System F Type Systems: From Theory to Practice

Theory
- Background on STLC
- Why STLC isn't enough?
- Why System F isn't enough?

Practice
- How to leverage these ideas in mainstream PLs?
- How do pragmatic concerns affect design decisions?

Simply Typed Lambda Calculus

$$
\begin{aligned}
e &::= x \\
&\mid \lambda\{x:t\}.\ e \\
&\mid e \quad e \\
&\mid n. \\
&\mid \mathrm{add1}\ e \\
t &::= \mathrm{int} \\
&\mid t \to t.
\end{aligned}
$$

- $(\lambda\{x:\mathrm{int}\}.\ \mathrm{add1}\ x)\ 42$ ✓.

$(\lambda\{x:\mathrm{int}\}.\ \mathrm{add1}\ x)\ (\lambda\{x:\mathrm{int}\}.\ x)$ ✗

Limitation of STLC

- An Identity function.

$(\lambda\{x: int\}.\ x)\ 74$ ✓

$(\lambda\{x: int \to int\}.\ x)\ (\lambda\{x: int\}.\ x)$ ✓

$(\lambda\{x: int \to int\}.\ x)\ 74$ ✗

$\Rightarrow$ (remove types)

$(\lambda x.\ x)\ 74$

$(\lambda x.\ x)\ (\lambda x.\ x)$

$(\lambda x.\ x)\ 74.$

---

① Too Conservative

② Code Duplication.

- sort
- map
- .....

System F (Reynolds 74) (Girard 72).

$(\lambda\{x : int \quad\}.\ x)$

$(\lambda\{x : int \to int\}.\ x)$.

Insight: What's varying among the almost duplicated terms?

- λ abstracts over varying <u>terms</u>
  - term → [λ] → term.
- Λ abstracts over varying <u>types</u>
  - type → [Λ] → term.

Big Lambda Λ

$$e ::= x \mid \lambda\{x:t\}.e \mid e\ e \mid n \mid add\ e$$
$$\mid \Lambda t.e$$
$$\mid e[t]$$

$$t ::= int \mid t \to t$$
$$\mid \forall T.t$$
$$\mid T$$

$$(\Lambda T.\lambda\{x:T\}.x)[int]$$
$$\Rightarrow \lambda\{x:int\}.x$$

Just like substitution in λ app

```
let id = ΛT. λ{x:T}. x.
in (id [int → int])((id [int]) 74)
```

User Defined Types

$$
\begin{array}{l}
(\Lambda\, CN. \\
\quad \lambda\{add: CN \to CN \to CN\}. \\
\quad \lambda\{mag: CN \to real\}. \\
\quad \lambda\{i : CN\}. \\
\quad (mag\,(add\ i\ i)))
\end{array}
$$

$(real \times real)$

$(\lambda\{x: real \times real\}. \ldots\ldots)$

mag_rep

i_rep

- Can benefit from syntactic sugaring
- Not just type synonym (type is abstract)

Representation Theorem

Main idea:

> Representations of primitive types shouldn't affect program behavior.

For compiler writers:

- runtime representation of integer, etc.
- type Direction = North
  | South
  | East
  | West.

Adding Subtyping to System F

Why Subtyping?

$(\lambda\{x:\{count:int\}\}.\ x.count)\ \{count:5, extra:85\}$

id function

$(\lambda\{x:\{count:int\}\}.\ x)\ \{count:5, extra:85\}$

$\{count:int\} \to \{count:int\}$ ✗

$(\Lambda T.\ \lambda\{x:T\}.\ x)\ [\{count:int, extra:int\}]$ ✓ (...).

- Implicit Subtyping "forgets" actual type.
- $(\Lambda T. \ldots)$ can't "see through" T.

Problem arises if we want both

$(\lambda\{x:\{count:int\}\}.\ (x.count, x))$

$(\Lambda T.\ \lambda\{x:T\}.\ (??, x))$.

Bounded Quantification Cardelli & Wegner 85

idea: Explicit Subtyping in $(\Lambda T. \cdots)$

$e ::= \cdots$
$\mid \Lambda T \leq t. e$

$t ::= \cdots$
$\mid \forall T \leq t. t$

Ex

$(\Lambda\ T \leq \{count: int\}.\ \lambda\{x:T\}.\ (x.count, x))$
$[\{count: int, extra: int\}]$
$\{count: 74, extra: 85\}$

$\Rightarrow (\lambda\{x: \{count: int, extra: int\}\}.\ (x.count, x))$
$\{count: 74, extra: 85\}$

$\Rightarrow \ldots$

Bounded Quantification Cont.

