

Static Semantics of Hobbes

Alan Jeffrey
CTI, DePaul University
<http://fpl.cs.depaul.edu/ajeffrey/>

17 November 2003

1 Introduction

This is the static semantics specification of Hobbes, a strongly-typed class-based multi-threaded object-oriented language.

Hobbes is a 'pure' object-oriented language: everything is an object, and there are no imperative features such as variable assignment or while loops. Instead, recursion has to be used, for example, a recursive factorial function is written:

```
import "Base.hob";

class Factorial {
  // The non-tail-recursive implementation of factorial.
  method fact (n : Integer) : Integer {
    if (n <= 1) {
      return 1;
    } else {
      return n * this.fact (n-1);
    }
  }
}

object F : Factorial {
}

thread Main {
  let x : Integer = F.fact (5);
}
```

```
    Out.println ("5! = " + $x);  
}
```

A program can contain any number of class, interface, object and thread declarations.

Week 6 begin

Week 6 end

The static semantics is specified by defining *typing judgements* such as:

$$\Gamma \vdash B : T$$

where Γ is a *typing context* giving information such as the types of variables. For example:

$$b : \text{Boolean}, x : \text{Integer}, y : \text{Integer} \vdash \text{if } (b) \{ \text{return } x; \} \text{ else } \{ \text{return } y \} : \text{Integer}$$

The main typing judgement is one which says that an entire program P is well-typed:

$$\Gamma \vdash P : \text{program}$$

For example:

Nothing : Unit \vdash thread Main { let x = 1+2; return Nothing; } : program

Hobbes is *strongly typed*, meaning that typings do not ‘go wrong’ at run-time. This is formally proved as a *subject reduction* proof.

2 Conventions used in this specification

This specification is based on the *Hobbes core language specification*.

2.1 Naming conventions

We let:

- A ranges over TypeVar.
- B and C range over Block.
- D and E range over Exp.

Week 8 begin

Week 8 end

- F, G and H range over Field.
- M and N range over Method.
- P and Q range over Program.
- R, S, T and U range over Type.
- V and W range over Val.
- X and Y range over Var.
- i and j range over IntegerLiteral.
- l and m range over MethodId.
- s and t range over CharSeq.
- u ranges over Mutability.
- a, b, c, d and e range over GlobalId.
- f, g, x, y and z range over LocalId.

For example, one possible Program is:

```
thread a { let X = V.m (W1, ..., Wn); B }
```

3 Typing contexts

A *typing context* gives information required during type checking (in an implementation, it is called a *symbol table*). For example, it includes the types for free variables, such as:

```
b : Boolean, x : Integer, y : Integer ⊢ if (b) { return x; } else { return y } : Integer
```

It also contains information about class, interface, local type, native class, object and thread declarations.

3.1 Definition of a typing context

A *typing context* is given by the grammar:

Week 7 begin

Week 8 begin

Week 8 end

Context ::=
ContextAtomic ", "...," ContextAtomic

ContextAtomic ::=
VarTyped |
TypeVar |
ClassVar |
ContextClass |
ContextInterface |
ContextNative |
ContextObject |
ContextThread

Week 8 begin

Week 9 begin

Week 9 end

Week 6 begin

Week 6 end

ClassVar ::=
"type" LocalId "<#" TypeGlobal ";"

Week 9 begin

Week 9 end

ContextClass ::=
"class" GlobalId
TypeParams
ExtendsClass
ImplementsInterfaces

Week 8 begin

Week 8 end

Week 9 begin

ExtendedClass

```
"{"  
  FieldAbstract ... FieldAbstract  
  MethodAbstract ... MethodAbstract  
"}"
```

Week 9 end

Week 5 begin

Week 5 end

Week 6 begin

ContextInterface ::=

```
"interface" GlobalId  
TypeParams  
ExtendsInterfaces  
"{"  
  MethodAbstract ... MethodAbstract  
"}"
```

Week 8 begin

Week 8 end

Week 6 end

ContextNative ::=

```
"native" "class" GlobalId  
TypeParams  
"{"  
  MethodAbstract ... MethodAbstract  
"}"
```

Week 8 begin

Week 8 end


```
ContextObject ::=
  "object" GlobalId
  TypeParams
  ":" Type
```

Week 8 begin

Week 8 end

```
ContextThread ::=
  "thread" GlobalId
```

We let Γ and Δ range over `Context`, and let γ and δ range over `ContextAtomic`.

