Compiler project

- Demos coming up Dec 13-16: sign up!
  - Submit your code with your demo
- Due date: December 16
  - 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.

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 ⇒ 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

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

```java
g() throws A, B {
    try h() {
        catch B => S3
    }
}
```

```java
h() throws A, B = {
    throw A
}
```

- need to store callee-save regs on walk
- lookup complicated if exc names = classes – need dispatch

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 ($x \rightarrow 2^n x$)
  - instanceof $T$, $(T)o$, typecase $o$ of $T_1 \Rightarrow s_1$ | $T_2 \Rightarrow s_2$
    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.

Coroutine iterators

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

C#: CLU-style iterators (generators)

```java
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)
    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 \textit{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>

---

Metaobjects

• Some languages (Smalltalk, Java, …) expose classes as objects (metaobjects)
  – query methods, fields, inheritance structure...
  – good for building compilers, run-time adapters, serialization code... \textit{not regular code}
• Metaobject protocol: methods exposed for querying classes, other type-level entities
• Java 1.5+: parametric polymorphism not reflected – really JVM metaobjects

---

Generalized LR parsing

• Some parser generators (e.g. PPG) support \textit{grammar inheritance} to support language extension
  – Problem: LALR grammars are not very extensible
• GLR parsing: conflicts resolved late by forking the parser stack. Compiler must reconcile alternate parsing results.
• Another nice idea: parser feedback to lexer to identify next legal tokens

---

Path and object sensitivity

• \textbf{Flow-insensitive:} same information throughout code (type checking)
• \textbf{Flow-sensitive:} information per program point
• \textbf{Context-sensitive:} information per calling context
• \textbf{Path-sensitive:} information per execution path leading to program point.
• \textbf{Object-sensitive:} information per method receiver object. Helps with points-to analysis.
Abstract interpretation

- Many forward analyses can be viewed as instances of abstract interpretation
- Idea: analysis ~ running the program, but mapping actual program state to a simplified abstract state.
  - Example: points-to analysis using abstract heap, a relation on "variables" and "objects".
- Transfer function is an abstraction of computation. Maps input abstraction to an output abstraction that includes all feasible concrete outputs.
- Convergence = run loops until abstract state converges.
- A rich, general mathematical structure for explaining and developing program analyses.

Attribute grammars

- Essentially a type system for program analysis and synthesis, with extra constraints in rules.
- Typing rules generate additional information about program (analysis results, output machine code, ...)
- Iterative constraint solving, not recursive type checking – information flows up and down in AST in complex ways.
- Examples:
  - Synthesizer Generator (Teitelbaum): a Cornell compiler framework based on attribute grammars.
  - JastAdd: a Java compiler based on rewriting attribute grammars.

Optimizing for locality

- 100+-fold speed difference between memory and cache ⇒ locality is crucial for performance.
- Some important tricks for matrices:
  1. Transpose matrices so loops go across rows.
  2. Pad rows to avoid cache conflicts
  3. Rewrite nested loops with outer loops over blocks, inner loop within each block.

Scalarization

- Avoiding indirection improves locality: scalarization
- Passing objects, records, tuples as ordinary arguments, returning copies on stack

  ```java
class Point { int x, y }
f(Point p): p ⇒ f(int px, int py) : int, int
```

- Inlining objects and arrays into referencing structures

  ```java
class Line {Point p1, p2} ⇒
class Line {int x1, y1, x2, y2}
```

- Requires exact type and escape analysis (or careful language design), has GC implications
**Instruction scheduling**

- Key: want to keep every pipeline stage of processor busy.
- Order of instructions matters; hard to predict effect.
  - Start load instructions early
  - Intel: hardware translates instructions to RISC-like micro-ops.
- Instruction scheduling: low-level optimization on assembly code.
  - Reorder instructions subject to dependencies between instructions (topological sort, need alias analysis...)
  - Scheduling is traversing dependency DAG on instructions
    - heuristics to start important work early, keep functional units busy.
    - Knowing ISA is not enough.
- Need to schedule before and after register allocation.

**Type-preserving compilation**

- Idea: compiler propagates types to compiled code. **Verifier** checks to see compiled code is safe.
  - Code consumer doesn’t have to trust compiler or compiled code.
  - Examples: Java bytecode, Typed Assembly Language (TAL).
- Bytecode verification is a dataflow analysis.
  - Dataflow values = mapping from locals, stack locations to types.
- Challenge: low-level code needs complex types.
  (Type of stack pointer? program counter?)

**Closing thoughts**

- Ability to build or modify compilers opens new ways to solve problems. Valuable knowledge!
  - Many uses for domain-specific languages—look for opportunities.
  - C, Java are reasonable target languages – let someone else write the optimizer! (except: exceptions, threads, coroutines, dispatching, transactions, ...)
- Possible next steps:
  - CS 6110: Advanced programming languages (theory, SP12)
  - CS 4110: Programming languages (language features, FA12)
  - TA this course in FA13