CubeX with Laziness

Ross Tate

November 26, 2013

0.1 Laziness

This extension of the CubeX employs lazy evaluation in the language. All expressions should be evaluated lazily (only when their values are needed). In addition, we've made the following changes/simplifications (highlighted throughout the spec with "**"):

1. There is an added `select` method on the `Boolean` type. (See spec for details.)
2. Certain operators should appropriately short-circuit. (See spec for details.)
3. Interfaces do not implement any methods.
4. There is only single interface inheritance. (No class inheritance or multiple inheritance.)
5. There is no Intersection type ($\tau \cap \tau$).
6. There are no `if` statements, no `while` loops, and no `for` loops.
7. There are no `Iterables`, and thus no `divide`, no `modulo`, no `onwards`, no `through`, and no `Strings`.
8. Input and output should now be of type `Integer`, not `Iterable⟨String⟩`. That is, instead of `void print_line(char* str, int len)`, `int next_line_length()`, and `void read_line(char* buffer)`, you'll have `int read_int()` and `void print_int(int output)`.

0.2 Grading

This extension will NOT be graded in stages. You should prepare for ALL tests to include:

- Statements
- Non-generic functions
- Generic functions
- Non-Generic classes without inheritance
- Generic classes without inheritance
- Generic classes with single interface inheritance.

No tests will include Generic classes with single interface inheritance + interface method implementations, Generic classes with class inheritance, or Generic classes with multiple interface inheritance. However, we will not test for the exclusion of these or any other CubeX language features that have been dropped specifically for this Laziness extension (e.g. We will not test to make sure a `while` loop does not parse/type-check/execute.) You are free to keep pre-existing features in your langauge so long as their inclusion does not break any tests pertaining to this modified spec. Your implementation will be graded on correctness (a single call to `print_int(int output)` should be made with the correct return value of the program) and performance (memory management and optimizations outlined in PA4 and PA5 should still be applied).

1 Lexing and Parsing

1.1 Core and Full Languages

$\nu_v ::=$ variable/function/method names $\quad \nu_c ::=$ class/interface names $\quad \nu_p ::=$ type-parameter names $\quad \nu_{vc} ::= \nu_v \mid \nu_c$

kind context	Θ	$::=$	$\nu_p, \ldots, \nu_p$
type context	Γ	$::=$	$\nu_v : \tau, \ldots, \nu_v : \tau$
** type	τ	$::=$	$\nu_p \mid \nu_c\langle\tau, \ldots, \tau\rangle \mid \top \mid \bot$
type scheme	σ	$::=$	$\langle\Theta\rangle(\Gamma) : \tau$
** expression	e	$::=$	$\nu_v \mid \nu_{vc}\langle\tau, \ldots, \tau\rangle(e, \ldots, e) \mid e.\nu_v\langle\tau, \ldots, \tau\rangle(e, \ldots, e) \mid \textbf{true} \mid \textbf{false} \mid n \mid e \ ? \ e \ : \ e$
** statement	s	$::=$	$\{s \ldots s\} \mid \nu_v := e; \mid \textbf{return}\ e;$
** interface	i	$::=$	$\textbf{interface}\ \nu_c\langle\Theta\rangle\ \textbf{extends}\ \tau\ \{\textbf{fun}\ \nu_v\sigma; \ldots \textbf{fun}\ \nu_v\sigma;\}$
class	c	$::=$	$\textbf{class}\ \nu_c\langle\Theta\rangle(\Gamma)\ \textbf{extends}\ \tau\ \{s \ldots s\ \textbf{super}(e, \ldots, e);\ \textbf{fun}\ \nu_v\sigma\ s \ldots \textbf{fun}\ \nu_v\sigma\ s\}$
program	p	$::=$	$s \mid s \ldots s\ p \mid \textbf{fun}\ \nu_v\sigma\ s \ldots \textbf{fun}\ \nu_v\sigma\ s\ p \mid i\ p \mid c\ p$
function context	Δ	$::=$	$\varnothing \mid \Delta, \nu_{vc}\sigma$
class context	Ψ	$::=$	$\varnothing \mid \Psi, \textbf{interface}\ \nu_c\langle\Theta\rangle\ \textbf{extends}\ \tau\ \{\Delta; \nu_v, \ldots, \nu_v\} \mid \Psi, \textbf{class}\ \nu_c\langle\Theta\rangle\ \textbf{extends}\ \tau\ \{\Delta\}$

Figure 1: **Lazy** Cubex Core Language Grammar, which gives the grammar for the Cubex core language, along with definitions for the formal constructs Δ and Ψ, which are not part of the language per se but are useful in our formalization of the type system. Note that p stands for "program" and defines a syntactically-valid program in the Cubex core language. Note also that lists, represented with an elipsis, may consist of zero, one, or more elements. So $a, \ldots, a$ may be the empty string, a, or some list of as separated by commas.

