Bit-Level Object Oriented Programming in Kava

David F. Bacon

IBM T.J. Watson Research Center

Research Goal

- A single high-level programming language
  - Compilable to software
  - Synthesizable to hardware (an orderly subset)
- Avoid need to switch domains
- Modular abstraction
  - Write synthesizable modules
    - Using non-synthesizable techniques (like malloc)
    - Modules can be step-wise refined until synthesizable
- At high level, synthesis and real-time are the same

Bridging to VHDL to Java Gap

- Concurrency: Java is global and MP-oriented
  - Process-like concurrency abstractions (Guava)
- Finite Space: In Java, all objects on the heap
  - Region-based memory management
  - Static analysis techniques
  - Reference counted, concurrent memory allocation
- Finite Time
  - Static analysis
- Bit-Level Abstraction (Kava)

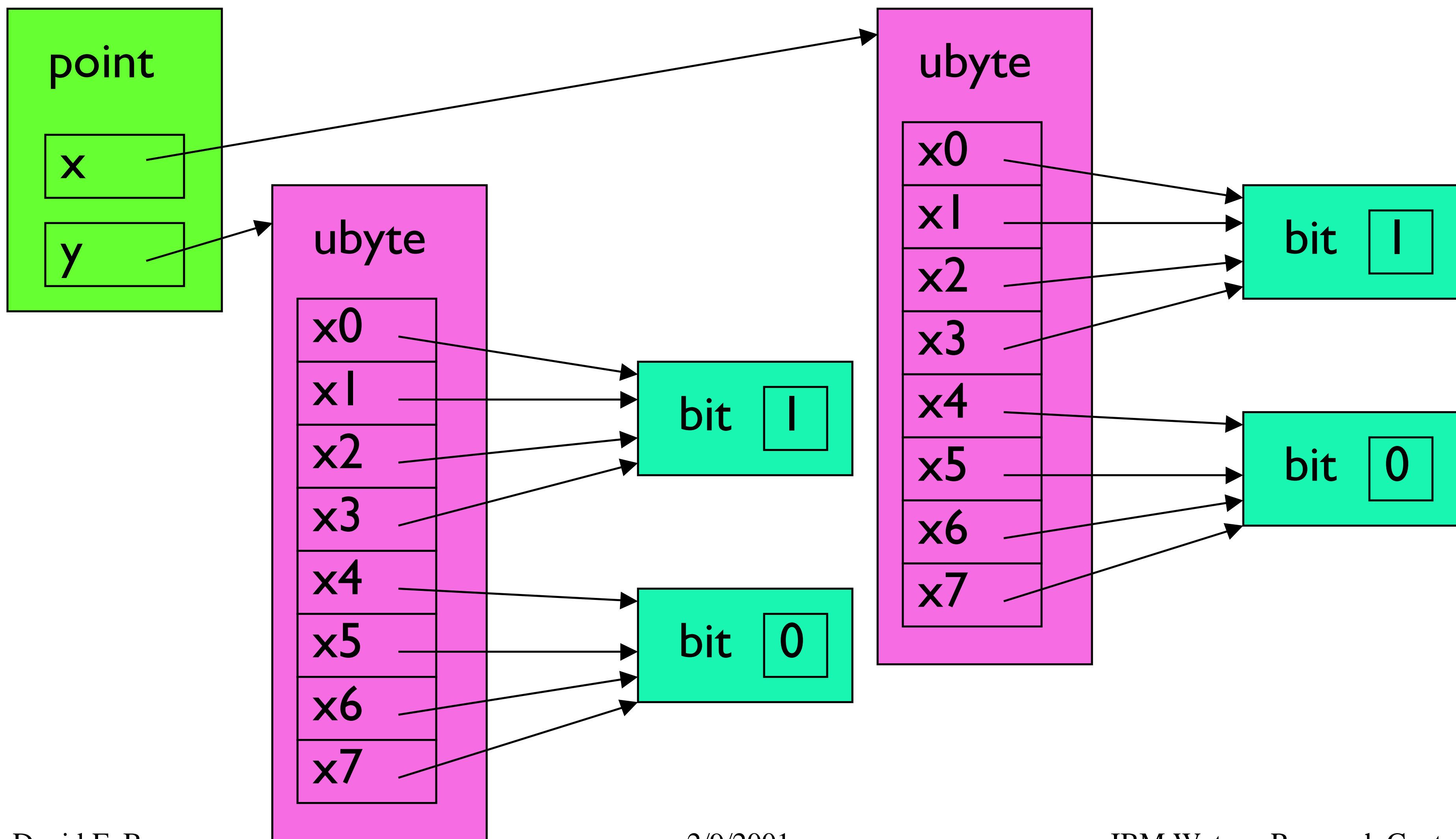
The Kava Problem

- Distinction between objects and primitives
  - In C++ and Java, totally non-uniform
  - In Smalltalk, integers look like objects
    - But can't be defined within the language
- "Unclean" model
- Creates pressure for additional primitives
- Some language semantics hardware defined

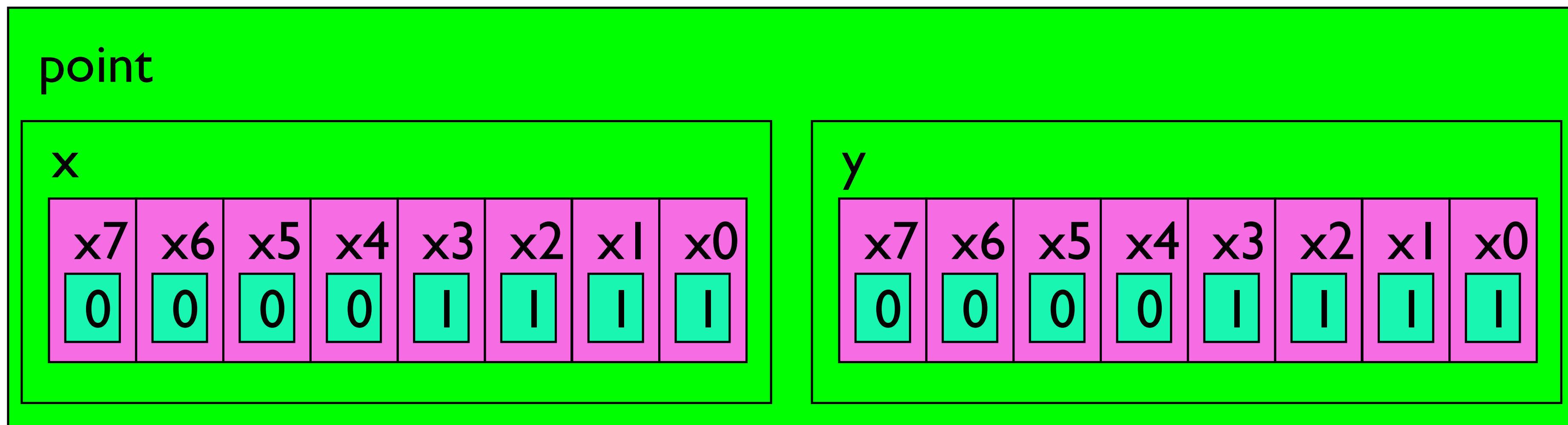
Solution: Bit-Level OO Programming

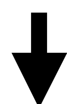
- Extend OO abstraction down to the bit level
- Eliminate primitive/object distinction
  - Expose value semantics to programmer
- Allow user-defined primitives
- Work in progress
  - Defined as a backward-compatible Java extension
  - Implementation in progress

