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** | $\overrightarrow{\tau} \xrightarrow{k} \sigma$ | $C\langle\overrightarrow{\tau}\rangle$
Ground types τ, σ	::=	**dynamic** | **Object** | **Null** | **Type** | **num** | **bool** | $\overrightarrow{\textbf{dynamic}} \xrightarrow{+} \textbf{dynamic}$ | $C\langle\overrightarrow{\textbf{dynamic}}\rangle$
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** $C\langle\overrightarrow{\tau}\rangle()$ | **op**$(\overrightarrow{e})$ | $e(\overrightarrow{e})$ | $e.m$ | **this**$.x$ | $x = e$ | **this**$.x = e$ | **throw** | e **as** τ | e **is** 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** $C\langle\overrightarrow{T}\rangle$ **extends** $G\langle\overrightarrow{\tau}\rangle$ $\{\overrightarrow{vd}\}$
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.

$$\frac{\begin{array}{c} cd = \textbf{class}\ C\langle\overrightarrow{T}\rangle\ \textbf{extends}\ G\langle\overrightarrow{\tau}\rangle\ \{vd_0, \ldots, vd_n\} \\ (C : \textbf{class}\ C\langle\overrightarrow{T}\rangle\ \textbf{extends}\ G\langle\overrightarrow{\tau}\rangle\ \{ce_0, \ldots, \overrightarrow{ce}_n\}) \in \Phi \\ \Delta = \overrightarrow{T} \quad \Gamma_i = \begin{cases} \Gamma_{C\langle\overrightarrow{T}\rangle} & \text{if } vd_i \text{ is a method} \\ \Gamma & \text{if } vd_i \text{ is a field} \end{cases} \\ \Phi, \Delta, \Gamma_i \vdash_{ce} vd_i : ce_i \Uparrow \Gamma_i' \qquad \text{for } i \in 0, \ldots, n \end{array}}{\Phi, \Gamma \vdash_c cd \Uparrow \Gamma}$$

Override checking:

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

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 $C\langle T_0, \ldots, T_n\rangle$ which extends $G\langle\tau_0, \ldots, \tau_k\rangle$, 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.

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

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

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

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

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

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

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

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

Method and field absence

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

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

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**.

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

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**.

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

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.

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

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.

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

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.

$$\frac{\begin{array}{c} cd = \textbf{class}\ C\langle\overrightarrow{T}\rangle\ \textbf{extends}\ G\langle\overrightarrow{\tau}\rangle\ \{vd_0, \ldots, vd_n\} \\ (C : \textbf{class}\ C\langle\overrightarrow{T}\rangle\ \textbf{extends}\ G\langle\overrightarrow{\tau}\rangle\ \{ce_0, \ldots, \overrightarrow{ce}_n\}) \in \Phi \\ \Delta = \overrightarrow{T} \quad \Gamma_i = \begin{cases} \Gamma_{C\langle\overrightarrow{T}\rangle} & \text{if } vd_i \text{ is a method} \\ \Gamma & \text{if } vd_i \text{ is a field} \end{cases} \\ \Phi, \Delta, \Gamma_i \vdash_{ce} vd_i : ce_i \Uparrow \Gamma_i' \qquad \text{for } i \in 0, \ldots, n \end{array}}{\Phi, \Gamma \vdash_c cd \Uparrow \Gamma}$$

Override checking:

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

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 $C\langle T_0, \ldots, T_n\rangle$ which extends $G\langle\tau_0, \ldots, \tau_k\rangle$, 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.

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

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

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

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

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

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.