We distinguish between the Cubex core language, which is specified by the grammar in Figure 1.1, and the full language. The full language differs from the core language in a number of ways:

- It includes the following unary, binary, and ternary operators, listed in order of precedence, which are short for method calls on arbitrary classes (their signatures are those that are given for the built-in classes at the end of this document). All operators are left-associative.
  1. Unary prefixes `-` and `!` short for `negative` and `negate` respectively.
  2. ** Multiplicative operator `*` short for `times`, which should short-circuit if either of its operands is 0.
  3. Additive `+` and `-` short for `plus` and `minus` respectively.
  4. Inequality operators `<`, `<=`, `>=`, and `>` short for `lessThan` with an additional `Boolean` parameter indicating strictness, and with order reversed for `>=` and `>`.
  5. Equational operators `==` and `!=`, short for `equals` and `equals` followed by `negate`.
  6. ** Boolean operators `&` short for `and` followed by `|` short for `or`, both of which should appropriately short-circuit.
- Characters in the core language which have no obvious ASCII equivalent are represented in the full language in ASCII as follows:
  - `Thing` for $\top$ and `Nothing` for $\bot$.
  - `&` for $\wedge$ and `|` for $\vee$.
  - `<` for $\langle$ and `>` for $\rangle$.
- Expressions (denoted e in the grammar) may be surrounded by parentheses.
- If there are no type parameters (i.e. when $\langle\tau, \ldots, \tau\rangle$ is the empty list), a class/interface/function specification can drop the $\langle\rangle$.
- When calling a function/method/constructor with no type parameters, the $\langle\rangle$ may be dropped.
- If a class or interface just extends $\top$, the **extends** clause can be omitted.
- If the call to **super** has no arguments, it can be omitted.

- If a function implementation is simply **return** of some expression, the **return** can be abbreviated to =. For example,

```
fun foo(y : Integer) : Integer return y * 2;
```

  may be abbreviated to

```
fun foo(y : Integer) : Integer = y * 2;
```

Finally, there is no name hiding. If a judgement requires binding a name twice, that judgement is ill formed and does not hold. However, all the different kinds of context are different namespaces. So a name can be bound both in Δ and Γ. To clarify, Γ and $\hat{\Gamma}$ use the same namespace, so a variable must not be bound in both.

1.2 Name, Keywords, Literals, etc.

The grammar uses ν with subscripts to denote both various kinds of names. We enforce a distinction between these kinds. Variables and function/method names (ν_v) are a lower-case letter followed by letters, numbers, and/or underscores. Class/interface names (ν_c) are an upper-case letter followed by one or more letters, numbers, and/or underscores. Type parameters (ν_p) are an upper-case letter alone. In the formalism we refer to all of these collectively as just ν.

Key words (i.e. any words receiving special treatment in the core language) cannot be used as names or variables.

Integer literals (denoted in the grammar by n) are in decimal only.

Finally, note that the grammar contains various other syntactic artifacts (semicolons, brackets, etc.) whose only legal uses in the language (outside of comments and string literals) are fully specified by their role in the grammar.

1.3 Whitespace and Comments

Unless it occurs inside a string literal, white space is disregarded except for its role of separating names and other tokens (e.g. "`: =`" is not the same as "`:=`" and only the latter can be used in a variable assignment). Unless it occurs inside a string literal, `#` starts a comment that extends to the end of the line. Unless it occurs inside a string literal, `‘` starts a comment that extends to the matching `’`, where these comments can be nested inside each other. Comments are disregarded except for their role of separating names and other tokens.

2 Validating

The following formalizes when a program is valid, incorporating type validity, type checking, and class/interface validation. Judgements take the form "(context) $\vdash$ (property)". Ψ indicates the classes and interfaces in scope, what they directly inherit, and what their methods are. Θ indicates the type parameters in scope. Δ indicates the function names in scope and what their type scheme is. Γ typically indicates the immutable variable names in scope and what their type is. $\hat{\Gamma}$ typically indicates the mutable variable names in scope and what their type is. The following is a summary of the judgements used in this formalism and where their definitions can be found.