Uniform Object Model



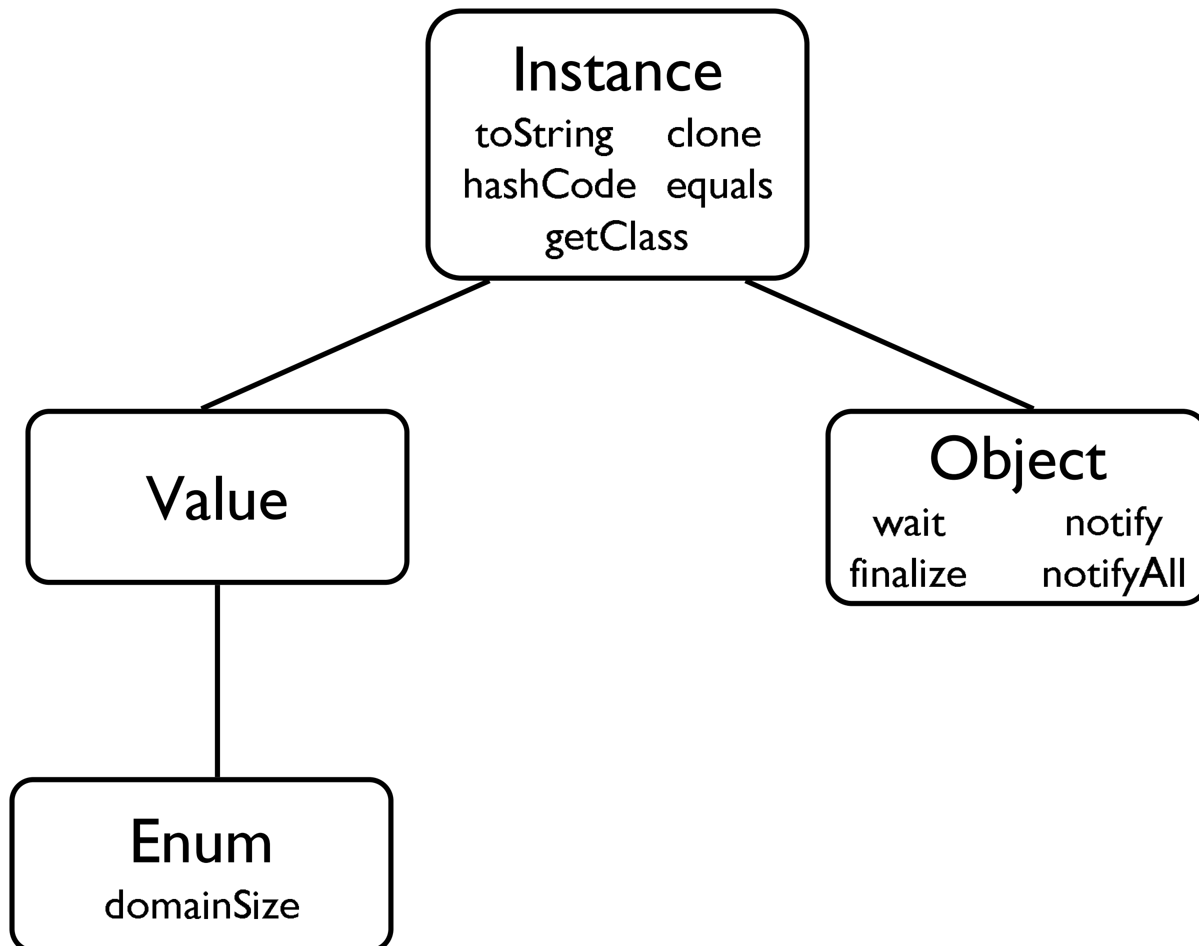
Efficient Implementation





point(15,15) 0000111100001111 (16-bit halfword)

Class Hierarchy



Enumerations

```
enum day {
    Monday, Tuesday, Wednesday, Thursday,
    Friday, Saturday, Sunday;

    public boolean weekend() {
        return this == Saturday || this == Sunday;
    }
}

day d = Monday;
if (d.weekend())  System.out.println("Let's party");
```

Enumeration-Based Arrays

```
int happiness[day] = { 0,1,4,3,7,10,8 };

if (happiness[Monday] > happiness[Saturday])
    System.out.println("Try going sailing!");
```

- Subscripted by enumerations instead of ints
- Fixed size
- Never require run-time bounds checks
- Base case for numeric types
  - Can define int without need for int

