Part IX : Object-Oriented Dispatch Methods

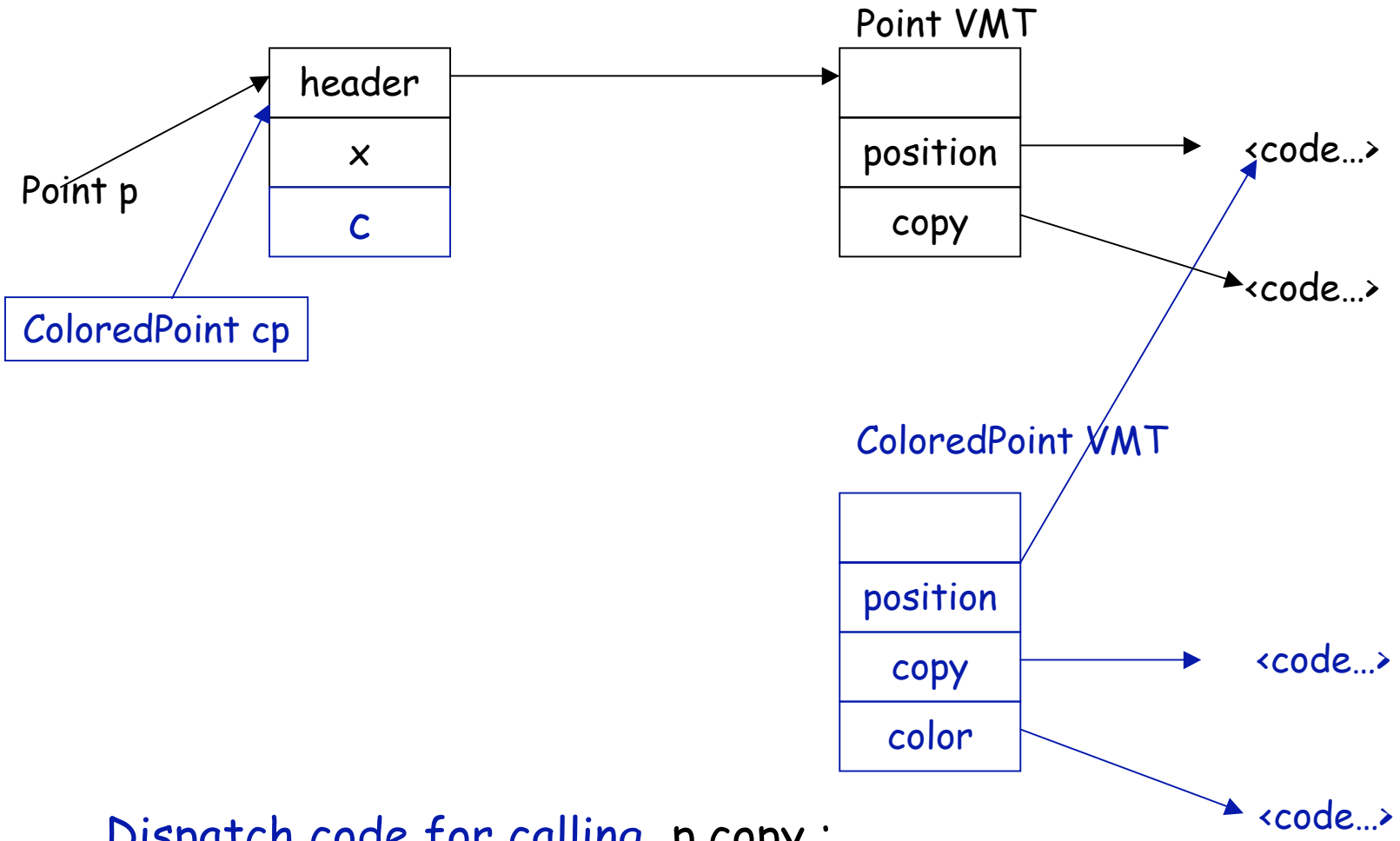
- OO languages support *dynamic binding*: A method call will invoke code which depends on the run-time type of the receiver object.
- Because of subtyping, the run-time type may differ from the static type, known at compile time.
- Problem: How to do dynamic disptach efficiently.