Judgement	Meaning	Figure
$\Psi \mid \Theta \vdash \tau <: \tau'$	τ is a subtype of τ'	2
$\Psi \vdash \nu\langle\Theta\rangle$ **extends** τ	generic class/interface $\nu\langle\Theta\rangle$ directly inherits τ	2
$\Psi \mid \Theta \vdash \Gamma <: \Gamma'$	Γ is a subcontext of Γ'	2
$\Psi \mid \Theta \vdash \sigma \approx \sigma'$	σ and σ' are equivalent type schemes	
$\Psi \mid \Theta \vdash \tau.\nu : \sigma$	method ν of type τ has type scheme σ	3
$\Psi \vdash \hat{\nu}\langle\Theta\rangle.\nu : \sigma$	generic class/interface $\hat{\nu}\langle\Theta\rangle$ has method ν with type scheme σ	3
$\Psi \mid \Theta \vdash \tau$	τ is a valid type	4
$\Psi \mid \Theta \vdash_{\hat{\tau}} \tau$	τ is an inheritable type with constructable component $\hat{\tau}$	4
$\Psi \mid \Theta \vdash \Gamma$	Γ is a valid type context	4
$\Psi \mid \Theta \vdash \sigma$	σ is a valid type scheme	4
$\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e : \tau$	expression e has type τ	5
$\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash s : \hat{\Gamma}'$	mutable variables $\hat{\Gamma}'$ are available after valid statement s	6
$\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^b_\tau s : \hat{\Gamma}'$	like above and all returns have type τ with $b =$ **true** guaranteeing a return	7
$\Psi \mid \Theta \vdash \tau : \Delta$	Δ are all the methods of τ	8
$\Psi \vdash \hat{\nu}\langle\Theta\rangle : \Delta$	Δ are the direct methods of generic class/interface $\hat{\nu}\langle\Theta\rangle$	8
$\Psi \mid \Theta \vdash \tau : \nu$	τ provides an implementation of method ν	9
$\Psi \mid \Delta \mid \Gamma \vdash i : \Psi'$	i is a valid interface with signature Ψ'	10
$\Psi \mid \Delta \mid \Gamma \vdash c : \Psi' \mid \Delta'$	c is a valid class with signature Ψ' and constructor Δ'	10
$\Psi \mid \Delta \mid \Gamma \vdash p$	p is a valid program	11
$\vdash p$	p is a valid program in the initial environment	11

$$\boxed{\Psi \mid \Theta \vdash \tau <: \tau \qquad \Psi \vdash \nu\langle\Theta\rangle \textbf{ extends } \tau \qquad \Psi \mid \Theta \vdash \Gamma <: \Gamma \qquad \Psi \mid \Theta \vdash \sigma \approx \sigma}$$

$$\frac{}{\Psi \mid \Theta \vdash \nu <: \nu} \qquad \frac{}{\Psi \mid \Theta \vdash \bot <: \tau} \qquad \frac{}{\Psi \mid \Theta \vdash \tau <: \top}$$

$$\frac{\text{for all } i, \quad \Psi \mid \Theta \vdash \tau_i <: \tau_i' \quad \text{and} \quad \Psi \mid \Theta \vdash \tau_i' <: \tau_i}{\Psi \mid \Theta \vdash \nu\langle\tau_1, \ldots, \tau_n\rangle <: \nu\langle\tau_1', \ldots, \tau_n'\rangle}$$

$$\frac{\Psi \vdash \nu\langle\nu_1, \ldots, \nu_n\rangle \textbf{ extends } \tau' \quad \Psi \mid \Theta \vdash \tau'[\nu_1 \mapsto \tau_1, \ldots, \nu_n \mapsto \tau_n] <: \tau}{\Psi \mid \Theta \vdash \nu\langle\tau_1, \ldots \tau_n\rangle <: \tau} \qquad \frac{\textbf{interface } \nu\langle\Theta\rangle \textbf{ extends } \tau \ \{\ldots\} \text{ in } \Psi}{\Psi \vdash \nu\langle\Theta\rangle \textbf{ extends } \tau} \qquad \frac{\textbf{class } \nu\langle\Theta\rangle \textbf{ extends } \tau \ \{\ldots\} \text{ in } \Psi}{\Psi \vdash \nu\langle\Theta\rangle \textbf{ extends } \tau}$$