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

Dynamic Semantics

NOTE: *This doesn't capture null derefences yet.*

For the dynamic semantics, we extend the term language as follows:

$$
\begin{array}{lll}
\text{Types } \tau, \sigma & ::= & \ldots \mid \langle \tau \rangle \\
\text{Values } v & ::= & l \mid i \mid \text{tt} \mid \text{ff} \mid \textbf{null} \\
\text{Object members } (om) & ::= & \textbf{var}\ x : [\tau]\ =\ v \mid f(\overrightarrow{x : \tau}) : \tau = s \\
\text{Heap values } (hv) & ::= & \langle v \rangle_\tau \mid \textbf{object}_\tau\{\overrightarrow{om}\} \mid (\overrightarrow{x : [\tau]}) : [\sigma] \Rightarrow s \\
\text{Expressions } e & ::= & \ldots \mid l \mid *l \mid \textbf{object}_\tau\{\overrightarrow{vd}\} \mid \textbf{do}\{\, s \,\} \\
 & & \mid l.x \mid l.x = e \mid *l = e
\end{array}
$$

Note that values are simply a refinement of expressions, and that object value members are refinements of variable declarations in which variables are constrained to be values.

The type language is extended with the type of boxed values.

Heap values consist of boxed values, object instances, and heap allocated closures.

We extend the expression language with: labels representing pointers to heap objects; label dereferences for unboxing values; object expressions representing new objects with their fields under evaluation; a **do** expression encapsulating a partially evaluated statement block as an expression; field load from a label; field write to a label; and assignment to a heap boxed slot.

We reuse the substitution notation to allow the dynamic semantics to replace the primitives referring to **this** in the syntax with the obvious corresponding versions operating on labels.

The dynamic semantics is defined in terms of the following auxilliary structures.

$$
\begin{array}{lll}
\text{Class table } (CT) & ::= & \epsilon \mid CT, cd \\
\text{Heaps } (H) & ::= & \epsilon \mid H, l \mapsto v
\end{array}
$$

We treat heaps as partial maps from labels to values. Throughout the dynamic semantics we assume that all introduced labels are fresh.

Expression evaluation: $\{CT, H \mid e\} \longmapsto \{H' \mid res\}$

Expression evaluation relates class tables, heaps, and an expression, to a result. A result is either an expression or a type error, indicated by **Error**.

$$
\text{Results } res \quad ::= \quad e \mid \textbf{Error}
$$

To simply the presentation, we treat results as a meta-level monad, and define the bind operation to handle propagation of both type errors and exceptions as follows.

$$
\begin{array}{rcl}
x \Leftarrow \textbf{Error in } e & = & \textbf{Error} \\
x \Leftarrow \textbf{throw in } e & = & \textbf{throw} \\
x \Leftarrow e_0 \textbf{ in } e & = & [e_0/x]e
\end{array}
$$

Creating a new empty object produces a new object expression.

$$
\frac{}{\{CT, H \mid \textbf{new Object}<>() \} \longmapsto \{H \mid \textbf{object}_{\textbf{Object}}\{\}\}}
$$

Creating a new object proceeds by creating a new object of the superclass, merging in the fields of the subclass (replacing any overrides). The actual type arguments are substituted for the parameters.

$$
\frac{\begin{array}{c} CT(C) = \textbf{class } C<T_0, \ldots, T_n> \textbf{ extends } G<\overrightarrow{\sigma}> \{\overrightarrow{vd}\} \\ \{CT, H \mid \textbf{new } G<[\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]\overrightarrow{\sigma}>() \} \longmapsto \{H' \mid \textbf{object}_{\tau_g}\{\overrightarrow{vd_s}\}\} \\ \overrightarrow{vd_c} = [\tau_0, \ldots, \tau_n/T_0, \ldots, T_n]((\overrightarrow{vd_s} \setminus \overrightarrow{vd}) \cup \overrightarrow{vd}) \end{array}}{\{CT, H \mid \textbf{new } C<\tau_0, \ldots, \tau_n>() \} \longmapsto \{H' \mid \textbf{object}_{C<\overrightarrow{\tau}>}\{\overrightarrow{vd_c}\}\}}
$$

An object expression for which all fields have been fully evaluated is placed in the heap bound to a fresh label. The value of the object expression is the label to which the object value is bound.

$$
\frac{\begin{array}{c} om'_i = \begin{cases} [l_n/\textbf{this}]om_i & \text{if } om_i \text{ is a method} \\ om_i & \text{otherwise} \end{cases} \\ H' = H, l_n \mapsto \textbf{object}_{C<\overrightarrow{\tau}>}\{om'_0, \ldots, om'_n\} \end{array}}{\{CT, H \mid \textbf{object}_{C<\overrightarrow{\tau}>}\{om_0, \ldots, om_n\}\} \longmapsto \{H' \mid l_n\}}
$$

