PRELIMINARY DRAFT

Syntax

Terms and types. Note that we allow types to be optional in certain positions (currently function arguments and return types, and on variable declarations). Implicitly these are either inferred or filled in with dynamic.

There are explicit terms for dynamic calls and loads, and for dynamic type checks.

Fields can only be read or set within a method via a reference to this, so no dynamic set operation is required (essentially dynamic set becomes a dynamic call to a setter). This just simplifies the presentation a bit. Methods may be externally loaded from the object (either to call them, or to pass them as closurized functions).

Type identifiers	::=	$C, G, T, S, \ldots$
Arrow kind (k)	::=	$+, -$
Types τ, σ	::=	T | **dynamic** | **Object** | **Null** | **Type** | **num** | **bool** | $\overrightarrow{\tau} \xrightarrow{k} \sigma$ | $C\!<\!\overrightarrow{\tau}\!>$
Optional type ($[\tau]$)	::=	$\_$ | τ
Term identifiers	::=	$a, b, x, y, m, n, \ldots$
Primops (ϕ)	::=	$+, - \ldots \| \| \ldots$
Expressions e	::=	x | i | tt | ff | **null** | **this** | $(\overrightarrow{x : [\tau]}) : [\sigma] \Rightarrow s$ | **new** $C\!<\!\overrightarrow{\tau}\!>()$ | **op**$(\overrightarrow{e})$ | $e(\overrightarrow{e})$ | **dcall**$(e, \overrightarrow{e})$ | $e.m$ | **dload**(e, m) | **this**$.x$ | $x = e$ | **this**$.x = e$ | **throw** | e **as** τ | e **is** τ | **check**(e, τ)
Declaration (vd)	::=	**var** $x : [\tau] = e$ | $f(\overrightarrow{x : \tau}) : \tau = s$
Statements (s)	::=	vd | e | **if** (e) **then** s_1 **else** s_2 | **return** e | $s; s$
Class decl (cd)	::=	**class** $C\!<\!\overrightarrow{T}\!>$ **extends** $G\!<\!\overrightarrow{\tau}\!>$ $\{\overrightarrow{vd}\}$
Toplevel decl (td)	::=	vd | cd
Program (P)	::=	**let** $\overrightarrow{td}$ **in** s

Type contexts map type variables to their bounds.

Class signatures describe the methods and fields in an object, along with the super class of the class. There are no static methods or fields.

The class hierararchy records the classes with their signatures.

The term context maps term variables to their types. I also abuse notation and allow for the attachment of an optional type to term contexts as follows: Γ_σ refers to a term context within the body of a method whose class type is σ.

$$
\begin{array}{lll}
\text{Type context } (\Delta) & ::= & \epsilon \mid \Delta, T <: \tau \\
\text{Class element } (ce) & ::= & \textbf{var } x : \tau \mid \textbf{fun } f : \sigma \\
\text{Class signature } (Sig) & ::= & \textbf{class } C{<}\overrightarrow{T}{>} \textbf{ extends } G{<}\overrightarrow{\tau}{>} \{\overrightarrow{ce}\} \\
\text{Class hierarchy } (\Phi) & ::= & \epsilon \mid \Phi, C \ : \ Sig \\
\text{Term context } (\Gamma) & ::= & \epsilon \mid \Gamma, x \ : \ \tau
\end{array}
$$

Subtyping

Variant Subtyping

We include a special kind of covariant function space to model certain dart idioms. An arrow type decorated with a positive variance annotation (+) treats **dynamic** in its argument list covariantly: or equivalently, it treats **dynamic** as bottom. This variant subtyping relation captures this special treatment of dynamic.

$$\frac{}{\Phi, \Delta \vdash \ \textbf{dynamic} <:^{+} \tau}$$

$$\frac{\Phi, \Delta \vdash \ \sigma <: \tau \quad \sigma \neq \textbf{dynamic}}{\Phi, \Delta \vdash \ \sigma <:^{+} \tau}$$

$$\frac{\Phi, \Delta \vdash \ \sigma <: \tau}{\Phi, \Delta \vdash \ \sigma <:^{-} \tau}$$

Invariant Subtyping

Regular subtyping is defined in a fairly standard way, except that generics are uniformly covariant, and that function argument types fall into the variant subtyping relation defined above.

$$\frac{}{\Phi, \Delta \vdash \ \tau <: \textbf{dynamic}}$$

$$\frac{}{\Phi, \Delta \vdash \ \tau <: \textbf{Object}}$$

$$\frac{}{\Phi, \Delta \vdash \ \textbf{bottom} <: \tau}$$

$$\frac{}{\Phi, \Delta \vdash \ \tau <: \tau}$$

$$\frac{(S \ : \ \sigma) \in \Delta \quad \Phi, \Delta \vdash \ \sigma <: \tau}{\Phi, \Delta \vdash \ S <: \tau}$$

$$\frac{\Phi, \Delta \vdash \ \sigma_i <:^{k_1} \tau_i \quad i \in 0, \ldots, n \qquad \Phi, \Delta \vdash \ \tau_r <: \sigma_r \qquad (k_0 = \text{-}) \vee (k_1 = +)}{\Phi, \Delta \vdash \ \tau_0, \ldots, \tau_n \xrightarrow{k_0} \tau_r <: \sigma_0, \ldots, \sigma_n \xrightarrow{k_1} \sigma_r}$$

$$\frac{\Phi, \Delta \vdash \ \tau_i <: \sigma_i \quad i \in 0, \ldots, n}{\Phi, \Delta \vdash \ C\!<\!\tau_0, \ldots, \tau_n\!> <: C\!<\!\sigma_0, \ldots, \sigma_n\!>}$$

$$\frac{(C : \textbf{class } C\!<\!T_0, \ldots, T_n\!> \textbf{ extends } C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> \{\ldots\}) \in \Phi \qquad \Phi, \Delta \vdash \ [\tau_0, \ldots, \tau_n / T_0, \ldots, T_n] C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> <: G\!<\!\sigma_0, \ldots, \sigma_m\!>}{\Phi, \Delta \vdash \ C\!<\!\tau_0, \ldots, \tau_n\!> <: G\!<\!\sigma_0, \ldots, \sigma_m\!>}$$

Typing

Field lookup

$$\frac{(C : \textbf{class } C\!<\!T_0, \ldots, T_n\!> \textbf{ extends } C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \qquad \textbf{var } x : \tau \in \overrightarrow{c\acute{e}}}{\Phi \vdash C\!<\!\tau_0, \ldots, \tau_n\!>.x \ \rightsquigarrow_f \ [\tau_0, \ldots, \tau_n / T_0, \ldots, T_n]\tau}$$

$$\frac{(C : \textbf{class } C\!<\!T_0, \ldots, T_n\!> \textbf{ extends } C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \quad x \notin \overrightarrow{c\acute{e}} \qquad \Phi \vdash C'\!<\!\upsilon_0, \ldots, \upsilon_k\!>.x \ \rightsquigarrow_f \ \tau}{\Phi \vdash C\!<\!\tau_0, \ldots, \tau_n\!>.x \ \rightsquigarrow_f \ [\tau_0, \ldots, \tau_n / T_0, \ldots, T_n]\tau}$$

