H# Official Documentation

The official specification for H#

Contents

Abstract

This document is the official document for the H-Sharp (H#) programming language. The H stands for hybrid - as this is a multi-paradigm programming language heavily inspired by C# and Scala. This document will only contain the grammar of the language, the operational semantics as well as the type semantics of the language. The operational semantics may be explained with code samples - but obvious uses will not be explained.

This document will also contain the byte-instruction semantics. The official compiler, is written in C# for productivity purposes while the Virtual Machine is implemented using C++. At no point will this document be documenting the internal processes of those applications.

Grammar

The offical H# grammar. Note: $\mathit{Id} \in \textsc{VarEnv}$ and $\mathit{TypeId} \in \textsc{TypeEnv}$. Both refer to the same grammatical definition, defined by the regular expression:

$$(\_|[\texttt{a}-\texttt{Z}])^{+}(\_|[\texttt{0}-\texttt{9}]|[\texttt{a}-\texttt{Z}])^{*}$$

If multiple elements can occur - and it'd be convenient, the element may be suffixed with 'n' to show it may contain n of such elements. $n \in \{0, 1, 2, \dots\}$. The notation $e_0\texttt{,} \dots \texttt{,} e_n$ represents the one to nth element with a specific separator. The full grammar is defined as follows:

$$
\begin{array}{lll}
\mathit{CompileUnit} & ::= & \mathit{CompileUnitElement} \\
\\
\mathit{CompileUnitElement} & ::= & \mathit{CompileUnitElement}\ \mathit{CompileUnitElement} \\
 & | & \mathit{Directive} \mid \mathit{Scope} \mid \mathit{ScopeElement} \mid \mathit{Declaration} \\
\\
\mathit{Scope} & ::= & \{\ \mathit{ScopeElement}\ \} \\
\\
\mathit{ScopeElement} & ::= & \mathit{ScopeElement}\ \mathit{ScopeElement} \\
 & | & \mathit{Expr}\texttt{;} \mid \mathit{Statement} \mid \mathit{VarDeclaration}\texttt{;} \\
\\
\mathit{Expr} & ::= & \mathit{Expr} \mid \texttt{(}\mathit{Expr}\texttt{)} \mid \mathit{Scope} \mid \mathit{LambdaExpr} \\
 & | & \mathit{Expr}\ \mathit{BinaryOp}\ \mathit{Expr} \mid \mathit{UnaryOp}\ \mathit{Expr} \mid \mathit{Id}\ \mathit{UnaryOp} \\
 & | & \mathit{Expr}\ \texttt{?}\ \mathit{Expr}\ \texttt{:}\ \mathit{Expr} \\
 & | & \mathit{Id} \mid \mathit{Name} \mid \texttt{this} \mid \texttt{base} \\
 & | & \mathit{Expr}\texttt{(}\mathit{Argument}\texttt{)} \mid \mathit{Expr}\texttt{[}\mathit{Expr}\texttt{]} \\
 & | & \texttt{new}\ \mathit{TypeId}\texttt{(}\mathit{Argument}\texttt{)} \mid \texttt{new}\ \mathit{TypeId}\texttt{[}\mathit{Argument}\texttt{]} \\
 & | & \mathit{Expr}\texttt{.}\mathit{Id} \mid \mathit{Expr}\texttt{?.}\mathit{Id} \\
 & | & \mathit{Expr}\texttt{.}\mathit{Id}\texttt{(}\mathit{Argument}\texttt{)} \mid \mathit{Expr}\texttt{?.}\mathit{Id}\texttt{(}\mathit{Argument}\texttt{)} \\
 & | & \mathit{Expr}\ \texttt{is}\ \mathit{TypeID} \mid \mathit{Expr}\ \texttt{is null}\ \mathit{Expr}\ \texttt{is}\ \mathit{TypeID}\ \mathit{Id} \\
 & | & \mathit{Expr}\ \texttt{is}\ \mathit{TypeID}\ \mathit{Id}\ \texttt{where}\ \mathit{Expr} \\
 & | & \mathit{Expr}\ \texttt{is not}\ \mathit{TypeID} \mid \mathit{Expr}\ \texttt{is not null} \\
 & | & \mathit{Expr}\ \texttt{is not}\ \mathit{TypeID}\ \mathit{Id}\ \texttt{where}\ \mathit{Expr} \\
 & | & \mathit{Expr}\ \texttt{as}\ \mathit{TypeId} \\
 & | & \texttt{(}\mathit{TypeID}\texttt{)}\ \mathit{Expr} \\
 & | & \texttt{sizeof(}\mathit{TypeId}\texttt{)} \mid \texttt{addressof(}\mathit{Id}\texttt{)} \mid \texttt{typeof(}\mathit{Expr}\texttt{)} \\
 & | & \mathit{Assignment} \\
 & | & \mathit{Literal} \\
\\
\mathit{LambdaExpr} & ::= & \texttt{(}\mathit{Param}\texttt{)}\ \texttt{=>}\ \mathit{Expr} \\
\\
\mathit{Directive} & ::= & \texttt{type}\ \mathit{Id}\ \texttt{=}\ \mathit{TypeId}\texttt{;} \\
 & | & \texttt{using}\ \mathit{Name}\texttt{;} \mid \texttt{using}\ \mathit{TypeId}\ \texttt{from}\ \mathit{Name}\texttt{;} \\