An object expression for which some prefix of the fields have been evaluated but with at least one field which is still unevaluated, makes progress by stepping the expression bound to the field for one step.

$$
\frac{\begin{array}{c} vd_c = \textbf{var } x : [\tau] \ = \ e \qquad \{CT, H \mid e\} \longmapsto \{H' \mid res\} \\ r_o = e' \Leftarrow res \textbf{ in object}_{C<\overrightarrow{\tau}>}\{\overrightarrow{om}, \textbf{var } x : [\tau] \ = \ e', \overrightarrow{vd}\} \end{array}}{\{CT, H \mid \textbf{object}_{C<\overrightarrow{\tau}>}\{\overrightarrow{om}, vd_c, \overrightarrow{vd}\}\} \longmapsto \{H' \mid r_o\}}
$$

There is an arbitrary set of primitive operations which the static semantics and dynamic semantics are parameterized over. Evaluating a primitive application proceeds by evaluating the arguments to values, and then producing a result value. Note that primitives do not modify the heap.

$$\frac{\text{If } \mathbf{op} \text{ applied to } v_0, \ldots, v_n \text{ produces } v_r}{\{CT, H \mid \mathbf{op}(v_0, \ldots, v_n)\} \longmapsto \{H \mid v_r\}}$$

$$\frac{\{CT, H \mid e_0\} \longmapsto \{H' \mid res_0\} \quad res = e'_0 \Leftarrow res_0 \textbf{ in } \mathbf{op}(\overrightarrow{v}, e'_0, \overrightarrow{e})}{\{CT, H \mid \mathbf{op}(\overrightarrow{v}, e_0, \overrightarrow{e})\} \longmapsto \{H' \mid res\}}$$

Functions are evaluated by evaluating first the applicand and then the arguments.

$$\frac{\{CT, H \mid e_f\} \longmapsto \{H' \mid res_f\} \quad res = e'_f \Leftarrow res_f \textbf{ in } e'_f(\overrightarrow{e})}{\{CT, H \mid e_f(\overrightarrow{e})\} \longmapsto \{H' \mid res\}}$$

$$\frac{\{CT, H \mid e_a\} \longmapsto \{H' \mid res_a\} \quad res = e'_a \Leftarrow res_a \textbf{ in } v_f(\overrightarrow{v}, e'_a, \overrightarrow{e})}{\{CT, H \mid v_f(\overrightarrow{v}, e_a, \overrightarrow{e})\} \longmapsto \{H' \mid res\}}$$

A function valued label applied to values reduces by substituting the arguments for the parameters into the body statement, and returning a statement block expression.

$$\frac{H(l_f) = (x_0 : \tau_0, \ldots, x_n : \tau_n) : \tau_r \Rightarrow s}{\{CT, H \mid l_f(v_0, \ldots, v_n)\} \longmapsto \{H \mid \mathbf{do}\{\, [v_0, \ldots, v_n / x_0, \ldots, x_n]s \,\}\}}$$

A do statement steps by stepping the suspended statement.

$$\frac{\{CT, H \mid s\} \longmapsto_s \{H' \mid s'\}}{\{CT, H \mid \mathbf{do}\{\, s \,\}\} \longmapsto \{H' \mid \mathbf{do}\{\, s' \,\}\}}$$

If evaluating a statement hits a runtime type error, the expression evaluates to a type error.

$$\frac{\{CT, H \mid s\} \longmapsto_s \{H' \mid \mathbf{Error}\}}{\{CT, H \mid \mathbf{do}\{\, s \,\}\} \longmapsto \{H' \mid \mathbf{Error}\}}$$

A do statement that reaches the return of a value evaluates to the value.

$$\frac{}{\{CT, H \mid \mathbf{do}\{\,\mathbf{return}\ v\,\}\} \longmapsto \{H \mid v\}}$$