$$\frac{}{\Psi \mid \Theta \vdash \varnothing <: \varnothing} \qquad \frac{\Psi \mid \Theta \vdash \tau <: \tau' \quad \Psi \mid \Theta \vdash \Gamma <: \Gamma'}{\Psi \mid \Theta \vdash \nu : \tau, \Gamma <: \nu : \tau', \Gamma'} \qquad \frac{\Psi \mid \Theta \vdash \Gamma <: \Gamma'}{\Psi \mid \Theta \vdash \nu : \tau, \Gamma <: \Gamma'}$$

$$\frac{\Psi \mid \Theta, \hat{\Theta} \vdash \Gamma <: \Gamma' \quad \Psi \mid \Theta, \hat{\Theta} \vdash \Gamma' <: \Gamma \quad \Psi \mid \Theta, \hat{\Theta} \vdash \tau <: \tau' \quad \Psi \mid \Theta, \hat{\Theta} \vdash \tau' <: \tau}{\Psi \mid \Theta \vdash \langle\hat{\Theta}\rangle(\Gamma) : \tau \approx \langle\hat{\Theta}\rangle(\Gamma') : \tau'}$$

Figure 2: Subtyping

$$\boxed{\Psi \mid \Theta \vdash \tau.\nu : \sigma \qquad \Psi \vdash \nu\langle\Theta\rangle.\nu : \sigma}$$

$$\frac{}{\Psi \mid \Theta \vdash \bot.\nu : \sigma} \qquad \frac{\Psi \vdash \hat{\nu}\langle\nu_1, \ldots, \nu_n\rangle.\nu : \sigma}{\Psi \mid \Theta \vdash \hat{\nu}\langle\tau_1, \ldots, \tau_n\rangle.\nu : \sigma[\nu_1 \mapsto \tau_1, \ldots, \nu_n \mapsto \tau_n]}$$

$$\frac{\textbf{interface } \hat{\nu}\langle\Theta\rangle \textbf{ extends } \tau \ \{\ldots; \nu\sigma; \ldots\} \text{ in } \Psi}{\Psi \vdash \hat{\nu}\langle\Theta\rangle.\nu : \sigma} \qquad \frac{\textbf{class } \hat{\nu}\langle\Theta\rangle \textbf{ extends } \tau \ \{\ldots, \nu\sigma, \ldots\} \text{ in } \Psi}{\Psi \vdash \hat{\nu}\langle\Theta\rangle.\nu : \sigma}$$

Figure 3: Method Lookup

$$\boxed{\Psi \mid \Theta \vdash \tau \qquad \Psi \mid \Theta \vdash_\tau \tau \qquad \Psi \mid \Theta \vdash \Gamma \qquad \Psi \mid \Theta \vdash \sigma}$$

$$\frac{\Psi \mid \Theta \vdash_{\hat{\tau}} \tau}{\Psi \mid \Theta \vdash \tau} \qquad \frac{}{\Psi \mid \Theta \vdash_\top \top} \qquad \frac{}{\Psi \mid \Theta \vdash \bot} \qquad \frac{\nu \text{ in } \Theta}{\Psi \mid \Theta \vdash \nu}$$

$$\frac{\textbf{interface } \nu\langle\nu_1, \ldots, \nu_n\rangle \textbf{ extends } \tau \ \{\ldots\} \text{ in } \Psi \quad \text{for all } i, \quad \Psi \mid \Theta \vdash \tau_i}{\Psi \mid \Theta \vdash_\top \nu\langle\tau_1, \ldots, \tau_n\rangle}$$

$$\frac{\textbf{class } \nu\langle\nu_1, \ldots, \nu_n\rangle \textbf{ extends } \tau \ \{\ldots\} \text{ in } \Psi \quad \text{for all } i, \quad \Psi \mid \Theta \vdash \tau_i}{\Psi \mid \Theta \vdash_{\nu\langle\tau_1, \ldots, \tau_n\rangle} \nu\langle\tau_1, \ldots, \tau_n\rangle}$$

$$\frac{\text{for all } i, \quad \Psi \mid \Theta \vdash \tau_i}{\Psi \mid \Theta \vdash \nu_1 : \tau_1, \ldots, \nu_n : \tau_n} \qquad \frac{\Psi \mid \Theta, \hat{\Theta} \vdash \Gamma \quad \Psi \mid \Theta, \hat{\Theta} \vdash \tau}{\Psi \mid \Theta \vdash \langle\hat{\Theta}\rangle(\Gamma) : \tau}$$

