Compiler project

• Due date: May 19, 11:30am
  –Hard deadline.
  –No room for error—plan early and often
  –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 T₁ ⇒ s₁ | T₂ ⇒ s₂

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

• Many languages allow *exceptions*: alternate return paths from a function
  – null pointer, overflow, emptyStack,...
• Function either terminates *normally* or with an exception
  – *total* functions $\Rightarrow$ 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 (see Map.get, etc.)
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
- lookup complicated if exc names = classes – need dispatch
Coroutine iterators

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

C# : CLU-style iterators (generators)

```csharp
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 for the price of 1

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

```java
foreach (c.contains(int x) && d.contains(x))
    { ... }
```
Stack-allocating coroutines

- Client and coroutine share same stack
  - Frame pointer and stack pointer in different stack frames!
  - Can’t do this in JVM

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

- Modal abstractions are concise and efficient:

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>JMatch</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>204</td>
<td>112</td>
<td>45%</td>
</tr>
<tr>
<td>LinkedList</td>
<td>249</td>
<td>155</td>
<td>38%</td>
</tr>
<tr>
<td>HashMap</td>
<td>434</td>
<td>158</td>
<td>64%</td>
</tr>
<tr>
<td>TreeMap</td>
<td>805</td>
<td>472</td>
<td>41%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1692</td>
<td>897</td>
<td>47%</td>
</tr>
</tbody>
</table>

- 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 current VMs (JVM, CLR, LLVM...)