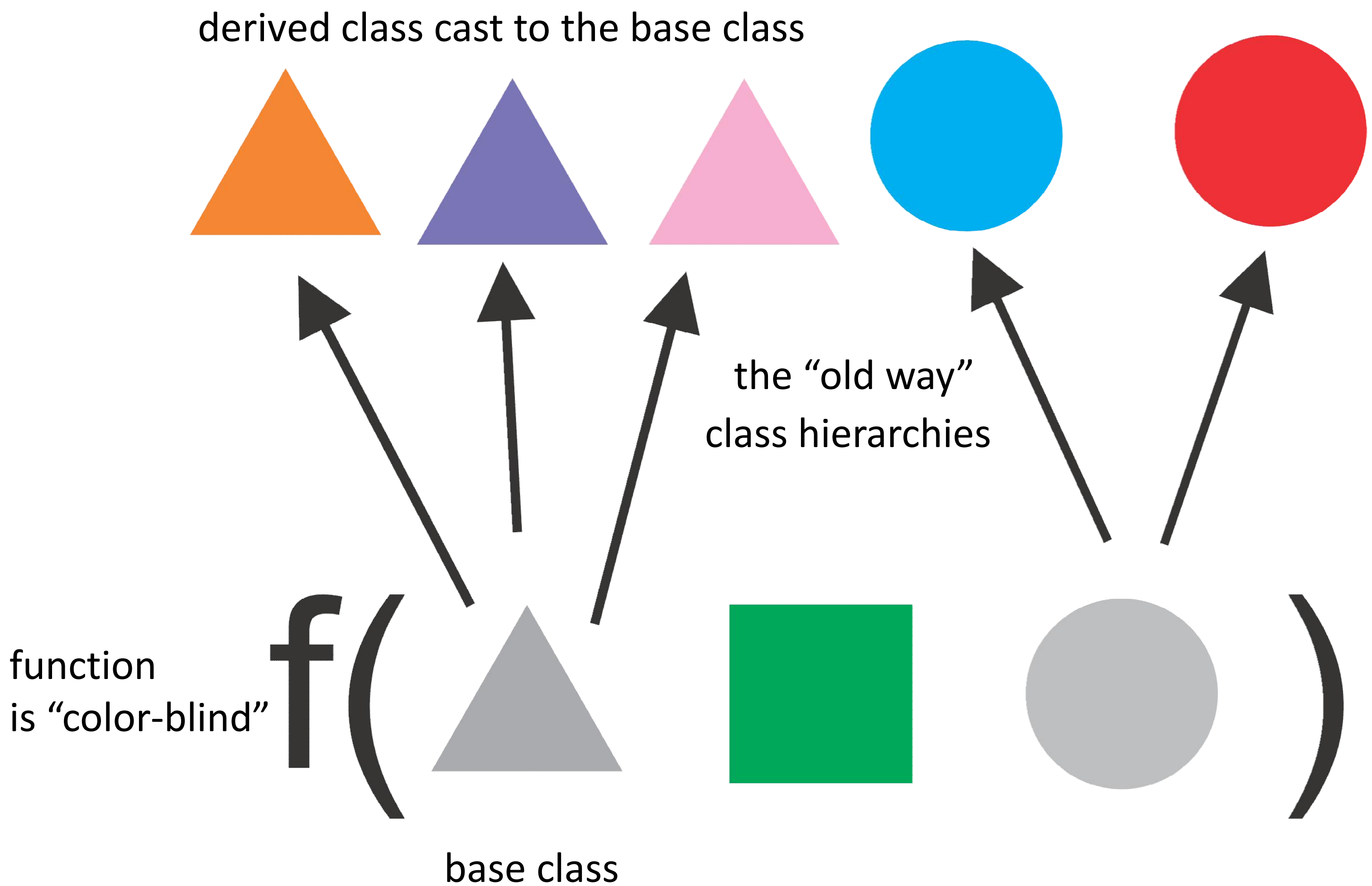
Generic programming

templating vs. class hierarchies

Generic programming

templating vs. class hierarchies

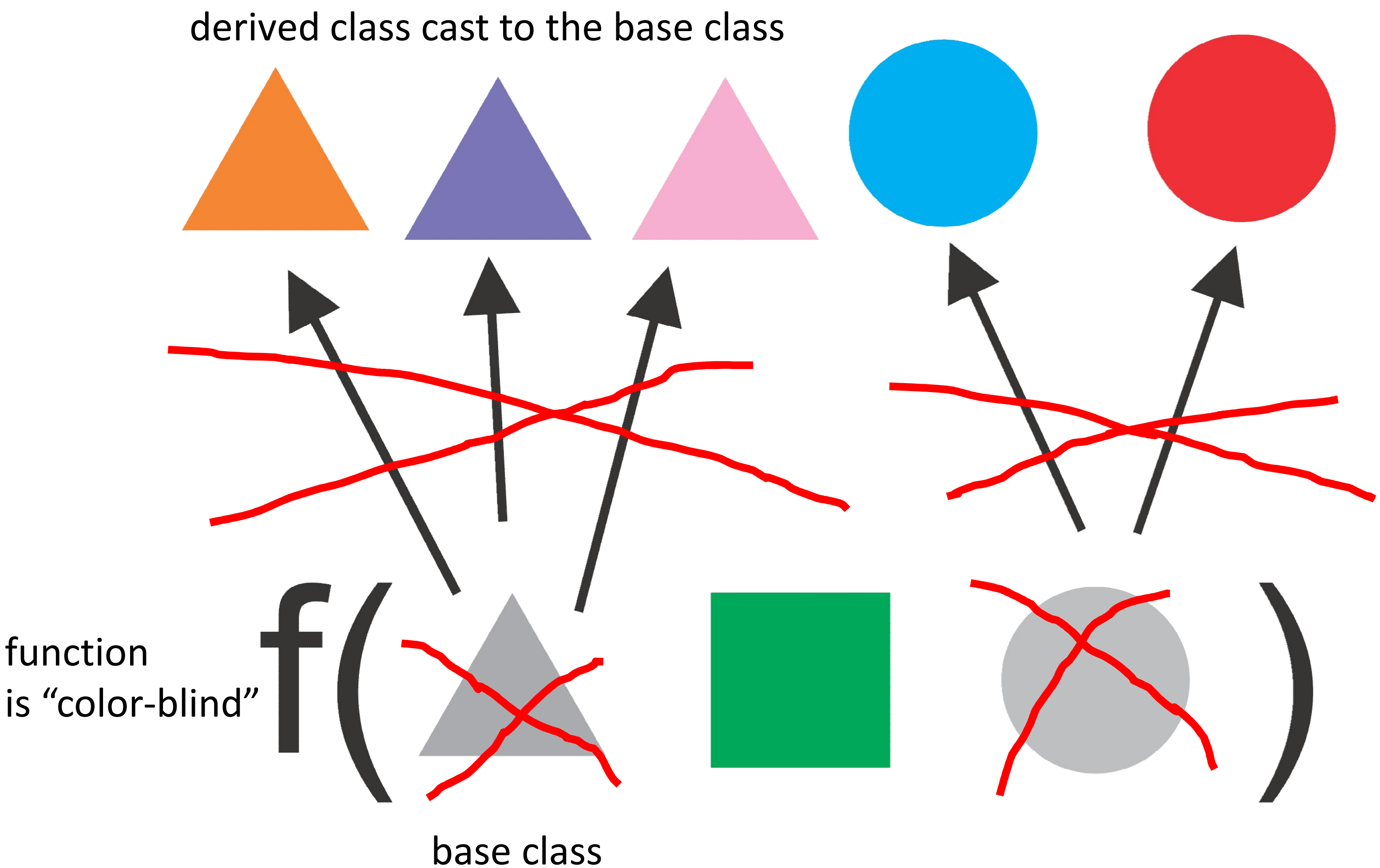
Generic programming



Generic programming

paradigm fails in the context of basic types, and various function objects

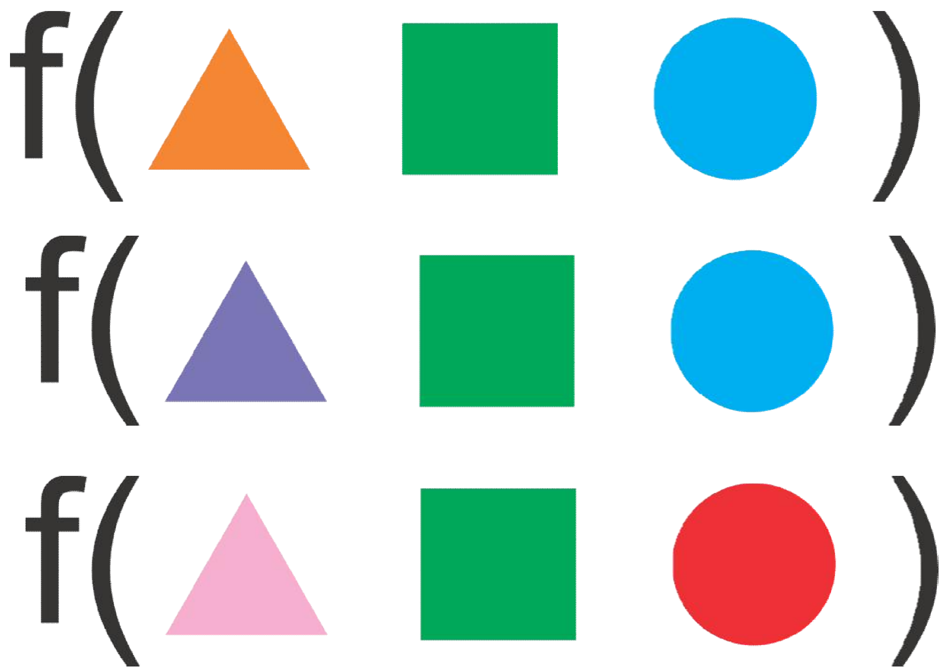
classes and dependences proliferate... many issues



Generic programming

You write

Compiler writes whichever necessary



to achieve this we need

- templates

but

- auto type helpful
- compile time introspection quite useful

we are making full use of generic programming if we have ways of passing functions

- since c++98, custom functors do the job, but too much overhead