- "See through" Quantified type $(\Lambda X \leq \ldots)$
- Express input/output dependencies.
- The Cardelli & Wegner 85 paper covers existential type as well.
- Easy type checking; inference... probably not

Limitation of Bounded Quantification

Canning et al. 89

- Goal: represent objects with recursive types + record types.

$$\text{Point} = \text{Rec pnt}.\{x: \text{Real}, y: \text{Real}, \text{move}: \text{Real} \times \text{Real} \to \text{pnt}, \text{lesseq}: \text{pnt} \to \text{bool}\}$$

$$\frac{\Gamma, s \le t \vdash T_1 \le T_2}{\Gamma \vdash \text{Rec } s.T_1 \le \text{Rec } t.T_2} \qquad \frac{\Gamma \vdash A_2 \le A_1 \quad \Gamma \vdash B_1 \le B_2}{\Gamma \vdash A_1 \to B_1 \le A_2 \to B_2}$$

- $\text{Moveable} = \text{Rec mv}.\{\text{move}: \text{Real} \times \text{Real} \to \text{mv}\}$

  $\text{Point} \le \text{Moveable}$ ✓

  - $\wedge T \le \text{Moveable}.\ \lambda\{x:T\}.\ x.\text{move}(1,1)$

    $\forall T \le \text{Moveable}.\ T \to \text{Moveable}$ (want T)

- $\text{Comparable} = \text{Rec cp}.\{\text{lesseq}: \text{cp} \to \text{bool}\}$

  $\text{Point} \le \text{Comparable}$ requires $\text{cp} \le \text{pnt}$ ✗

Limitations of Bounded Quantification Cont

- Where's the problem?
  - Moveable = Rec mv. {move: Real×Real → mv}
  - Comparable = Rec cp. {lesseq: cp → bool}
  - Could be any Moveable/Comparable.
    ⇒ needs more specific types.
- Attempt 2: (leave it to the users)
  - Moveable = ∀ mv. {move: Real×Real → mv}
  - Comparable = ∀ cp. {lesseq: cp → bool}

  Point ≤ Moveable [Point] ✓
  Point ≤ Comparable [Point] ✓

  Λ[T ≤ Moveable [T]]. λ{x: T} x.move (1, 1)

  ✗

F-bounded Quantification Canning et al. 89

$$\forall t \leq F[t]. \sigma$$

Allows t in the bound.

$$\Lambda\, T \leq \text{Comparable}[T]. \lambda\{x:T\}. \lambda\{y:T\}.$$
$$\text{if } x.\text{lesseq } y \text{ then } x \text{ else } y$$
$$: \forall\, T \leq \text{Comparable}[T]. T \to T \to T.$$

Speculations for semantics development.

- semantics-by-translation (Breazu-Tannen et al.)

$$\forall s \leq t. \sigma$$
$$\Rightarrow \forall s. (s \to t) \to \sigma$$

scope.

- more in Canning et al. 89.

Into the practical world

$$Comparable = \Lambda T.\{ lesseq : T \to bool \}$$

```
interface Comparable<T> {
    int compare(T other);
}
```

Java Generics

But it wasn't always like this

One path of attempts:

- Pizza (Odersky & Wadler 97)
- GJ (Odersky et al. 98).

Java pre-generics history

```
interface Comparable {
    int compare(Object other);
}

class Num implements Comparable {
    int val;

    int compare(Object other) {
        Num otherNum = (Num) other;
        ... this.val ... otherNum.val ...
    }
}
```

- Programmers must be careful...