Figure 4: Type Validity

$\boxed{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e : \tau}$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e : \tau \qquad \Psi \mid \Theta \vdash \tau <: \tau'}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e : \tau'}$$

$$\frac{\nu : \tau \text{ in } \Gamma}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash \nu : \tau}$$

$$\frac{\begin{array}{c}\text{for all } i, \qquad \Psi \mid \Theta \vdash \tau_i \\ \nu\langle \nu_1, \ldots, \nu_m \rangle(\hat{\nu}_1 : \hat{\tau}_1, \ldots, \hat{\nu}_n : \hat{\tau}_n) : \tau \text{ in } \Delta \\ \text{for all } i, \quad \Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e_i : \hat{\tau}_i[\nu_1 \mapsto \tau_1, \ldots, \nu_m \mapsto \tau_m]\end{array}}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash \nu\langle \tau_1, \ldots, \tau_m \rangle(e_1, \ldots, e_n) : \tau[\nu_1 \mapsto \tau_1, \ldots, \nu_m \mapsto \tau_m]}$$

$$\frac{\begin{array}{c}\text{for all } i, \qquad \Psi \mid \Theta \vdash \tau_i \\ \Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e : \hat{\tau} \\ \Psi \mid \Theta \vdash \hat{\tau}.\nu : \langle \nu_1, \ldots, \nu_m \rangle(\hat{\nu}_1 : \hat{\tau}_1, \ldots, \hat{\nu}_n : \hat{\tau}_n) : \tau \\ \text{for all } i, \quad \Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e_i : \hat{\tau}_i[\nu_1 \mapsto \tau_1, \ldots, \nu_m \mapsto \tau_m]\end{array}}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e.\nu\langle \tau_1, \ldots, \tau_m \rangle(e_1, \ldots, e_n) : \tau[\nu_1 \mapsto \tau_1, \ldots, \nu_m \mapsto \tau_m]}$$

$$\frac{}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash \textbf{true} : \texttt{Boolean}\langle\rangle} \qquad \frac{}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash \textbf{false} : \texttt{Boolean}\langle\rangle}$$

$$\frac{}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash n : \texttt{Integer}\langle\rangle}$$

** $$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e : \texttt{Boolean} \qquad \Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e_1 : \tau \qquad \Psi \mid \Theta \mid \Delta \mid \Gamma \vdash e_2 : \tau}{\Psi \mid \Theta \mid \Delta \mid \Gamma \vdash (e \ ? \ e_1 \ : \ e_2) : \tau}$$

The parentheses above have only been included to prevent confusion between the two colons.

Figure 5: Type Checking Expressions

$\boxed{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \Gamma \vdash s : \Gamma}$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash s : \hat{\Gamma}' \qquad \Psi \mid \Theta \vdash \hat{\Gamma}' <: \hat{\Gamma}''}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash s : \hat{\Gamma}''} \qquad \frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash s : \hat{\Gamma}', \nu : \tau, \nu' : \tau', \hat{\Gamma}''}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash s : \hat{\Gamma}', \nu' : \tau', \nu : \tau, \hat{\Gamma}''}$$

$$\frac{\text{for all } i, \quad \Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash s_i : \hat{\Gamma}_i}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}_0 \vdash \{s_1 \ldots s_n\} : \hat{\Gamma}_n}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \tau}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash \nu \coloneqq e; : \hat{\Gamma}, \nu : \tau} \qquad \frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma}, \nu : \tau, \hat{\Gamma}' \vdash e : \tau'}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}, \nu : \tau, \hat{\Gamma}' \vdash \nu \coloneqq e; : \hat{\Gamma}, \nu : \tau', \hat{\Gamma}'}$$