- since c++0x, std library support for functors – much better
- since c++11, lambda functions – as meta as it gets
- since c++14, generic lambdas – as if we're writing python

let's see some examples...

Auto typing

C++98

```
int a=5;
int b=7; //change to 7.2, change int to float
int c=a+b; //change into float
```

Python

```
a=5
b=7 #change to 7.2, nothing else to be done
c=a+b
```

Auto typing

C++11

```cpp
auto a=5;
auto b=7; //change to 7.2
auto c=a+b;
```

Python

```python
a=5
b=7 #change to 7.2, nothing else to be done
c=a+b
```

Auto typing

C++11

"auto" doesn't work in function arguments

```cpp
auto a=5;
auto b=7; //change to 7.2
auto c=a+b;
```

as return type only since c++14

```cpp
decltype(a) d = c;
```

since c++11 more "introspective"

Python

```python
a=5
b=7 #change to 7.2, nothing else to be done
c=a+b
```

Templates

pre C++98

```
int f(int a, int b){
  return a+b;
}

float f(float a, float b){
  return a+b;
}
```

Python

```
def f(a, b):
  return a+b #does the job
```

Templates

C++98

```
template<typename T>
T f(T a, T b){
  return a+b;
}
```

Python

```
def f(a, b):
  return a+b #does the job
```

Templates

C++14

```
template<typename T1, typename T2>
auto f(T1 a, T2 b){
  return a+b;
}
```

Python

```
def f(a, b):
  return a+b #does the job
```

Templates

C++98

```
template<typename T>
void f(T X){
  X();
}

void g() {
  cout << "whateva";
}

f(g); //compiles!
```