Method lookup

$$\frac{(C : \textbf{class } C\!<\!T_0, \ldots, T_n\!> \textbf{ extends } C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \qquad \textbf{fun } m : \sigma \in \overrightarrow{c\acute{e}}}{\Phi \vdash C\!<\!\tau_0, \ldots, \tau_n\!>.m \ \rightsquigarrow_m \ [\tau_0, \ldots, \tau_n / T_0, \ldots, T_n]\sigma}$$

$$\frac{(C : \textbf{class } C\!<\!T_0, \ldots, T_n\!> \textbf{ extends } C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \quad m \notin \overrightarrow{c\acute{e}} \qquad \Phi \vdash C'\!<\!\upsilon_0, \ldots, \upsilon_k\!>.m \ \rightsquigarrow_m \ \sigma}{\Phi \vdash C\!<\!\tau_0, \ldots, \tau_n\!>.m \ \rightsquigarrow_m \ [\tau_0, \ldots, \tau_n / T_0, \ldots, T_n]\sigma}$$

Method and field absence

$$\frac{(C : \textbf{class } C\!<\!T_0, \ldots, T_n\!> \textbf{ extends } C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \quad x \notin \overrightarrow{c\acute{e}} \quad \Phi \vdash x \notin C'\!<\!\upsilon_0, \ldots, \upsilon_k\!>}{\Phi \vdash x \notin C\!<\!\tau_0, \ldots, \tau_n\!>}$$

$$\frac{(C : \textbf{class } C\!<\!T_0, \ldots, T_n\!> \textbf{ extends } C'\!<\!\upsilon_0, \ldots, \upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \quad m \notin \overrightarrow{c\acute{e}} \quad \Phi \vdash m \notin C'\!<\!\upsilon_0, \ldots, \upsilon_k\!>\tau\sigma}{\Phi \vdash m \notin C\!<\!\tau_0, \ldots, \tau_n\!>}$$

Expression typing: $\Phi, \Delta, \Gamma \vdash e \ : \ [\tau] \Uparrow \tau'$

Expression typing is a relation between typing contexts, a term (e), an optional type ($[\tau]$), and a type (τ'). The general idea is that we are typechecking a term (e) and want to know if it is well-typed. The term appears in a context, which may (or may not) impose a type constraint on the term. For example, in **var** $x : \tau \ = \ e$, e appears in a context which requires it to be a subtype of τ, or to be coercable to τ. Alternatively if e appears as in **var** $x : \_ \ = \ e$, then the context does not provide a type constraint on e. This "contextual" type information is both a constraint on the term, and may also provide a source of information for type inference in e. The optional type $[\tau]$ in the typing relation corresponds to this contextual type information. Viewing the relation algorithmically, this should be viewed as an input to the algorithm, along with the term. The process of checking a term allows us to synthesize a precise type for the term e which may be more precise than the type required by the context. The type τ' in the relation represents this more precise, synthesized type. This type should be thought of as an output of the algorithm. It should always be the case that the synthesized (output) type is a subtype of the checked (input) type if the latter is present. The checking/synthesis pattern allows for the propagation of type information both downwards and upwards.

It is often the case that downwards propagation is not useful. Consequently, to simplify the presentation the rules which do not use the checking type require that it be empty ($\_$). This does not mean that such terms cannot be checked when contextual type information is supplied: the first typing rule allows contextual type information to be dropped so that such rules apply in the case that we have contextual type information, subject to the contextual type being a supertype of the synthesized type:

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow \sigma \qquad \Phi, \Delta \vdash \ \sigma <: \tau}{\Phi, \Delta, \Gamma \vdash e \ : \ \tau \Uparrow \sigma}$$

The implicit downcast rule also allows this when the contextual type is a subtype of the synthesized type, corresponding to an implicit downcast.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow \sigma \qquad \Phi, \Delta \vdash \ \tau <: \sigma}{\Phi, \Delta, \Gamma \vdash e \ : \ \tau \Uparrow \tau}$$

Variables are typed according to their declarations:

$$\frac{}{\Phi, \Delta, \Gamma[x \ : \ \tau] \vdash x \ : \ \_ \Uparrow \tau}$$

Numbers, booleans, and null all have a fixed synthesized type.

$$\frac{}{\Phi, \Delta, \Gamma \vdash i \ : \ \_ \Uparrow \textbf{num}}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash \text{ff} \ : \ \_ \Uparrow \textbf{bool}}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash \text{tt} \ : \ \_ \Uparrow \textbf{bool}}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash \textbf{null} \ : \ \_ \Uparrow \textbf{bottom}}$$

A **this** expression is well-typed if we are inside of a method, and σ is the type of the enclosing class.

$$\frac{\Gamma = \Gamma'_{\sigma}}{\Phi, \Delta, \Gamma \vdash \textbf{this} \ : \ \_ \Uparrow \sigma}$$

A fully annotated function is well-typed if its body is well-typed at its declared return type, under the assumption that the variables have their declared types.

$$\frac{\Gamma' = \Gamma[\overrightarrow{x} \ : \ \overrightarrow{\tau}] \qquad \Phi, \Delta, \Gamma' \vdash s \ : \ \sigma \Uparrow \Gamma'}{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : \tau}) : \sigma \Rightarrow s \ : \ \_ \Uparrow \overrightarrow{\tau} \bar{\rightarrow} \sigma}$$

A function with a missing argument type is well-typed if it is well-typed with the argument type replaced with **dynamic**.

$$\frac{\Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \mathbf{dynamic}, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ [\tau] \Uparrow \tau_f}{\Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \_, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ [\tau] \Uparrow \tau_f}$$

A function with a missing argument type is well-typed if it is well-typed with the argument type replaced with the corresponding argument type from the context type. Note that this rule overlaps with the previous: the formal presentation leaves this as a non-deterministic choice.

$$\frac{\begin{array}{c}\tau_c = \upsilon_0, \ldots, \upsilon_n \xrightarrow{k} \upsilon_r \\ \Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \upsilon_i, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ \tau_c \Uparrow \tau_f\end{array}}{\Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \_, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ \tau_c \Uparrow \tau_f}$$

A function with a missing return type is well-typed if it is well-typed with the return type replaced with **dynamic**.

$$\frac{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \mathbf{dynamic} \Rightarrow s \ : \ [\tau_c] \Uparrow \tau_f}{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \_ \Rightarrow s \ : \ [\tau_c] \Uparrow \tau_f}$$

A function with a missing return type is well-typed if it is well-typed with the return type replaced with the corresponding return type from the context type. Note that this rule overlaps with the previous: the formal presentation leaves this as a non-deterministic choice.

$$\frac{\begin{array}{c}\tau_c = \upsilon_0, \ldots, \upsilon_n \xrightarrow{k} \upsilon_r \\ \Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \upsilon_r \Rightarrow s \ : \ \tau_c \Uparrow \tau_f\end{array}}{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \_ \Rightarrow s \ : \ \tau_c \Uparrow \tau_f}$$

Instance creation creates an instance of the appropriate type.

$$\frac{\begin{array}{c}(C : \mathbf{class}\ C\!<\!T_0, \ldots, T_n\!>\ \mathbf{extends}\ C'\!<\!\upsilon_0, \ldots, \upsilon_k\!>\ \{\ldots\}) \in \Phi \\ \mathrm{len}(\overrightarrow{\tau}) = n + 1\end{array}}{\Phi, \Delta, \Gamma \vdash \mathbf{new}\, C\!<\!\overrightarrow{\tau}\!>() \ : \ \_ \Uparrow C\!<\!\overrightarrow{\tau}\!>}$$

Members of the set of primitive operations (left unspecified) can only be applied. Applications of primitives are well-typed if the arguments are well-typed at the types given by the signature of the primitive.

$$\frac{\mathbf{op} \ : \ \overrightarrow{\tau} \to \sigma \qquad \Phi, \Delta, \Gamma \vdash e \ : \ \tau \Uparrow \tau'}{\Phi, \Delta, \Gamma \vdash \mathbf{op}(\overrightarrow{e}) \ : \ \_ \Uparrow \sigma}$$

