Background to OO Genericity

(1) Inclusion Polymorphism (original Java 1994)

```
class Cell extends Object{
  Object fst;
 Object getFst() {return this.fst;}
 void setFst(Object o) { this.fst=o;}
}

Cell a; Int b;
a.fst=b;//safe
b=(Int)a.getFst();//runtime check
```

- Can store into generic Object via safe upcast
- Retrieval requires downcast.

Problem : *runtime overhead* and *maybe unsafe*

(2) Parametric Type (GJ) (Bracha et al. OOPSLA98)

Type parameter to denote the class field type

```
class Cell⟨X⟩ extends Object{
  X fst;
 X getFst() {return this.fst;}
 void setFst(X v) {this.fst=v}
}
class Pair⟨X,Y⟩ extends Cell⟨X⟩{
  Y snd;
 void setSnd(Y v) {this.snd=v}
}
Int i; Cell⟨Int⟩ a;
a.setFst(i); //safe
i=a.getFst(); //safe
```

Supports different specialized instances:

Cell<Int>, Pair<Int, Float>, Pair<Num, String>

Parametric Type requires Invariant Subtyping

- Cell<t1> <: Cell<t2> if t1=t2
- Pair<t1,t2> <: Pair<t3,t4> if t1=t3 and t2=t4
- Pair<t1,t2> <: Cell<t3> if t1=t3

Reason:

- Reading requires co-variant subtyping
- Writing requires contra-variant subtyping

Problem with GJ's solution:

Cell<Int> is not a subtype of Cell<Num>

Cell<Int> is a subtype of Cell
(a raw type where field type information is lost!)

(3) Declaration-Site Variance (Eiffel,C#)

field for read-only

field for read/write

```
class ROCell⟨X⟩ extends Object{
  ⊕X fst;
  X getFst() {return this.fst;}
}

class RWCell⟨X⟩ extends Object{
  ⊙X fst;
  X getFst(){return this.fst;}
  void setFst(X v) {this.snd=v;}
}
```

ROCell<Int> <: ROCell<Num>

RWCell<Int> <: RWCell<Int>

Problem : duplication and fixed variance for entire class.

(4) Use-Site Variance (Thorup+Torgensen ECOOP99)

Class Declaration: the variance is parameterized

Object Use-site: the variance is instantiated based on the current use

```
class Cell⟨αX⟩ extends Object{
  αX fst;
}
void main() {
  Cell⟨⊕Int⟩ a=··· //α is instantiated for reading
  Cell⟨⊖Int⟩ b=···//α is instantiated for writing
  Int c = a.fst;//read-use
  b.fst= c; //write-use
··· }
```

Variant ParametricTypes (Igarashi+Viroli ECOOP02)

Read and write access (invariant subtyping):

$\mathtt{Cell}\langle\odot\mathtt{t1}\rangle <: \mathtt{Cell}\langle\odot\mathtt{t2}\rangle$ if $\mathtt{t1}=\mathtt{t2}$

Read-only access (co-variant subtyping):

$\mathtt{Cell}\langle\oplus\mathtt{t1}\rangle <: \mathtt{Cell}\langle\oplus\mathtt{t2}\rangle$ if $\mathtt{t1} <: \mathtt{t2}$

Write-only access (contra-variant subtyping):