Figure 6: Type Checking Statements

$\boxed{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \Gamma \vdash^b_\tau s : \Gamma}$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^{\textbf{true}}_\tau s : \hat{\Gamma}'}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^{\textbf{false}}_\tau s : \hat{\Gamma}'}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^b_\tau s : \hat{\Gamma}' \qquad \Psi \mid \Theta \vdash \hat{\Gamma}' <: \hat{\Gamma}''}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^b_\tau s : \hat{\Gamma}''} \qquad \frac{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^b_\tau s : \hat{\Gamma}', \nu : \hat{\tau}, \nu' : \hat{\tau}', \hat{\Gamma}''}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^b_\tau s : \hat{\Gamma}', \nu' : \hat{\tau}', \nu : \hat{\tau}, \hat{\Gamma}''}$$

$$\frac{\begin{array}{c}\text{for all } i, \qquad \Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash^{b_i}_\tau s_i : \hat{\Gamma}_i \\ b = \textbf{true} \text{ implies there exists } i \text{ with } b_i = \textbf{true}\end{array}}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}_0 \vdash^b_\tau \{s_1 \ldots s_n\} : \hat{\Gamma}_n}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \hat{\tau}}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^{\textbf{false}}_\tau \nu \coloneqq e; : \hat{\Gamma}, \nu : \hat{\tau}} \qquad \frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma}, \nu : \hat{\tau}, \hat{\Gamma}' \vdash e : \hat{\tau}'}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma}, \nu : \hat{\tau}, \hat{\Gamma}' \vdash^{\textbf{false}}_\tau \nu \coloneqq e; : \hat{\Gamma}, \nu : \hat{\tau}', \hat{\Gamma}'}$$

$$\frac{\Psi \mid \Theta \mid \Delta \mid \Gamma, \hat{\Gamma} \vdash e : \tau}{\Psi \mid \Theta \mid \Delta \mid \Gamma \mid \hat{\Gamma} \vdash^{\textbf{true}}_\tau \textbf{return } e; : \hat{\Gamma}}$$

Figure 7: Type Checking Returns

$\boxed{\Psi \mid \Theta \vdash \tau : \Delta \qquad \Psi \vdash \nu\langle\Theta\rangle : \Delta}$

$$\frac{\begin{array}{c}\Psi \vdash \hat{\nu}\langle\nu_1, \ldots, \nu_n\rangle : \Delta \qquad \Psi \vdash \hat{\nu}\langle\nu_1, \ldots, \nu_n\rangle \textbf{ extends } \tau \qquad \Psi \mid \nu_1, \ldots, \nu_n \vdash \tau : \Delta' \\ \text{for all } \nu\sigma \text{ in } \Delta, \quad \text{for all } \nu'\sigma' \text{ in } \Delta', \quad \nu = \nu' \quad \text{implies} \quad \Psi \mid \nu_1, \ldots, \nu_n \vdash \sigma \approx \sigma'\end{array}}{\Psi \mid \Theta \vdash \hat{\nu}\langle\tau_1, \ldots, \tau_n\rangle : (\Delta \cup \Delta')[\nu_1 \mapsto \tau_1, \ldots, \nu_n \mapsto \tau_n]}$$

$$\frac{\textbf{interface } \hat{\nu}\langle\Theta\rangle \textbf{ extends } \tau\ \{\Delta\} \text{ in } \Psi}{\Psi \vdash \hat{\nu}\langle\Theta\rangle : \Delta} \qquad \frac{\textbf{class } \hat{\nu}\langle\Theta\rangle \textbf{ extends } \tau\ \{\Delta\} \text{ in } \Psi}{\Psi \vdash \hat{\nu}\langle\Theta\rangle : \Delta}$$

Figure 8: Method-Context Lookup

$\boxed{\Psi \mid \Theta \vdash \tau : \nu}$

$$\frac{\textbf{class } \hat{\nu}\langle\nu_1, \ldots, \nu_n\rangle \textbf{ extends } \tau\ \{\ldots, \nu\sigma, \ldots\} \text{ in } \Psi}{\Psi \vdash \hat{\nu}\langle\tau_1, \ldots, \tau_n\rangle : \nu}$$

