CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 40: Bounded polymorphism and other Java extensions

30 Nov 07

Whither language research?

Language support for:

- building correct systems
- building secure systems
- building large, maintainable systems
- future architectural evolution
  - Distributed computation
  - Multicore/multiprocessor systems

(first three items:) Knowing intent helps

- Performance no longer a driving concern!

2

Some language projects

- Jif: adding security policies (confidentiality and integrity) to the Java type system
- PolyJ: parametric polymorphism in Java
- J&: nested inheritance and intersection for package-level extensibility
- JMatch: abstraction-preserving pattern matching and iteration

3

Jif - Java information flow

- Annotate (Java) programs with labels
- Variables have type + label

  int {L} x;

- Label L can contain policies

```
int {Alice→Bob} x;
   // Alice thinks Bob should be allowed to learn x
int {Alice←Bob} x;
     // Alice thinks Bob should be allowed to affect x
```

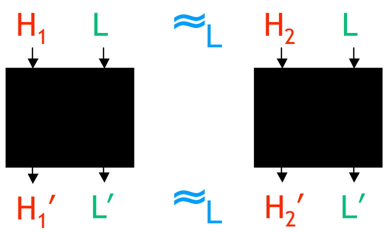
- Allow x=y if $L_y \sqsubseteq L_x$, x+y has label $L_x \sqcup L_y$
- Parametric polymorphism and dependent types used to provide genericity, power to create principals and policies at run time.

4

Noninterference

"Low-security behavior of the program is not affected by any high-security data."

Goguen & Meseguer 1982



Confidentiality: high = confidential, low = public

Integrity: low = trusted, high = untrusted

5

Proving noninterference

- Define $s \approx_L s'$ if s and s' are same on all components $\sqsubseteq L$
- Nonintefeference: $s \approx_L s' \Rightarrow [\![s]\!] \approx_L [\![s']\!]$
- Prove diagram holds in operational semantics

6

Bounded type parameters

```
class HashMap[K,V] implements Map[K,V] {
   bool add(K key, V value) { int i = key.hashCode(); ... }
}
```

- Hash table code must be able to compute hash value for values of type K: can't apply HashMap to every type!
- Idea: *constrain* parameter type K to ensure it has the necessary operation: constrained parametric polymorphism
- Java 1.5 has supertype bounds. E.g., key type *K* okay if subtype of

```
interface Hashable { int hashCode(); }

class HashMap[K extends Hashable, V] { ... }
```

7

Type parameter bounds

```
class HashMap[K extends Hashable, V] { ... }
```

ObjectT(HashMap) = $\lambda K \leq$Hashable::type.λV::type.μS.{add: $K * V \rightarrow$ bool,...}

- $F_{\leq}$ type context:
  $\Delta = \alpha_1 \leq \tau_1$::type,...,$\alpha_n \leq \tau_n$::type
- General idea: typing contexts can carry constraints on types (and on values!)
- Can extend to higher kinds ($F_{\omega\leq}$) but subtyping rules are tricky

8

F-bounded polymorphism

- F-bounded polymorphism: τ can mention α in type constraint $\alpha \leq \tau$

```
class HashMap[K extends Comparable[K], V]

interface Comparable[K] {
   int compare(K k);
}
```

- Payoff: more precise bounds, don't have to write comparison against Object.

9

Parameterized classes

```
class HashMap[K≤Comparable[K],V] implements Map[K,V]{
   static Hashmap() {…}
   bool add(K key, V value) { int i = key.hashCode(); … }
}
```

- Defines parameterized type ObjectT(HashMap): type of objects
- What is value of *class* object?
  $\Lambda K \leq \text{Comparable}[K]::\text{type}.\Lambda V::\text{type}.\{\ldots\text{static methods}\ldots\}$
  $:\forall K \leq \text{Comparable}[K]::\text{type}.\forall V::\text{type}.\{\ldots\text{static methods}\ldots\}$

$\tau ::= X \mid B \mid \tau_1 \rightarrow \tau_2 \mid \tau_1\, \tau_2 \mid \lambda X \leq \tau'::K.\tau \mid \forall X \leq \tau'::K.\tau$

