1 Generic Wyvern

Wyvern is a new statically typed programming language being developed for secure web applications. There are two popular and well-documented methods [**?**,**?**] for implementing generic types.

- *Type Parameters*: Types are parametrised by generic type names that are defined at runtime. This allows programmers to reuse generic code. Many popular languages use generic type parameters such as *Java*, *C#* and *Scala*.
- *Type Members*: Types and objects may contain type members in the same manner as normal field or method members. These can be subtyped by more precise types to provide generic behaviour.

There have long been attempts to formalise a sound small-step semantics for type members in a structural setting for Scala, which have as yet all been unsuccessful due to issues where well-typed expressions are lost during subject reduction. In 2014 Amin et.al. were able to formalise a big step semantics for Scala style type members. While this demonstrates the soundness of the kinds of programs we wish to type check, a small-step semantics offers powerful advantages when reasoning about the behaviour of programs. For this reason we formalise type members with a small step semantics. We build upon the work of Amin et. al. [**?**] to formalise type members in structurally typed languages using a small step semantics.

1.1 Transitivity, Narrowing and Type Members

Two properties that one might reasonably expect to occur naturally in a structurally typed language are *Subtype Transitivity* and *Environment Narrowing* (Figure **??**). Subtype transitivity is a familiar property, and

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad \text{(Subtype Transitivity)}$$

$$\frac{\Gamma, (x : U) \vdash T <: T' \qquad \Gamma \vdash S <: U}{\Gamma, (x : S) \vdash T <: T'} \quad \text{(Environment Narrowing)}$$

Fig. 1. Subtype Transitivity and Environment Narrowing

environment narrowing simply expresses the expectation that we can treat a variable in an environment as having a more precise type without changing the type relationships within that environment.

Since transitivity is often used in proving subject reduction, it is a problem if a type system lacks this property. The issue with transitivity arises from the introduction of type member lower bounds, and their contra-variance. The following example demonstrates this.

```
A = {z ⇒ type N : ⊥ .. ⊤}

B = {z ⇒ type N : ⊥ .. ⊤
         def meth1(x : ⊤){return new{z ⇒}}:⊤}

S = {z ⇒ type L : A .. ⊤
         val f : A}

T = {z ⇒ type L : A .. ⊤
         val f : z.L}

U = {z ⇒ type L : B .. ⊤
         val f : z.L}
```

Here `S` subtypes `T` and `T` subtypes `U`, but `S` does not subtype `U`. Because of the contra-variance of the lower bound of type member `L`, `A` subtypes `z.L` in `T` but not `U`. Amin et al. [**?**] attempt to reconcile this by narrowing the type

$$\frac{\Gamma, z : \{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 <: \overline{\sigma}_2}{\Gamma \vdash \{z \Rightarrow \overline{\sigma}_1\} \ <: \ \{z \Rightarrow \overline{\sigma}_2\}} \quad \text{(Structural Subtyping)}$$

Fig. 2. Subtype Transitivity and Environment Narrowing

of `z` in the larger types `T` and `U` by using the subtype rule in Figure **??**. Here we type check the declaration types of the larger type with a smaller receiver. While this allows for subtype transitivity, it introduces environment narrowing that proves unsound in a small step semantics. Amin et. al. were however able to reconcile both transitivity and narrowing within a big step semantics.

1.2 Path Equality

To further complicate matters, well-formedness can be lost when reducing field accesses.

Julian: Example?

2 Type System

2.1 Syntax

In this section we present the Wyvern Type Members Syntax in Figure **??**.

$$\begin{array}{llll}
e & ::= & x & \textit{expression} \\
 & \mid & \texttt{new}\ \{z \Rightarrow \overline{d}\} & \\
 & \mid & e.m(e) & \\
 & \mid & e.f & \\
 & \mid & e \trianglelefteq T & \\
 & \mid & l & \\
p & ::= & x & \textit{paths} \\
 & \mid & l & \\
 & \mid & p \trianglelefteq T & \\
v & ::= & l & \textit{value} \\
 & \mid & v \trianglelefteq T & \\
d & ::= & \texttt{val}\ f : T = p & \textit{declaration} \\
 & \mid & \texttt{def}\ m(x : T) = e : T & \\
 & \mid & \texttt{type}\ L : T..T & \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x : T & \textit{Environment} \\
A & ::= & \varnothing \mid A,\ <: T & \textit{Assumption Context}
\end{array}$$