Python

```
def f(g):
    g()

def g():
    print "whateva"

f(g) #voila!
```

Templates

C++98

```
template<typename T, typename P, int i>
void f(P p){
  T::some_static_method();
  p.other_method(i);
}

template<typename T, int i>
void f<myclass3>(P p){
  T::some_static_method();
  p.other_method3(i);
}
```

Python

```
def f(T, p, i):
  T.my_method() #yes, we are passing a class as an argument!
  if p.__class__ == my_class3:
    p.other_method3(i)
  else:
    p.other_method(i)
```

Templates

C++98

```
template<typename T, typename P, int i>
void f(P p){
  T::some_static_method();
  if (typeid(p).name() == "myclass3")
    p.other_method3(i);
  else
    p.other_method(i);
}
```

Can't compile!!!

Python

```
def f(T, p, i):
  T.my_method() #yes, we are passing a class as an argument!
  if p.__class__ == my_class3:
    p.other_method3(i)
  else:
    p.other_method(i)
```

Templates

C++17

```
template<typename T, typename P, int i>
void f(P p){
  T::some_static_method();
  if constexpr(typeid(p).name() == "myclass3")
    p.other_method3(i);
  else
    p.other_method(i);
}
```

whatever is declare “constexpr”
is evaluated at compile-time!! (since c++11)

Python

```
def f(T, p, i):
  T.my_method() #yes, we are passing a class as an argument!
  if p.__class__ == my_class3:
    p.other_method3(i)
  else:
    p.other_method(i)
```

Templates

C++17

```
template<typename T, typename P, int i>
void f(P p){
  T::some_static_method();
  if constexpr(typeid(p).name() == "myclass3")
    p.other_method3(i);
  else
    p.other_method(i);
}
```

whatever is declare “constexpr”
is evaluated at compile-time!! (since c++11)

Python

```
def f(T, p, i, method_name=''):
  T.my_method() #we are passing a class as an argument!
  if hasattr(p,other_method):
    p.other_method(1)
  elif hasattr(p,other_method3):
    p.other_method3(i)
  else: getattr(p,method_name)(i) #just tell me which method to call
```

Functional meta-programming

treating functions as data

why is it useful?

C++

```
void C(int x5, int x6) {
  cout << x5 << x6 << endl;
}

void B(int x3, int x4, int x5, int x6){
  cout << x3 << x4 << endl;
  C(x5,x6);
}

void A(int x1, int x2, int x3, int x4, int x5, int x6){
  cout << x1 << x2 << endl;
  B(x3,x4,x5,x6);
}

A(1,2,3,4,5,6);
```

C++98 function objects!!!

```
void C(int x5, int x6){
  cout << x5 << x6;
}

template<typename T>
void B(int x3, int x4, T X){
  cout << x3 << x4;
  X();
}

template<typename T>
void A(int x1, int x2, T X){
  cout << x1 << x2;
  X();
}

template<typename T>
struct BB {
    int x3, x4;
    T C;
    BB(int x3_, int x4_, T C_):x3(x3_),x4(x4_),C(C_) {};
    void operator ()() { B(x3,x4,C); };
};

struct CC {
    int x5, x6;
    CC(int x5_, int x6_):x5(x5_),x6(x6_) {};
    void operator ()() { C(x5,x6); };
};

A(x1,x2, BB<CC>(x3,x4, CC(x5,x6))); //prints 123456
```

Functional meta-programming

treating functions as data

C++0x

automatic function objects

parital evaluation made easy

since c++0x

```
void C(int x5, int x6){
  cout << x5 << x6;
}

template<typename T>
void B(int x3, int x4, T X){
  cout << x3 << x4;
  X();
}

template<typename T>
void A(int x1, int x2, T X){
  cout << x1 << x2;
  X();
}
int x1=1, x2=2, x3=3, x4=4, x5=5, x6=6;
auto CC = bind(C, x5, x6);
auto BB = bind( B<decltype(ref(CC))>, x3, x4, ref(CC) );
A(x1,x2, BB); //prints 123456
```

c++11 → (points to `auto CC = bind(C, x5, x6);`)

Functional meta-programming

treating functions as data

Lambda functions –
passing snippet of code
through a scope wormhole!!!