Extending Java with generics Odersky & Wadler 97

- new syntax
- translate to old Java.

```
interface Comparable<T> {
    int compare(T other);
}

class Num implements Comparable<Num>{
    int val;
    int compare(Num otherNum){
        ... this.val ... otherNum.val ...
    }}
```

Homogeneous Translation

```
interface Comparable {
  int compare$(Object o);}

class Num implements Comparable{
  int val;
  int compare(Num...){...}
  int compare$(Object o){
    return this.compare(
        (Num) o);
  }}
```

Heterogeneous Translation

```
interface Comparable_Num {
  int compare(Num o); }

class Num imp. Comparable_Num{
  int val;
  int compare(Num o){
    ...
  }}
```

More Translation Examples

```
<T implements Comparable<T>> T min(T a, T b){
    if (a.compare(b) <= 0) return a;
    else return b;}.

Num x = min(new Num(89), new Num(97));
```

Homogeneous Translation

```
Comparable min(Comparable a, Comparable b){
    ... compare$ ... }.

Num x = (Num) min(new Num(89), new Num(97));
```

Heterogeneous Translation

```
Num min_Num(Num a, Num b){...}

Num x = min(new Num(89), new Num(97));
```

Homogeneous Translation and Java Array

```
<T> T[] copy(T[] arr) {
    T[] cpy = new T[arr.length];
    for(...) {...}
    return cpy; }
```

⇒

```
Object[] copy(Object[] arr) {
    Object[] cpy = new Object[arr.length];
    ... }
```

- ISSUE ①: `int[]` ⊭ `Object[]`

Solution:

```
abstract class Array{
  int length();
  Object get(int i);
  void set(int i, Object o);}
```

Array_Obj Array_int ...

Homogeneous Translation & Java Array Cont.

```
Array copy(Array arr) {
  Array cpy = new Array_obj(arr.length());
  ... }
```

- Concrete Array subclasses only at creation
- Casting from 'int' to 'Integer' needed in Array_int::get
- Relies on array being covariant for reference types, i.e., 'String[] ≤ Object[]'

Generics & Subtyping

- $(X \leq Y) \overset{?}{\Rightarrow} (Foo\langle X \rangle \leq Foo\langle Y \rangle)$

```
class Cell<T>{
    T x;
    Cell(T x){ this.x = x;}
    T get() { return this.x;}
    void set(T x){ this.x = x;}
}

Cell<String> str = new Cell("9");
Cell<Object> obj = str;
obj.set(new Integer(89));
String s = str.get();
```

- Generics must be invariant.

Generics & Casting

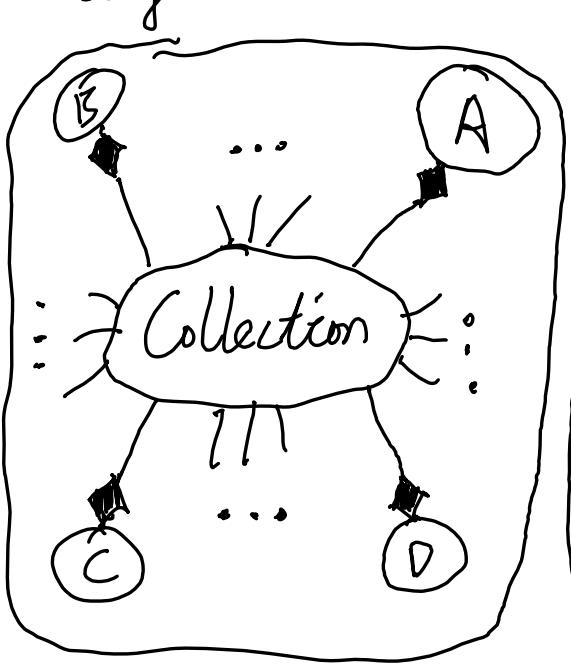
```
class Cell<T> {
    ...
    boolean equals(Object o) {
        if (o instanceof Cell) {
            Cell<X> other = (Cell) o;
            return this.x.equals(other.x);
        }
    ....}
}
```