Function applications are well-typed if the applicand is well-typed and has function type, and the arguments are well-typed.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow \overrightarrow{\tau_a} \xrightarrow{k} \tau_r \qquad \Phi, \Delta, \Gamma \vdash e_a \ : \ \tau_a \Uparrow \tau'_a \quad \text{for } e_a, \tau_a \in \overrightarrow{e_a}, \overrightarrow{\tau_a}}{\Phi, \Delta, \Gamma \vdash e(\overrightarrow{e_a}) \ : \ \_ \Uparrow \tau_r}$$

Application of an expression of type **dynamic** is well-typed if the arguments are well-typed at any type.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow \textbf{dynamic} \qquad \Phi, \Delta, \Gamma \vdash e_a \ : \ \_ \Uparrow \tau'_a \quad \text{for } e_a \in \overrightarrow{e_a}}{\Phi, \Delta, \Gamma \vdash e(\overrightarrow{e_a}) \ : \ \_ \Uparrow \textbf{dynamic}}$$

A method load is well-typed if the term is well-typed, and the method name is present in the type of the term.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow \sigma \qquad \Phi \vdash \sigma.m \rightsquigarrow_m \tau}{\Phi, \Delta, \Gamma \vdash e.m \ : \ \_ \Uparrow \tau}$$

A method load from a term of type **dynamic** is well-typed if the term is well-typed.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \textbf{dynamic} \Uparrow \tau}{\Phi, \Delta, \Gamma \vdash e.m \ : \ \_ \Uparrow \textbf{dynamic}}$$

A field load from **this** is well-typed if the field name is present in the type of **this**.

$$\frac{\Gamma = \Gamma_\tau \quad \Phi \vdash \tau.x \rightsquigarrow_f \sigma}{\Phi, \Delta, \Gamma \vdash \textbf{this}.x \ : \ \_ \Uparrow \sigma}$$

An assignment expression is well-typed so long as the term is well-typed at a type which is compatible with the type of the variable being assigned.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ [\tau] \Uparrow \sigma \qquad \Phi, \Delta, \Gamma \vdash x \ : \ \sigma \Uparrow \sigma'}{\Phi, \Delta, \Gamma \vdash x = e \ : \ [\tau] \Uparrow \sigma}$$

A field assignment is well-typed if the term being assigned is well-typed, the field name is present in the type of **this**, and the declared type of the field is compatible with the type of the expression being assigned.

$$\frac{\Gamma = \Gamma_\tau \qquad \Phi, \Delta, \Gamma \vdash e : [\tau] \Uparrow \sigma \qquad \Phi \vdash \tau.x \rightsquigarrow_f \sigma' \qquad \Phi, \Delta \vdash \sigma <: \sigma'}{\Phi, \Delta, \Gamma \vdash \textbf{this}.x = e : \_ \Uparrow \sigma}$$

A throw expression is well-typed at any type.

$$\frac{}{\Phi, \Delta, \Gamma \vdash \textbf{throw} : \_ \Uparrow \sigma}$$

A cast expression is well-typed so long as the term being cast is well-typed. The synthesized type is the cast-to type. We require that the cast-to type be a ground type.

$$\frac{\Phi, \Delta, \Gamma \vdash e : \_ \Uparrow \sigma \qquad \tau \text{ is ground}}{\Phi, \Delta, \Gamma \vdash e \textbf{ as } \tau : \_ \Uparrow \tau}$$

An instance check expression is well-typed if the term being checked is well-typed. We require that the cast to-type be a ground type.

$$\frac{\Phi, \Delta, \Gamma \vdash e : \_ \Uparrow \sigma \qquad \tau \text{ is ground}}{\Phi, \Delta, \Gamma \vdash e \textbf{ is } \tau : \_ \Uparrow \textbf{bool}}$$

Declaration typing: $\Phi, \Delta, \Gamma \vdash_d vd \Uparrow \Gamma'$

Variable declaration typing checks the well-formedness of the components, and produces an output context Γ' which contains the binding introduced by the declaration.

A simple variable declaration with a declared type is well-typed if the initializer for the declaration is well-typed at the declared type. The output context binds the variable at the declared type.

$$\frac{\Phi, \Delta, \Gamma \vdash e : \tau \Uparrow \tau'}{\Phi, \Delta, \Gamma \vdash_d \textbf{var } x : \tau \ = \ e \Uparrow \Gamma[x \ : \ \tau]}$$

A simple variable declaration without a declared type is well-typed if the initializer for the declaration is well-typed at any type. The output context binds the variable at the synthesized type (a simple form of type inference).

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow \tau'}{\Phi, \Delta, \Gamma \vdash_d \textbf{var}\ x : \_ \ = \ e \Uparrow \Gamma[x \ : \ \tau']}$$

A function declaration is well-typed if the body of the function is well-typed with the given return type, under the assumption that the function and its parameters have their declared types. The function is assumed to have a contravariant (precise) function type. The output context binds the function variable only.

$$\frac{\tau_f = \overrightarrow{\tau_a} \stackrel{-}{\rightarrow} \tau_r \qquad \Gamma' = \Gamma[f \ : \ \tau_f] \qquad \Gamma'' = \Gamma'[\overrightarrow{x} \ : \ \overrightarrow{\tau_a}] \qquad \Phi, \Delta, \Gamma'' \vdash s \ : \ \tau_r \Uparrow \Gamma_0}{\Phi, \Delta, \Gamma \vdash_d f(\overrightarrow{x : \tau_a}) : \tau_r = s \Uparrow \Gamma'}$$

Statement typing: $\Phi, \Delta, \Gamma \vdash s \ : \ \tau \Uparrow \Gamma'$

The statement typing relation checks the well-formedness of statements and produces an output context which reflects any additional variable bindings introduced into scope by the statements.

A variable declaration statement is well-typed if the variable declaration is well-typed per the previous relation, with the corresponding output context.

$$\frac{\Phi, \Delta, \Gamma \vdash_d vd \Uparrow \Gamma'}{\Phi, \Delta, \Gamma \vdash vd \ : \ \tau \Uparrow \Gamma'}$$

An expression statement is well-typed if the expression is well-typed at any type per the expression typing relation.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow \tau}{\Phi, \Delta, \Gamma \vdash e \ : \ \tau \Uparrow \Gamma}$$

A conditional statement is well-typed if the condition is well-typed as a boolean, and the statements making up the two arms are well-typed. The output context is unchanged.

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \textbf{bool} \Uparrow \sigma \qquad \Phi, \Delta, \Gamma \vdash s_1 \,:\, \tau_r \Uparrow \Gamma_1 \qquad \Phi, \Delta, \Gamma \vdash s_2 \,:\, \tau_r \Uparrow \Gamma_2}{\Phi, \Delta, \Gamma \vdash \textbf{if}\ (e)\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,:\, \tau_r \Uparrow \Gamma}$$

A return statement is well-typed if the expression being returned is well-typed at the given return type.

