System F Type Systems: From Theory to Practice

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

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

Simply Typed Lambda Calculus

terms

$$e ::= x \mid \lambda\{x:t\}.\ e \mid e\ \ e \mid n. \mid \text{add1}\ \ e$$

types

$$t ::= \text{int} \mid t \to t.$$

Ex

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

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

Limitation of STLC

- An Identity function.

$(\lambda\{x: int\}.\ x)\ 74$ ✓.

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

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

- But without types, they all reduce to values

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

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

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

- Too Conservative.

$\Rightarrow$ Code Duplication.

- sort
- map
- .....

System F (Reynolds 74) (Girard 72).

Intuition

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

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

What's varying? Types.

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

Big Lambda Λ Grammar

$$
\begin{aligned}
e &::= x \mid \lambda\{x:t\}.\, e \mid e\ e \mid n \mid \mathrm{add1}\ e \\
&\mid \Lambda T.\, e \qquad \text{(Big Lambda)} \\
&\mid e[t] \qquad \text{(Type application)}
\end{aligned}
$$

$$
\begin{aligned}
t &::= \mathrm{int} \mid t \to t \\
&\mid \forall T.\, t \qquad \text{(Universal/forall type)} \\
&\mid T \qquad \text{(Type variable)}
\end{aligned}
$$

Big Lambda Semantics

- Type application is similar to term application

Substitution

$$\text{let } id = \Lambda T.\ \lambda\{x:T\}.\ x.$$
$$\text{in } (id\ [int \to int]\ (\lambda\{x:int\}.\ x))$$
$$(id\ [int]\ 74)$$

$$\Rightarrow (((\Lambda T.\ \lambda\{x:T\}.\ x)\ [int \to int]\ (\lambda \dots))$$
$$((\Lambda T.\ \lambda\{x:T\}.\ x)\ [int]\ 74)$$

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

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

$$\Rightarrow 74$$

User Defined Types

Warmup example

```
class Counter {
    Counter() {...}
    Counter inc() {...}
    int get() {...}}
-------------------------------
new Counter().inc().get()
```

System F Example

$$
\begin{array}{l}
(\wedge Counter.\ \lambda\{new : unit \to Counter\}. \\
\qquad \lambda\{inc : Counter \to Counter\}. \\
\qquad \lambda\{get : Counter \to int\}. \\
\qquad get\ (inc\ (new\ ())))
\end{array}
$$

int

$(\lambda\{x : unit\}. \cdots)$

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

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

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.

Background: Record types & Subtyping

Record types

$e ::= \cdots$
$\quad | \{l_1 : e_1, \ldots, l_n : e_n\}$
$\quad | \; e.l_i$

$t ::= \cdots$
$\quad | \{l_1 : t_1, \ldots, l_n : t_n\}$

$\{count : 74\}.count \Rightarrow 74$

Why Subtyping?

$\{count : int, extra : int\}$

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

Won't typecheck!

$$\frac{\forall i \in [1, n],\ \Gamma \vdash s_i \le t_i}{\Gamma \vdash \{l_1 : s_1, \ldots, l_n : s_n\} \le \{l_1 : t_1, \ldots, l_n : t_n, \ldots\}}$$

Adding Subtyping to System F

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

type: $\{count:int\} \to int \times \{count:int\}$

Implicit Subtyping "forgets" actual type.

term: $\wedge T.\ \lambda\{x:T\}.(x.count, x)$

type: $\forall T.\ T \to int \times T$

$(\wedge T. \ldots)$ can't "see through" T.

So close...