$$\begin{array}{llll}
T & ::= & \{z \Rightarrow \overline{\sigma}\} & \textit{type} \\
 & \mid & p.L & \\
 & \mid & \top & \\
 & \mid & \bot & \\
\sigma & ::= & \texttt{val}\ f : T & \textit{decl type} \\
 & \mid & \texttt{def}\ m : T \to T & \\
 & \mid & \texttt{type}\ L : T..T & \\
E & ::= & \bigcirc & \textit{eval context} \\
 & \mid & E.m(e) & \\
 & \mid & p.m(E) & \\
 & \mid & E.f & \\
 & \mid & E \trianglelefteq T & \\
d_v & ::= & \texttt{val}\ f : T = v & \textit{declaration value} \\
 & \mid & \texttt{def}\ m(x : T) = e : T & \\
 & \mid & \texttt{type}\ L : T..T = T & \\
\mu & ::= & \varnothing \mid \mu,\ l \mapsto \{z \Rightarrow \overline{d}\} & \textit{store} \\
\Sigma & ::= & \varnothing \mid \Sigma,\ l : \{\mathtt{z} \Rightarrow \overline{\sigma}\} & \textit{store type}
\end{array}$$

Fig. 3. Syntax

Expressions (e): Expressions are variables (x), new expressions (`new` $\{z \Rightarrow \overline{d}\}$), method calls ($e.m(e)$), field accesses ($e.f$), expression upcasts ($e \trianglelefteq T$) and locations ($l$) in the store. The only expression that differs from tradition is the explicit upcasts on expressions. We use upcasts to avoid the narrowing issues described in Section We employ a similar strategy with regard to the explicitly upcast expression $e \trianglelefteq T$. Here e may have a more precise type than T, but to avoid narrowing we maintain the type T.

Types (T): Types are restricted to structural types ($\{z \Rightarrow \overline{\sigma}\}$), type member selections on paths ($p.L$), ($\top$) the top type at the top of the type lattice that represents the empty type and $\bot$ the bottom type at the bottom

of the type lattice that represents the type containing al possible declaration labels with $\top$ in the contra-variant type position, and $\bot$ in the covariant type position.

Paths (p): Paths are expressions that type selections may be made on. We restrict these to variables (x), locations (l) and upcast paths ($p \trianglelefteq T$).

Values (v): Values in our type system are locations (l) and upcast values ($v \trianglelefteq T$).

Declarations (d): Declarations may be fields (`val`), methods (`def`) or type members (`type`). These are all standard.

Declaration Types (σ): Declaration types may be field (`val`), method (`def`) or type members (`type`).

Declaration Values (d_v): Declaration values are similar to declarations, except we require field initializers to be values.

On top of these, we also include an evaluation context E, an environment Γ that maps variables to types, a store μ that maps locations to objects, a store type Σ used to type check the store and an assumption context A that is used to type check recursive types ([**?**]) that consists of a list of type pairs.

2.2 Semantics

In this section we describe the Wyvern Type Members semantics.

Path Functions Julian: I'll finish describing these functions later once they are finalized.

$$p \equiv p \quad \text{(Eq-Refl)} \qquad \frac{p_1 \equiv p_2}{p_2 \equiv p_1} \quad \text{(Eq-Sym)} \qquad \frac{p_1 \equiv p_2 \qquad p_2 \equiv p_3}{p_1 \equiv p_3} \quad \text{(Eq-Trans)} \qquad \frac{p_1 \equiv p_2}{p_1 \equiv p_2 \trianglelefteq T} \quad \text{(Eq-Path)}$$

Fig. 4. Path Equivalence

In Figure **??** we describe our path equivalence judgement. Two paths are equivalent if they wrap up the same variable or location. This is useful when trying to compare two types that are not technically the same types, but are derived from the same object.

Subtyping Subtyping is given in Figure **??**. We use a modified version of the subtyping relation described by Amin et. al. [**?**]. We remove the reflexivity rule, however this can be inferred from the other subtyping rules. We modify the S-Struct rule for structural subtyping by upcasting the self variable of the larger type to avoid narrowing. This can be seen in S-Struct in Figure **??**. If we didn't do this, we would have to type check the declaration types of the larger type with a smaller receiver, resulting in narrowing. We also need to be able subtype two path equivalent selection types(S-Path in Figure **??**). Since we can have two paths that are equivalent in that they lead to the same location or variable while having different types, we need a way to subtype path equivalent type selections. We require that they have the appropriate contra-variant and covariant relationships between their lower and upper bounds respectively. To deal with recursive types, we use an assumption context A which we use to type check the bounds of the selection type. For this we introduce the S-Assume rule. S-Select-Upper and S-Select-Lower are our rules for subtyping selection types. To supertype or subtype a selection type, a type needs to supertype or subtype its upper or lower bounds respectively. We then use S-Top and S-Bottom to type check the subtyping and supertyping of the top ($\top$) and bottom ($\bot$) types.

