SWEN 423 Mock Term Test 1Model Answers
Extending FJ with Generics
[..]
Section 1: Grammar and general considerations

```
e  ::= x | e.<Ts>m(es) | new C<Ts>(es) | e.x
T  ::= C<Ts> | X
GD ::= X₁ extends Ts₁..Xₙ extends Tsₙ
cd ::= class C<GD> implements Ts{ Fs K Ms }
  | interface C<GD> extends Ts{ MH₁; .. MHₖ; }
K  ::= C(T₁ x₁ .. Tₙ xₙ){this.x₁=x₁;..this.xₙ=xₙ;}
F  ::= T x;
M  ::= MH{return e;}
MH ::= <GD> T m(T₁ x₁ .. Tₙ xₙ)
v  ::= new T(vs)
Ɛv ::= □ | Ɛv.<Ts>m(es) | v.<Ts>m(vs,Ɛv,es)| new T(vs,Ɛv,es) | Ɛv.x
Γ  ::= x₁ : T₁ .. xₙ : Tₙ
```

In addition to the FJ well formedness criteria, we add the following:
-The main expression does not contain any X or any x
-All generic type names X contained in the left hand side of a generic declaration GD are unique.
-All T in the Ts contained in the right hand side of a generic declaration GD can not be of form X (but can contain some X, as in “List<X>”).
-The types Ts in the extends or implements list can not be of form X.
-In a class or interface C<GD>, all generic type names X in the methods GD' are disjoint from the generic type names introduced in GD.
-Forall the 'X extends Ts' in a GD, all T in Ts are of form C<_>, where C is an interface declaration in cds.

We assume the conventional notation dom(cds,C)=ms, returning the list of names directly defined in C.

[Q1; 10 points]:
Consider the following examples of regular Java with generics:

```
class Sorter<T extends Comparable<T> & Serializable>{
  public <L extends List<T>> L sortAndSave(L l){
    l=this.<L>sort(l);
    l=this.save(l);
    return l;
  }
  public <L> L save(L l){/**/}  public <L> L sort(L l){/**/}  }
```

This is correct Java code. Explain in the detail the meaning of such code, focus on the role of '&' and the role of 'this.<L>sort('
& allows generic constraints to bound on multiple interfaces, so that only a T that implement both Comparable<T> and Serializable can be used.

[Q2; 10 points]:
The code above is correct Java but does not respect the restriction of GFJ. Rewrite the code above so that it is a syntactically correct in GFJ.

```
class Sorter<T extends Comparable<T> & Serializable<>>{
  public <L extends List<T>> L sortAndSave(L l){return
this.<L>save(this.<L>sort(l));}
  public <L> L save(L l){/**/}  public <L> L sort(L l){/**/}  }
```

Note how we can not declare local variables in GFJ.

[Q3; 5 points]:
In your words, describe the role of GD. How is C<GD> different w.r.t. C<Ts>?
What is the role of X?
Use correct terminology.

GD is the generic declarations: it is similar to formal parameter declaration, where a type and a variable name are introduced. Here the X correspond to the variable name, while Ts corresponds to the type name.
In C<Ts>, Ts is similar to passing actual parameters in a method call, and each individual T corresponds to a value that satisfy an individual parameter.

Section 2: Small step reduction

We now show the small step reduction for GFJ. As you will notice, it is very similar to FJ reduction

[Q4; 5 points]: There are two minor typos in the conventional rule (ctx). What are they?

$$\text{(ctx)}\ \frac{e_1 \longrightarrow e_2}{\mathcal{E}v[e_1] \longrightarrow \mathcal{E}v[e_2]}$$

[Q5; 5 points]: There are three minor typos also in (f-access). What are they?

$$\text{(f-access)}\ \frac{\texttt{class C<\_> \_\{ T}_1\ x_1\texttt{;..T}_n\ x_n\texttt{; K Ms \} in cds}}{\texttt{new C<Ts>(}v_1..v_n\texttt{).}x_i \longrightarrow v_i}$$