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

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

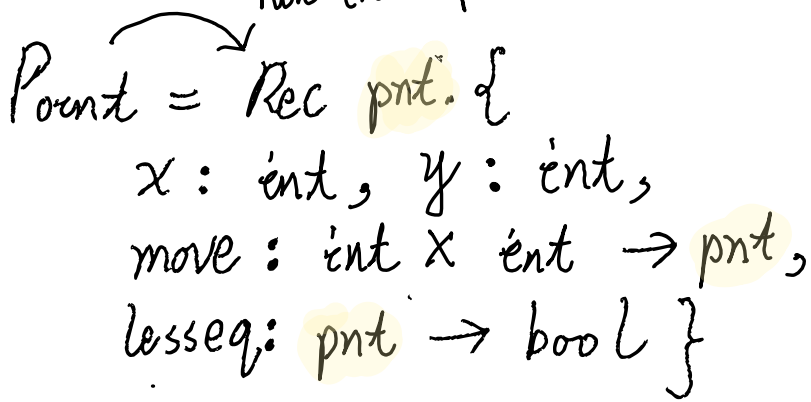
$\Rightarrow (\lambda\{x: \{count: int, extra: int\}\}.\ (x.count, x))$
$\{count: 74, extra: 85\}$

$\Rightarrow \ldots$

Background: Objects & Recursive Types

- Goal: Model Objects.

```
class Point {
    int x, y;
    Point move(int dx, int dy) {...}
    boolean lesseq(Point other) {...}
}
```



Background: Function Subtyping

- Conditions for $A_1 \to B_1 \leq A_2 \to B_2$
  $A_1 \leq A_2 \wedge B_1 \leq B_2$ (?)

$(\lambda \{f: Fruit \to Fruit\}.$
$(f\ banana, f\ apple))$

$: (Fruit \to Fruit) \to (Fruit \times Fruit)$

What types of 'f' is acceptable?

f_1: Apple $\to$ Orange $\not\leq$ Fruit $\to$ Fruit

f_2: Top $\to$ Orange $\leq$ Fruit $\to$ Fruit

f_3: Top $\to$ Top $\not\leq$ Fruit $\to$ Fruit

Limitations of Bounded Quantification, Canning et al. 89

Function subtyping:

$$\frac{\Gamma \vdash A_2 \le A_1 \quad \Gamma \vdash B_1 \le B_2}{\Gamma \vdash A_1 \to B_1 \le A_2 \to B_2}$$

Recursive type subtyping

$$\frac{\Gamma, s \le t \vdash T_1 \le T_2}{\Gamma \vdash \text{Rec } s.T_1 \le \text{Rec } t.T_2}$$

Model "Interface"

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

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

- $\text{Comparable} = \text{Rec } cp.\{\text{lesseq}: cp \to \text{bool}\}$
  $\text{Point} \le \text{Comparable}$ requires $cp \le pnt$ ✗

Limitations of Bounded Quantification Cont

- Where's the problem?
  - Moveable = Rec mv. {move: Real×Real→mv}
  - Comparable = Rec cp. {lesseq: cp → bool}

Type system is right!

mv/cp could be any Moveable/Comparable

⇒ need more specific types.

Limitation of Bounded Quantification Cont.

Attempt 2: Leave it to the users

- $Moveable \simeq \forall mv.\ \{move: Real \times Real \to mv\}$
- $Comparable \simeq \forall cp.\ \{lesseq: cp \to bool\}$

$Point \leq Moveable[Point]$ ✓.

$Point \leq Comparable[Point]$ ✓.

But we can't use it as we'd like to

$\Lambda T \leq Moveable[T]. \lambda\{x:T\}\ x.move(1,1)$

not in scope

F-bounded Quantification Canning et al. 89

Def

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

Allows t in the bound.

Ex

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

: $\forall\, T \leq \text{Comparable}[T].\ T \to T \to T$

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
- translates to old Java.

Ex

```
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 other); }

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

Heterogeneous Translation

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

class Num implements Comparable_num {
    int val;
    int compare(Num o) {...} }
```

Generic Methods

```
<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; }
```