The Single-Inheritance Case

- Dynamic dispath is relatively easy to implement in the single inheritance case, where every class has at most one superclass.
- Example:

```
class Point {
   private int x;
   Point (int x) { this.x = x; }
   int position() { return x; }
   Point copy (int delta) {
      return new Point (position() + delta);
   }
}
class ColoredPoint extends Point {
   private Color c;
   ColoredPoint (int x, Color c) { super(x); this.c = c }
   Color color () { return c; }
   ColoredPoint copy (int delta) {
      return new ColoredPoint (x + delta, c);
   }
}
```

Graphic 1



Dispatch code for calling p.copy :

p.header.copy()

Method Dispatch

- The VMT dispatch scheme is predominant in single-inheritance situations.
- Single inheritance languages:
  - Simula, Smalltalk, Modula-3, Ada 95, Object Oberon, ...
- Multiple inheritance languages:
  - Eiffel, Beta, C++, ...
- Languages with structural subtyping
  - Cecil, Self, Pict, ...
- Hybrids -- single inheritance + interfaces:
  - Java, Objective C

Techniques for multiple inheritance and hybrids

- Trampolines
- Row-displacement tables
- Inline caching

Multiple Inheritance Example

```
class Point {
  private int x;
  Point (int x) { this.x = x; }
  int position () { return x; }
  Point copy (int delta) {
    return new Point (position() + delta);
  }
}

class Colored {
  private Color c;
  Colored (Color c) { this.c = c; }
  Color color () { return c; }
}

class ColoredPoint extends Colored, Point {
  ColoredPoint (int x, Color c) {
    Point.super (x); Colored.super (c);
  }
  ColoredPoint copy (int delta) {
    return new ColoredPoint (position() + delta, color());
  }
}
```

Trampolines

- Idea: Have multiple entry points for references, one per base class.
- Every entry point gets a header field pointing to a VMT
- When passing from a subclass to its superclass, we update the object pointer to point to the correct entry point.
- Overriding makes it necessary to move from an entry point to the start of the enclosing object.
- This is achieved by a *trampoline* method which, when called, returns the reference of the enclosing object by subtracting a known offset from the entry point.
- This technique has been used in Beta and C++.

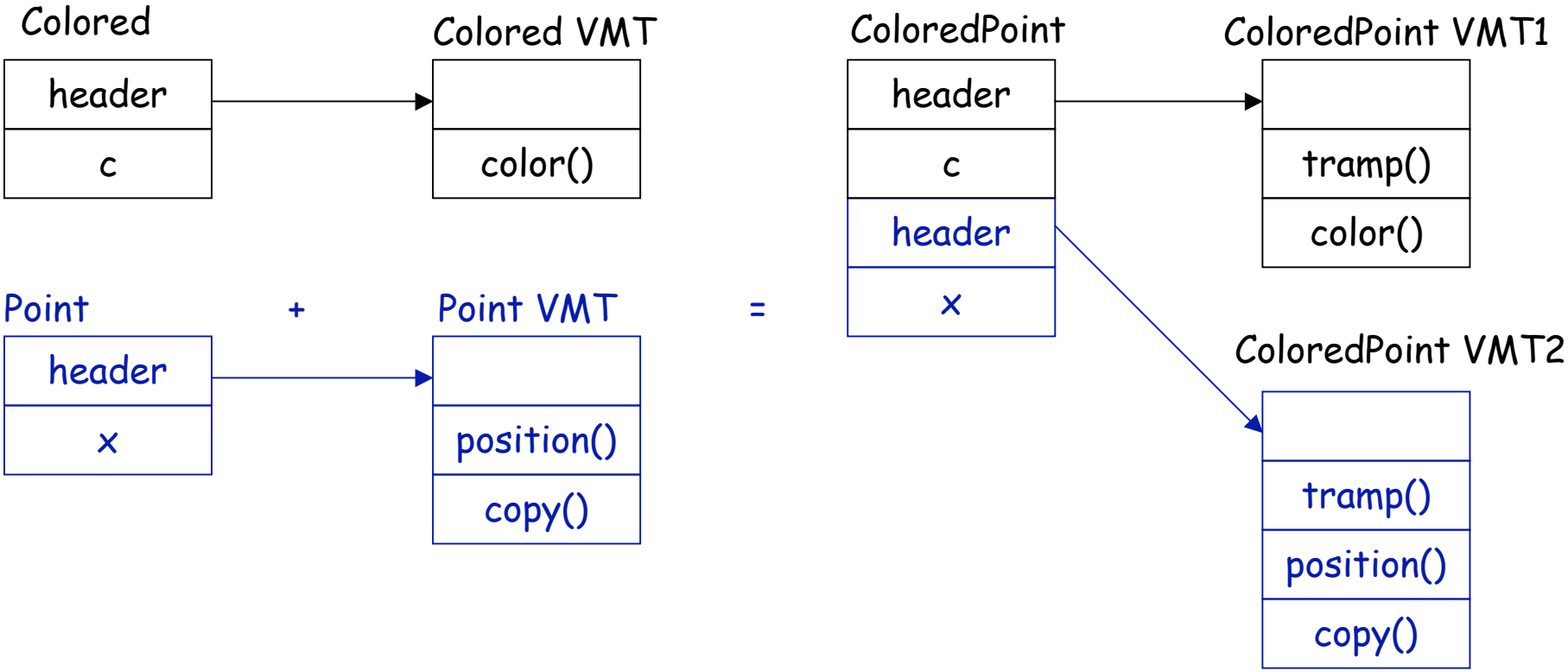
Trampolines (2)