Well-Formedness Julian: Do we need to worry about this?

Type Expansion and Membership We use a modified version of the type expansion (Figure **??**) definition and the same membership (Figure **??**) definition from Amin et al [**?**]. Expansion of types is used to extract the set of declaration types for a type. E-Struct is the expansion rule for structural types. Structural types simply expand to their defined types. The expansion of selection types (E-Sel) are slightly more complicated.

$$\boxed{A; \Sigma; \Gamma \vdash S <: T}$$

$$\frac{(S <: T) \in A}{A; \Sigma; \Gamma \vdash S \texttt{<:} T} \quad \text{(S-Assume)} \qquad \frac{A; \Sigma; \Gamma, z : \{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 <: \ [z \trianglelefteq \{z \Rightarrow \overline{\sigma}_2\}/z]\overline{\sigma}_2}{A; \Sigma; \Gamma \vdash \{z \Rightarrow \overline{\sigma}_1\} \ <: \ \{z \Rightarrow \overline{\sigma}_2\}} \quad \text{(S-Struct)}$$

$$\frac{p_1 \equiv p_2 \qquad A; \Sigma; \Gamma \vdash p_1 \ni \texttt{type}\ L : S_1..U_1 \quad A; \Sigma; \Gamma \vdash p_2 \ni \texttt{type}\ L : S_2..U_2 \qquad A, (p_1.L <: p_2.L); \Sigma; \Gamma \vdash S_2 <: \ S_1 \qquad A, (p_1.L <: p_2.L); \Sigma; \Gamma \vdash U_1 \ <: \ U_2}{A; \Sigma; \Gamma \vdash p_1.L \ <: \ p_2.L} \quad \text{(S-Path)}$$

$$\frac{A; \Sigma; \Gamma \vdash p \ni \texttt{type}\ L : S..U \qquad A; \Sigma; \Gamma \vdash S <: U \qquad A; \Sigma; \Gamma \vdash U <: T}{A; \Sigma; \Gamma \vdash p.L \ <: \ T} \quad \text{(S-Select-Upper)}$$

$$\frac{A; \Sigma; \Gamma \vdash p \ni \texttt{type}\ L : S..U \qquad A; \Sigma; \Gamma \vdash S <: U \qquad A; \Sigma; \Gamma \vdash T <: S}{A; \Sigma; \Gamma \vdash T \ <: \ p.L} \quad \text{(S-Select-Lower)}$$

$$A; \Sigma; \Gamma \vdash T \texttt{<:} \top \quad \text{(S-Top)} \qquad A; \Sigma; \Gamma \vdash \bot \texttt{<:} T \quad \text{(S-Bottom)}$$

$$\boxed{A; \Sigma; \Gamma \vdash \sigma <: \sigma'}$$

$$A; \Sigma; \Gamma \vdash \texttt{val}\ f : T <: \texttt{val}\ f : T \quad \text{(S-Decl-Val)} \qquad \frac{A; \Sigma; \Gamma \vdash S' <: S \qquad A; \Sigma; \Gamma \vdash T <: T'}{A; \Sigma; \Gamma \vdash \texttt{def}\ m : S \to T <: \texttt{def}\ m : S' \to T'} \quad \text{(S-Decl-Def)}$$

$$\frac{A; \Sigma; \Gamma \vdash S' <: S \qquad A; \Sigma; \Gamma \vdash U <: U'}{A; \Sigma; \Gamma \vdash \texttt{type}\ L : S..U \ <: \ \texttt{type}\ L : S'..U'} \quad \text{(S-Decl-Type)}$$

Fig. 5. Subtyping

Selection types expand to the expansion of the upper bound. The expansion of the upper bound is type checked with a less precise type (the upper bound) that the selection type. As with our subtyping rule for structural types, we need to prevent narrowing of types at type expansion. For this reason we upcast the self variable z to the original type U. The top type expands to the empty set (E-Top). The membership judgement is used to determine the membership of a declaration type for an expression. We type check the expression, expand the expression's type and in the case of path expansion, substitute out the self variable.

Expression Typing Expression typing is fairly straight forward and is given in Figure **??**. Variables (T-Var) are typed with their types in the environment and locations (T-Loc)in the store type. New expressions (T-New) are typed as a collection of declaration types that correspond to their declarations. Method calls (T-Meth) are typed as their return type provided the arguments subtype the method parameter types. Field accesses (T-Field) are typed as the field type for the receiver. Upcasts (T-Type) are typed as the upcast type if the upcast expression appropriately subtypes the upcast type.

