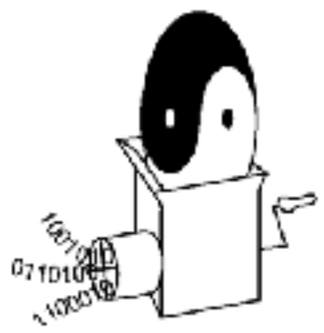


CS 4120/5120
Introduction to Compilers

Andrew Myers

Cornell University

Lecture 37

Run-Time Type Discrimination, Variants, and Nonlocal Control Flow

2 May 2018

Compiler project

- Due date: May 20, 2pm
  - –**Hard deadline**.
  - –No room for error—plan early and often, aim to finish a at least couple of days early
  - –Got test cases?
- QiXi, full-featured OO Xi UI lib now available
- Compiler competition!
  - –Correctness, speed, compiler engineering
  - –Winners receive plaque, bragging rights.

Run-time type discrimination

- How to discover types at run time?
  - n tag bits ⇒Tag 2^n-1 primitives, align memory to 2^{n-2} words, some performance hit, range limitation on ints ($x \rightarrow 2^n x$)
- o instanceof T, (T)o, typecase o of $T_1 \Rightarrow s_1 \mid T_2 \Rightarrow s_2$

1. look up DT pointer, class descriptor in hash table containing type relationships (may be filled lazily)
2. (SI only, separate compilation) Record superclasses sequentially in DT (**display**). instanceof C ⇒ check if class at depth depth(C) is C.
3. (Single inheritance only) in-order traversal of hierarchy with classes numbered sequentially ⇒ all subclasses of C in contiguous range. Test class index in range with single unsigned comparison.
4. Quick range test (ala #2) can be done even with MI using **PQ-trees**.

Exceptions

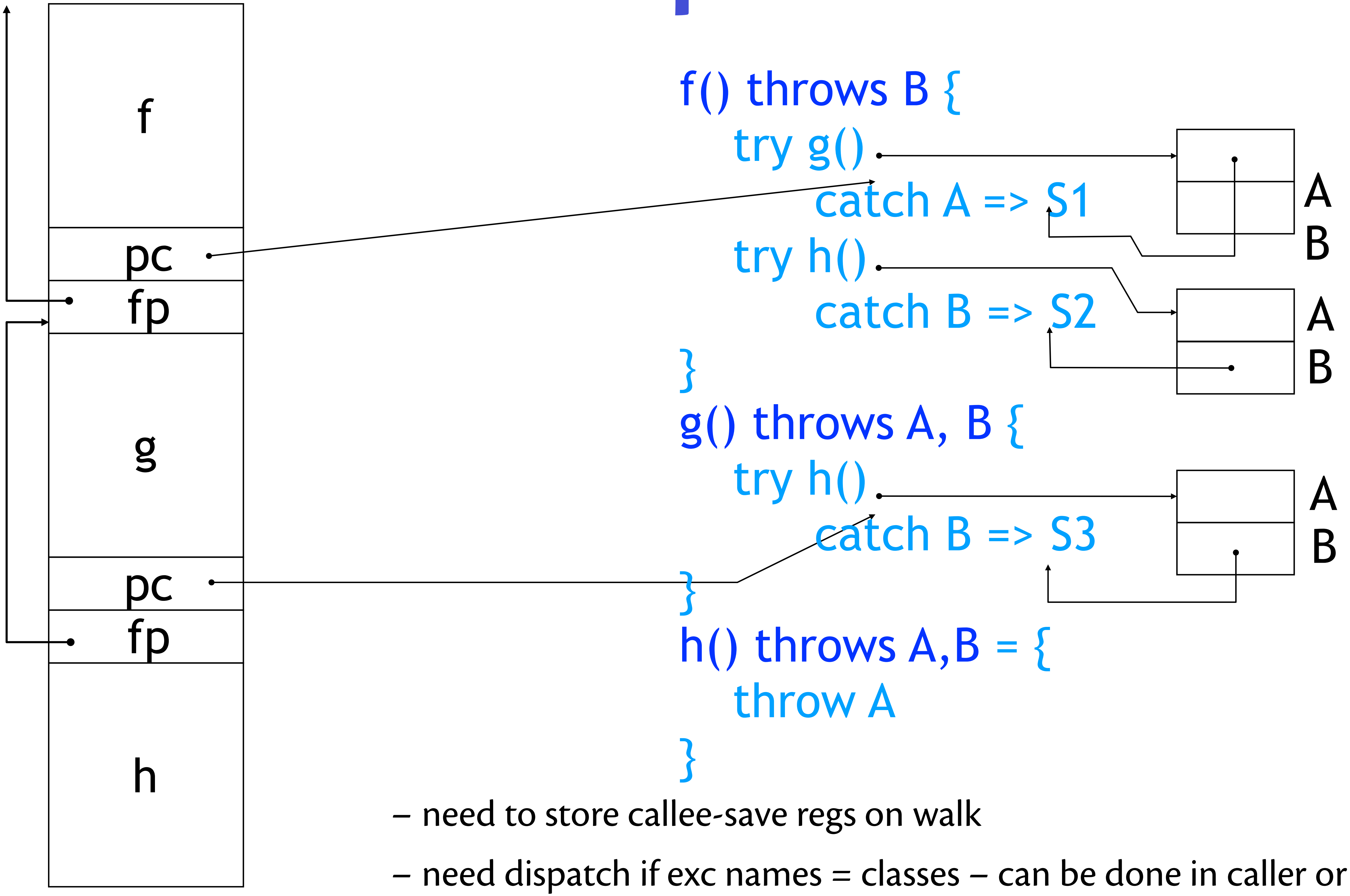
- Most languages allow *exceptions:* alternate return paths from a function
  - null pointer, overflow, emptyStack,...
- Function either terminates *normally* or with an exception
- – *total* functions ⇒ robust software
- – normal case code separated from unusual cases
  - no ignorable encoding of error conditions in result (e.g., null)
- Exception propagates *dynamically* to nearest enclosing `try`...`catch` statement (up call tree)
  - Tricky to implement dynamic exceptions efficiently
  - Result: underused by programmers (e.g., `Map.get`)