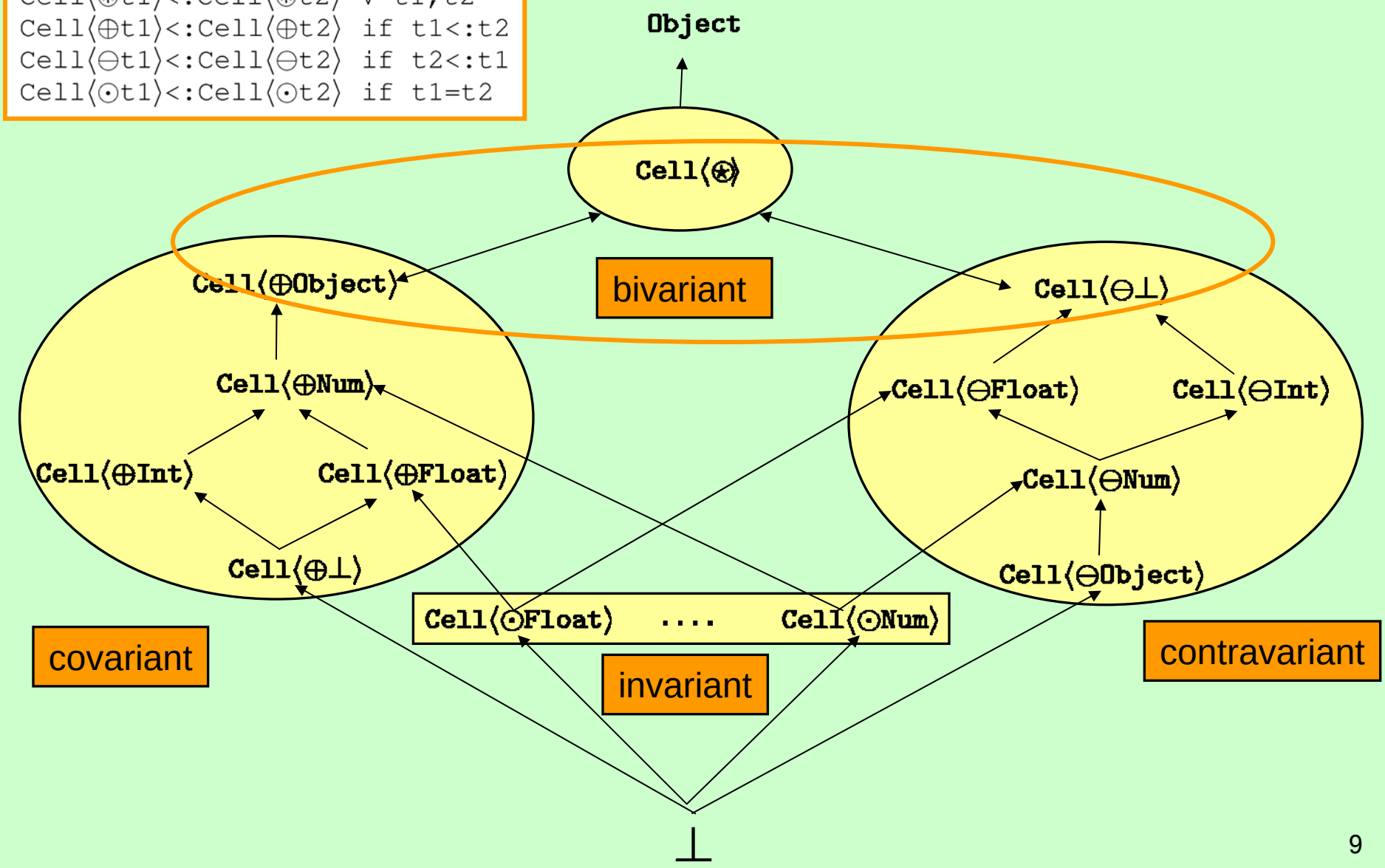
$\mathtt{Cell}\langle\ominus\mathtt{t1}\rangle <: \mathtt{Cell}\langle\ominus\mathtt{t2}\rangle$ if $\mathtt{t2} <: \mathtt{t1}$

No access (bivariant subtyping):

$\mathtt{Cell}\langle\circledast\mathtt{t1}\rangle <: \mathtt{Cell}\langle\circledast\mathtt{t2}\rangle \ \forall\ \mathtt{t1},\mathtt{t2}$

Rich Variant Subtype Hierarchy

```
Cell⟨⊛t1⟩<:Cell⟨⊛t2⟩ ∀ t1,t2
Cell⟨⊕t1⟩<:Cell⟨⊕t2⟩ if t1<:t2
Cell⟨⊖t1⟩<:Cell⟨⊖t2⟩ if t2<:t1
Cell⟨⊙t1⟩<:Cell⟨⊙t2⟩ if t1=t2
```



Wildcard Types from Java 5

VPT to Wildcard Types from Java 1.5 :

Cell<⊛Num> corresponds to Cell <?>
Cell<⊕Num> corresponds to Cell <? extends Num>
Cell<⊖Num> corresponds to Cell <? super Num>
Cell<⊙Int> corresponds to Cell<Int>