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** | $(\overrightarrow{x : [\tau]}) : [\sigma] \Rightarrow e$ | **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** $G\!<\!\overrightarrow{S}\!>$ $\{\overrightarrow{vd}\}$
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.

Type context (Δ)	::=	$\epsilon \mid \Delta, T <: \tau$
Class element (ce)	::=	$\textbf{var}\ x : \tau \mid \textbf{fun}\ f : \tau$
Class signature (Sig)	::=	$\textbf{class}\ C{<}\overrightarrow{T}{>}\ \textbf{extends}\ G{<}\overrightarrow{S}{>}\ \{\overrightarrow{ce}\}$
Class hierarchy (Φ)	::=	$\epsilon \mid \Phi, C\ :\ Sig$
Term context (Γ)	::=	$\epsilon \mid \Gamma, x\ :\ \tau$

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.

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

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

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

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

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

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

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

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

Field and Method lookup

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

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

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

Typing

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.

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

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

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

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

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.

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

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

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

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.

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

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

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

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.

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

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.

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

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

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

A dynamic call expression is well-typed so long as the applicand and the arguments are well-typed at any type.

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

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 dynamic method load is well typed so long as the term is well-typed.

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

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.

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

A throw expression is well-typed at any type.

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

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.

TODO(leafp): specify ground types

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

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.

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

A check expression is well-typed so long as the term being checked is well-typed. The synthesized type is the target type of the check.

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

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.

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

Elaboration

These are the same rules, extended with a translated term which corresponds to the original term with the additional dynamic type checks inserted to reify the static unsoundness as runtime type errors.

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.

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

TODO(leafp): specify ground types

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

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

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

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.

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

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

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

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

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