A do statement that reaches a throw evaluates to a throw.

$$\frac{}{\{CT, H \mid \mathbf{do}\{\,\mathbf{return\ throw}\,\}\} \longmapsto \{H \mid \mathbf{throw}\}}$$

A method load from an expression evaluates compositionally.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o\ \mathbf{in}\ e'.m}{\{CT, H \mid e.m\} \longmapsto \{H' \mid res\}}$$

A method load from a label loads the appropriate method and allocates it as a closure in the heap.

$$\frac{H(l) = \mathbf{object}_\tau\{\ldots, m(\overrightarrow{x:\tau}) : \tau = s, \ldots\} \quad H' = H, l_f \mapsto (\overrightarrow{x:\tau}) : \tau \Rightarrow s}{\{CT, H \mid l.m\} \longmapsto \{H' \mid l_f\}}$$

A field load from a label loads the appropriate field value.

$$\frac{H(l) = \mathbf{object}_\tau\{\ldots, \mathbf{var}\ x : \tau\ =\ v_x, \ldots\}}{\{CT, H \mid l.x\} \longmapsto \{H \mid v_x\}}$$

Assignment of an expression to a labeled slot evaluates compositionally.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o\ \mathbf{in}\ *l = e'}{\{CT, H \mid *l = e\} \longmapsto \{H' \mid res\}}$$

Assignment of a value to a labeled slot replaces the contents of the slot with the new value.

$$\frac{H' = H, l \mapsto \langle v \rangle_\tau}{\{CT, H \mid *l = v\} \longmapsto \{H' \mid v\}}$$

Unboxing of a labeled slot looks up the slot and evaluates to its contents.

$$\frac{H(l) = \langle v \rangle_\tau}{\{CT, H \mid *l\} \longmapsto \{H \mid v\}}$$

Assignment of an expression to a field evaluates compositionally.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o \textbf{ in } l.x = e'}{\{CT, H \mid l.x = e\} \longmapsto \{H' \mid res\}}$$

Assignment of a value to a field replaces the contents of the field the new value.

$$\frac{H(l) = \textbf{object}_\tau\{\ldots, \textbf{var } x : \tau = v_x, \ldots\} \quad H' = H, l \mapsto \textbf{object}_\tau\{\ldots, \textbf{var } x : \tau = v, \ldots\}}{\{CT, H \mid l.x = v\} \longmapsto \{H' \mid v\}}$$

A type cast on an expression evaluates compositionally.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o \textbf{ in } e' \textbf{ as } \tau}{\{CT, H \mid e \textbf{ as } \tau\} \longmapsto \{H' \mid res\}}$$

A type cast of a null value always succeeds.

$$\frac{}{\{CT, H \mid \textbf{null as } \tau\} \longmapsto \{H \mid \textbf{null}\}}$$

Casting a value to a type which is a subtype of its runtime type evaluates to the value.

$$\frac{\sigma = \mathit{typeof}(H, v) \qquad \Phi = \mathit{sigof}(CT) \quad \Phi, \epsilon \vdash \sigma <: \tau}{\{CT, H \mid v \textbf{ as } \tau\} \longmapsto \{H \mid v\}}$$

Casting a non-null value to a type which is not a subtype of its runtime type evaluates to a throw.

$$\frac{\sigma = \mathit{typeof}(H, v) \qquad \Phi = \mathit{sigof}(CT) \\ v \neq \mathbf{null} \qquad \mathbf{not}\ \Phi, \epsilon \vdash\ \sigma <: \tau}{\{CT, H\ \mid\ v\ \mathbf{as}\ \tau\} \longmapsto \{H\ \mid\ \mathbf{throw}\}}$$

Checking that a value is a type which is a subtype of its runtime type evaluates to true.

$$\frac{\sigma = \mathit{typeof}(H, v) \qquad \Phi = \mathit{sigof}(CT) \\ \Phi, \epsilon \vdash\ \sigma <: \tau}{\{CT, H\ \mid\ v\ \mathbf{as}\ \tau\} \longmapsto \{H\ \mid\ \mathrm{tt}\}}$$