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \tau_r \Uparrow \tau}{\Phi, \Delta, \Gamma \vdash \textbf{return}\ e \,:\, \tau_r \Uparrow \Gamma}$$

A sequence statement is well-typed if the first component is well-typed, and the second component is well-typed with the output context of the first component as its input context. The final output context is the output context of the second component.

$$\frac{\Phi, \Delta, \Gamma \vdash s_1 \,:\, \tau_r \Uparrow \Gamma' \qquad \Phi, \Delta, \Gamma' \vdash s_2 \,:\, \tau_r \Uparrow \Gamma''}{\Phi, \Delta, \Gamma \vdash s_1; s_2 \,:\, \tau_r \Uparrow \Gamma''}$$

Class member typing: $\Phi, \Delta, \Gamma \vdash_{ce} vd : ce \Uparrow \Gamma'$

A class member is well-typed with a given signature (ce) taken from the class hierarchy if the signature type matches the type on the definition, and if the definition is well-typed.

$$\frac{\Phi, \Delta, \Gamma \vdash_d \textbf{var}\ x : [\tau] \ = \ e \Uparrow \Gamma'}{\Phi, \Delta, \Gamma \vdash_{ce} \textbf{var}\ x : [\tau] \ = \ e : \textbf{var}\ x : [\tau] \Uparrow \Gamma'}$$

$$\frac{\begin{array}{c} vd = f(x_0 : \tau_0, \ldots, x_n : \tau_n) : \tau_r = s \\ \sigma_e = \tau_0, \ldots, \tau_n \xrightarrow{+} \tau_r \\ \Phi, \Delta, \Gamma \vdash_d vd \Uparrow \Gamma' \end{array}}{\Phi, \Delta, \Gamma \vdash_{ce} vd : \textbf{fun}\ f : \sigma_e \Uparrow \Gamma'}$$

Class declaration typing: $\Phi, \Gamma \vdash_c cd \Uparrow \Gamma'$

A class declaration is well-typed with a given signature (*Sig*) taken from the class hierarchy if the signature matches the definition, and if each member of the class is well-typed with the corresponding signature from the class signature. The members are checked with the generic type parameters bound in the type context, and with the type of the current class set as the type of **this** on the term context Γ.

$$\frac{\begin{array}{c} cd = \textbf{class}\ C{<}\overrightarrow{T}{>}\ \textbf{extends}\ G{<}\overrightarrow{\tau}{>}\ \{vd_0, \ldots, vd_n\} \\ (C : \textbf{class}\ C{<}\overrightarrow{T}{>}\ \textbf{extends}\ G{<}\overrightarrow{\tau}{>}\ \{ce_0, \ldots, \overrightarrow{ce}_n\}) \in \Phi \\ \Delta = \overrightarrow{T} \quad \Gamma_c = \Gamma_{C<\overrightarrow{T}>} \\ \Phi, \Delta, \Gamma_c \vdash_{ce} vd_i : ce_i \Uparrow \Gamma' \qquad \text{for } i \in 0, \ldots, n \end{array}}{\Phi, \Gamma \vdash_c cd \Uparrow \Gamma'}$$

Override checking:

$\Phi \vdash C{<}T_0, \ldots, T_n{>}\ \textbf{extends}\ G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow ce$

The override checking relation is the primary relation that checks the consistency of the class hierarchy. We assume a non-cyclic class hierarchy as a syntactic pre-condition. The override check relation checks that in a class declaration $C{<}T_0, \ldots, T_n{>}$ which extends $G{<}\tau_0, \ldots, \tau_k{>}$, the definition of an element with signature ce is valid.

A field with the type elided is a valid override if the same field with type **dynamic** is valid.

$$\frac{\Phi \vdash C{<}T_0, \ldots, T_n{>}\ \textbf{extends}\ G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow \textbf{var}\ x : \textbf{dynamic}}{\Phi \vdash C{<}T_0, \ldots, T_n{>}\ \textbf{extends}\ G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow \textbf{var}\ x : \_}$$

A field with a type τ is a valid override if it appears in the super type with the same type.

$$\frac{\Phi \vdash G{<}\tau_0, \ldots, \tau_k{>}.x \rightsquigarrow_f \tau}{\Phi \vdash C{<}T_0, \ldots, T_n{>}\ \textbf{extends}\ G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow \textbf{var}\ x : \tau}$$

A field with a type τ is a valid override if it does not appear in the super type.

$$\frac{\Phi \vdash x \notin G{<}\tau_0, \ldots, \tau_k{>}}{\Phi \vdash C{<}T_0, \ldots, T_n{>}\ \textbf{extends}\ G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow \textbf{var}\ x : \tau}$$

A method with a type σ is a valid override if it does not appear in the super type.

$$\frac{\Phi \vdash f \notin G{<}\tau_0, \ldots, \tau_k{>}}{\Phi \vdash C{<}T_0, \ldots, T_n{>}\,\textbf{extends}\,G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow \textbf{fun}\ f : \sigma}$$

A method with a type σ is a valid override if it appears in the super type, and σ is a subtype of the type of the method in the super class.

$$\frac{\begin{array}{c}\Phi \vdash G{<}\tau_0, \ldots, \tau_k{>}.f\ \rightsquigarrow_m\ \sigma_s \\ \Phi, \Delta \vdash\ \sigma <: \sigma_s\end{array}}{\Phi \vdash C{<}T_0, \ldots, T_n{>}\,\textbf{extends}\,G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow \textbf{fun}\ f : \sigma}$$

Toplevel declaration typing: $\Phi, \Gamma \vdash_t td \Uparrow \Gamma'$

Top level variable declarations are well-typed if they are well-typed according to their respective specific typing relations.

$$\frac{\Phi, \epsilon, \Gamma \vdash_d vd \Uparrow \Gamma'}{\Phi, \Gamma \vdash_t vd \Uparrow \Gamma'}$$

$$\frac{\Phi, \Gamma \vdash_c cd \Uparrow \Gamma'}{\Phi, \Gamma \vdash_t cd \Uparrow \Gamma'}$$

Well-formed class signature: $\Phi \vdash Sig\ \textbf{ok}$

The well-formed class signature relation checks whether a class signature is well-formed with respect to a given class hierarchy Φ.

The **Object** signature is always well-formed.

$$\frac{}{\Phi \vdash \textbf{Object ok}}$$

A signature for a class C is well-formed if its super-class signature is well-formed, and if every element in its signature is a valid override of the super-class.

$$\frac{\begin{array}{c} Sig = \textbf{class}\ C{<}\overrightarrow{T}{>}\ \textbf{extends}\ G{<}\tau_0,\ldots,\tau_k{>}\ \{ce_0,\ldots,ce_n\} \\ (G : Sig') \in \Phi \qquad \Phi \vdash Sig'\ \textbf{ok} \\ \Phi \vdash C{<}\overrightarrow{T}{>}\,\textbf{extends}\,G{<}\tau_0,\ldots,\tau_k{>} \Leftarrow ce_i \qquad \text{for } ce_i \in ce_0,\ldots,ce_n \end{array}}{\Phi \vdash Sig\ \textbf{ok}}$$

Well-formed class hierarchy: $\vdash \Phi\ \textbf{ok}$

A class hierarchy is well-formed if all of the signatures in it are well-formed with respect to it.

$$\frac{\Phi \vdash Sig\ \textbf{ok} \text{ for } Sig\ \in \Phi}{\vdash \Phi\ \textbf{ok}}$$

Program typing: $\Phi \vdash P$

Program well-formedness is defined with respect to a class hierarchy Φ. It is not specified how Φ is produced, but the well-formedness constraints in the various judgments should constrain it appropriately. A program is well-formed if each of the top level declarations in the program is well-formed in a context in which all of the previous variable declarations have been checked and inserted in the context, and if the body of the program is well-formed in the final context. We allow classes to refer to each other in any order, since Φ is pre-specified, but do not model out of order definitions of top level variables and functions. We assume as a syntactic property that the class hierarchy Φ is acyclic.

$$\frac{\begin{array}{ll} \Gamma_0 = \epsilon & \Phi, \Gamma_i \vdash_t td_i \Uparrow \Gamma_{i+1} \quad \text{for } i \in 0,\ldots,n \\ & \Phi, \epsilon, \Gamma_{n+1} \vdash s : \tau \Uparrow \Gamma'_{n+1} \end{array}}{\Phi \vdash \textbf{let}\ td_0,\ldots,td_n\ \textbf{in}\ s}$$

Elaboration

Elaboration is a type driven translation which maps a source Dart term to a translated term which corresponds to the original term with additional dynamic type checks inserted to reify the static unsoundness as runtime type errors. For the translation, we extend the source language slightly as follows.

$$\text{Expressions } e \quad ::= \quad \ldots \mid \mathbf{dcall}(e, \overrightarrow{e}) \mid \mathbf{dload}(e, m) \mid \mathbf{check}(e, \tau)$$

The expression language is extended with an explicitly checked dynamic call operation, and explicitly checked dynamic method load operation, and a runtime type test. Note that while a user level cast throws an exception on failure, the runtime type test term introduced here produces a hard type error which cannot be caught programmatically.