Note, It is ok for Ts in new C<Ts> to be unused

$$\text{(m-call)}\ \frac{\texttt{class C<GD> \_ \{ \_ <GD'>}T_0\ \texttt{m(}T_1\ x_1..T_n\ x_n\texttt{)\{return e;\} \_ \} in cds} \quad e'=e[GD=Ts][GD'=Ts'][this=new\ C<Ts>(vs)][x_1=v_1]..[x_n=v_n]}{\texttt{new C<Ts>(vs).<Ts'>m(}v_1..v_n\texttt{)} \longrightarrow e'}$$

#Define e[GD=Ts] = e' e[X=T] = e T[X=T']=T"

$e[X_1 \text{ extends } Ts_1 .. X_n \text{ extends } Ts_n = T_1..T_n] = e[X_1=T_1]..[X_n=T_n]$

$x[X=T] = x$
$e.<T_1..T_n>m(e_1..e_k)[X=T] = e.<T_1[X=T]..T_n[X=T]>m(e_1[X=T]..e_k[X=T])$
$\text{new } T(e_1..e_n)[X=T'] = \text{new } T[X=T'](e_1[X=T']..e_n[X=T'])$
$e.x[X=T] = e[X=T].x$

$C<T_1..T_n>[X=T] = C<T_1[X=T]..T_n[X=T]>$
$X[X=T] = T$
$X[X'=T] = X \text{ with } X!=X'$

[Q6; 5 points]: Define the conventional variable substitution e[x=e'] for GFJ.

#Define e[x=e'] = e"
$x[x=e] = e$
$x[x'=e] = x \text{ with } x != x'$
$e_0.<Ts>m(e_1 .. e_k)[x=e] = e_0[x=e].<Ts>m(e_1[x=e].. e_k[x=e])$
$\text{new } C<Ts>(e_1 .. e_n)[x=e] = \text{new } C<Ts>(e_1[x=e]..e_n[x=e])$
$e_0.x[x=e] = e_0[x=e].x$
Ts need to be mentioned and not captured with '_' because we need to keep it the same to the left and the right of the '='

[Q7; 10 points]:
Since the main expression must not contain any X,
also the T used with the notation e[X=T] will never contain any X.
The current definition of e[X=T] works well under this assumption, but it would produce strange results in case there was some X inside of T that is also present inside of e.
According to the former definition of e[X=T], what is the result of the following operations:
(A)-new C<W,D<Y>>(new K<Z>())[W=K<Z>][Z=J<>]
 ==new C<K<Z>,D<Y>>(new K<Z>())[Z=J<>]
 ==new C<K<J<>>,D<Y>>(new K<J<>>())
(B)-new C<W,D<Y>>(new K<Z>())[Z=J<>][W=K<Z>]
==new C<W,D<Y>>(new K<J<>>())[W=K<Z>]
==new C<K<Z>,D<Y>>(new K<J<>>())

Note how the order of replacements was relevant because Z was both at the left (Z=J<>) and at the right (W=K<Z>):
new C<K<J<>>,D<Y>>(new K<J<>>()) //first replace W, then Z
new C<K<Z>,D<Y>>(new K<J<>>()) //first replace Z, then W

Section 3: Class Type system

```
_______
#Define canonical(cds,GD) = Ts;cds'
  canonical(cds,ø) = ø;ø
  canonical(cds,X extends Ts GD) = C<> Ts'; cd cds'
    cd = interface C<> extends Ts[X=C<>]{Ms}
    forall T in Ts[X=C<>]  cds cd |- T : ok
    Ts'; cds' = canonical(cds cd,GD[X=C<>])
    overrideOk(C<> Ts[X=C<>])
    forall m in dom(cd,C<>) exists T in Ts m in dom(cds,T)
```