$e ::= x \mid \lambda x{:}\tau.e \mid e_1\; e_2 \mid \Lambda X \leq \tau'::K.e \mid e\,[\tau]$

10

PolyJ

Parametric polymorphism with *structural* bounds on *class* (http://www.cs.cornell.edu/polyj)

- Can constrain on static methods, constructors, …

```
class HashMap[K,V] implements Map[K,V]
    where K { int compare(K);   K();   }
```

  - Can write new T()
  - Can instantiate on int & friends
- Structural: parameters don't have to declare implementation of bounding interface
  - little value to separate abstraction for bound!
- Implemented, not chosen for Java standard…

11

A useful direction?

- Type instantiation is a binary operation C(T)
- Programmer may control neither C nor T
  - Not reasonable to expect T to implement a bounding interface
  - Not reasonable to expect T even to implement the right methods
- Should be able to define binding at instantiation

  `HashMap[String with compare=lcCompare, Object]`

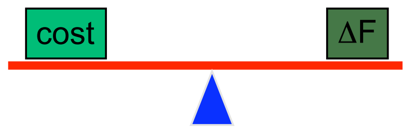
  - Can view as dependent type with optional arguments:

  HashMap = λk::type, v::type, compare: t*t->int. {…}

12

Scalable extensibility

- Principle: To extend a software system should require writing code proportional to change in functionality.



- Current languages lack this property!
- Our approach: a new package-level inheritance mechanism: *nested inheritance*

13

Example: a scalable compiler

Changes to the compiler should be proportional to changes in the language.

- Most compiler passes are **sparse**
- Can't exploit this

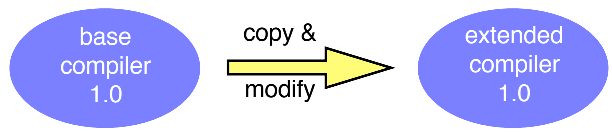
Types

Operations

	+	**if**	x	*e*.f	=
name resolution					
type checking					
exception checking					
constant folding					