We also extend typing contexts slightly by adding an internal type to method signatures.

$$\text{Class element } (ce) \quad ::= \quad \mathbf{var}\ x : \tau \mid \mathbf{fun}\ f : \tau \triangleleft \sigma$$

A method signature of the form $\mathbf{fun}\ f : \tau \triangleleft \sigma$ describes a method whose public interface is described by σ, but which has an internal type τ which is a subtype of σ, but which is properly covariant in any type parameters. The elaboration introduces runtime type checks to mediate between the two types. This is discussed further in the translation of classes below.

Field lookup

$$\frac{(C : \mathbf{class}\ C\langle T_0, \ldots, T_n\rangle\ \mathbf{extends}\ C'\langle \upsilon_0, \ldots, \upsilon_k\rangle\ \{\overrightarrow{ce}\}) \in \Phi \quad \mathbf{var}\ x : \tau \in \overrightarrow{ce}}{\Phi \vdash C\langle \tau_0, \ldots, \tau_n\rangle.x \rightsquigarrow_f [\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]\tau}$$

$$\frac{(C : \mathbf{class}\ C\langle T_0, \ldots, T_n\rangle\ \mathbf{extends}\ C'\langle \upsilon_0, \ldots, \upsilon_k\rangle\ \{\overrightarrow{ce}\}) \in \Phi \quad x \notin \overrightarrow{ce} \quad \Phi \vdash C'\langle \upsilon_0, \ldots, \upsilon_k\rangle.x \rightsquigarrow_f \tau}{\Phi \vdash C\langle \tau_0, \ldots, \tau_n\rangle.x \rightsquigarrow_f [\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]\tau}$$

Method lookup

$$\frac{(C : \mathbf{class}\ C\langle T_0, \ldots, T_n\rangle\ \mathbf{extends}\ C'\langle \upsilon_0, \ldots, \upsilon_k\rangle\ \{\overrightarrow{ce}\}) \in \Phi \quad \mathbf{fun}\ m : \tau \triangleleft \sigma \in \overrightarrow{ce}}{\Phi \vdash C\langle \tau_0, \ldots, \tau_n\rangle.m \rightsquigarrow_m [\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]\tau \triangleleft [\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]\sigma}$$

$$\frac{(C : \mathbf{class}\ C\langle T_0, \ldots, T_n\rangle\ \mathbf{extends}\ C'\langle \upsilon_0, \ldots, \upsilon_k\rangle\ \{\overrightarrow{ce}\}) \in \Phi \quad m \notin \overrightarrow{ce} \quad \Phi \vdash C'\langle \upsilon_0, \ldots, \upsilon_k\rangle.m \rightsquigarrow_m \tau \triangleleft \sigma}{\Phi \vdash C\langle \tau_0, \ldots, \tau_n\rangle.m \rightsquigarrow_m [\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]\tau \triangleleft [\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]\sigma}$$

Method and field absence

$$\frac{(C : \textbf{class } C\!<\!T_0,\ldots,T_n\!> \textbf{ extends } C'\!<\!\upsilon_0,\ldots,\upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \quad x \notin \overrightarrow{c\acute{e}} \quad \Phi \vdash x \notin C'\!<\!\upsilon_0,\ldots,\upsilon_k\!>}{\Phi \vdash x \notin C\!<\!\tau_0,\ldots,\tau_n\!>}$$

$$\frac{(C : \textbf{class } C\!<\!T_0,\ldots,T_n\!> \textbf{ extends } C'\!<\!\upsilon_0,\ldots,\upsilon_k\!> \{\overrightarrow{c\acute{e}}\}) \in \Phi \quad m \notin \overrightarrow{c\acute{e}} \quad \Phi \vdash m \notin C'\!<\!\upsilon_0,\ldots,\upsilon_k\!>\tau\sigma}{\Phi \vdash m \notin C\!<\!\tau_0,\ldots,\tau_n\!>}$$

Type translation

To translate covariant generics, we essentially want to treat all contravariant occurrences of type variables as **dynamic**. The type translation $\lfloor\tau\rfloor$ implements this. It is defined in terms of the dual operator $\lceil\tau\rceil$ which translates positive occurences of type variables as **dynamic**.

$$\begin{aligned}
\lfloor T\rfloor &= T \\
\lfloor\tau_0,\ldots,\tau_n \xrightarrow{k} \tau_r\rfloor &= \lceil\tau_0\rceil,\ldots,\lceil\tau_n\rceil \xrightarrow{k} \lfloor\tau_r\rfloor \\
\lfloor C\!<\!\tau_0,\ldots,\tau_n\!>\rfloor &= C\!<\lfloor\tau_0\rfloor,\ldots,\lfloor\tau_n\rfloor> \\
\lfloor\tau\rfloor &= \tau \text{ if } \tau \text{ is base type.}
\end{aligned}$$

$$\begin{aligned}
\lceil T\rceil &= \textbf{dynamic} \\
\lceil\tau_0,\ldots,\tau_n \xrightarrow{k} \tau_r\rceil &= \lfloor\tau_0\rfloor,\ldots,\lfloor\tau_n\rfloor \xrightarrow{k} \lceil\tau_r\rceil \\
\lceil C\!<\!\tau_0,\ldots,\tau_n\!>\rceil &= C\!<\lceil\tau_0\rceil,\ldots,\lceil\tau_n\rceil> \\
\lceil\tau\rceil &= \tau \text{ if } \tau \text{ is base type.}
\end{aligned}$$

Expression typing: $\Phi, \Delta, \Gamma \vdash e : [\tau] \Uparrow e' : \tau'$

For subsumption, the elaboration of the underlying term carries through.

$$\frac{\Phi, \Delta, \Gamma \vdash e : \_ \Uparrow e' : \sigma \qquad \Phi, \Delta \vdash \sigma <: \tau}{\Phi, \Delta, \Gamma \vdash e : \tau \Uparrow e' : \sigma}$$

In an implicit downcast, the elaboration adds a check so that an error will be thrown if the types do not match at runtime.

$$\frac{\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow e' \ : \ \sigma \qquad \Phi, \Delta \vdash \ \tau <: \sigma}{\Phi, \Delta, \Gamma \vdash e \ : \ \tau \Uparrow \mathbf{check}(e', \tau) \ : \ \tau}$$

$$\frac{}{\Phi, \Delta, \Gamma[x \ : \ \tau] \vdash x \ : \ \_ \Uparrow x \ : \ \tau}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash i \ : \ \_ \Uparrow i \ : \ \mathbf{num}}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash \mathrm{ff} \ : \ \_ \Uparrow \mathrm{ff} \ : \ \mathbf{bool}}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash \mathrm{tt} \ : \ \_ \Uparrow \mathrm{tt} \ : \ \mathbf{bool}}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash \mathbf{null} \ : \ \_ \Uparrow \mathbf{null} \ : \ \mathbf{bottom}}$$

$$\frac{\Gamma = \Gamma'_\sigma}{\Phi, \Delta, \Gamma \vdash \mathbf{this} \ : \ \_ \Uparrow \mathbf{this} \ : \ \sigma}$$