Figure 9: Method-Implemented Check

$\boxed{\Psi \mid \Delta \mid \Gamma \vdash i : \Psi \qquad \Psi \mid \Delta \mid \Gamma \vdash c : \Psi \mid \Delta}$

$$\text{**}\ \frac{\begin{array}{c}\Psi \mid \Theta \vdash_{\top} \tau \\ \Psi' = \textbf{interface } \nu\langle\Theta\rangle \textbf{ extends } \tau\ \{\nu_1\sigma_1, \ldots, \nu_n\sigma_n\} \\ \text{for all } i, \qquad \Psi, \Psi' \mid \Theta \vdash \sigma_i \\ \Psi, \Psi' \mid \Theta \vdash \nu\langle\Theta\rangle : \Delta' \\ \text{for all } \bar{\nu}\langle\bar{\Theta}\rangle(\bar{\Gamma}) : \bar{\tau} \text{ in } \Delta', \quad \text{for all } \bar{\nu}' \text{ in } \bar{\Theta}, \quad \bar{\nu}' \text{ not in } \Theta\end{array}}{\Psi \mid \Delta \mid \Gamma \vdash \textbf{interface } \nu\langle\Theta\rangle \textbf{ extends } \tau\ \{\textbf{fun } \nu_1\sigma_1; \ldots; \textbf{fun } \nu_n\sigma_n;\} : \Psi'}$$

$$\frac{\begin{array}{c}\Psi \mid \Theta \vdash_{\hat{\tau}} \tau \\ \Psi' = \textbf{class } \nu\langle\Theta\rangle \textbf{ extends } \tau\ \{\nu_1\sigma_1; \ldots; \nu_n\sigma_n;\} \qquad \Delta' = \nu\langle\Theta\rangle(\hat{\Gamma}) : \nu\langle\Theta\rangle \\ \text{for all } i, \qquad \Psi, \Psi' \mid \Theta \vdash \sigma_i \\ \Psi, \Psi' \mid \Theta \vdash \hat{\Gamma} \qquad \hat{\Gamma}_0 = \hat{\Gamma} \qquad \text{for all } i, \qquad \Psi, \Psi' \mid \Theta \mid \Delta, \Delta' \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash s_i : \hat{\Gamma}_i \\ \hat{\tau}(e_1, \ldots, e_k) = \top() \qquad \text{or} \qquad \Psi, \Psi' \mid \Theta \mid \Delta, \Delta' \mid \Gamma, \hat{\Gamma}_m \vdash \hat{\tau}(e_1, \ldots, e_k) : \hat{\tau} \\ \Psi, \Psi' \mid \Theta \vdash \nu\langle\Theta\rangle : \Delta'' \\ \text{for all } \bar{\nu}\langle\bar{\Theta}\rangle(\bar{\Gamma}) : \bar{\tau} \text{ in } \Delta'', \qquad \text{for all } \bar{\nu}' \text{ in } \bar{\Theta}, \qquad \bar{\nu}' \text{ not in } \Theta \\ \text{for all } i, \quad \sigma_i = \langle\Theta_i\rangle(\Gamma_i) : \tau_i \quad \Psi, \Psi' \mid \Theta, \Theta_i \mid \Delta, \Delta', \Delta'' \mid \Gamma, \hat{\Gamma}_m \mid \Gamma_i \vdash_{\tau_i}^{\textbf{true}} \hat{s}_i : \Gamma'_i \\ \text{for all } \hat{\nu}\sigma \text{ in } \Delta'', \qquad \Psi, \Psi' \mid \Theta \vdash \nu\langle\Theta\rangle : \hat{\nu}\end{array}}{\Psi \mid \Delta \mid \Gamma \vdash \textbf{class } \nu\langle\Theta\rangle(\hat{\Gamma}) \textbf{ extends } \tau\ \{s_1 \ldots s_m\ \textbf{super}(e_1, \ldots, e_k);\ \textbf{fun } \nu_1\sigma_1\ \hat{s}_1\ \ldots\ \textbf{fun } \nu_n\sigma_n\ \hat{s}_n\} : \Psi' \mid \Delta'}$$

Figure 10: Class and Interface Checking

$\boxed{\Psi \mid \Delta \mid \Gamma \vdash p \qquad \vdash p}$

$$\text{**}\ \frac{\Psi \mid \varnothing \mid \Delta \mid \Gamma \mid \varnothing \vdash^{\mathbf{true}}_{\texttt{Integer}} s : \hat{\Gamma}}{\Psi \mid \Delta \mid \Gamma \vdash s}$$

$$\text{**}\ \frac{\hat{\Gamma}_0 = \varnothing \qquad \text{for all } i, \qquad \Psi \mid \varnothing \mid \Delta \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash^{b_i}_{\texttt{Integer}} s_i : \hat{\Gamma}_i \qquad \Psi \mid \Delta \mid \Gamma, \hat{\Gamma}_n \vdash p}{\Psi \mid \Delta \mid \Gamma \vdash s_1 \ldots s_n\ p}$$