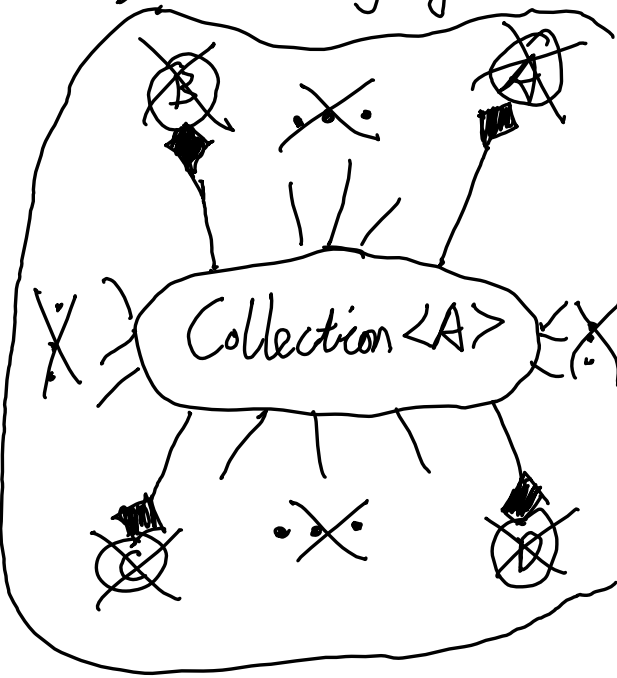
- $\Gamma \vdash (\text{Cell})\, o : \exists x.\, \text{Cell}\langle x \rangle$
- Generics have no RTTI after homogeneous translation.

Pizza seems to be self-consistent in terms of language features... BUT

The generic legacy Problem Odersky et al. 98

- Libraries are main beneficiaries of generics
- Legacy code uses libraries without generics
- Easy to rewrite libraries, not legacy code.





Odersky et al. 98 proposes GJ to

- Add generics to Java
- Address legacy code compatibility issues

Raw Types in GJ

- Insight:
  - Legacy code already uses dangerous "pre-generics" idioms (Object + manual casting).
  - Allow this idiom with generics, but generate warnings to prevent new usage.

Warning Generation

```
Cell<String> str = new Cell<String>("98");
Cell raw = str;
Cell<Integer> i = raw;
raw.set(new Integer(98));
```

Generates unchecked warning on:

- method call to raw type if erasure changes parameter type.
- assignment to raw type if erasure changes lhs type

Raw Type and casting

```
class Cell<T> {
    ......
    boolean equals(Object o) {
        if (o instanceof Cell) {
            Cell other = (Cell) o;
            return this.x.equals(other.x);
    ......}.
```

- Naturally replaces existential type in Pizza.
- Cell behaves like a restricted version of Cell<Object>.

Retrofitting in GJ

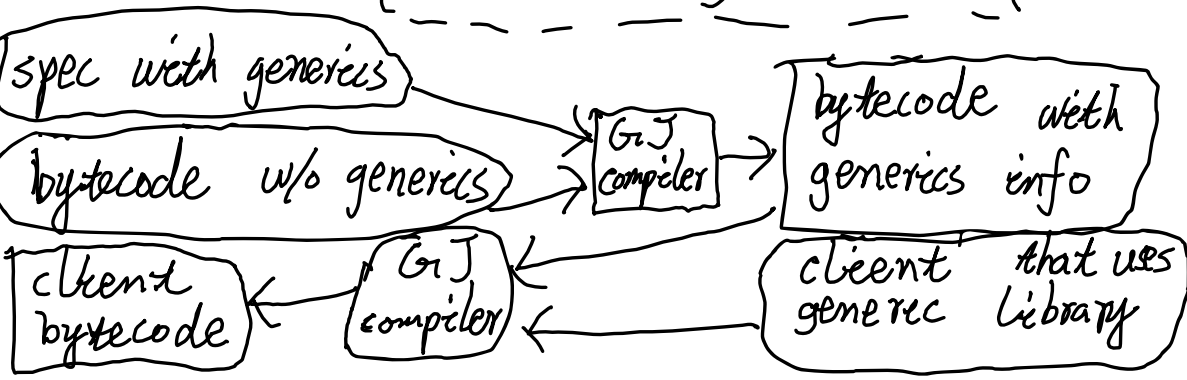
- Still needs to rewrite old libraries with generics. Collection → Collection<T>
- Generics translates to old idiom anyway
- Trust old libraries, leave them alone.

How to tell GJ compiler?

- "retrofitting mode"

retrofitting spec file.