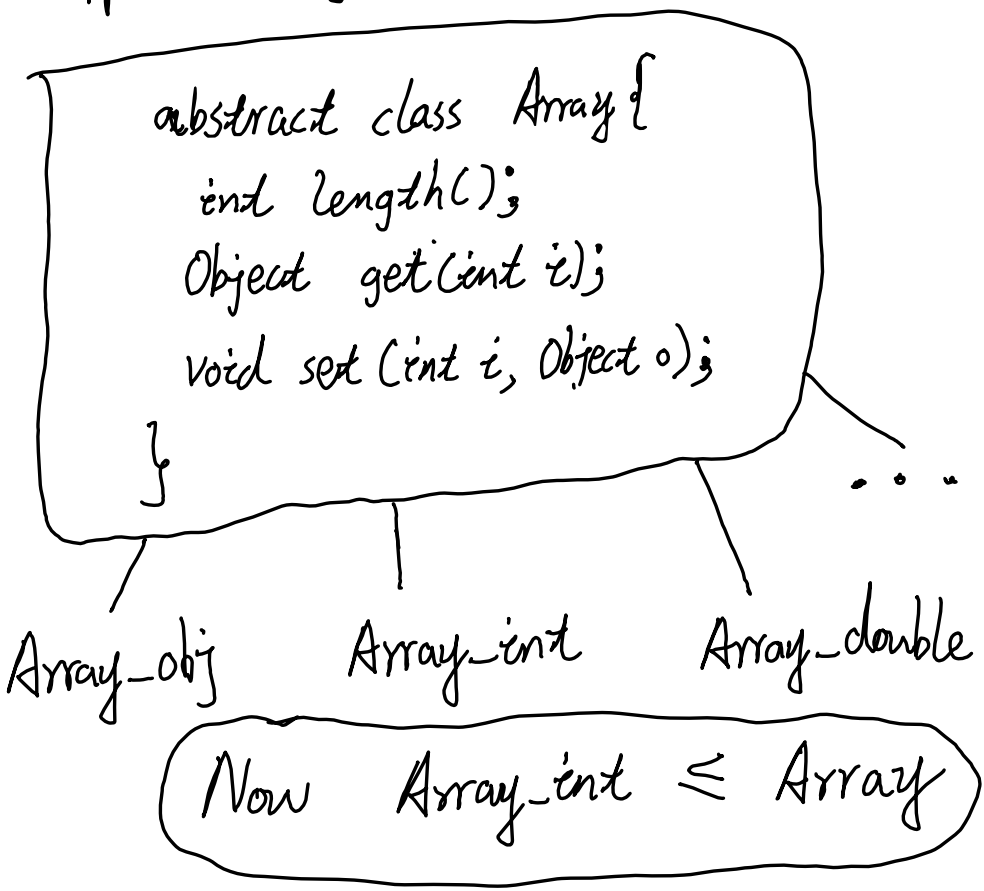
⇒

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

- ISSUE: `int[]` $\not\subseteq$ `Object[]`

Homogeneous Translation & Java Array Cont.

- Wrapper Array class



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

Generics & Subtyping

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

Assume yes

```
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(); //ERROR.
```