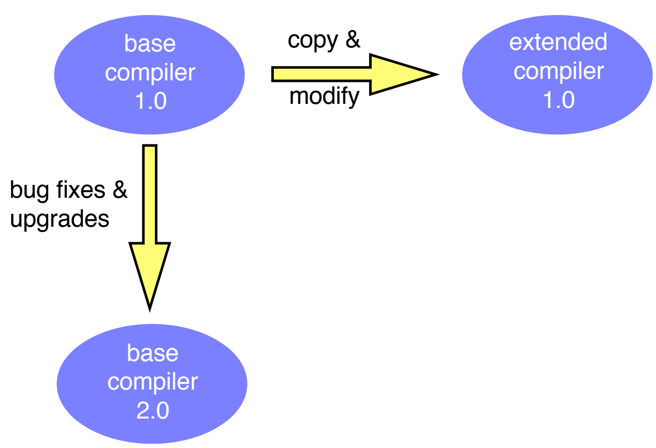
14

In place modification



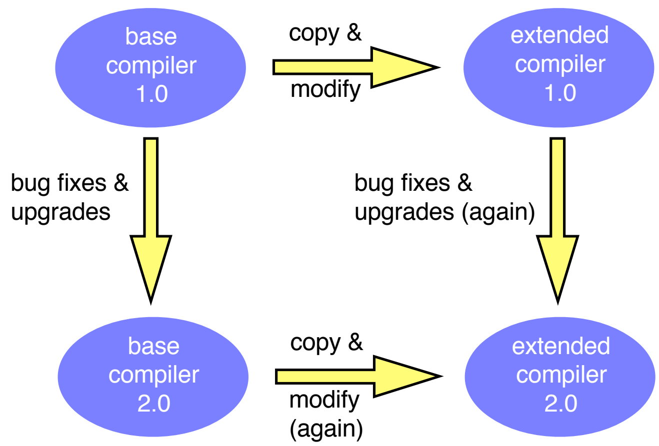
15

In place modification

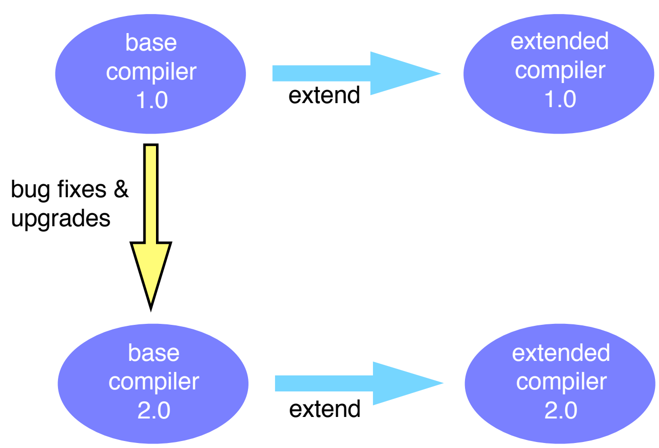


16

In place modification

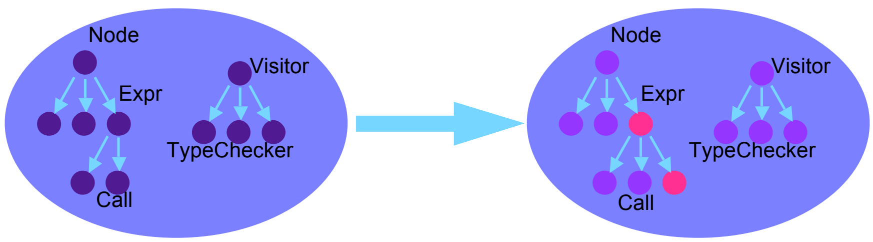


Idea: inherit the compiler



Inheritance

Write code only for the new functionality.
Inherit the rest: *modular*



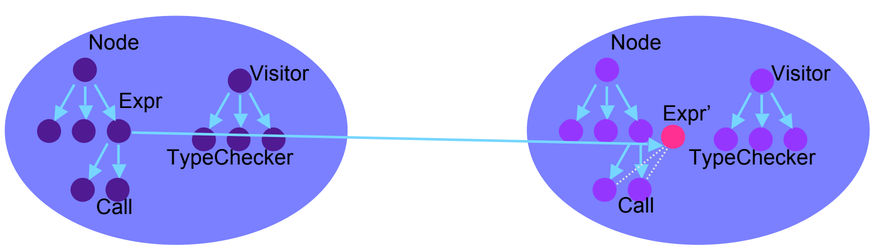
base compiler

extended compiler

19

Limitations of inheritance

Inheritance works on individual classes



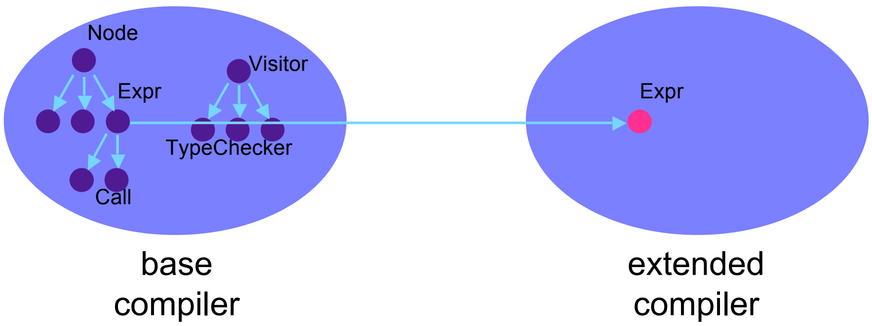
base compiler

extended compiler

20

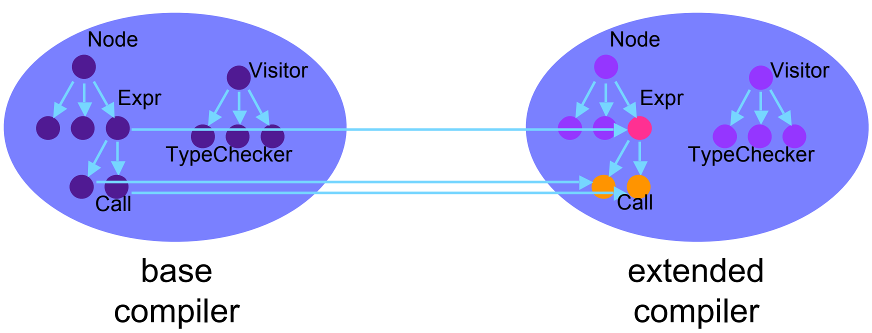
Nested inheritance

Idea: Write code just describing changes to package



Extensions are inherited

- Nested subclasses inherit changes
- Relationships among classes preserved
- Key: statically type-safe