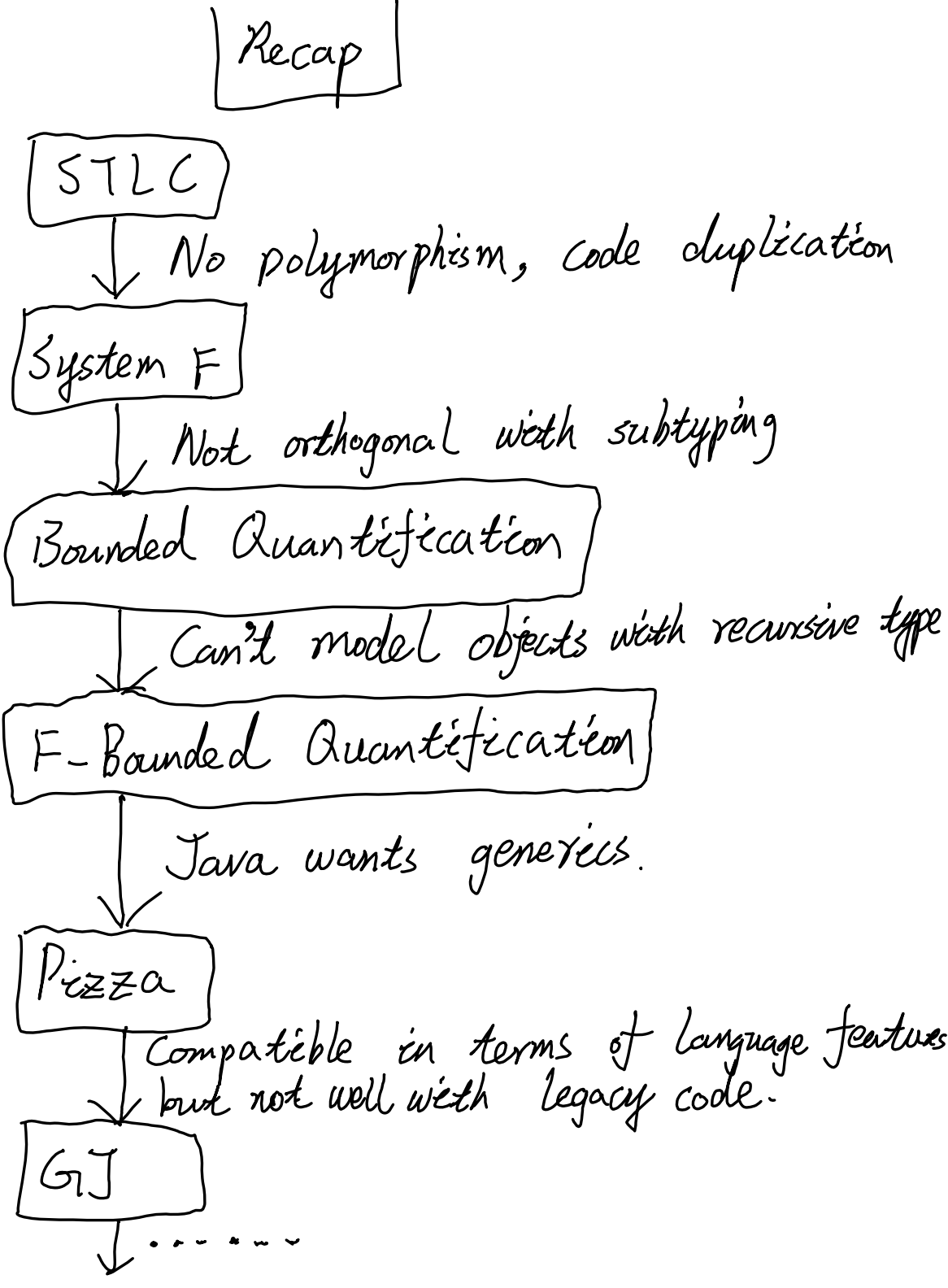
```
class ArrayList<T> {
    void add(T x);
    T    get(int i);
    ......
}
```