Checking that a value is a type which is not a subtype of its runtime type evaluates to false.

$$\frac{\sigma = \mathit{typeof}(H, v) \qquad \Phi = \mathit{sigof}(CT) \\ \mathbf{not}\ \Phi, \epsilon \vdash\ \sigma <: \tau}{\{CT, H\ \mid\ v\ \mathbf{as}\ \tau\} \longmapsto \{H\ \mid\ \mathrm{ff}\}}$$

A dynamic call of an expression evaluates compositionally.

$$\frac{\{CT, H\ \mid\ e\} \longmapsto \{H'\ \mid\ \mathit{res}_o\} \\ \mathit{res} = e' \Leftarrow \mathit{res}_o\ \mathbf{in}\ \mathbf{dcall}(e', \overrightarrow{e})}{\{CT, H\ \mid\ \mathbf{dcall}(e, \overrightarrow{e})\} \longmapsto \{H'\ \mid\ \mathit{res}\}}$$

$$\frac{\{CT, H\ \mid\ e_0\} \longmapsto \{H'\ \mid\ \mathit{res}_o\} \\ \mathit{res} = e' \Leftarrow \mathit{res}_o\ \mathbf{in}\ \mathbf{dcall}(v, v_0, \ldots, v_k, e'_0, \ldots, e_j)}{\{CT, H\ \mid\ \mathbf{dcall}(v, v_0, \ldots, v_k, e_0, \ldots, e_j)\} \longmapsto \{H'\ \mid\ \mathit{res}\}}$$

A dynamic call of a label applied values, where the label is bound in the heap to a closure of the appropriate arity, steps to a normal application of the same label, with casts applied to each of the arguments.

NOTE: *This is convenient for specifying the semantics, but may be harder to fit into a proof. May reconsider*

NOTE: *There's a subtle point here about which subtyping relation to use, and what to do when the types of the arguments are non-ground. This needs some further thought.*

$$\frac{H(l) = (x_0 : \tau_0, \ldots, x_n : \tau_n) : \tau_r \Rightarrow s}{\{CT, H\ \mid\ \mathbf{dcall}(l, v_0, \ldots, v_n)\} \longmapsto \{H\ \mid\ l(v_0\ \mathbf{as}\ \tau_0, \ldots, v_n\ \mathbf{as}\ \tau_n)\}}$$

A dynamic call of a label applied to values, where the label is not bound in the heap to a closure of the appropriate arity, steps to a throw.

$$\frac{H(l) \neq (x_0 : \tau_0, \ldots, x_n : \tau_n) : \tau_r \Rightarrow s}{\{CT, H \mid \mathbf{dcall}(l, v_0, \ldots, v_n)\} \longmapsto \{H \mid \mathbf{throw}\}}$$

A dynamic load of a method evaluates compositionally.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o \textbf{ in } \mathbf{dload}(e', m)}{\{CT, H \mid \mathbf{dload}(e, m)\} \longmapsto \{H' \mid res\}}$$

A dynamic load from a label bound to an object with the appropriate method present evaluates to the same thing as a regular load of that label and method.

NOTE: *Again, re-using the regular load here is convenient, but may be ugly in the proof. If so, reconsider*

$$\frac{H(l) = \mathbf{object}_\tau\{\ldots, m(\overrightarrow{x : \tau}) : \tau = s, \ldots\}}{\{CT, H \mid \mathbf{dload}(l, m)\} \longmapsto \{H \mid l.m\}}$$

A check of an expression evaluates compositionally.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o \textbf{ in } \mathbf{check}(e', \tau)}{\{CT, H \mid \mathbf{check}(e, \tau)\} \longmapsto \{H' \mid res\}}$$

A check of a null value always succeeds.

$$\frac{}{\{CT, H \mid \mathbf{check}(\mathbf{null}, \tau)\} \longmapsto \{H \mid \mathbf{null}\}}$$

Checking a value at a type which is a subtype of its runtime type evaluates to the value.