- Generics must be <u>invariant</u>.
  type variables must match <u>exactly</u>

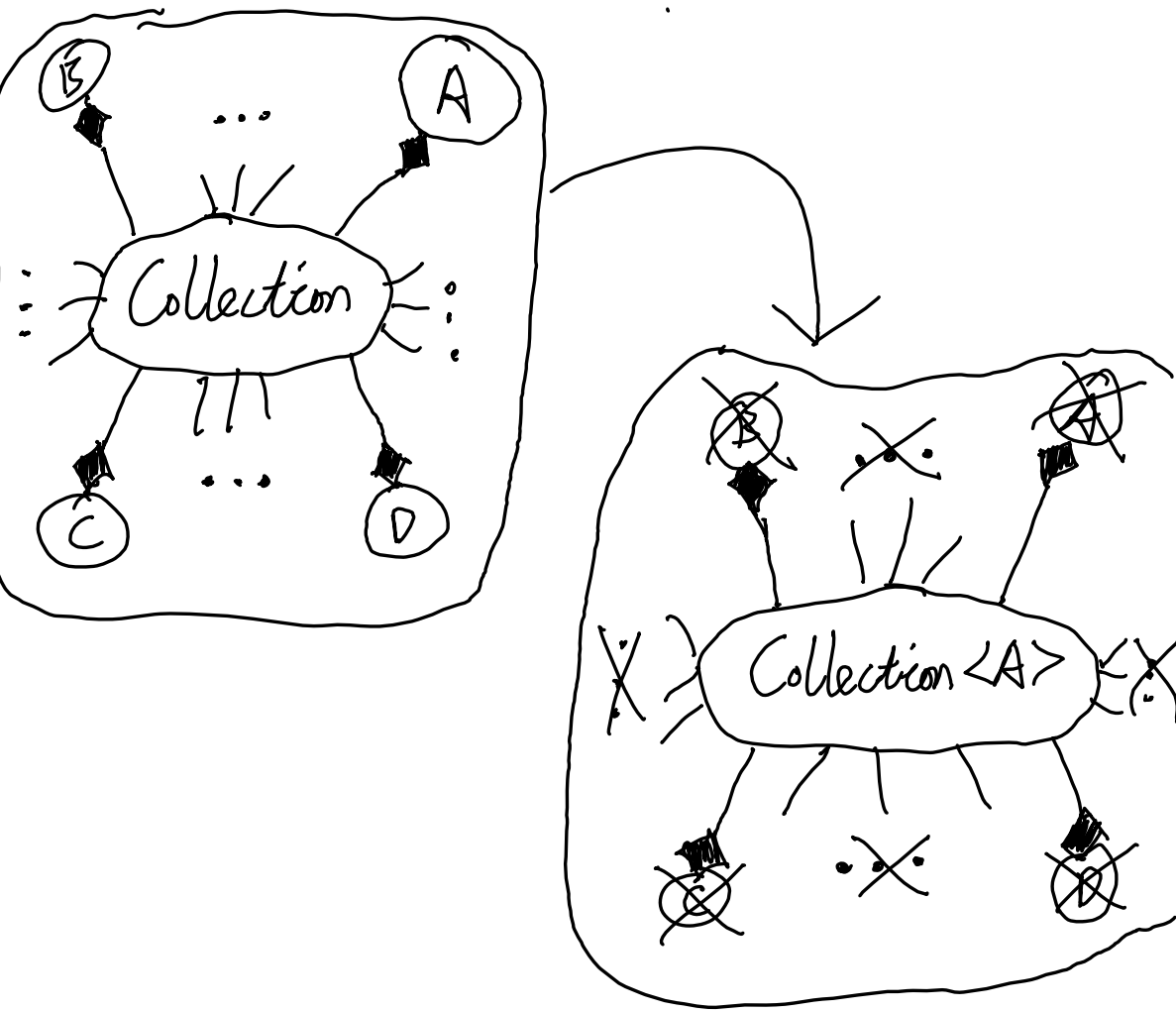
Homogeneous translation & Casting

```
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



Odersky et al. 98 proposes GJ to

- Add generics to Java
- Address legacy code compatibility issues

Raw Types in GJ

Goal:

Old code + new library.

Observation:

Legacy code uses "Object & Casting" idiom

Idea:

Allow this idiom with generics