- Advantages: Reasonable performance even in the worst case, both fields and methods can be multiply inherited.
- Disadvantages: Added cost of trampline methods, covariant datastructures are not supported. E.g. in Java:

```
ColoredPoint[] <: Point[]
```

- This can't work with trampolines since we would have to update every pointer in the array.

Graphic 2



Point p; ColoredPoint cp;

p = cp ⇒ p = cp + 8;

cp = p ⇒ cp = p.tramp(p)

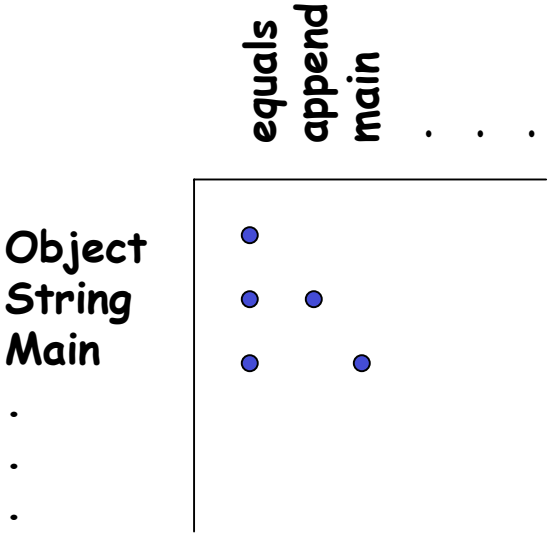
tramp(p) = p

tramp(p) = p - 8

Row Displacement Tables

Conceptually, the task of dynamic dispatch is as follows:

- Given a number of classes and a number of methods, find the method code corresponding to a given class and a given method name.
- If we number both classes and methods, this task can be reduced to a single indexing operation in a two dimensional table.



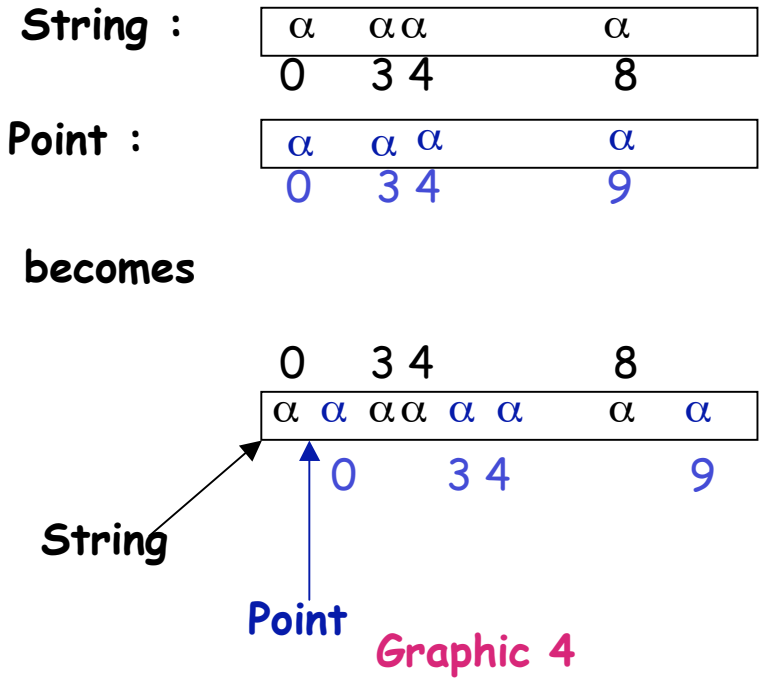
- Problem : This table would grow very big : Typical application : 500 classes, 2000 unique method names.