Reduction Reduction is given in Figure **??**. Reduction is new (R-New), method (R-meth), field (R-Field) and context (R-Context) reduction. New expressions are reduced (R-New) once field initializers are reduced to values. Method calls on values with a value type parameter are reduced (R-Meth) using the $Leadsto_m$ judgement. We use the $Leadsto_m$ judgement to extract the method body from the store, and wrap it up in well-formed type layers. Field reduction (R-Field) is performed the same way, using the $Leadsto_f$ judgement. Context reduction (R-Context) is standard.

3 Proving Type Soundness

Our type system makes several modifications based on previous work [**?**]. As discussed before,

$$\boxed{A; \Sigma; \Gamma \vdash T \ \mathbf{wf}}$$

$$\frac{A; \Sigma; \Gamma \vdash p \ni \texttt{type}\ L : S..U \qquad A; \Sigma; \Gamma \vdash \texttt{type}\ L : S..U \ \mathbf{wf}}{A; \Sigma; \Gamma \vdash p.L \ \mathbf{wf}} \ \text{(WF-Sel)}$$

$$\frac{A; \Sigma; \Gamma, z : \{z \Rightarrow \overline{\sigma}\} \vdash \overline{\sigma} \ \mathbf{wf} \qquad \forall j \neq i,\ dom(\sigma_j) \neq dom(\sigma_i)}{A; \Gamma; \Sigma \vdash \{z \Rightarrow \overline{\sigma}\} \ \mathbf{wf}} \ \text{(WF-Struct)} \qquad A; \Sigma; \Gamma \vdash \top \ \mathbf{wf} \ \ \text{(WF-Top)}$$

$$A; \Sigma; \Gamma \vdash \bot \ \mathbf{wf} \ \ \text{(WF-Bot)}$$

$$\boxed{A; \Sigma; \Gamma \vdash \sigma \ \mathbf{wf}}$$

$$\frac{A; \Sigma; \Gamma \vdash T : \mathbf{wf}}{A; \Sigma; \Gamma \vdash \texttt{val}\ f : T \ \mathbf{wf}} \ \text{(WF-Val)} \qquad \frac{A; \Sigma; \Gamma \vdash T : \mathbf{wf} \qquad A; \Sigma; \Gamma \vdash S : \mathbf{wf}}{A; \Sigma; \Gamma \vdash \texttt{def}\ m : S \to T \ \mathbf{wf}} \ \text{(WF-Def)}$$

$$\frac{A; \Sigma; \Gamma \vdash S : \mathbf{wfe} \ \vee \ S = \bot \qquad A; \Sigma; \Gamma \vdash U : \mathbf{wfe} \qquad A; \Sigma; \Gamma \vdash S <: U}{A; \Sigma; \Gamma \vdash \texttt{type}\ L : S..U \ \mathbf{wf}} \ \text{(WF-Type)}$$

$$\boxed{A; \Sigma \vdash \Gamma \ \mathbf{wf}}$$

$$\frac{\forall x \in dom(\Gamma), A; \Sigma; \Gamma \vdash \Gamma(x) \ \mathbf{wf}}{\Sigma \vdash \Gamma \ \mathbf{wf}} \ \text{(WF-Environment)}$$

$$\boxed{\Sigma \ \mathbf{wf}}$$

$$\frac{\forall l \in dom(\Sigma), \varnothing; \Sigma; \varnothing \vdash \Sigma(l) \ \mathbf{wf}}{\Sigma \ \mathbf{wf}} \ \text{(WF-Store-Context)}$$

$$\frac{\forall l \in dom(\mu), \varnothing; \Sigma; \varnothing \vdash \mu(l) : \Sigma(l)}{\Sigma \vdash \mu \ \mathbf{wf}} \ \text{(WF-Store)}$$

Fig. 6. Well-Formedness

4 Future Work

As part of the development of Generic Wyvern, we simplified the concept of paths in the same way that Amin et. al. [**?**] did. This was due to the complexity of both the path equality and narrowing problems. In the future we hope to expand the notion of paths to include field accesses. We also intend to introduce *Intersection* and *Union* types.

5 Conclusion

We have developed and proven sound a small-step semantics for type members in structurally type languages.