```
class ArrayList<T> {...}
ArrayList<Integer> ns = new ArrayList<Integer>();
ArrayList          raw = new ArrayList();
```

Warning Generation

Ex

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

Generates unchecked warning on;

- assignment to raw type
  if lhs type contains type variable.

- method call to raw type
  if parameter contains type variable.

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

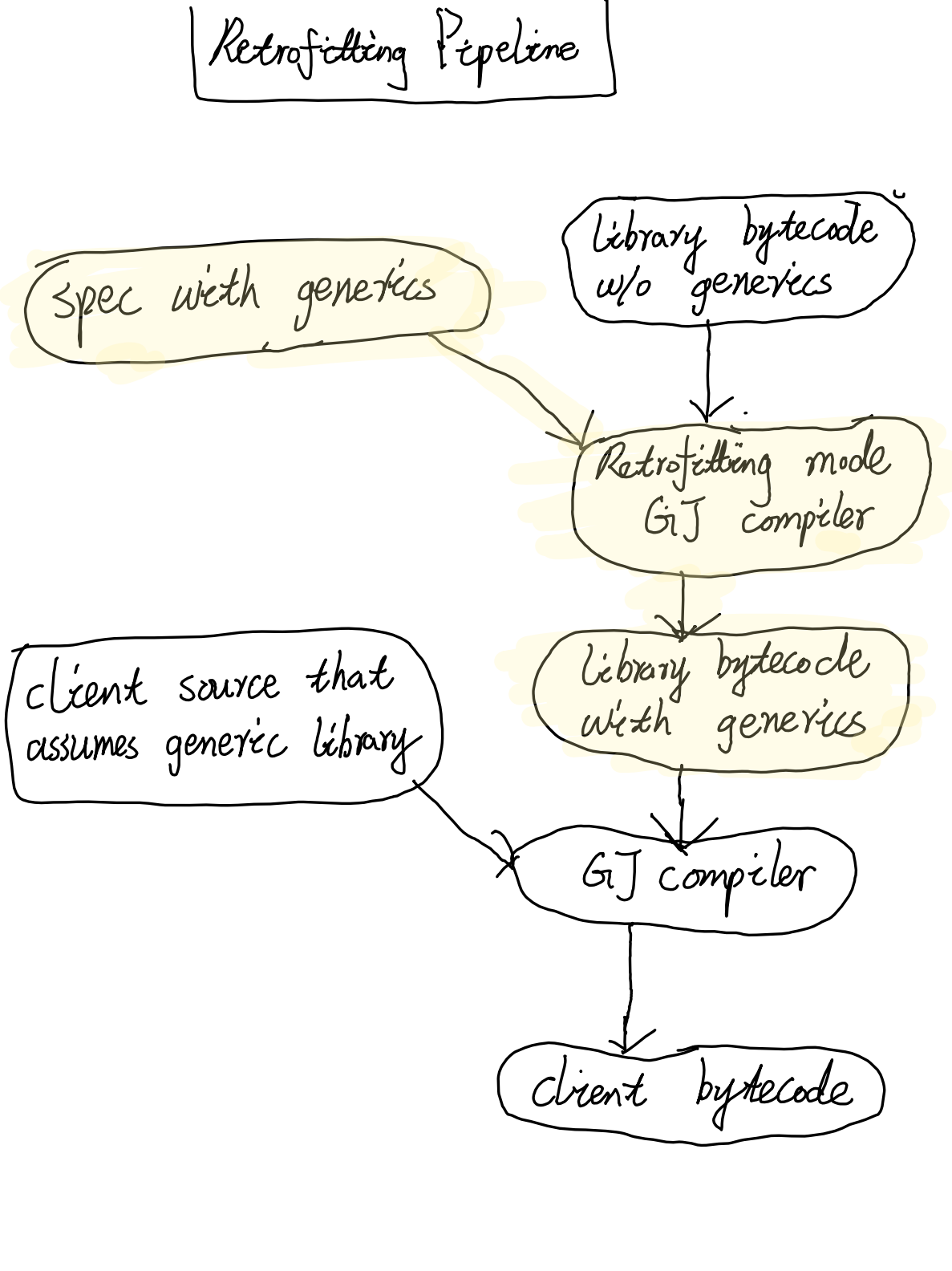
- Goal:
  New code + old libraries
- Observation:
  Generics translates to old idiom anyway
- Idea:
  Trust old libraries. Leave them alone

How does compiler know?

- "retrofitting mode"

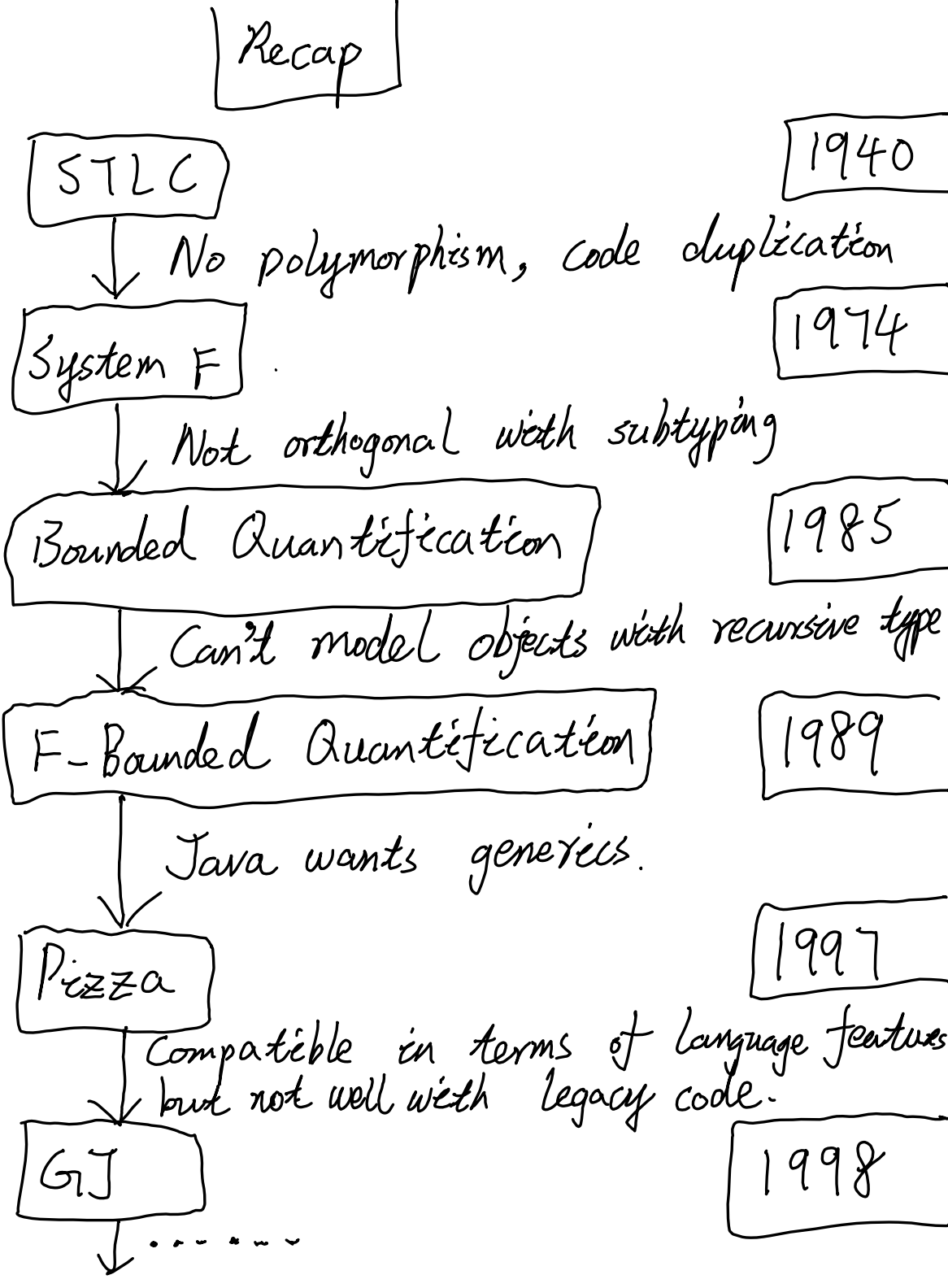
retrofitting spec file.

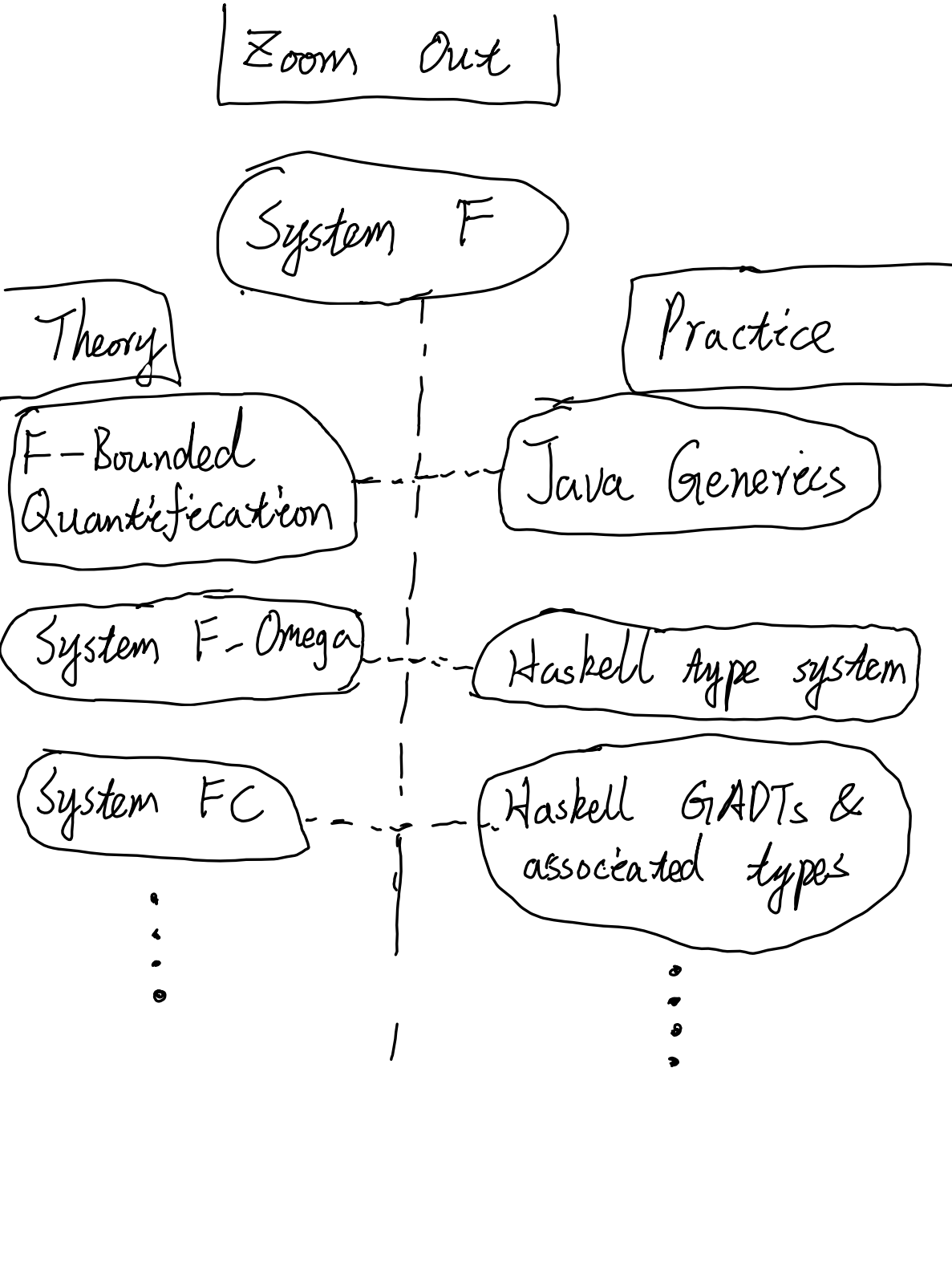
```
class ArrayList<T> {
    void add(T x);
    T    get(int i);
    ..... }
```


Retrofitting Pipeline
Library bytecode w/o generics
Spec with generics
Retrofitting mode GJ compiler
Library bytecode with generics
client source that assumes generic library
GJ compiler
client bytecode

Summary on code Compatibility

Library state	Client Assumption	How
Collection<T>	Collection	Raw Types
Collection	Collection<T>	Retrofitting
Collection	Collection	Programmers' caution
Collection<T>	Collection<T>	compiler checks


Recap
STLC
1940
No polymorphism, code duplication
System F
1974
Not orthogonal with subtyping
Bounded Quantification
1985
Can't model objects with recursive type
F-Bounded Quantification
1989
Java wants generics.
Pizza
1997
Compatible in terms of language features
but not well with legacy code.
GJ
1998


Zoom Out
System F
Theory
Practice
F-Bounded Quantification
Java Generics
System F-Omega
Haskell type system
System FC
Haskell GADTs & associated types