 & | & \mathit{NamespaceDecl} \\
\\
\mathit{NamespaceDecl} & ::= & \texttt{namespace}\ \mathit{Name}\ \mathit{Scope} \\
\\
\mathit{Statement} & ::= & \mathit{Assignment}\texttt{;} \mid \mathit{ControlStatement} \mid \mathit{MatchStatement}\texttt{;} \\
 & | & \mathit{TryCatchStatement} \\
\\
\mathit{ControlStatement} & ::= & \texttt{if}\ \mathit{Expr}\ \mathit{Scope} \\
 & | & \texttt{if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else}\ \mathit{Scope} \\
 & | & \texttt{if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else if}\ \mathit{Expr}\ \mathit{Scope} \\
 & | & \texttt{if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else}\ \mathit{Scope} \\
 & | & \texttt{while}\ \mathit{Expr}\ \mathit{Scope} \\
 & | & \texttt{do}\ \mathit{Scope}\ \texttt{while}\ \mathit{Expr}\texttt{;} \\
 & | & \texttt{for (}\mathit{Assignment}\texttt{;}\ \mathit{Expr}\texttt{;}\ \mathit{Expr}\texttt{)}\ \mathit{Scope} \\
 & | & \texttt{for (}\mathit{VarDeclaration}\texttt{;}\ \mathit{Expr}\texttt{;}\ \mathit{Expr}\texttt{)}\ \mathit{Scope} \\
 & | & \texttt{for (}\mathit{TypeId}\ \mathit{Id}\ \texttt{:}\ \mathit{Expr}\texttt{)}\ \mathit{Scope} \\
 & | & \texttt{throw}\ \mathit{TypeID}\texttt{(}\mathit{Argument}\texttt{)} \\
 & | & \texttt{return}\ \mathit{Expr}\texttt{;} \\
 & | & \texttt{break;}
\end{array}
$$

```
MatchStatement      ::= Expr match { MatchCase }

MatchCase           ::= MatchCase, MatchCase
                     |  case Literal => Expr | case TypeId => Expr
                     |  case TypeId Id => Expr | case TypeId Id when Expr => Expr
                     |  case (MatchCaseId) => Expr
                     |  case (MatchCaseId) when Expr => Expr
                     |  case TypeId (MatchCaseId) => Expr
                     |  case TypeId (MatchCaseId) when Expr => Expr
                     |  case _ => Expr

MatchCaseId         ::= MatchCaseId, MatchCaseId
                     |  Id | _

TryCatchStatement   ::= try Scope catch (TypeId Id) Scope
                     |  try Scope catch (TypeId Id) when Expr Scope
                     |  try Scope catch (TypeId Id) Scope finally Scope
                     |  try Scope catch (TypeId Id) when Expr Scope finally Scope

Assignment          ::= Id = Expr
                     |  Id += Expr | Id -= Expr
                     |  Id *= Expr | Id /= Expr
                     |  Id &= Expr | Id |= Expr
                     |  Id %= Expr

Declaration         ::= VarDecl | FuncDecl
                     |  ClassDecl | StaticClassDecl | TraitDecl
                     |  UnionDecl | EnumDecl | StructDecl
                     |  TraitUniversal

ClassDecl           ::= Modifier class Id ClassBody
                     |  Modifier class Id : ParamType ClassBody
                     |  Modifier class Id(Param) ClassBody
                     |  Modifier class Id(Param) : ParamType ClassBody

StructDecl          ::= Modifier struct Id ClassBody
                     |  Modifier struct Id : ParamType ClassBody
                     |  Modifier struct Id(Param) ClassBody
                     |  Modifier struct Id(Param) : ParamType ClassBody

StaticClassDecl     ::= object Id ClassBody
                     |  AccessMod object Id ClassBody
                     |  object Id(Param) ClassBody
                     |  AccessMod object Id(Param) ClassBody

TraitDecl           ::= trait Id ClassBody
                     |  AccessMod trait Id ClassBody

TraitUniversal      ::= trait TypeId for TypeId Scope

ClassBody           ::= ; | { ClassMember }

ClassMember         ::= ClassMember ClassMember
                     |  Id(Param) FuncBody | AcessMod Id(Param) FuncBody
                     |  VarDecl; | AcessMod VarDecl;
                     |  FuncDecl | ClassDecl | UnionDecl | EnumDecl
                     |  event TypeId id; | AccessMod event TypeId id;
```

