SWEN 423 Mock Term Test 1

Released 8am Friday 31 July Due 8am Saturday 1 August

Total 120 points, capped at 100; so if you get 105 or 110 points is exactly like getting 100%.

Extending FJ with Generics

We now extend FJ with generic methods and generic classes.
To make this extension easier, we **do not** model the way Java can infer generic types for generic method invocation. In the same way, we override the semantic of Java syntax "Foo<>",: in Java such syntax asks to infer the generic parameters for Foo, while in our FJ extension, it denotes the application of zero type parameters to the class name Foo.

Note how there is a distinction between class and interface names C and type names T.
Also, note a difference w.r.t. Java: a type with zero generic arguments is written as "C<>", while in Java we would just write "C". This is similar to what already happens for a class with an empty set of implements: in FJ we would write "class C implements {" but in Java we would omit the "implements" keyword too.
Thus, a type either finishes with "<..>" or it is an X: a generic type name.

Generic declarations GD are of form 'X extends Ts', where we do not specify the separator between the various T in Ts. In the Java concrete syntax, the symbol '&' is used.
Note the difference between lowecase x (variable names) and uppercase X (generic type names).

Section 1: Grammar and general considerations

Here we define GFJ, an extension of FJ with generics:

```
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('

[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.

[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.

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

$$\frac{e_1 \rightarrow e_2}{\mathcal{E}v[e^1] \rightarrow \mathcal{E}v[e]}\ (ctx)$$

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

$$\frac{\text{class C<\_> \{ } C_1\ x_1;..C_n\ x_n;\ K\ Ms\ \}\ \text{in cds}}{\text{new C<Ts>}(v_1..v_n).x_i \rightarrow v_i}\ (f\text{-access})$$

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

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

#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.

[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<>]
(B)-new C<W,D<Y>>(new K<Z>())[Z=J<>][W=K<Z>]

Section 3: Class Type system

Of course, the most interesting part of the extension with Generics is the type system.
Inside of a method declaration, some X can be used in the types.
Instead of List<Number<>> we may encounter List<X>, or even X directly.
This is the most challenging problem when generics are involved.
The solution we propose here involve growing the class table to contain extra interfaces. For example, in the case of a single X extends Ts, an extra interface CanonicalX implementing Ts will be used in place of X to type the method declaration.
Those canonical interfaces are added to cds when typing a method, thus we will keep the class table cds as an explicit parameter, since it is not a constant across all the type system.
The formal definition of canonical relies on overrideOk, whose definition is a little different with respect to the one seen before.
Here we check that is ok for list of Types to be involved in some subtyping relation.
The symbol ⌀ is the empty set.

```
_______
#Define canonical(cds,GD) = Ts;cds'
  canonical(cds,⌀) = ⌀;⌀
  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.

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<>>{}
```

[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<>>{}
```

[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();}
```

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

$$
\begin{array}{ll}
 & C\langle GD\rangle \vdash M_1 : ok \;..\; C\langle GD\rangle \vdash M_n : ok \\
 & dom(cds,C') \subseteq dom(cds,C) \text{ forall } C'\langle\_\rangle \text{ in } Ts \\
 & canonical(cds,GD) = Ts';cds' \\
 & cds,cds' \vdash overrideOk(C\langle Ts'\rangle\ Ts[GD=Ts']) \\
 & \text{forall } T \text{ in } Ts \;\; cds\; cds' \vdash T[GD=Ts'] : ok \\
 & \text{forall } MH\; e \text{ in } M_1..M_n \;\; cds\; cds' \vdash MH[GD=Ts'] : ok \\
 & \text{forall } T\; x; \text{ in } Fs \;\; cds\; cds' \vdash T[GD=Ts'] : ok \\
(cd) & \text{---} \\
 & class\ C\langle GD\rangle\ implements\ Ts\{Fs\ K\ M_1..M_n\} : ok
\end{array}
$$

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

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'

[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.

Section 4: Expression Type system

Rule (x-t) is standard

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

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

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.

$$\frac{\begin{array}{l} cds;\Gamma \vdash e_0 : C_0\langle Ts_0\rangle \ldots cds;\Gamma \vdash e_n : C_n\langle Ts_n\rangle \\ \_\ C_0\langle GD\rangle\ \_\ \{\_\ \langle GD'\rangle T\ m(T_1\ x_1 .. T_n\ x_n)\ \_\ \_\}\ in\ cds \\ forall\ i\ in\ 1..n\ C_i\langle Ts_i\rangle = T_i[GD=Ts_0][GD'=Ts'] \\ cds \vdash Ts'\ okWith\ GD' \end{array}}{cds;\Gamma \vdash e_0.\langle Ts'\rangle m(e_1 \ldots e_n) : T[GD=Ts_0][GD'=Ts']}\ (m\text{-}t)$$

[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?

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

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

$$\frac{T \leq T' \quad \Gamma \vdash e : T}{\Gamma \vdash e : T'}\ (sub)$$

$$\frac{}{T \leq T}\ (sub\text{-}refl)$$

$$\frac{i\ in\ 1..n \quad \_\ C\langle GD\rangle\ \_\ T_1..T_n\ \{\_\}\ in\ cds}{C\langle Ts\rangle \leq T_i[GD=Ts]}\ (sub\text{-}direct)$$

$$\frac{T_1 \leq C_2 \quad T_2 \leq T_3}{T_1 \leq T_3}\ (sub\text{-}trans)$$

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