$$\frac{\sigma = \mathit{typeof}(H, v) \qquad \Phi = \mathit{sigof}(CT) \quad \Phi, \epsilon \vdash \sigma <: \tau}{\{CT, H \mid \check{v}\tau\} \longmapsto \{H \mid v\}}$$

Checking a non-null value at a type which is not a subtype of its runtime type evaluates to a runtime type error.

$$\frac{\sigma = \mathit{typeof}(H, v) \qquad \Phi = \mathit{sigof}(CT) \qquad v \neq \textbf{null} \qquad \textbf{not } \Phi, \epsilon \vdash \ \sigma <: \tau}{\{CT, H \ \mid \ \textbf{check}(v, \tau)\} \longmapsto \{H \ \mid \ \textbf{Error}\}}$$

Statement evaluation: $\{CT, H \ \mid \ s\} \longmapsto_s \{H \ \mid \ res\}$

Statement evaluation relates class tables, heaps, and a statement to a result. A result is either a statement or a type error, indicated by **Error**.

Statement results $res \quad ::= \quad s \mid \textbf{Error}$

As with expressions, we simplify the rules by treating results as a meta-level monad. We define the bind operation to handle propagation of both type errors and exceptions out of expressions as follows.

$$\begin{array}{rcl} x \Leftarrow \textbf{Error in } s & = & \textbf{Error} \\ x \Leftarrow \textbf{throw in } s & = & \textbf{return throw} \\ x \Leftarrow e_0 \textbf{ in } s & = & [e_0/x]s \end{array}$$

Statement evaluation proceeds by first re-associating statements to find the first statement in the sequence.

$$\frac{\{CT, H \ \mid \ s_1; (s_2; s_3)\} \longmapsto_s \{H' \ \mid \ res\}}{\{CT, H \ \mid \ (s_1; s_2); s_3\} \longmapsto_s \{H' \ \mid \ res\}}$$

Evaluating a variable definition steps by evaluating the initializer to a value. If an exception or runtime type error is hit, then the bind operation discards the rest of the continuation (s_k) and propagates the exception/error.

$$\frac{\{CT, H \ \mid \ e\} \longmapsto \{H' \ \mid \ res_o\} \qquad res = e' \Leftarrow res_o \textbf{ in var } x : \tau \ = \ e'; s_k}{\{CT, H \ \mid \ \textbf{var } x : \tau \ = \ e; s_k\} \longmapsto_s \{H' \ \mid \ res\}}$$

Evaluating a variable definition with a value initializer and no continuation statement simply returns null.

$$\frac{}{\{CT, H \ \mid \ \textbf{var } x : \tau \ = \ v\} \longmapsto_s \{H' \ \mid \ \textbf{return null}\}}$$

Evaluating a variable definition with a value initializer and a continuation statement allocates a new heap slot for the variable and places the boxed value in the slot. All uses of the variable in the continuation are replaced with a dereference (unbox) of the label.

$$\frac{H' = H, l \mapsto \langle \tau \rangle_v}{\{CT, H \mid \textbf{var}\ x : \tau\ =\ v; s_k\} \longmapsto_s \{H' \mid [*l/x]s_k\}}$$

Evaluating a local function declaration with no continuation simply returns null.

$$\frac{}{\{CT, H \mid f(\overrightarrow{x : \tau}) : \sigma = s\} \longmapsto_s \{H' \mid \textbf{return null}\}}$$

Evaluating a local function declaration with a continuation statement allocates a closure in the heap bound to a fresh label, and replaces all recursive uses of the variable in the closure and in the continuation statement with the label.

$$\frac{H' = H, l \mapsto [l/f](\overrightarrow{x : \tau}) : \sigma \Rightarrow s}{\{CT, H \mid f(\overrightarrow{x : \tau}) : \sigma = s; s_k\} \longmapsto_s \{H' \mid [l/f]s_k\}}$$

Evaluating an expression statement steps by evaluating the expression, propagating errors and exceptions as usual.

$$\frac{\begin{array}{c}\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \\ res = e' \Leftarrow res_o\ \textbf{in}\ e'; s_k\end{array}}{\{CT, H \mid e; s_k\} \longmapsto_s \{H \mid res\}}$$