$\boxed{A; \Sigma; \Gamma \vdash T \ \mathbf{wfe}}$

$$\frac{A; \Sigma; \Gamma \vdash T \ \mathbf{wf} \qquad A; \Sigma; \Gamma \vdash T \prec \overline{\sigma}}{A; \Sigma; \Gamma \vdash T \ \mathbf{wfe}} \quad \text{(WFE)}$$

Fig. 7. Well-Formed and Expanding Types

$\boxed{A; \Sigma; \Gamma \vdash T \prec \overline{\sigma}}$

$$A; \Sigma; \Gamma \vdash \{z \Rightarrow \overline{\sigma}\} \prec_z \overline{\sigma} \quad \text{(E-Struct)} \qquad \frac{A; \Sigma; \Gamma \vdash p \ni \texttt{type}\ L : S..U \qquad A; \Sigma; \Gamma \vdash U \prec_z \overline{\sigma}}{A; \Sigma; \Gamma \vdash p.L \prec_z [z \trianglelefteq U/z]\overline{\sigma}} \quad \text{(E-Sel)}$$

$$A; \Sigma; \Gamma \vdash \top \prec_z \varnothing \quad \text{(E-Top)}$$

Fig. 8. Expansion

$\boxed{A; \Sigma; \Gamma \vdash e \ni \sigma}$

$$\frac{A; \Sigma; \Gamma \vdash p : T \qquad A; \Sigma; \Gamma \vdash T \prec_z \overline{\sigma} \qquad A; \sigma_i \in \overline{\sigma}}{A; \Sigma; \Gamma \vdash p \ni [p/z]\sigma_i} \quad \text{(M-Path)}$$

$$\frac{A; \Sigma; \Gamma \vdash e : T \qquad A; \Sigma; \Gamma \vdash T \prec_z \overline{\sigma} \qquad \sigma_i \in \overline{\sigma} \qquad z \notin \sigma_i}{A; \Sigma; \Gamma \vdash e \ni \sigma_i} \quad \text{(M-Exp)}$$

Fig. 9. Membership

$\boxed{A; \Sigma; \Gamma \vdash e : T}$

$$\frac{x \in dom(\Gamma)}{A; \Sigma; \Gamma \vdash x : \Gamma(x)} \quad \text{(T-Var)} \qquad \frac{l \in dom(\Sigma)}{A; \Sigma; \Gamma \vdash l : \Sigma(l)} \quad \text{(T-Loc)} \qquad \frac{A; \Sigma; \Gamma, z : \{z \Rightarrow \overline{\sigma}\} \vdash \overline{d} : \overline{\sigma}}{A; \Sigma; \Gamma \vdash \texttt{new}\ \{z \Rightarrow \overline{d}\} : \{z \Rightarrow \overline{\sigma}\}} \quad \text{(T-New)}$$

$$\frac{A; \Sigma; \Gamma \vdash e_0 \ni \texttt{def}\ m : S \to T \qquad A; \Sigma; \Gamma \vdash e_0 : T_0 \qquad A; \Sigma; \Gamma \vdash e_1 : S' \qquad A; \Sigma; \Gamma \vdash S' <: S}{A; \Sigma; \Gamma \vdash e_0.m(e_1) : T} \quad \text{(T-Meth)}$$

$$\frac{A; \Sigma; \Gamma \vdash e : S \qquad A; \Sigma; \Gamma \vdash e \ni \texttt{val}\ f : T}{A; \Sigma; \Gamma \vdash e.f : T} \quad \text{(T-Field)} \qquad \frac{A; \Sigma; \Gamma \vdash e : S \qquad A; \Sigma; \Gamma \vdash S <: T}{A; \Sigma; \Gamma \vdash e \trianglelefteq T : T} \quad \text{(T-Type)}$$

Fig. 10. Expression Typing

$\boxed{\mu \mid e \ \rightarrow \mu' \mid e'}$

$$\frac{l \notin dom(\mu) \qquad \mu' = \mu, l \mapsto \{\mathtt{z} \Rightarrow \overline{d_v}\}}{\mu \mid \texttt{new}\ \{\mathtt{z} \Rightarrow \overline{d_v}\} \ \rightarrow \mu' \mid l} \quad \text{(R-New)} \qquad \frac{\mu : \Sigma \qquad \mu; \Sigma \vdash v_1 \rightsquigarrow_{m(v_2)} e}{\mu \mid v_1.m(v_2) \ \rightarrow \mu \mid e} \quad \text{(R-Meth)}$$

$$\frac{\mu : \Sigma \qquad \mu; \Sigma \vdash v_1 \rightsquigarrow_f v_2}{\mu \mid v_1.f \ \rightarrow \mu \mid v_2} \quad \text{(R-Field)} \qquad \frac{\mu \mid e \ \rightarrow \ \mu' \mid e'}{\mu \mid E[e] \ \rightarrow \mu' \mid E[e']} \quad \text{(R-Context)}$$

Fig. 11. Reduction