$$\begin{array}{lll}
\mathit{VarDecl} & ::= & \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \mathit{TypeId}\texttt{[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \mathit{TypeId}\texttt{[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \mathit{TypeId}\texttt{[]}\ \mathit{Id}\,\texttt{=}\,\mathit{ValueListInitializer} \mid \mathit{StorageMod}\,\mathit{TypeId}\texttt{[]}\ \mathit{Id}\,\texttt{=}\,\mathit{ValueListInitializer} \\
 & \mid & \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Initializer} \mid \mathit{StorageMod}\ \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Initializer} \\
 & \mid & \mathit{TypeId}\ \mathit{Id} \mid \mathit{StorageMod}\ \mathit{TypeId}\ \mathit{Id} \\
 & \mid & \mathit{TypeId}\texttt{[]}\ \mathit{Id} \mid \mathit{StorageMod}\ \mathit{TypeId}\texttt{[]}\ \mathit{Id} \\
 & \mid & \mathit{LambdaType}\ \mathit{Id}\ \texttt{=}\ \mathit{LambdaExpr} \\
 & \mid & \mathit{TupleDecl}
\end{array}$$

$$\begin{array}{lll}
\mathit{TupleDecl} & ::= & \texttt{(}\mathit{ParamType}\texttt{)}\ \mathit{Id}\ \texttt{=}\ \texttt{(}\mathit{Argument}\texttt{)} \\
 & \mid & \texttt{(}\mathit{Param}\texttt{)}\ \texttt{=}\ \texttt{(}\mathit{Argument}\texttt{)} \\
 & \mid & \texttt{(}\mathit{ParamType}\texttt{)[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \texttt{(}\mathit{Param}\texttt{)[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr}
\end{array}$$

$$\begin{array}{lll}
\mathit{FuncDecl} & ::= & \mathit{Id}\texttt{(}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{AccessMod}\ \mathit{Id}\texttt{(}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{Id}\ \texttt{= (}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{AccessMod}\ \mathit{Id}\ \texttt{= (}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{AccessMod}\ \texttt{const}\ \mathit{Id}\ \texttt{= (}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody}
\end{array}$$