$$\frac{\begin{array}{c}\Delta' = \Delta, \nu_1\langle\Theta_1\rangle(\Gamma_1) : \tau_1, \ldots, \nu_n\langle\Theta_n\rangle(\Gamma_n) : \tau_n \\ \text{for all } i, \qquad \Psi \mid \Theta_i \vdash \Gamma_i \qquad \Psi \mid \Theta_i \vdash \tau_i \qquad \Psi \mid \Theta_i \mid \Delta' \mid \Gamma \mid \Gamma_i \vdash^{\mathbf{true}}_{\tau_i} s_i : \hat{\Gamma}_i \\ \Psi \mid \Delta' \mid \Gamma \vdash p\end{array}}{\Psi \mid \Delta \mid \Gamma \vdash \mathbf{fun}\ \nu_1\langle\Theta_1\rangle(\Gamma_1) : \tau_1\ s_1 \ldots \mathbf{fun}\ \nu_n\langle\Theta_n\rangle(\Gamma_n) : \tau_n\ s_n\ p}$$

$$\frac{\Psi \mid \Delta \mid \Gamma \vdash i : \Psi' \qquad \Psi, \Psi' \mid \Delta \mid \Gamma \vdash p}{\Psi \mid \Delta \mid \Gamma \vdash i\ p} \qquad \frac{\Psi \mid \Delta \mid \Gamma \vdash c : \Psi' \mid \Delta' \qquad \Psi, \Psi' \mid \Delta, \Delta' \mid \Gamma \vdash p}{\Psi \mid \Delta \mid \Gamma \vdash c\ p}$$

$$\frac{\begin{array}{ll}\text{**}\ \Psi_0 = & \mathbf{class}\ \texttt{Boolean}\langle\rangle\ \mathbf{extends}\ \top\ \{ \\ & \quad \texttt{negate}\langle\rangle() : \texttt{Boolean}\langle\rangle; \\ & \quad \texttt{and}\langle\rangle(\texttt{that} : \texttt{Boolean}\langle\rangle) : \texttt{Boolean}\langle\rangle; \\ & \quad \texttt{or}\langle\rangle(\texttt{that} : \texttt{Boolean}\langle\rangle) : \texttt{Boolean}\langle\rangle; \\ & \quad \texttt{lessThan}\langle\rangle(\texttt{that} : \texttt{Boolean}\langle\rangle, \texttt{strict} : \texttt{Boolean}\langle\rangle) : \texttt{Boolean}\langle\rangle; \\ & \quad \texttt{equals}\langle\rangle(\texttt{that} : \texttt{Boolean}\langle\rangle) : \texttt{Boolean}\langle\rangle; \\ & \}, \\ & \mathbf{class}\ \texttt{Integer}\langle\rangle\ \mathbf{extends}\ \top\ \{ \\ & \quad \texttt{negative}\langle\rangle() : \texttt{Integer}\langle\rangle; \\ & \quad \texttt{times}\langle\rangle(\texttt{factor} : \texttt{Integer}\langle\rangle) : \texttt{Integer}\langle\rangle; \\ & \quad \texttt{plus}\langle\rangle(\texttt{summand} : \texttt{Integer}\langle\rangle) : \texttt{Integer}\langle\rangle; \\ & \quad \texttt{minus}\langle\rangle(\texttt{subtrahend} : \texttt{Integer}\langle\rangle) : \texttt{Integer}\langle\rangle; \\ & \quad \texttt{lessThan}\langle\rangle(\texttt{that} : \texttt{Integer}\langle\rangle, \texttt{strict} : \texttt{Boolean}\langle\rangle) : \texttt{Boolean}\langle\rangle; \\ & \quad \texttt{equals}\langle\rangle(\texttt{that} : \texttt{Integer}\langle\rangle) : \texttt{Boolean}\langle\rangle; \\ & \} \\ \text{**}\ \Delta_0 = & \varnothing \\ \text{**}\ \Gamma_0 = & \texttt{input} : \texttt{Integer}\langle\rangle\end{array}}{\begin{array}{c}\Psi_0 \mid \Delta_0 \mid \Gamma_0 \vdash p \\ \hline \vdash p\end{array}}$$

Figure 11: Program Checking