Example: UI toolkit

- Language: J& (pronounced "jet")
  - Java + nested inheritance and intersection
- Nested classes are static
  - works for packages too

```
class UI {
  class Window { Point position; ... }
  class Button extends Window { ... }
  void draw(Button b) { ... }
  Button clickMe() {
    return new Button("Click me");
  }
}
```

```
class FancyUI extends UI {
  class Window {
    int border;
  }
  void draw(Button b) {
    ... b.border ...
  }
}
```


Nested class inheritance

Methods, fields, and nested classes are inherited

```
class UI {
  class Window { Point position; ... }
  class Button extends Window { ... }
  void draw(Button b) { ... }
  Button clickMe() {
    return new Button("Click me");
  }
}
```

```
class FancyUI extends
  class
    Point position;
    int border;
  }
  void draw(Button b) {
    ... b.border ...
  }
  Button clickMe() {
    return new Button("Click me");
  }
  class Button extends Window { ... }
```

Class overriding

Nested classes can also be overridden

```
class UI {
  class Window { Point position; ... }
  class Button extends Window { ... }
  void draw(Button b) { ... }
  Button clickMe() {
    return new Button("Click me");
  }
}
```

```
class FancyUI extends
  class
    Point position;
    int border;
  }
  void draw(Button b) {
    ... b.border ...
  }
  Button clickMe() {
    return new Button("Click me");
  }
  class Button extends Window { ... }
```

25

Type name interpretation

Type names reinterpreted in inheriting context

Button here is UI.Button

Button here is FancyUI.Button

```
class UI {
  class Window { Point position; ... }
  class Button extends Window { ... }
  void draw(Button b) { ... }
  Button clickMe() {
    return new Button("Click me");
  }
}
```

```
class FancyUI extends
  class
    Point position;
    int border;
  }
  void draw(Button b) {
    ... b.border ...
  }
  Button clickMe() {
    return new Button("Click me");
  }
  class Button extends Window { ... }
```

26

Dependent classes

Key to soundness:

Button = UI[this].button
(dependent type!)

UI u = **new** FancyUI();
UI.Button b = **new** UI.Button();
u.draw(b); // illegal, unsafe call accessing b.border

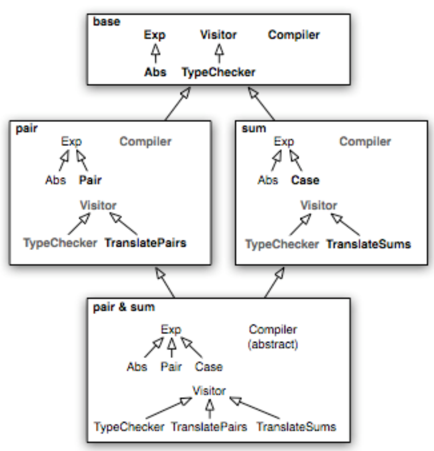
final UI u = **new** FancyUI();
u.Button b = **new** u.Button();
u.draw(b); // this call is OK

27

Nested intersection in J&

```
package pair extends base
package sum extends base
package pair_sum extends
   pair & sum
```

- Intersect packages, classes to obtain union of functionality
- Intersection is recursive in hierarchy
- Conflicts create abstract members to resolve



28

Intersection results

- Ported Polyglot compiler framework from Java to J&
  - Got rid of design patterns, factories needed for extensibility in Java: 32→28kLOC

Compiler extension	LOC	+J_o	+carray	+covret	+autoboxing
Coffer (type state)	2642	63	34	66	86
J_o (pedagogy)	436		34	37	46
constant arrays	122			31	34
covariant returns	214				53
autoboxing	347				

- Similar results with Pastry P2P framework

29

Summary

Current work:

- Type-level adaptation of new functionality
- Implementation of class sharing in J& and application to real software (Polyglot,…)

Future work:

- Dynamic software upgrading
- Schema evolution for families of persistent objects

30

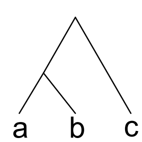
Pattern matching for OO

- Goal: deep "ML-like" pattern matching for an imperative, object-oriented language
- JMatch : Java + (iterable, abstract) pattern matching
- Small set of features gives:
  - Pattern matching
  - Typecase
  - Views
  - First-class patterns
  - Support for iteration abstractions (ala CLU, Sather, ICON)

31

Limitations of ML patterns

```
datatype tree = Leaf | Node of tree*tree

case t of
  Node(Node(a,b), c) => Node(b,c)
| _ => t
```



- Data structure unification compiled into efficient code
  - But: no use outside module w/o violating data abstraction
- Not generic! Can't use for:
  datatype BST = Leaf | Node of tree*tree*value
  datatype RBTree = Leaf | Node of tree*tree*value*color
  type ATree = value array

32

Modal abstractions

- *Modes:* different directions of evaluation. ML:

  datatype intlist = Nil | Cons of int * intlist

  Cons_F : int*intlist→intlist, Cons_B: intlist→int*intlist

  Equational relationship:

  $\text{Cons}_B(\text{Cons}_F(x,y)) = (x,y)$

  $\text{Cons}_F(\text{Cons}_B(z)) = z$

  is a relation: Cons $\subseteq$ int*intlist*intlist

- Logic programming ideas adapted from Prolog to JMatch:
  - Pattern matching via *user-defined modal abstractions*
  - Implementation of relation as boolean formulas

33

JMatch list matching

Modal abstraction declaration:

```
List cons(int head, List tail)
        returns(result)        /* forward mode */
returns(head, tail)    /* backward mode */
```

Mode unknowns

- Two operations in one!

(Separate) implementation of both modes:

```
class List {
    int head; List tail;
    static List cons(int h, List t) {
        List ret = new List(); ret.head = h;
        ret.tail = t; return ret;
    } returns(h, t) {
        h = head; t = tail; return;
    }
}
```

34

Invertible computation

- Modal abstraction implementable using a boolean formula for its relation
- Compiler automatically generates code for different modes

```
class List {
    int head; List tail;
    List(int h, List t) returns(h, t) (
        head = t && tail = t
    )
    static List cons(int h, List t) returns(h, t) (
        result = List(h, t)
    )
}
```

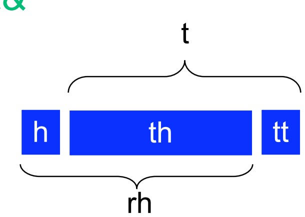
```
Forward:
head = h;
tail = t;

Backward:
h = head;
t = tail;
```

```
Forward:
result = new List_F(h, t);

Backward:
(h, t) = List_R(result);
```

Snoc lists

```
class List {
    List head;
    int tail;
    List(List h, int t) returns(h, t) ( head = h && tail = t )
    static List cons(int h, List t) returns(h, t) (
        t = List(List th, int tt) &&
        List rh = cons(h, th) &&
        result = List(rh, tt)
    )
)
```

Local unknowns

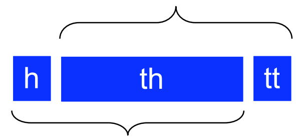


36

Evaluation

- JMatch may reorder evaluation of conjuncts

```
static List cons(int h, List t) returns(h, t) (
        t = List(List th, int tt) &&
        List rh = cons(h, th) &&
        result = List(rh, tt)
)
```



- Simple rule for reasoning about side effects: always choose leftmost solvable conjunct
- Unification with local propagation
  - no unknowns inside values, unlike most logic programming languages
- Disjunctions always evaluated left-to-right