C++11

```
void C(int x5, int x6){
  cout << x5 << x6;
}

template<typename T>
void B(int x3, int x4, T X){
  cout << x3 << x4;
  X();
}

template<typename T>
void A(int x1, int x2, T X){
  cout << x1 << x2;
  X();
}
int x1=1, x2=2, x3=3, x4=4, x5=5, x6=6;

A(x1,x2, [&](){ B(x3,x4, [&](){ C(x5,x6); } ); } );
```

Functional meta-programming

treating functions as data

Python

```
def C(x5, x6):
  print x5, x6

def B(x3, x4, X):
  print x3, x4,
  X()

def A(x1,  x2,  X):
  print x1, x2,
  X()

x1,x2,x3,x4,x5,x6=1,2,3,4,5,6

A(x1,x2, partial(B, x3,x4, partial(C, x5, x6))) #prints 123456

A(x1,x2, lambda: B(x3,x4, lambda: C(x5,x6))) #prints 123456
```

Variadic templates

C++11

```
template<typename T>
int Wrapper(T t) { t(); return 0; }

template <typename ... F >
void A(F... f ){
  for(int i=0; i<10; i++)
    int x[] = {Wrapper(f)...};
}
void C() {
 cout << "C";
}
void B() {
  cout << "B";
}
void D() {
  cout << "D";
}
A(C, B); //prints CBCBCBCBCBCB...
A(C, B, D); //prints CBDCBDCBDCBD...
A(C, B, D, D, B); //prints CBDDBCBDDBCBDDB....
```

List/aggregate initializer

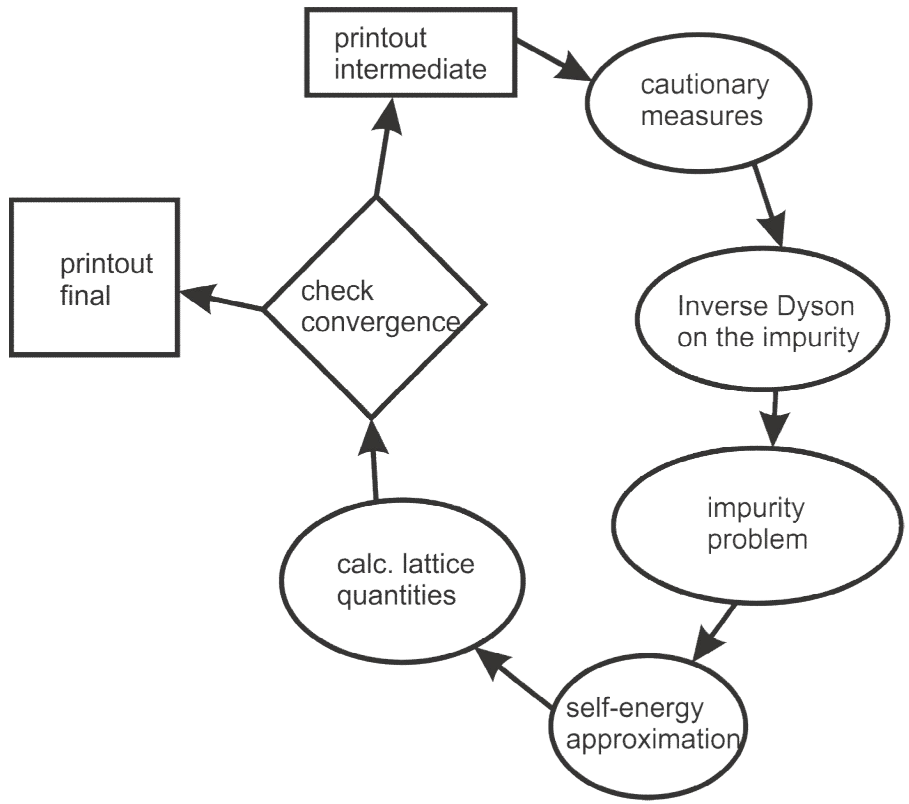
Variadic templates

Python

```
def A(*args):
  for i in range(10):
    for f in args:
      f()

def B():
 print "B"

def C():
 print "C"

def D():
 print "D"

A(C, B) #prints CBCBCBCBCBCB...
A(C, B, D) #prints CBDCBDCBDCBD...
A(C, B, D, D, B) #prints CBDDBCBDDBCBDDB....
```