A valuable expression statement with no continuation evaluates to a return of null.

$$\frac{}{\{CT, H \mid v\} \longmapsto_s \{H \mid \textbf{return null}\}}$$

A valuable expression statement with a continuation evaluates to the continuation.

$$\frac{}{\{CT, H \mid v; s_k\} \longmapsto_s \{H \mid s_k\}}$$

A conditional statement evaluates by stepping the expression to a value.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o \textbf{ in if } (e') \textbf{ then } s_1 \textbf{ else } s_2; s_k}{\{CT, H \mid \textbf{if } (e) \textbf{ then } s_1 \textbf{ else } s_2; s_k\} \longmapsto_s \{H \mid res\}}$$

A conditional statement with true as the scrutinee steps to the composition of the true branch statement and the continuation statement.

$$\frac{}{\{CT, H \mid \textbf{if } (\text{tt}) \textbf{ then } s_1 \textbf{ else } s_2; s_k\} \longmapsto_s \{H \mid s_1; s_k\}}$$

A conditional statement with false as the scrutinee steps to the composition of the false branch statement and the continuation statement.

$$\frac{}{\{CT, H \mid \textbf{if } (\text{ff}) \textbf{ then } s_1 \textbf{ else } s_2; s_k\} \longmapsto_s \{H \mid s_2; s_k\}}$$

A return statement steps the returned expression to a value.

$$\frac{\{CT, H \mid e\} \longmapsto \{H' \mid res_o\} \quad res = e' \Leftarrow res_o \textbf{ in return } e'; s_k}{\{CT, H \mid \textbf{return } e; s_k\} \longmapsto_s \{H' \mid res\}}$$

A return of a value with a continuation discards the continuation.

$$\frac{}{\{CT, H \mid \textbf{return } v; s_k\} \longmapsto_s \{H \mid \textbf{return } v\}}$$

Top level evaluation: $\{CT, H \mid P\} \longmapsto_t \{H' \mid s\}$

To support mutually recursive classes (which may in turn refer to top level declarations in their field initializers), top level variables and functions must be pre-allocated in the heap before executing code. Initializers for fields are accumulated as assignment statements.

A program with no top level declarations evaluates by evaluating its main statement.

$$\frac{}{\{CT, H \mid \textbf{let } \epsilon \textbf{ in } s_m\} \longmapsto_t \{H \mid s_m\}}$$

A program with a top level variable declaration evaluates by placing a slot for the variable in the heap initialized to null, replacing all uses of the variable

with dereferences of the slot, and pre-pending an assignment to the slot to the continuation for the rest of the program.

$$\frac{H' = H, l \mapsto \langle \mathbf{null} \rangle_\tau \qquad \{CT, H' \mid [*l/x]\mathbf{let}\ \overrightarrow{td}\ \mathbf{in}\ s_m\} \longmapsto_t \{H'' \mid s_k\}}{\{CT, H \mid \mathbf{let\ var}\ x : \tau\ =\ e, \overrightarrow{td}\ \mathbf{in}\ s_m\} \longmapsto_t \{H'' \mid *l = e; s_k\}}$$

A program with a top level function declaration evaluates by allocating a closure in the heap and replacing all uses of the function variable (including recursive references) with the label. No additional computation is added to the program.

$$\frac{\begin{array}{c} H' = H, l \mapsto [l/f](\overrightarrow{x : \tau}) : \sigma \Rightarrow s \\ \{CT, H' \mid [l/f]\mathbf{let}\ \overrightarrow{td}\ \mathbf{in}\ s_m\} \longmapsto_t \{H'' \mid s_k\} \end{array}}{\{CT, H \mid \mathbf{let}\ f(\overrightarrow{x : \tau}) : \sigma = s, \overrightarrow{td}\ \mathbf{in}\ s_m\} \longmapsto_t \{H'' \mid s_k\}}$$

Top level class definitions are dropped (they should already be present in CT).