Graphic 3

Compacting the Table

- The two-dimensional table is occupied only sparsely, because any given class implements only a small subset of all methods.
- We can get a better space utilization by overlaying the rows of this table like a set of combs.

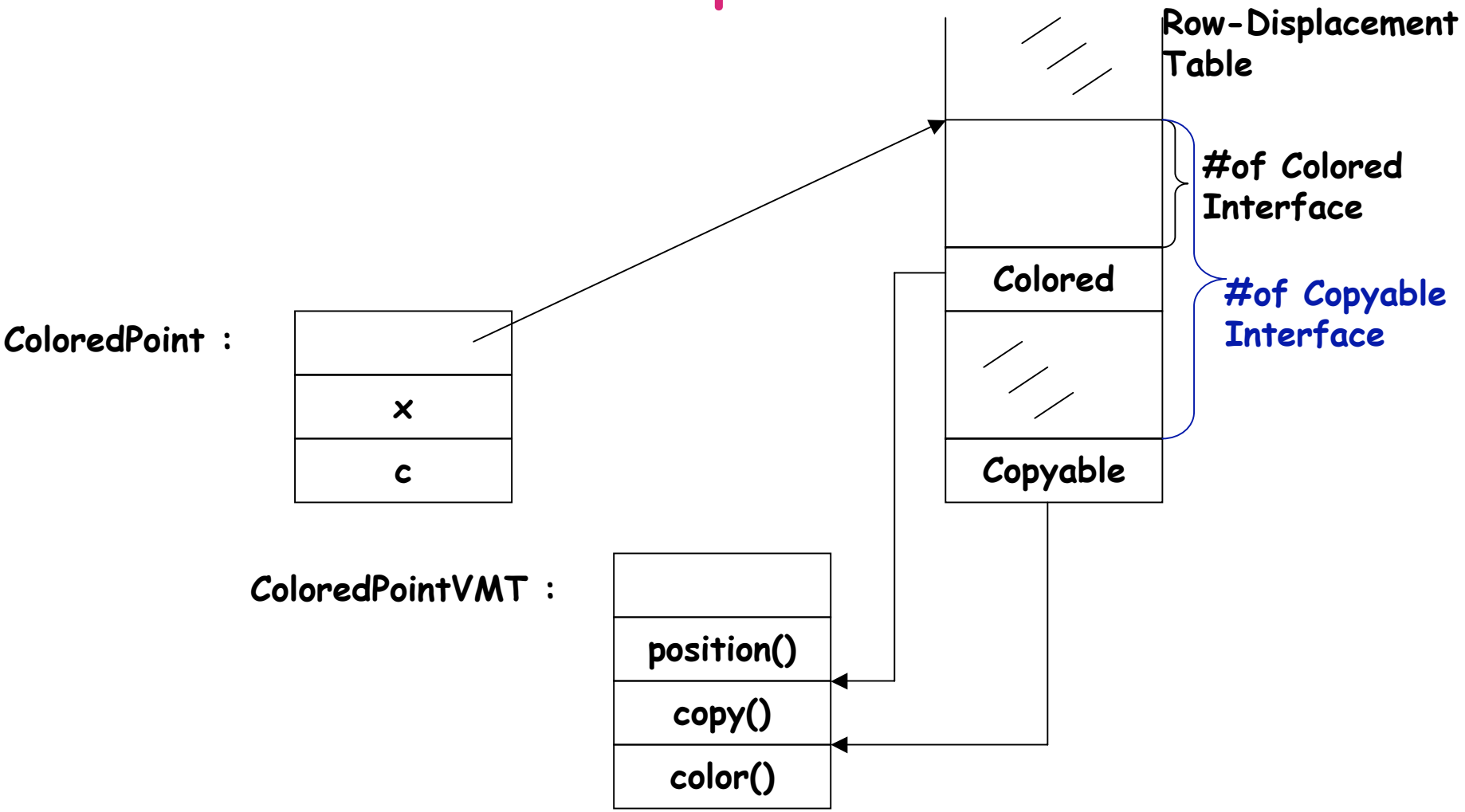
Compacting the Table (2)



Graphic 4

- Variation for Java : Index the table with classes and interfaces instead of classes and methods. A table entry points back to the place in a VMT where the interface 's method are implemented.
- This technique has been used in some high-speed Java implementations.