$$\begin{array}{lll}
\mathit{FuncBody} & ::= & \mathit{Scope} \mid \texttt{=>}\ \mathit{Expr}
\end{array}$$

$$\begin{array}{lll}
\mathit{FuncParam} & ::= & \mathit{FuncParam}\texttt{,}\ \mathit{FuncParam} \mid \mathit{Param} \\
 & \mid & \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Literal}
\end{array}$$

$$\begin{array}{lll}
\mathit{UnionDecl} & ::= & \texttt{union}\ \mathit{Id}\ \{\ \mathit{UnionMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{union}\ \mathit{Id}\ \{\ \mathit{UnionMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{static union}\ \mathit{Id}\ \{\ \mathit{UnionMember}\ \}
\end{array}$$

$$\begin{array}{lll}
\mathit{UnionMember} & ::= & \mathit{UnionMember}\ \mathit{UnionMember} \\
 & \mid & \mathit{TypeId}\ \mathit{Id};
\end{array}$$

$$\begin{array}{lll}
\mathit{EnumDecl} & ::= & \texttt{enum}\ \mathit{Id}\ \{\ \mathit{EnumBodyMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{enum}\ \mathit{Id}\ \{\ \mathit{EnumBodyMember}\ \} \\
 & \mid & \texttt{enum}\ \mathit{Id}\texttt{(}\mathit{EnumMember}\texttt{)}\{\ \mathit{EnumBodyMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{enum}\ \mathit{Id}\ \texttt{(}\mathit{EnumMember}\texttt{)}\ \{\ \mathit{EnumBodyMember}\ \}
\end{array}$$

$$\begin{array}{lll}
\mathit{EnumBodyMember} & ::= & \mathit{EnumMember} \mid \mathit{FuncDecl} \mid \mathit{FuncDecl}\ \mathit{EnumBodyMember}
\end{array}$$

$$\begin{array}{lll}
\mathit{EnumMember} & ::= & \mathit{EnumMember}\texttt{,}\ \mathit{EnumMember} \\
 & \mid & \mathit{Id} \mid \mathit{Id}\ \texttt{=}\ \mathit{LiteralNoNull}
\end{array}$$

$$
\begin{array}{lcl}
\mathit{Initializer} & ::= & \mathit{ValueListInitializer} \mid \{\ \mathit{KeyValueListElement}\ \} \\
 & \mid & \{\ \mathit{IdValueListElement}\ \} \\
\mathit{ValueListInitializer} & ::= & \{\ \mathit{ValueListElement}\ \} \\
\mathit{ValueListElement} & ::= & \mathit{Expr} \mid \mathit{ValueListElement},\ \mathit{ValueListElement} \\
\mathit{IdValueListElement} & :: & \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{IdValueListElement},\ \mathit{IdValueListElement} \\
\mathit{KeyValueListElement} & :: & \texttt{[}\mathit{Expr}\texttt{]}\ \texttt{=}\ \mathit{Expr} \mid \mathit{KeyValueListElement},\ \mathit{KeyValueListElement} \\
\mathit{Modifier} & ::= & \mathit{StorageMod} \mid \mathit{AccessMod} \mid \mathit{TypeMod} \mid \mathit{CompilerHintMod} \mid \epsilon \\
 & \mid & \mathit{AccessMod}\ \mathit{StorageMod} \mid \mathit{AccessMod}\ \texttt{abstract}\ \texttt{variant} \\
 & \mid & \mathit{AccessMod}\ \mathit{CompilerHintMod} \mid \mathit{AccessMod}\ \mathit{StorageMod}\ \mathit{CompilerHintMod} \\
 & \mid & \mathit{AccessMod}\ \mathit{StorageMod}\ \mathit{CompilerHintMod} \mid \mathit{AccessMod}\ \mathit{TypeMod} \\
\mathit{StorageMod} & ::= & \texttt{const} \mid \texttt{static} \mid \texttt{override} \mid \texttt{virtual} \mid \texttt{lazy} \\
\mathit{TypeMod} & ::= & \texttt{variant} \mid \texttt{abstract} \\
\mathit{CompilerHintMod} & ::= & \texttt{inline} \mid \texttt{final} \mid \texttt{constexpr} \\
\mathit{AccessMod} & ::= & \texttt{public} \mid \texttt{private} \mid \texttt{protected} \mid \texttt{internal} \mid \texttt{external}
\end{array}
$$