Note how thanks to last two premises of canonical all M in Ms must be the same of one of the of the methods in one of the extended interfaces Ts.

```
_______
#Define cds|-overrideOk(Ts)      cds|-overrideOk(T,T',m)
  cds|-overrideOk(Ts)
    forall T,T' in Ts, forall m in dom(cds,T) ∩ dom(cds,T')
      cds|-overrideOk(T,T',m)
  cds|-overrideOk(C₀<Ts₀>,C₁<Ts₁>,m)
    _ C₀<GD₀> _{_ <GD'> T₀ m(T₁ x₁..Tₙ xₙ)_ _} in cds
    _ C₁<GD₁> _{_ <GD'> T'₀ m(T'₁ x₁..T'ₙ xₙ)_ _} in cds
    forall i in 0..n Tᵢ[GD₀=Ts₀]=T'ᵢ[GD₁=Ts₁]

_______
#Define GD[X=T]=GD'    Ts[X=T]
  (X₁ extends Ts₁..Xₙ extends Tsₙ)[X=T] = X₁ extends Ts₁[X=T]..Xₙ extends Tsₙ[X=T]
  (T₁..Tₙ)[X=T] = T₁[X=T]..Tₙ[X=T]
```

Rules (T-ok), (GD-T) and (GD-E) check that a type is well formed with respect to the declared bounds. For example if C is declared as
  “class C<X extends Shape<>>{_}”
then the type “C<String<>>” will not be ok, and
“String<> okWith X extends Shape<>” will not hold.

```
        cds|- Ts okWith GD
        _ C<GD> _{_} in cds
(T-ok)--------------------------------
        cds|-C<Ts> : ok
```

```
       cds|- Ts okWith GD[X=C<Ts'>]
       cds|- C<Ts'> ≤ Tᵢ forall i in 1..n
       cds|- Tᵢ : ok forall i in 1..n
(GD-T)-----------------------------------------
       cds|- C<Ts'> Ts okWith X extends T₁..Tₙ GD

(GD-E)-------------------------------------
       cds|- ∅ okWith ∅
```

[Q8; 10 points]:
Note in the formal definition of canonical the line "Ts'; cds' = canonical(cds cd, GD[X=C<>])".
What would change if that line was instead just "Ts'; cds' = canonical(cds cd, GD)"?
Provide a minimal call to canonical where this different definition would produce a different result.

canonical(_,X extends A<>, Y extends B<X>) creates
interface X1<> extends A<> {} interface Y1<> extends B<X1<>> {}
in the original definition, but it would instead create
interface X1<> extends A<> {} interface Y1<> extends B<X> {}
if we removed the X=C<> part.

Rule (id) is quite involved, in addition of the checks already present in FJ, we also check that all the type mentioned in the interface are parameterized correctly,
using cds |- T : ok and cds|- Ts okWith GD

```
    dom(cds,C') ⊆ dom(cds,C) forall C'<_> in Ts
    canonical(cds,GD) = Ts';cds'
    cds cds'|-overrideOk(C<Ts'>,Ts[GD=Ts'])
    forall T in Ts   cds cds'|- T[GD=Ts'] : ok
    forall MH in MHs   cds cds'|- MH[GD=Ts']: ok
(id)-------------------------------------------------------
    interface C<GD> extends Ts{MHs} : ok

        canonical(cds,GD)=Ts;cds'
        forall i in 0..n cds cds'|- Tᵢ[GD=Ts] : ok
(MH-ok)-----------------------------------
        cds|-<GD> T₀ m(T₁ x₁..Tₙ xₙ) : ok
```

_____
#Define MH[GD=Ts] MH[X=T]
MH[X₁ extends Ts₁ .. Xₙ extends Tsₙ = T₁..Tₙ] = MH[X₁=T₁]..[Xₙ=Tₙ]
<GD> T₀ m(T₁ x₁..Tₙ xₙ)[X=T] = <GD[X=T]> T₀[X=T] m(T₁[X=T] x₁..Tₙ[X=T] xₙ)

[Q9; 5 points]:
Describe how the rule (id) uses canonical and overrideOk.
In particular, the formal definition of canonical check for overrideOk internally. How this impact rule (id)?