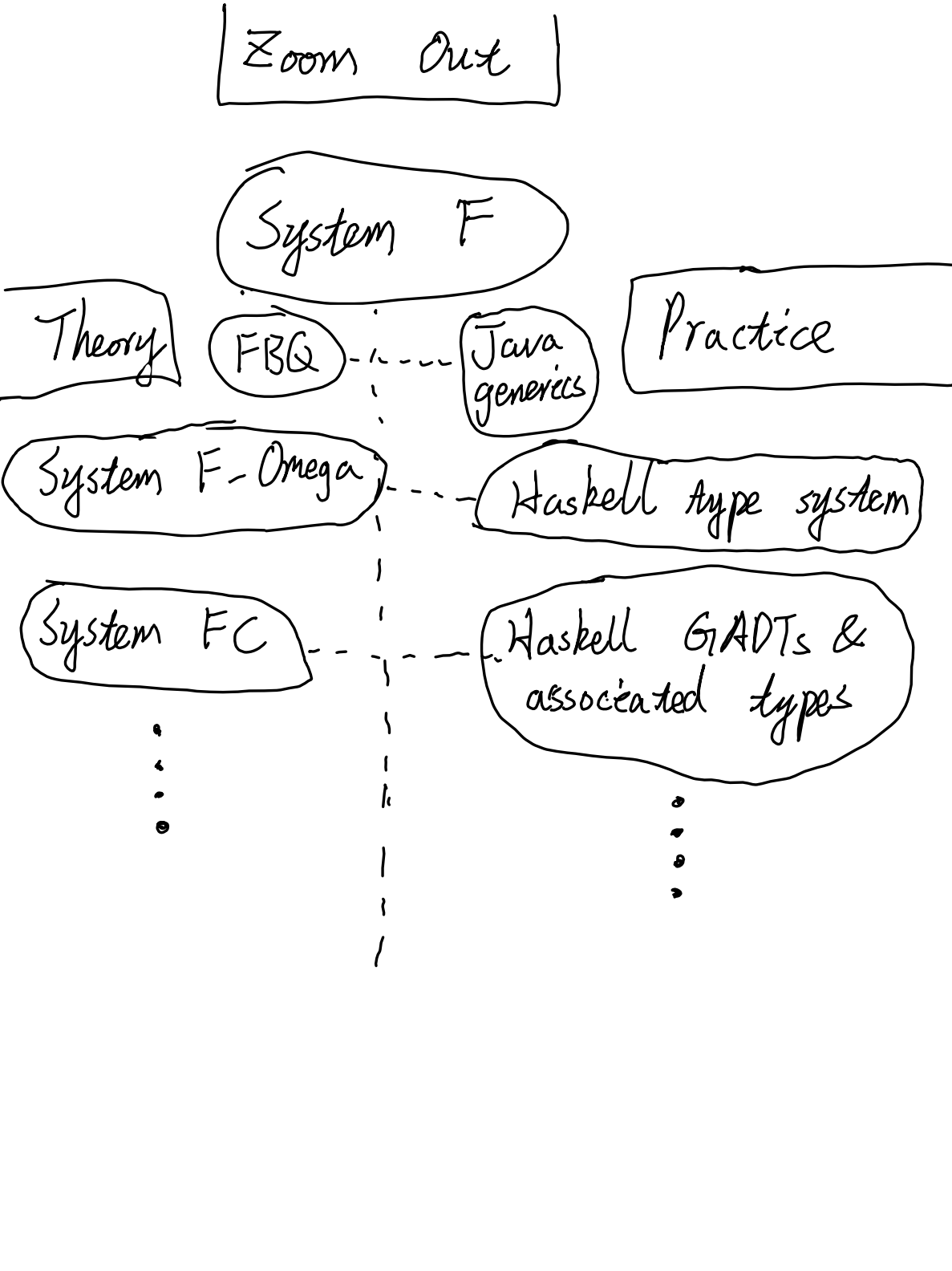


More on GJ

Library	Client	How.
Collection	Collection<T>	retrofitting
Collection<T>	Collection	raw type
Collection	Collection	programmers' caution
Collection<T>	Collection<T>	compiler.

- "int[] ⊄ Object[]" issue not mentioned.
  - Array wrapper like Pizza.
  - forbids using primitives for generic parameters.
- Runtime security loophole due to type erasure
  - 'IntList extends List<Integer>' enforces type.
- Exception to invariant generics
  - '<A> List<A> empty() {...}'
  - '<A> Cell<A> make(A x){...}' and 'null'
  - bottom type '*': List<*> ≤ List<T>.


Recap
STLC
No polymorphism, code duplication
System F
Not orthogonal with subtyping
Bounded Quantification
Can't model objects with recursive type
F-Bounded Quantification
Java wants generics.
Pizza
Compatible in terms of language features
but not well with legacy code.
GJ
.....


Zoom Out
System F
Theory
FBQ
Java generics
Practice
System F-Omega
Haskell type system
System FC
Haskell GADTs & associated types