A fully annotated function elaborates to a function with an elaborated body. The rest of the function elaboration rules fill in the reified type using contextual information if present and applicable, or **dynamic** otherwise.

$$\frac{\Gamma' = \Gamma[\overrightarrow{x} \ : \ \overrightarrow{\tau}] \qquad \Phi, \Delta, \Gamma' \vdash s \ : \ \sigma \Uparrow s' \ : \ \Gamma'}{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : \tau}) : \sigma \Rightarrow s \ : \ \_ \Uparrow (\overrightarrow{x : \tau}) : \sigma \Rightarrow s' \ : \ \overrightarrow{\tau} \ \bar{\rightarrow} \ \sigma}$$

$$\frac{\Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \mathbf{dynamic}, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ [\tau] \Uparrow e_f \ : \ \tau_f}{\Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \_, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ [\tau] \Uparrow e_f \ : \ \tau_f}$$

$$\frac{\begin{array}{c}\tau_c = \upsilon_0, \ldots, \upsilon_n \xrightarrow{k} \upsilon_r \\ \Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \upsilon_i, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ \tau_c \Uparrow e_f \ : \ \tau_f\end{array}}{\Phi, \Delta, \Gamma \vdash (x_0 : [\tau_0], \ldots, x_i : \_, \ldots, x_n : [\tau_n]) : [\sigma] \Rightarrow s \ : \ \tau_c \Uparrow e_f \ : \ \tau_f}$$

$$\frac{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \mathbf{dynamic} \Rightarrow s \ : \ [\tau_c] \Uparrow e_f \ : \ \tau_f}{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \_ \Rightarrow s \ : \ [\tau_c] \Uparrow e_f \ : \ \tau_f}$$

$$\frac{\begin{array}{c}\tau_c = \upsilon_0, \ldots, \upsilon_n \xrightarrow{k} \upsilon_r \\ \Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \upsilon_r \Rightarrow s \ : \ \tau_c \Uparrow e_f \ : \ \tau_f\end{array}}{\Phi, \Delta, \Gamma \vdash (\overrightarrow{x : [\tau]}) : \_ \Rightarrow s \ : \ \tau_c \Uparrow e_f \ : \ \tau_f}$$

$$\frac{\begin{array}{c}(C : \mathbf{class}\ C{<}T_0, \ldots, T_n{>}\ \mathbf{extends}\ C'{<}\upsilon_0, \ldots, \upsilon_k{>}\ \{\ldots\}) \in \Phi \\ \mathrm{len}(\overrightarrow{\tau}) = n + 1\end{array}}{\Phi, \Delta, \Gamma \vdash \mathbf{new}\, C{<}\overrightarrow{\tau}{>}() \ : \ \_ \Uparrow \mathbf{new}\, C{<}\overrightarrow{\tau}{>}() \ : \ C{<}\overrightarrow{\tau}{>}}$$

$$\frac{\mathbf{op} \ : \ \overrightarrow{\tau} \to \sigma \qquad \Phi, \Delta, \Gamma \vdash e \ : \ \tau \Uparrow e' \ : \ \tau'}{\Phi, \Delta, \Gamma \vdash \mathbf{op}(\overrightarrow{e}) \ : \ \_ \Uparrow \mathbf{op}(\overrightarrow{e'}) \ : \ \sigma}$$

Function application of an expression of function type elaborates to either a call or a dynamic (checked) call, depending on the variance of the applicand. If the applicand is a covariant (fuzzy) type, then a dynamic call is generated.

$$\frac{\begin{array}{c}\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow e' \ : \ \overrightarrow{\tau_a} \xrightarrow{k} \tau_r \\ \Phi, \Delta, \Gamma \vdash e_a \ : \ \tau_a \Uparrow e'_a \ : \ \tau'_a \quad \text{for } e_a, \tau_a \in \overrightarrow{e_a}, \overrightarrow{\tau_a} \\ e_c = \begin{cases} e'(\overrightarrow{e'_a}) & \text{if } k = - \\ \mathbf{dcall}(e', \overrightarrow{e'_a}) & \text{if } k = + \end{cases}\end{array}}{\Phi, \Delta, \Gamma \vdash e(\overrightarrow{e_a}) \ : \ \_ \Uparrow \ e_c \ : \ \tau_r}$$

Application of an expression of type **dynamic** elaborates to a dynamic call.

$$\frac{\begin{array}{c}\Phi, \Delta, \Gamma \vdash e \ : \ \_ \Uparrow e' \ : \ \mathbf{dynamic} \\ \Phi, \Delta, \Gamma \vdash e_a \ : \ \_ \Uparrow e'_a \ : \ \tau'_a \quad \text{for } e_a \in \overrightarrow{e_a}\end{array}}{\Phi, \Delta, \Gamma \vdash e(\overrightarrow{e_a}) \ : \ \_ \Uparrow \mathbf{dcall}(e', \overrightarrow{e'_a}) \ : \ \mathbf{dynamic}}$$

$$\frac{\begin{array}{c}\Phi, \Delta, \Gamma \vdash e \ : \ \mathbf{dynamic} \Uparrow e' \ : \ \tau \\ \Phi, \Delta, \Gamma \vdash e_a \ : \ \_ \Uparrow e'_a \ : \ \tau_a \quad \text{for } e_a \in \overrightarrow{e_a}\end{array}}{\Phi, \Delta, \Gamma \vdash \mathbf{dcall}(e, \overrightarrow{e_a}) \ : \ \_ \Uparrow \mathbf{dcall}(e', \overrightarrow{e'_a}) \ : \ \mathbf{dynamic}}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \_ \Uparrow e' \,:\, \sigma \qquad \Phi \vdash \sigma.m \rightsquigarrow_m \sigma \triangleleft \tau}{\Phi, \Delta, \Gamma \vdash e.m \,:\, \_ \Uparrow e'.m \,:\, \tau}$$

A method load from a term of type **dynamic** elaborates to a dynamic (checked) load.

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \mathbf{dynamic} \Uparrow e' \,:\, \tau}{\Phi, \Delta, \Gamma \vdash e.m \,:\, \_ \Uparrow \mathbf{dload}(e', m) \,:\, \mathbf{dynamic}}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \mathbf{dynamic} \Uparrow e' \,:\, \tau}{\Phi, \Delta, \Gamma \vdash \mathbf{dload}(e, m) \,:\, \_ \Uparrow \mathbf{dload}(e', m) \,:\, \mathbf{dynamic}}$$

$$\frac{\Gamma = \Gamma_\tau \quad \Phi \vdash \tau.x \rightsquigarrow_f \sigma}{\Phi, \Delta, \Gamma \vdash \mathbf{this}.x \,:\, \_ \Uparrow \mathbf{this}.x \,:\, \sigma}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, [\tau] \Uparrow e' \,:\, \sigma \qquad \Phi, \Delta, \Gamma \vdash x \,:\, \sigma \Uparrow x \,:\, \sigma'}{\Phi, \Delta, \Gamma \vdash x = e \,:\, [\tau] \Uparrow x = e' \,:\, \sigma}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_\tau \qquad \Phi, \Delta, \Gamma \vdash e \,:\, [\tau] \Uparrow e' \,:\, \sigma \\ \Phi \vdash \tau.x \rightsquigarrow_f \sigma' \qquad \Phi, \Delta \vdash \sigma <: \sigma'\end{array}}{\Phi, \Delta, \Gamma \vdash \mathbf{this}.x = e \,:\, \_ \Uparrow \mathbf{this}.x = e \,:\, \sigma}$$