[Q10; 5 points]: consider the following program; is it accepted by GFJ? If not, what is that does not holds?

```
interface A{ A m();}
interface I<X extends A>{}
```

is it accepted by GFJ?
No, it does not even respect the syntax of GFJ. It should be

```
interface A<>{ A<> m();}
interface I<X extends A<>>{}
```

This version would be correct.
[Q11; 5 points]: consider the following program; is it accepted by GFJ? If not, what is that does not holds?

```
interface A<>{ A<> m();}
interface B<>{ B<> m();}
interface I<X extends A<> & B<>>{}
```

No, even the fixed version would not be accepted. Informally, the reason is that method m() is defined in both A and B but with different return types.
Formally, rule (id) does not hold because inside of canonical(_, X extends A<> & B<>)
overrideOk(..) does not hold: we locate two methods m() with different signature

[Q12; 5 points]: consider the following program; is it accepted by GFJ? If not, what is that does not holds?

```
interface A<>{ A<> m();}
interface B<>{ B<> m();}
interface I<X extends A<>> extends A<>{ B<> m();}
```

No, even the fixed version would not be accepted. Informally, the reason is that method m() is defined in both A and I but with different return types.
Formally, rule (id) does not hold because overrideOk(..) does not hold: we locate two methods m() with different signature. Note how overrideOk is called with 'C<Ts'>,Ts[GD=Ts']', thus also the methods of I are included in the check.

Rule (cd) is very similar to rule (id), but also requires for methods to be well typed.

$$\frac{\begin{array}{l}\texttt{C<GD> |- } M_1\texttt{: ok .. C<GD> |- } M_n\texttt{: ok}\\ \texttt{dom(cds,C') } \subseteq \texttt{ dom(cds,C) forall C'<\_> in Ts}\\ \texttt{canonical(cds,GD) = Ts';cds'}\\ \texttt{cds,cds'|-overrideOk(C<Ts'> Ts[GD=Ts'])}\\ \texttt{forall T in Ts cds cds'|- T[GD=Ts'] : ok}\\ \texttt{forall MH e in } M_1\texttt{..}M_n\texttt{ cds cds'|- MH[GD=Ts']: ok}\\ \texttt{forall T x; in Fs cds cds'|- T[GD=Ts']: ok}\end{array}}{\texttt{class C<GD> implements Ts\{Fs K } M_1\texttt{..}M_n\texttt{\} : ok}}\ (cd)$$

$$\frac{\begin{array}{l}\texttt{forall i in 0..n T'}_i\texttt{=}T_i\texttt{[GD GD'=Ts']}\\ \texttt{canonical(cds, GD GD') = Ts';cds'}\\ \texttt{GD= } X_1\texttt{ extends } Ts_1\texttt{ .. } X_n\texttt{ extends } Ts_n\\ \texttt{T=C<}X_1\texttt{..}X_n\texttt{>[GD GD'=Ts']}\\ \texttt{cds cds';this:T } x_1\texttt{:T'}_1\texttt{..}x_n\texttt{:T'}_n\texttt{ |- e[GD GD'=Ts'] : T'}_0\end{array}}{\texttt{C<GD>|- <GD'>}T_0\texttt{ m(}T_1\ x_1\texttt{..}T_n\ x_n\texttt{)\{return e;\} : ok}}\ (meth)$$

Rule (meth) is the crucial rule of the whole type system.
Remember that notation [GD=Ts] only works as expected when Ts does not contains any X.

[Q13; 5 points]:
Explain why we can be sure that there is no X in Ts'
Ts' is returned by canonical. Looking to the definition of canonical we can see that the resulting Ts are all of form C<>, thus without any X.

[Q14; 5 points]:
Can the result of e[GD GD'=Ts'] contains an X? Can T'0 contains an X?
If so, produce an example class declaration where this can happens,
otherwise explain why we can be sure that there is no X in those terms.

No, neither the result of e[GD GD'=Ts'] nor T'0 can contain an X.
Otherwise, the rule (meth) would not be applicable: we do not have a way to type an expression that contains an X. For example if we have new C<X>() this would not be well typed with rule (n-t) becuase "cds|- Ts okWith GD" would fail with Ts=X: rule (GD-T) requires all T to be of form C<_>.
This in turn enforces that 'e' must only contains the X that are defined in GD or GD', otherwise an X would be left inside the result.
Since we know that e[GD GD'=Ts'] have no X, then the result of typing e[GD GD'=Ts'] must also have no X, since the type system is not introducing fresh types.