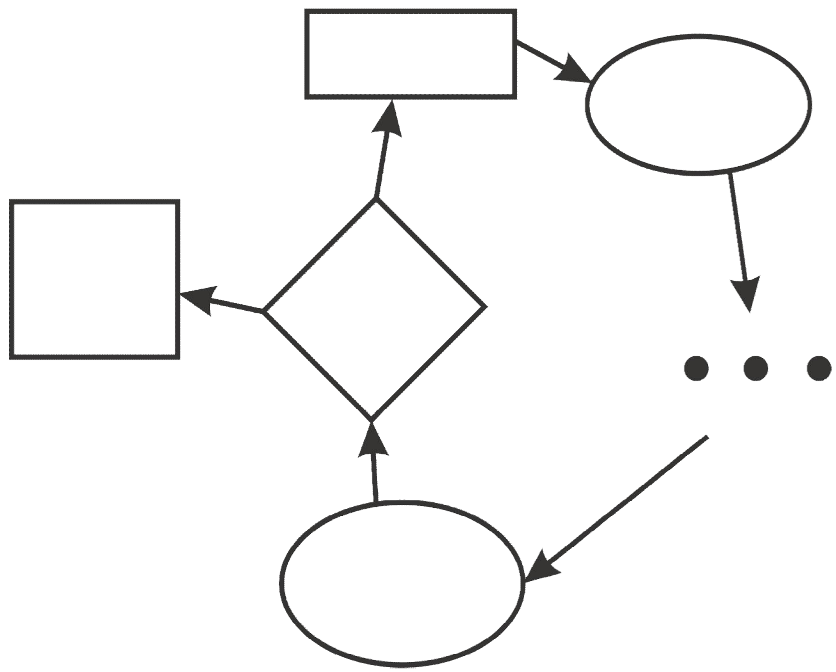
Generic DMFT loop

different shapes used in different ways



Leave the details to be determined later.

The basic structure is always the same! The number of circles may vary



Generic DMFT-like loop

C++11

```
template <typename Data, typename ... F >
bool generic_loop( Data data,
                   Mixer<Data> & mixer,
                   double accuracy,
                   int n_iterations,
                   F... f ){
  cout << "generic_loop!" << endl;
  for(int it=0; it<n_iterations; it++) {
    cout << "iteration: " << it << endl;
    int x[] = {Wrapper(f, data)...};

    if (mixer()<accuracy) {
      cout << "converged!!" << endl;
      data.dump_final();
      return true;
    }
    data.dump_intermediate(it);

  }
  cout << "didn't converge!!" << endl;
  return false;
```