$$\frac{\{CT, H \mid \mathbf{let}\ \overrightarrow{td}\ \mathbf{in}\ s_m\} \longmapsto_t \{H' \mid s_k\}}{\{CT, H \mid \mathbf{let}\ cd, \overrightarrow{td}\ \mathbf{in}\ s_m\} \longmapsto_t \{H' \mid s_k\}}$$

Program evaluation: $\{CT, H \mid P\} \longmapsto_p \{H \mid res\}$

Program evaluation relates class tables, heaps, and a program, to a result. A result is either a program or a type error, indicated by **Error**.

$$\text{Results } res \quad ::= \quad P \mid \mathbf{Error}$$

A program with top level declarations steps to a program with no top level declarations by allocating top level variables and setting up the initialization code.

$$\frac{\{CT, H \mid \mathbf{let}\ \overrightarrow{td}\ \mathbf{in}\ s_m\} \longmapsto_t \{H' \mid s_k\}}{\{CT, H \mid P\} \longmapsto_p \{H' \mid \mathbf{let}\ \epsilon\ \mathbf{in}\ s_k\}}$$

A program with no top level declarations can take a step by stepping the main statement to another statement. The result is a new program with an updated main.

$$\frac{\{CT, H \mid s_m\} \longmapsto_s \{H' \mid s'_m\}}{\{CT, H \mid \textbf{let } \epsilon \textbf{ in } s_m\} \longmapsto_p \{H' \mid \textbf{let } \epsilon \textbf{ in } s'_m\}}$$

A program with no top level declarations can take a step by stepping the main statement to a runtime type error. The result is a runtime type error.

$$\frac{\{CT, H \mid s_m\} \longmapsto_s \{H' \mid \textbf{Error}\}}{\{CT, H \mid \textbf{let } \epsilon \textbf{ in } s_m\} \longmapsto_p \{H' \mid \textbf{Error}\}}$$

Metatheory

This is a sketch of the intended meta-theory.

Soundness

We consider soundness with respect to a slight modification of the type system as written above. Specifically the variant subtyping relation is eliminated, and type parameters to generics are checked to see that they are only used covariantly.

The dynamic semantics as written above should be sound with respect to that (standard) type system.

Conjecture 1 (Progress) *If a program P is well-typed with respect to the standard variant of the static semantics ($\Phi \vdash P$) and CT and H are well-formed with respect to Φ (this needs to be defined, but roughly that Φ and CT have consistent views of the class hierarchy, and that H defines labels of the appropriate type) then either:*

- *P is of the form* **let** ϵ **in return throw**
- *P is of the form* **let** ϵ **in return** v
- $\{CT, H \mid P\} \longmapsto \{H' \mid res\}$

Conjecture 2 (Preservation) *If a program P is well-typed with respect to the standard variant of the static semantics ($\Phi \vdash P$) and CT and H are well-formed with respect to Φ (this needs to be defined, but roughly that Φ and CT have consistent views of the class hierarchy, and that H defines labels of the appropriate type), and $\{CT, H \mid P\} \longmapsto \{H' \mid P'\}$ then $\Phi \vdash P'$.*

The soundness result that we expect to be true is that the translation of a program according to the elaboration semantics, with method signatures replaced by their internal signatures, and with covariant function types translated to replace dynamic in their arguments with bottom, is well-typed in the standard static semantics without the variant subtyping.

Conjecture 3 (Soundness of the translation) *If $\Phi \vdash P \Uparrow P'$ then $\Phi' \vdash P''$ in the standard static semantics where Φ' is Φ with method signatures replaced with internal signatures and P'' is P' with covariant arrows translated to contravariant arrows as described above.*

NOTE: *The translation of covariant arrows to contravariant arrows should probably just be made part of the translation to avoid having to deal with it here*

Conjecture 4 (Correctness of the translation) *If $\Phi \vdash P \Uparrow P'$ and P reduces to a value v in the standard Dart semantics (not specified here yet), then either:*

- $\{CT, H \mid P\} \longmapsto^* \{H' \mid v\}$
- $\{CT, H \mid P'\} \longmapsto^* \{H' \mid \textbf{Error}\}$