User-Defined boolean Type

```
enum boolean {
    false, true;

    public boolean ! this { return not[this]; }

    public static final boolean not[boolean] = { true, false };

    public boolean this & boolean that { return and[this][that]; }

    public static final boolean and[boolean][boolean] =
        { { false, false }, { false, true } };
    ….
```

User-Defined bit Type

```
enum bit {
   zero, one;

  public boolean operator < bit b { return lt[this,b]; }
  public bit operator + bit b { return xor[this,b]; }
  public bit operator sum(bit b, bit c) { return sum[this,b,c]; }
  public bit operator carry(bit b, bit c) { return cry[this,b,c]; }

  static final boolean lt[bit,bit] = { { false, true }, { false, false } };
  static final bit xor[bit,bit]     = { { zero, one }, { one, zero } };

  static final bit sum[bit,bit,bit] = { { { zero, one }, { one, zero } },
                                        { { one, zero }, { zero, one } } };
```

For-Enumerators

- Ability to iterate over values of an enum type
  - C idiom relies on value outside of range
- Unary + operator
  - Advances enum value and returns true
  - Or returns false if last value already reached

for (*init*; *enumerate*) *body* --> *init;* do { *body* } while (*enumerate*)

```
for (day d = day.first; + d)
    total += happiness[day];
int lifestyle = total/day.number;
```

Values

- Generalized read-only objects
- Do not allow:
  - synchronization
  - address equality
  - modification outside of constructor
- Efficient
  - small values can be shared by copy (registers)
  - large values can be shared by reference
- Enumerations are a special kind of value

User-Defined ubyte

```
enum byteindex { x0, x1, x2, x3, x4, x5, x6, x7 };

final value ubyte {
    private bit data[byteindex];

    public boolean this < ubyte that {
        for (byteindex x = byteindex.last; -x)
            if (data[x] != that.data[x])
                return data[x] < that.data[x];
        return false;
    }
```

User-Defined ubyte (cont)

```
public ubyte ~ this {
    return new ubyte.complement(this);
}

private ubyte.complement(ubyte b) {
    for (byteindex x; +x)
        data[x] = ~ b.data[x];
}
```

User-Defined **ubyte** (cont)

```
public constructor this + ubyte that {
    return new ubyte.sum(this, that);
}

private ubyte.sum(ubyte a, ubyte b) {
    bit c;
    for (byteindex x; +x) {
        data[x] = c.sum(a.data[x], b.data[x], c);
        c = c.carry(a.data[x], b.data[x]);
    }
}
```

References and Pointers

- Pointers may be bound or unbound
  - Foo* x = null; x = new Foo();
- References are always bound (never null)
  - Foo y = new Foo();
- References are initialized by default
  - Foo z; ⟷ Foo z = new Foo();
  - int x; ⟷ int x = new int(); ⟷ int x = 0;

Class point

```
final value point {
    public ubyte x;
    public ubyte y;
    public point() { }
    public point(ubyte x, ubyte y) { this.x = x; this.y = y; }
    public point add(point that) {
        return new point(x+that.x, y+that.y);
    }
    …
}
```

Examples of Kava Value Types

- complex
- point
- ipaddress
- decimal
- interval
- date
- time
- timestamp

Advantages of Kava

- Eliminates primitive/object distinction
- Semantics of primitives definable inside language
- New primitives easily definable (loose numerics)
  - Efficiency requires machine instruction inlining
- New “core” data types can be defined
  - Aggregation of “primitives” in small values
- Abstraction extends to bit-field manipulation
- Simplifies VM instruction set

Systemic Effects

- Should reduce number of objects allocated
  - Many common things will become scalar values
  - Strings will only require one object
- Concurrency issue:
  - “Large” values must be atomically updated
  - Synergy with Guava
    - » Not a problem if language is race-free

Status and Future Plans

- Implementation underway
  - Kava-to-bytecode compiler (using Jikes)
  - Semantic expansion in JIT compiler (Jalapeno)
- Implement novel types
  - DSP, NPU, etc.
  - Compile to those processors
  - Compile to Verilog or VHDL