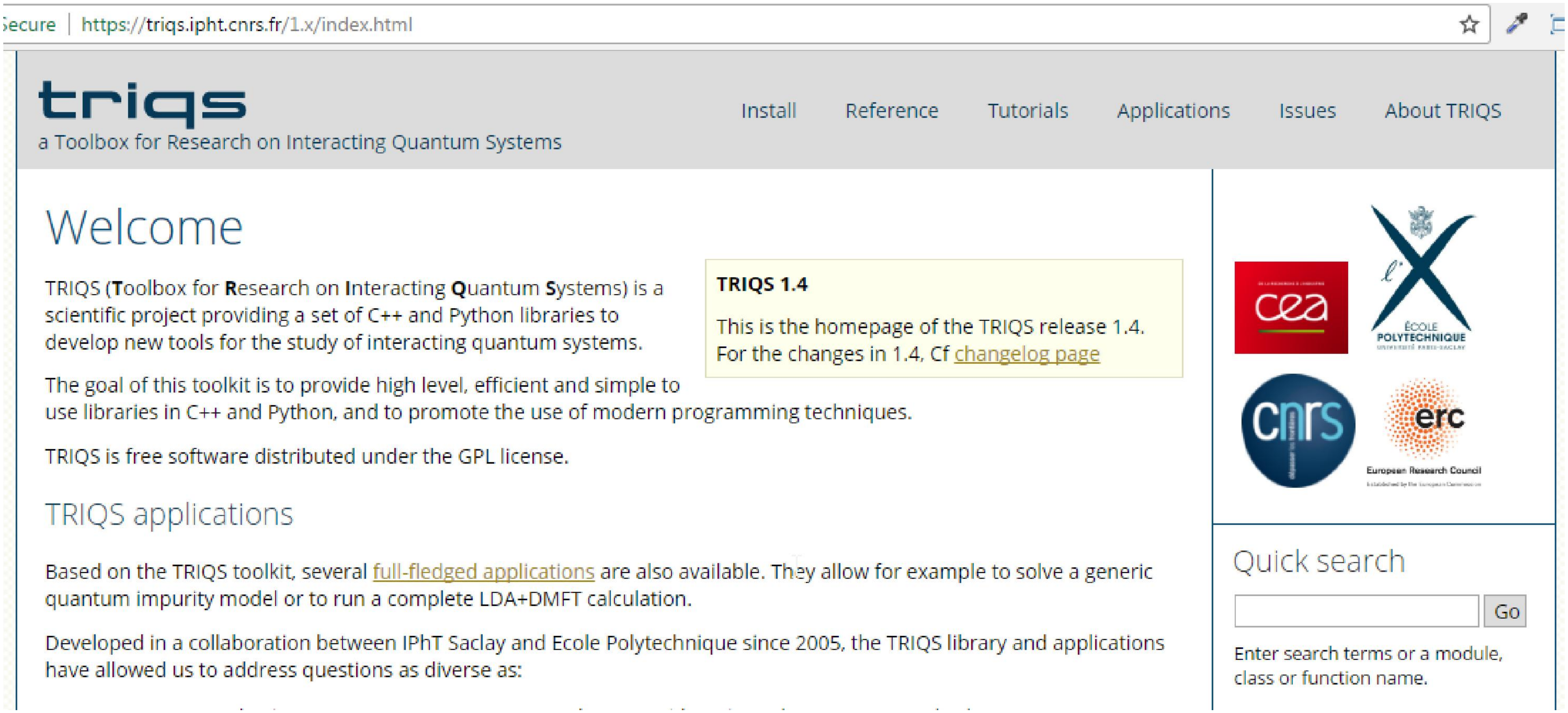
Generic DMFT-like loop

Python

```
def generic_loop( data,
                  mixer,
                  accuracy,
                  n_iterations,
                  *funcs ):
  print "generic_loop!"
  for it in range(n_iterations):
    print "iteration:",it

    for func in funcs:
      func(data)

    if mixer()<accuracy:
      print "converged!!"
      data.dump_final()
      return True
    data.dump_intermediate(it)

  print "didn't converge!!"
  return False
```

triqs

Toolbox for **R**esearch on **I**nteracting **Q**uantum **S**ystems



triqs

Toolbox for **R**esearch on **I**nteracting **Q**uantum **S**ystems

Main goals

- act as a bridge between c++ and python so as
  to allow for both painless manipulation
  of data (in python) and high-optimization of critical routines (c++)

- provide containers for common objects
  in condensed matter theory
  (multidimensional arrays (in c++) ,Green's functions,
  second-quantized Hamiltonians, etc.)

- provide generic implementation of common
  algorithms (monte carlo, Hilbert transform, FT, tail fitting...)