Graphic 5


Row-Displacement Table
#of Colored Interface
#of Copyable Interface
Colored
Copyable
ColoredPoint :
x
c
ColoredPointVMT :
position()
copy()
color()

Row Table Dispatch

- Code for `I obj; ... obj.meth():`

  `obj.header[I.number].meth()`

- How do we know that the table entry is used for the current class?
- We don't have to, because Java is statically typed!
- Advantage of row table dispatch: Good performance with average case = worst case.

Pipelining Considerations

- Both VMT dispatch and row-table dispatch include *pipeline bubbles.*
- New instructions can only be fetched after computing the dynamic method address.
- In modern multi-scalar processors with deep pipelines, this can be very costly.
- Example: Pipeline of length 6 (+ writeback), 4 way issue: 24 missed intructions!
- And things are getting worse with VLIW.

Inline caching

- Observation: Many calls always go to the same class.
- Idea: For every call instruction, remember the code that was branched to during the last execution of the instruction.
- Immediately jump to that code without using dynamic dispatch.
- At the start of the target code, we have to test whether we are in fact in the right class.
- If we are not, go back to the slower dynamic dispatch scheme.
- This scheme can be a big win, if calls always go to the same class.
- The scheme is a big loss if they don't.
- Examples?

Inline caching and the JVM

- Inline caching is used to implement calls of interface methods in Sun's JVM interpreter.
- The instruction **`invoke_interface`** has a field which contains the offset of the method entry that was invoked in the last execution of the instruction (relative to start of VMT).
- When **`invoke_interface`** is executed, it is checked first whether an entry for the called method is at the given offset.
- If not, all methods of the given object are searched linearly for one that matches the (name and type) of the called method.

Polymorphic Inline Caching

- Inline caching is an all-or-nothing optimization -- it's either very fast or no help at all (even slowing down computation).
- Better compromise: Keep a table of the last *n* jump targets.
- If the current target is in the table, jump directly, otherwise go through dynamic dispatch and add new target to table.
- If the table grows too large, revert to all dynamic dispatch.
- This scheme is described in Urs Hölzle's thesis.
- It is used in Self and the Hotspot implementation of Java.
- Advantage: Pipleline bubbles are avoided ⇒ potentially very good performance, better even than simple VMT dispatch for single inheritance.
- Disadvantage: Unpredictable: Can be (a little bit) worse than VMT dispatch in bad cases.

JIT Compilers

- Interpretation of Java bytecodes incurs overheads which result in poor performance.
- Distribution of Java classes as native code would improve performance, at the price of portability and security.
- JIT compilers are a way out of this dilemma.
- A JIT compiler compiles bytecode to native code, either at load time, or once the code is executed a number of times.
- In principle, JIT-compiled code can be faster than statically compiled native code since more informations are available at run-time than at compile-time. (E.g. which methods are called the most often? How many different methods are invoked by this call?

JIT Compilers (2)

- In practice, JIT compiled code is usually slower than native code
  - because the compilation overhead adds to the run-time cost.
  - because JIT compiler opimizations need to run fast, and are therefore less agressive than native code compiler optimizations.
- Tradeoff: Slow compiler + fast generated code or fast compiler and slow generated code?
- Related tradeoff: When to invoke JIT compiler?
  - Symantec: At first execution
  - Inprise : At 2nd execution
  - Hotspot : After 10000 executions
  - Client hotspot: After 1000 executions

Summary

- We have studied fundamental techniques for constructing compilers.
  - Syntax analysis (⇒ top-down and bottom-up)
  - Tree matching and manipulation (⇒ visitors)
  - (Byte-)code generation
- We know now how programs are represented internally
  - As token sequences
  - As trees with symbol tables and attributes
  - As code sequences
- We have experienced construction of a sizeable program where theory was translated into design and coding patterns.

What was missing

- Advanced type systems with genericity
  - How do we specify typing rules?
  - How can we implement generic functions and classes?
- Semantics-based compilation
  - how do we specify the meaning of programs in a given language?
  - how do we increase the likelyhood that compilers preserve meaning?
- Optimization
  - How can we generate executables that run faster?
  - Analysis techniques (e.g. Dataflow)
  - Optimization

Comments ?

- What aspects of the course did you like best? least?
- What topics do you want to know more about?
- What topics do you think should be treated in less detail?
- Did you learn more from the lecture or from the project?
- Comments are very welcome; they help us improve the course.