Param	::=	*Param*, *Param* $\vert$ *TypeId Id* $\vert$ `const` *TypeId Id* $\vert$ *ParamType Id*
ParamType	::=	*TypeId* $\vert$ *ParameterizedType* $\vert$ *ParamType*, *ParamType*
LambdaType	::=	`(`*ParamType*`):` *TypeId* $\vert$ *TypeId* : *TypeId*
ParameterizedType	::=	<*TypeId*>
Argument	::=	*Expr* $\vert$ *Expr*, *Expr*
Name	::=	*Id* $\vert$ *Name*.*Name*
BinaryOp	::=	`+` $\vert$ `-` $\vert$ `*` $\vert$ `/` $\vert$ % $\vert$ < $\vert$ > $\vert$ <= $\vert$ >= $\vert$ `==` $\vert$ `!=` $\vert$ $\Vert$ $\vert$ && $\vert$ $\vert$ $\vert$ & $\vert$ << $\vert$ >> $\vert$ `=>` $\vert$ `::` $\vert$ `??` $\vert$ `..`
UnaryOp	::=	`-` $\vert$ `!` $\vert$ `#` $\vert$ `++` $\vert$ `--` $\vert$ `*`
LiteralNoNull	::=	*IntLit* $\vert$ *FloatLit* $\vert$ *DoubleLit* $\vert$ *BoolLit* $\vert$ *CharLit* $\vert$ *StringLit*
Literal	::=	*LiteralNoNull* $\vert$ *NullLit*
Letter	::=	[`a-Z`]
Digit	::=	[`0-9`]
IntLit	::=	$Digit^+$
FloatLit	::=	$Digit^+$.$Digit^+$`f`
DoubleLit	::=	$Digit^+$.$Digit^+$
CharLit	::=	'*Letter*' $\vert$ '\Letter'
StringLit	::=	¨$(Letter \vert Digit)^*$¨
BoolLit	::=	`true` $\vert$ `false`
NullLit	::=	`null`

Operational Semantics

General Semantics

$$\textsc{VariableLookup}\quad \frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = v \neq (\ell, \omega, \sigma)}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}$$

$$\textsc{HeapObjectLookup}\quad \frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = (\ell, \omega, \sigma) \quad \sigma(\ell) = v \quad \omega = \mathtt{O}}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}$$

$$\textsc{HeapStringLookup}\quad \frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = (\ell, \omega, \sigma) \quad \sigma(\ell) = v \quad \omega = \mathtt{S}}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}$$

$$\textsc{HeapArrayLookup}\quad \frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = (\ell, \omega, \sigma) \quad \sigma(\ell) = v \quad \omega = \mathtt{A}}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}$$

Classes

Objects

Structs

Union

Enum

Traits

Type Semantics

$$\begin{aligned} \theta &= \mathit{TypeEnv} \\ \gamma &= \mathit{TypeLookupEnv} \\ \eta &= \mathit{ReferenceTypeEnv} \end{aligned}$$

$$\begin{aligned} &\theta, \gamma, \eta \vdash \mathit{Expr} \ : \ \mathit{Type} \\ &\theta, \gamma, \eta \vdash \mathit{Decl} \ : \ \theta, \gamma \\ &\theta, \gamma, \eta \vdash \mathit{Decl} \ : \ \theta, \gamma, \eta \end{aligned}$$

`int <: float <: double <: INumeric`

$$\texttt{typeof}(t, \gamma, \eta) = \begin{cases} \texttt{Ref}(\gamma(t)) & t \in \gamma, \gamma(t) \in \eta \\ \gamma(t) & t \in \gamma, \gamma(t) \notin \eta \\ t & \text{otherwise}^1 \end{cases}$$

$$\texttt{base}(\tau_1, \tau_2) = \begin{cases} \tau_1 & t_2 <: t_1 \\ \tau_2 & t_1 <: t_2 \end{cases}$$