$$\frac{}{\Phi, \Delta, \Gamma \vdash \mathbf{throw} \,:\, \_ \Uparrow \mathbf{throw} \,:\, \sigma}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \_ \Uparrow e' \,:\, \sigma \qquad \tau \text{ is ground}}{\Phi, \Delta, \Gamma \vdash e \;\mathbf{as}\; \tau \,:\, \_ \Uparrow e' \;\mathbf{as}\; \tau \,:\, \tau}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \_ \Uparrow e' \,:\, \sigma \qquad \tau \text{ is ground}}{\Phi, \Delta, \Gamma \vdash e \;\mathbf{is}\; \tau \,:\, \_ \Uparrow e' \;\mathbf{is}\; \tau \,:\, \mathbf{bool}}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e \,:\, \_ \Uparrow e' \,:\, \sigma}{\Phi, \Delta, \Gamma \vdash \mathbf{check}(e, \tau) \,:\, \_ \Uparrow \mathbf{check}(e', \tau) \,:\, \tau}$$

Declaration typing: $\Phi, \Delta, \Gamma \vdash_d vd \Uparrow vd' : \Gamma'$

Elaboration of declarations elaborates the underlying expressions.

$$\frac{\Phi, \Delta, \Gamma \vdash e : \tau \Uparrow e' : \tau'}{\Phi, \Delta, \Gamma \vdash_d \textbf{var}\ x : \tau\ =\ e \Uparrow \textbf{var}\ x : \tau'\ =\ e' : \Gamma[x\ :\ \tau]}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e : \_ \Uparrow e' : \tau'}{\Phi, \Delta, \Gamma \vdash_d \textbf{var}\ x : \_\ =\ e \Uparrow \textbf{var}\ x : \tau'\ =\ e' : \Gamma[x\ :\ \tau']}$$

$$\frac{\tau_f = \overrightarrow{\tau_a} \xrightarrow{-} \tau_r \qquad \Gamma' = \Gamma[f\ :\ \tau_f] \qquad \Gamma'' = \Gamma'[\overrightarrow{x}\ :\ \overrightarrow{\tau_a}] \qquad \Phi, \Delta, \Gamma'' \vdash s : \tau_r \Uparrow s' : \Gamma_0}{\Phi, \Delta, \Gamma \vdash_d f(\overrightarrow{x : \tau_a}) : \tau_r = s \Uparrow f(\overrightarrow{x : \tau_a}) : \tau_r = s' : \Gamma'}$$

Statement typing: $\Phi, \Delta, \Gamma \vdash s : \tau \Uparrow s' : \Gamma'$

Statement elaboration elaborates the underlying expressions.

$$\frac{\Phi, \Delta, \Gamma \vdash_d vd \Uparrow vd' : \Gamma'}{\Phi, \Delta, \Gamma \vdash vd : \tau \Uparrow vd' : \Gamma'}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e : \_ \Uparrow e' : \tau}{\Phi, \Delta, \Gamma \vdash e : \tau \Uparrow e' : \Gamma}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e : \textbf{bool} \Uparrow e' : \sigma \qquad \Phi, \Delta, \Gamma \vdash s_1 : \tau_r \Uparrow s_1' : \Gamma_1 \qquad \Phi, \Delta, \Gamma \vdash s_2 : \tau_r \Uparrow s_2' : \Gamma_2}{\Phi, \Delta, \Gamma \vdash \textbf{if}\ (e)\ \textbf{then}\ s_1\ \textbf{else}\ s_2 : \tau_r \Uparrow \textbf{if}\ (e')\ \textbf{then}\ s_1'\ \textbf{else}\ s_2' : \Gamma}$$

$$\frac{\Phi, \Delta, \Gamma \vdash e : \tau_r \Uparrow e' : \tau}{\Phi, \Delta, \Gamma \vdash \textbf{return}\ e : \tau_r \Uparrow \textbf{return}\ e' : \Gamma}$$

$$\frac{\Phi, \Delta, \Gamma \vdash s_1 : \tau_r \Uparrow s_1' : \Gamma' \qquad \Phi, \Delta, \Gamma' \vdash s_2 : \tau_r \Uparrow s_2' : \Gamma''}{\Phi, \Delta, \Gamma \vdash s_1; s_2 : \tau_r \Uparrow s_1'; s_2' : \Gamma''}$$

Class member typing: $\Phi, \Delta, \Gamma \vdash_{ce} vd : ce \Uparrow vd' : \Gamma'$

Elaborating class members is done with respect to a signature. The field translation simply translates the field as a variable declaration.

$$\frac{\Phi, \Delta, \Gamma \vdash_{d} \textbf{var } x : [\tau] \ = \ e \Uparrow vd' : \Gamma'}{\Phi, \Delta, \Gamma \vdash_{ce} \textbf{var } x : [\tau] \ = \ e : \textbf{var } x : [\tau] \Uparrow vd' : \Gamma'}$$

Translating methods requires introducing guard expressions. The signature provides an internal and an external type for the method. The external type is the original declared type of the method, and is the signature which the method presents to external clients. Because we implement covariant generics, clients may see an instantiation of this signature which will allow them to violate the contract expected by the implementation. To handle this, we rewrite the method to match an internal signature which is in fact covariant in the type parameters. This property is enforced in the override checking relation: from the perspective of this relation, there is simply another internal type which defines how to wrap the method with guards.

The translation insists that the internal and external types be function types of the appropriate arity, and that the external type is equal to the type of the declaration. The declaration is translated using the underlying function definition translation, but is then wrapped with guards to enforce the type contract, producing a valid function of the internal (covariant) type. The original body of the function is wrapped in a lambda function, which is applied using a dynamic call which checks that the arguments (which may have negative occurrences of type variables which are treated as **dynamic** in the internal type) are appropriate for the actual body. The original function returns a type τ_r which may be a super-type of the internal type (since negative occurrences of type variables must be treated as dynamic), and so we insert a check expression to guard against runtime type mismatches here.

This is a very simplistic translation for now. We could choose, in the case that the body returns a lambda, to push the checking down into the lambda (essentially wrapping it in place).

$$\frac{\begin{array}{c} vd = f(x_0 : \tau_0, \ldots, x_n : \tau_n) : \tau_r = s \\ \sigma_e = \tau_0, \ldots, \tau_n \xrightarrow{+} \tau_r \qquad \sigma_i = \upsilon_0, \ldots, \upsilon_n \xrightarrow{-} \upsilon_r \\ \Phi, \Delta, \Gamma \vdash_{d} vd \Uparrow f(x_0 : \tau_0, \ldots, x_n : \tau_n) : \tau_r = s' : \Gamma' \\ e_g = (x_0 : \tau_0, \ldots, x_n : \tau_n) : \tau_r \Rightarrow s' \\ s_g = \textbf{return } (\textbf{check}(\textbf{dcall}(e_g, x_0, \ldots, x_n), \upsilon_r)) \\ vd_g = f(x_0 : \upsilon_0, \ldots, x_n : \upsilon_n) : \upsilon_r = s_g \end{array}}{\Phi, \Delta, \Gamma \vdash_{ce} vd : \textbf{fun } f : \sigma_i \triangleleft \sigma_e \Uparrow vd_g : \Gamma'}$$