- provide a simplified and intuitive interface to MPI and HDF5

c++2py wrapper

- allows for dynamical linking of python code with precompiled c++ libraries
- the “wrapping” produces python modules with “pythonically” callable functions and classes
  - each c++ class gets a python version of itself and one may even choose which properties of the class will be visible in python
- basic types are simply equated between python and c++
  no need to invoke any special integers, floats, string, etc.
- std types also wrapped up naturally (e.g. vector -> list)
- wrapper is based on the intermediate representation of clang compiler and cmake project structure
- the python modules are generated automatically with a single command

triqs::array

- c++ std library has no convenient multidimensional array container
- triqs::array is analogous to numpy.array in python
  and allows for many of the same functionalities
- linear algebra made easy!

```
array<int, 2> A(2, 3);

foreach(A, [&A](size_t i, size_t j) { A(i, j) = i + j; });
std::cout << A;
// prints
// [[0,1,2]
//  [1,2,3]]

cout << get_shape(a); //prints (2,3) //introspective!

placeholder<0> i_;
placeholder<1> j_;
A(i_, j_) << i_ + j_; //lazy experssions
cout << A(0,1); //prints 1 //other elements are not yet evaluated!
cout << A;
// prints
// [[0,1,2]
//  [1,2,3]]
```

triqs::array

- c++ std library has no convenient multidimensional array container
- triqs::array is analogous to numpy.array in python
 and allows for many of the same functionalities
- linear algebra made easy!

```
array<int, 2> A(2, 3);

foreach(A, [&A](size_t i, size_t j) { A(i, j) = i + j; });
std::cout << A;
// prints
// [[0,1,2]
//  [1,2,3]]

auto B=A+0.2; //elementwise, automatic cast to array<float,2>
cout << B;
// prints
// [[0.2,1.2,2.2]
//  [1.2,2.2,3.2]]

array_view<int, 1> A1 = A(1, range()); //selects first row of A
cout << A1; //[1,2,3]
auto C = A * B; //matrix product!!!
```

triqs::array

- c++ std library has no convenient multidimensional array container
- triqs::array is analogous to numpy.array in python
 and allows for many of the same functionalities
- linear algebra made easy!

```
A = numpy.fromfunction(lambda i, j: i + j, (2, 3), dtype=int)
print A
# prints
# [[0 1 2]
#  [1 2 3]]

print numpy.shape(A)
# prints (2,3)

A1 = A[1,:]
print A1 #prints [1 2 3]

B = A+0.2
print B
# prints
# [[0.2 1.2 2.2]
#  [1.2 2.2 3.2]]

C = numpy.dot(A,B) #multiplication is elementwise! use dot
```

HDF5

Hierarchical **D**ata **F**ormat

- HDF5 is a data model, library, and file format for storing and managing data
- standard, **well maintained** and **widely used**
- supports an unlimited variety of data types, and is designed for flexible and **efficient I/O** and for **high volume** and **complex data**
- portable and is extensible
- HDF5 can be read and written in many languages

HDF5

Hierarchical **D**ata **F**ormat

```
A = HDFArchive('myfile.h5', 'w') # Opens a file in read/write mode
A['mu'] = 1.29
A['a'] = numpy.array([1,2,3])
A['obj'] = vars(obj) #obj instance of class myclass defined elsewher

#later...

A = HDFArchive('myfile.h5', 'r') # Opens a file in read mode
mu = A['mu']
print mu #prints 1.29
a = A['a']
print a #prints [1 2 3] #numpy.array ready to use!
obj = myclass.from_dict(A['obj']) #classmethod, lines...
print obj.a, obj.b #Loaded whole object in one command
```

Take away messages

- High-level programming makes life easier!
    - Generic and meta programming allow for separation between various levels of detail of an algorithm
    - no huge class hierarchies – instead small pieces we put together as we need them, when we need them

- C++ is no longer so low-level, but still allows for huge optimizations and should be used for computationally intensive routines.

- Combination of c++ and python ideal! Install triqs give it a try!

- Need multidim array in c++? triqs::array does the job

- No more text files, no more formatting or mysterious binaries – store data in HDF5 format!