Additionally, we note that θ is local to the expression while γ and η are global environments[2]. Additionally, $\eta \subset \gamma$ such that no element in η can be an atomic type and must be a type that is defined during compile-time. Another thing to note is τ consists of the tuple (ϕ, μ). Where ϕ is the set of all fields belonging to the type. Unless it's an atomic type, in which case this will be the empty set. μ is the set of all methods.

T-IntLit
$$\frac{i \in \mathbb{N}}{\theta, \gamma, \eta \vdash i \ : \ \texttt{int}}$$

T-Identifier
$$\frac{\tau = \theta(id) \qquad id \in \theta}{\theta, \gamma, \eta \vdash id \ : \ \tau}$$

T-Addition
$$\frac{\theta, \gamma, \eta \vdash e_1 \ : \ \tau_1 \qquad \theta, \gamma, \eta \vdash e_2 \ : \ \tau_2 \qquad \tau' = \texttt{base}(\tau_1, \tau_2) \qquad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ + } e_2 \ : \ \tau'}$$

T-Subtraction
$$\frac{\theta, \gamma, \eta \vdash e_1 \ : \ \tau_1 \qquad \theta, \gamma, \eta \vdash e_2 \ : \ \tau_2 \qquad \tau' = \texttt{base}(\tau_1, \tau_2) \qquad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ - } e_2 \ : \ \tau'}$$

T-Multiplication
$$\frac{\theta, \gamma, \eta \vdash e_1 \ : \ \tau_1 \qquad \theta, \gamma, \eta \vdash e_2 \ : \ \tau_2 \qquad \tau' = \texttt{base}(\tau_1, \tau_2) \qquad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ * } e_2 \ : \ \tau'}$$

T-Division
$$\frac{\theta, \gamma, \eta \vdash e_1 \ : \ \tau_1 \qquad \theta, \gamma, \eta \vdash e_2 \ : \ \tau_2 \qquad \tau' = \texttt{base}(\tau_1, \tau_2) \qquad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ / } e_2 \ : \ \tau'}$$

T-DeclVar
$$\frac{\theta, \gamma, \eta \vdash e \ : \ \tau \qquad \theta' = \theta[x \mapsto \tau'] \qquad \tau' = \texttt{typeof}(t, \gamma, \eta) \qquad \tau <: \tau'}{\theta, \gamma, \eta \vdash t \ x \texttt{ = } e \ : \ \theta', \gamma, \eta}$$

T-NewObject
$$\frac{\tau = \texttt{typeof}(t, \gamma, \eta) \qquad \theta, \gamma, \eta \vdash e_1, \ldots, e_n \ : \ \tau_1, \ldots, \tau_n}{\theta, \gamma, \eta \vdash \texttt{new } t\texttt{(}e_1, \ldots, e_n\texttt{)} \ : \ \tau}$$

T-FieldAccess
$$\frac{\theta, \gamma, \eta \vdash e \ : \ \tau \qquad \tau = (\phi, \mu) \qquad \tau' = \phi(id) \qquad id \in \phi}{\theta, \gamma, \eta \vdash e\texttt{.}id \ : \ \tau'}$$

T-MethodAccess
$$\frac{\theta, \gamma, \eta \vdash e \ : \ \tau \qquad \tau = (\phi, \mu) \qquad \tau' = \mu(id) \qquad id \in \mu}{\theta, \gamma, \eta \vdash e\texttt{.}id \ : \ \tau'}$$

When inferring the type of a scope - the whole set of control paths must be considered. Additionally, the last expression of a scope is returned to the calling scope.

[1] Atomic type, such as `int`, `bool` or `char`.

[2] With respect to current domain as elements in γ may have local definitions not globally visible.

Compilation Semantics

Bytecode Semantics