Class declaration typing: $\Phi, \Gamma \vdash_c cd \Uparrow cd' : \Gamma'$

Elaboration of a class requires that the class hierarchy Φ have a matching signature for the class declaration. Each class member in the class is elaborated using the corresponding class element from the signature.

$$\frac{\begin{array}{c} cd = \textbf{class } C\overrightarrow{<T>} \textbf{ extends } G\overrightarrow{<\tau>} \{vd_0, \ldots, vd_n\} \\ (C : \textbf{class } C\overrightarrow{<T>} \textbf{ extends } G\overrightarrow{<\tau>} \{ce_0, \ldots, \overrightarrow{ce}_n\}) \in \Phi \\ \Delta = \overrightarrow{T} \quad \Gamma_c = \Gamma_{C\overrightarrow{<T>}} \\ \Phi, \Delta, \Gamma_c \vdash_{ce} vd_i : ce_i \Uparrow vd_i' : \Gamma' \qquad \text{for } i \in 0, \ldots, n \\ cd' = \textbf{class } C\overrightarrow{<T>} \textbf{ extends } G\overrightarrow{<\tau>} \{\overrightarrow{vd'}\} \end{array}}{\Phi, \Gamma \vdash_c cd \Uparrow cd' : \Gamma'}$$

Override checking: $\Phi \vdash C<T_0, \ldots, T_n> \textbf{extends } G<\tau_0, \ldots, \tau_k> \Leftarrow ce$

Override checking remains largely the same, with the exception of additional consistency constraints on the internal signatures for methods.

$$\frac{\Phi \vdash C<T_0, \ldots, T_n> \textbf{extends } G<\tau_0, \ldots, \tau_k> \Leftarrow \textbf{var } x : \textbf{dynamic}}{\Phi \vdash C<T_0, \ldots, T_n> \textbf{extends } G<\tau_0, \ldots, \tau_k> \Leftarrow \textbf{var } x : \_}$$

$$\frac{\Phi \vdash G<\tau_0, \ldots, \tau_k>.x \rightsquigarrow_f \tau}{\Phi \vdash C<T_0, \ldots, T_n> \textbf{extends } G<\tau_0, \ldots, \tau_k> \Leftarrow \textbf{var } x : \tau}$$

$$\frac{\Phi \vdash x \notin G<\tau_0, \ldots, \tau_k>}{\Phi \vdash C<T_0, \ldots, T_n> \textbf{extends } G<\tau_0, \ldots, \tau_k> \Leftarrow \textbf{var } x : \tau}$$

For a non-override method, we require that the internal type τ be a subtype of $\lfloor\sigma\rfloor$ where σ is the declared type. Essentially, this enforces the property that the initial declaration of a method in the hierarchy has a covariant internal type.

$$\frac{\begin{array}{c} \Delta = T_0, \ldots, T_n \qquad \Phi, \Delta \vdash \tau <: \lfloor\sigma\rfloor \\ \Phi \vdash f \notin G<\tau_0, \ldots, \tau_k> \end{array}}{\Phi \vdash C<T_0, \ldots, T_n> \textbf{extends } G<\tau_0, \ldots, \tau_k> \Leftarrow \textbf{fun } f : \tau \triangleleft \sigma}$$

For a method override, we reqire two coherence conditions. As before, we require that the internal type τ be a subtype of the $\lfloor\sigma\rfloor$ where σ is the external type. Moreover, we also insist that the external type σ be a subtype of the external type of the method in the superclass, and that the internal type τ be a subtype of the internal type in the superclass. Note that it this last consistency property that ensures that covariant generics are "poisonous" in the sense that non-generic subclasses of generic classes must still have additional checks. For example, a superclass with a method of external type $\sigma_s = T \bar{\to} T$ will have internal type $\tau_s = \mathbf{dynamic} \bar{\to} T$. A subclass of an instantiation of this class with **num** can validly override this method with one of external type $\sigma = \mathbf{num} \bar{\to} \mathbf{num}$. This is unsound in general since the argument occurrence of T in σ_s is contra-variant. However, the additional consistency requirement is that the internal type of the subclass method must be a subtype of $[\mathbf{num}/T]\tau_s = \mathbf{dynamic} \bar{\to} \mathbf{num}$. This enforces the property that the overriden method must expect to be used at type $\mathbf{dynamic} \bar{\to} \mathbf{num}$, and hence must check its arguments (and potentially its return value as well in the higher-order case). This checking code is inserted during the elaboration of class members above.

$$\frac{\begin{array}{c}\Delta = T_0, \ldots, T_n \qquad \Phi, \Delta \vdash\ \tau <: \lfloor\sigma\rfloor \\ \Phi \vdash G{<}\tau_0, \ldots, \tau_k{>}.f\ \rightsquigarrow_m\ \tau_s \triangleleft \sigma_s \\ \Phi, \Delta \vdash\ \tau <: \tau_s \qquad \Phi, \Delta \vdash\ \sigma <: \sigma_s\end{array}}{\Phi \vdash C{<}T_0, \ldots, T_n{>}\,\mathbf{extends}\,G{<}\tau_0, \ldots, \tau_k{>} \Leftarrow \mathbf{fun}\ f : \tau \triangleleft \sigma}$$

Toplevel declaration typing: $\Phi, \Gamma \vdash_t td \Uparrow td' : \Gamma'$

Top level declaration elaboration falls through to the underlying variable and class declaration code.

$$\frac{\Phi, \epsilon, \Gamma \vdash_d vd \Uparrow vd' : \Gamma'}{\Phi, \Gamma \vdash_t vd \Uparrow vd' : \Gamma'}$$

$$\frac{\Phi, \Gamma \vdash_c cd \Uparrow cd' : \Gamma'}{\Phi, \Gamma \vdash_t cd \Uparrow cd' : \Gamma'}$$

Well-formed class signature: $\Phi \vdash Sig\,\mathbf{ok}$

$$\frac{}{\Phi \vdash \mathbf{Object\,ok}}$$

$$\frac{\begin{array}{c} \mathit{Sig} = \mathbf{class}\ C{<}\overrightarrow{T}{>}\ \mathbf{extends}\ G{<}\tau_0,\ldots,\tau_k{>}\ \{ce_0,\ldots,ce_n\} \\ (G : \mathit{Sig}') \in \Phi \qquad \Phi \vdash \mathit{Sig}'\ \mathbf{ok} \\ \Phi \vdash C{<}\overrightarrow{T}{>}\,\mathbf{extends}\,G{<}\tau_0,\ldots,\tau_k{>} \Leftarrow ce_i \qquad \text{for } ce_i \in ce_0,\ldots,ce_n \end{array}}{\Phi \vdash \mathit{Sig}\,\mathbf{ok}}$$

Well-formed class hierarchy: $\vdash \Phi\,\mathbf{ok}$

$$\frac{\Phi \vdash \mathit{Sig}\,\mathbf{ok} \text{ for } \mathit{Sig} \in \Phi}{\vdash \Phi\,\mathbf{ok}}$$

Program typing: $\Phi \vdash P \Uparrow P'$

$$\frac{\begin{array}{c} \Gamma_0 = \epsilon \qquad \Phi, \Gamma_i \vdash_t td_i \Uparrow td_i' : \Gamma_{i+1} \quad \text{for } i \in 0,\ldots,n \\ \Phi, \epsilon, \Gamma_{n+1} \vdash s : \tau \Uparrow s' : \Gamma'_{n+1} \end{array}}{\Phi \vdash \mathbf{let}\ td_0,\ldots,td_n\ \mathbf{in}\ s \Uparrow \mathbf{let}\ td_0',\ldots,td_n'\ \mathbf{in}\ s'}$$