Section 4: Expression Type system

Rule (x-t) is standard

$$\frac{}{cds;\Gamma \vdash x : \Gamma(x)}\ (x\text{-}t)$$

$$\frac{cds;\Gamma \vdash e_1 : T'_1 \ldots cds;\Gamma \vdash e_n : T'_n \quad class\ C\langle GD\rangle\ \_\ \{T_1\ x_1; \ldots T_n\ x_n;\ K\ Ms\}\ in\ cds \quad forall\ i\ in\ 1..n\ T'_i = T_i[GD=Ts] \quad cds \vdash Ts\ okWith\ GD}{cds;\Gamma \vdash new\ C\langle Ts\rangle(e_1 \ldots e_n) : C\langle Ts\rangle}\ (n\text{-}t)$$

Rule (n-t) relies on cds|- Ts okWith GD to check that the type parameters satisfy the generic declaration.
[Q15; 5 points]:
Provide an example cds, GD and Ts where cds|- Ts okWith GD holds, and one example where it does not holds.

The simpler examples will have Ts and GD of size 1, so we avoid the recursive calls.
Holds example: Ts=C<> and GD= X extends emptySet

$$\frac{}{cds \vdash C\langle\rangle\ okWith\ X\ extends\ emptySet}$$

Non holds example: Ts=C<> and GD= X extends Foo<> where class C<>{} in cds

$$\frac{cds \vdash C\langle\rangle \leq Foo\langle\rangle}{cds \vdash C\langle\rangle\ okWith\ X\ extends\ Foo\langle\rangle}$$

In this rule (instantiated from (GD-T) the premise does not hold since C<> is not a subtype of Foo<>.

```
     cds;Γ |- e₀ : C₀<Ts₀> .. cds;Γ |- eₙ : Cₙ<Tsₙ>
     _ C₀<GD> _ {_ <GD'>T m(T₁ x₁..Tₙ xₙ)_ _} in cds
     forall i in 1..n Cᵢ<Tsᵢ>=Tᵢ[GD=Ts₀][GD'=Ts']
     cds|- Ts' okWith GD'
(m-t)-------------------------------------
     cds;Γ |- e₀.<Ts'>m(e₁ .. eₙ) : T[GD=Ts₀][GD'=Ts']
```

[Q16; 10 points]:
Using correct terminology, explain the rule (m-t) in English.
In particular, carefully describe the role of “Ci<Tsi>=Ti[GD=Ts][GD'=Ts']”.
Would the variation “Ci<Tsi>=Ti[GD GD'=Ts Ts']” behave identically?

Explanation in English in the class.
Yes, such variation will behave identically, just look to the definition of [GD=Ts]

[Q17; 10 points]:
Similarly to rule (m-t), define rule (f-t) to type field access expressions.

ANSWER:

```
     cds;Γ |- e₀ : C<Ts>
     _ C<GD> _ { _ T x _ _ _} in cds
     cds|- Ts okWith GD
(f-t)-------------------------------------
     cds;Γ |- e₀.x : T[GD=Ts]
```

In the following you can find the conventional subsumption and subtyping rules.

```
       T ≤ T' Γ |- e : T
(sub)------------------------
       Γ |- e : T'


(sub-refl)---------------------
              T ≤ T

             i in 1..n
             _ C<GD> _ T₁..Tₙ {_} in cds
(sub-direct)-----------------------------
               C<Ts> ≤ Tᵢ[GD=Ts]
//Note: there is correctly no T0 here, so i in 1..n;
//that is why we had to add rule (sub-refl)

            T₁ ≤ T₂ T₂ ≤ T₃
(sub-trans)------------------
               T₁ ≤ T₃
```

[Q18; 5 points]:
There is a minor typos in of those last 3 rules. What is it?