Exceptions: goals

1. normal return adds little/no overhead
2. try/catch free if no exception
3. catching exception ~ cheap as checking for error value
    - C/C++: setjmp/longjmp. Try/catch expensive.

- **Static exception tables** (CLU):
    - insight: can map pc to handler in each function.
    - on exception: climb stack using return pc, look up exception handler at each stack frame (binary search on pc)

Example



```
f() throws B {
    try g()
        catch A => S1
    try h()
        catch B => S2
}
g() throws A, B {
    try h()
        catch B => S3
}
h() throws A,B = {
    throw A
}
```

– need to store callee-save regs on walk

– need dispatch if exc names = classes – can be done in caller or (with larger tables in throwing code)

Fast exceptions payoff

- Translating enhanced for-loops with exceptions!

```
for (T x : c) {
  … body …
}
```

Conventional translation:

```
Iterator<T> i = c.iterator()
while (i.hasNext()) {
  T x = i.next();
  …body…
}
```

Exception-based translation:

```
Iterator<T> i = c.iterator()
try {
  while (true) {
    T x = i.next();
    …body…
  }
} catch (NoSuchElement …) {}
```

- Exception-based translation replaces *N* calls to `hasNext()` with one `throw/catch`.
- Typically faster if exceptions are implemented well!

Coroutine iterators

- Another CLU idea: iteration via coroutines
- Now in C#, Python, Ruby, JMatch:

C# : CLU-style iterators (generators)

```
public static IEnumerable<int> elements() {
 if (left != null)
   foreach int x in left.elements())
     yield return x;
 yield return value;
 if (right != null)
   foreach (int x in right.elements())
     yield return x;
}
```

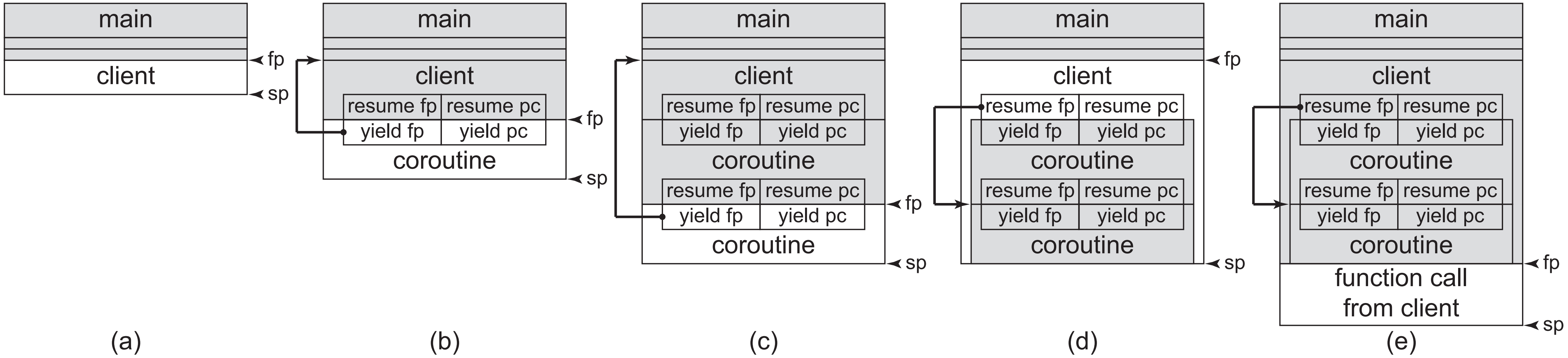
JMatch modal iterative abstractions:
2 methods for the price of 1

```
public boolean contains(int x) iterates(x) (
   left != null && left.contains(x)
|  x == value
|  right != null && right.contains(x)
)
```

```
foreach (c.contains(int x)) { … }
```

Stack-allocating coroutines

- Client and coroutine share same stack
  - –Frame pointer and stack pointer in different stack frames!
  - –Coroutine activation record can be stack-allocated if it doesn't escape
  - –Can't do this in JVM, but LLVM should support it



–*Tail-yield* optimization allows yielding values directly through a chain of coroutines

JMatch

- Modal abstractions are concise *and* efficient:

Expressiveness (LOC)

	Java	JMatch	Savings
ArrayList	204	112	45%
LinkedList	249	155	38%
HashMap	434	158	64%
TreeMap	805	472	41%
Total	1692	897	47%

Performance vs. C++ STL

Average 3% difference iterating 250k elements: LinkedList, HashMap, TreeMap vs. STL equivalent

- More results in paper, including vs. Java

Wanted: a robust high-performance back end for JMatch

Nonlocal control

- Language mechanisms for nonlocal control (exceptions, coroutines) are useful, can be implemented efficiently
- But: poorly supported by rigid stack discipline of most current VMs (JVM, CLR)

  Exception: LLVM directly supports exceptions and coroutines.