37

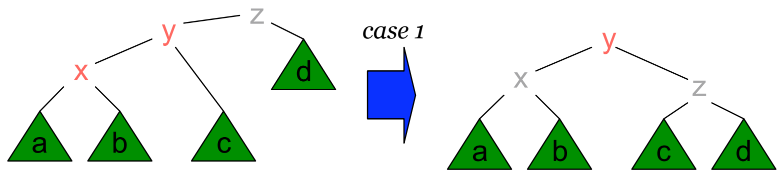
Pattern-matching statements

```
switch (x) {
   case List.cons(List.cons( _, int y1), int y2) :
        …y1 … y2 …
}

if (x = List.cons(List.cons( _, int y1), int y2)) {
   … y1 …y2 …
} else …

let List.cons(List.cons( _, int y1), int y2) = x;
… y1 … y2 …
```

38

Example: Red-black trees

```
static RBNode balance(int color, int value, RBTree left, RBTree right) {
  if (color == BLACK) {
    switch (value,left,right) {
      case int z, RBNode(RED,int y,
                      RBNode(RED,int x,RBTree a,RBTree b),RBTree c), RBTree
d:
      case z, RBNode(RED,x,a,RBNode(RED,y,b,c)),d:
      case x, c, RBNode(RED,z,RBNode(RED,y,a,b),d):
      case x, a, RBNode(RED,y,b,RBNode(RED,z,c,d)):
        return RBNode(RED,y,RBNode(BLACK,x,a,b),RBNode(BLACK,z,c,d));
    }
  }
  return RBNode(color, value, left, right);
}
```



Iterable modes

- Mode is *iterable* if underlying relation is many-to-one

```
interface Collection {
  boolean contains(Object x) iterates(x);
}
```

- Forward mode (default):
  tests for membership of particular x
- Backward mode: finds all x satisfying contains(x)
- while iterates over formula solutions
  `while (c.contains(Object x)) { … }`
- Type checker checks *multiplicity* to ensure multiple solutions not accidentally discarded (1 ≤ *)

Implementing iterators

```
class RBNode implements IntCollection, Tree {
  RBTree left, right; int value; boolean color;
  boolean contains(int x) iterates(x) (
    x < value && left.contains(value) ||
    x = value ||
    x > value && right.contains(value)
  )
}
```

- Disjunction || creates iteration
- Forward mode: efficient binary search
- Backward mode: in-order tree traversal

41

Implementing iterators in Java

- The backward mode:

```
class TreeIterator  implements Iterator {
  Iterator subiterator;
  boolean hasNext;
  Object current;
  int state;
  // states:
  //   1. Iterating through left child.
  //   2. Just yielded current node value
  //   3. Iterating through right child

  TreeIterator() {
    subiterator = RBTree.this.left.iterator();
    state = 1;
    preloadNext();
  }

  public boolean hasNext() {
    return hasNext;
  }

  public Object next() {
    if (!hasNext) throw new NoSuchElementException();
    Object ret = current;
    preloadNext();
    return ret;
  }
```

```
private void preloadNext() {
  loop: while (true) {
    switch (state) {
      case 1:
      case 3:
        hasNext = true;
        if (subiterator.hasNext()) {
          current = subiterator.next();
          return;
        } else {
          if (state == 1) {
            state = 2;
            current = RBTree.this.value;
            return;
          } else {
            hasNext = false;
            return;
          }
      case 2:
        subiterator = RBTree.right.iterator();
        state = 3;
        continue loop;
  }
```

42

Related work

- Views [Wadler87]
- First-class patterns [Fahndrich & Boyland 97], [Tullsen 00]
- Convenient iteration
  - CLU, Sather, ICON, Alma-0 : no pattern matching
- Modes
  - Mercury, μProlog
- Pattern matching for imperative languages
  - Pizza [Odersky and Wadler 97], predicate dispatch [Ernst98]
- Future work: exhaustiveness checking on patterns

43

JMatch summary

- A few mechanisms:
  - declaration of modal abstractions
  - automatic implementation using formulas
  - iterable patterns and formulas
- A lot of useful features:
  - deep pattern matching, typecase
  - views, first-class patterns
  - easy implementation of iteration abstractions
  - multiple return values
  - modal abstractions ⇒ simpler ADT specifications
- Implementation:
  www.cs.cornell.edu/projects/jmatch

  Using Polyglot extensible Java compiler framework [CC'03]
  (www.cs.cornell.edu/projects/polyglot)

44