3.2 Identifier of an atomic context

The *identifier* of an atomic typing context γ is the name $\text{id}(\gamma)$ that it provides a definition for:

Week 9 begin

Identifier Class Var: $\text{id}(\text{type } a < \# T;) = a$

Week 9 end

Identifier Class: $\text{id}(\text{class } a \dots) = a$

Week 6 begin

Identifier Interface: $\text{id}(\text{interface } a \dots) = a$

Week 6 end

Identifier Native: $\text{id}(\text{native class } a \dots) = a$

Identifier Object: $\text{id}(\text{object } a \dots) = a$

Identifier Thread: $\text{id}(\text{thread } a \dots) = a$

The identifier of a field F is the name $\text{id}(F)$ of the field, and the identifier of a method M is the name $\text{id}(M)$ of the method.

Identifier Field Abstract $\text{id}(u \text{ field } f : T;) = f$

Identifier Field Concrete $\text{id}(f = V) = f$

Identifier Method Abstract $\text{id}(\text{method } x.m \dots;) = m$

Identifier Method Concrete $\text{id}(\text{method } x.m \dots \{ B \}) = m$

3.3 Domain of a typing context

The *domain* of a typing context is all the identifiers with definitions in the context:

Domain Context: $\text{dom}(\gamma_1, \dots, \gamma_n) = \{\text{id}(\gamma_1), \dots, \text{id}(\gamma_n)\}$

3.4 Applying a typing context to an identifier

We define $\Gamma(x)$ to be the type of the variable declared in Γ , when $x \in \text{dom}(\Gamma)$. We define $\Gamma(a)$ to be the atomic context declared in Γ , when $a \in \text{dom}(\Gamma)$.

Context Apply Local: $(\Gamma, \gamma, \Delta)(x) = \gamma$, when $\text{id}(\gamma) = x$ and $x \notin \text{dom}(\Delta)$.

Context Apply Global: $(\Gamma, \gamma, \Delta)(a) = \gamma$, when $\text{id}(\gamma) = a$ and $a \notin \text{dom}(\Delta)$.

We define $\Gamma(U \leq T.m)$ to be the type of method m in type T , with self type U , and $\Gamma(U \leq T.f)$ to be the type of field f in type T , with self type U .

Context Apply Method: If $\Gamma(a) = \text{class } a[A_1, \dots, A_n] \cdots \text{extended } y \{ \cdots M \cdots \}$
and $\text{id}(M) = m$
then $\Gamma(U \leq a[T_1, \dots, T_n].m) = [y := U, A_1 := T_1, \dots, A_n := T_n]M$.

Week 9 begin

Week 9 end
Week 9 begin

Context Apply Field: If $\Gamma(a) = \text{class } a[A_1, \dots, A_n] \cdots \text{extended } y \{ \cdots F \cdots \}$
and $\text{id}(f) = F$
then $\Gamma(U \leq a[T_1, \dots, T_n].f) = [y := U, A_1 := T_1, \dots, A_n := T_n]F.$

Context Apply Method Interface:

If $\Gamma(a) = \text{interface } a[A_1, \dots, A_n] \cdots \{ \cdots M \cdots \}$ and $\text{id}(M) = m$
then $\Gamma(U \leq a[T_1, \dots, T_n].m) = [A_1 := T_1, \dots, A_n := T_n]M.$

Context Apply Method Superclass:

If $\Gamma(a) = \text{class } a[A_1, \dots, A_n] \text{ extends } T \cdots \{ M_1 \dots M_i F_1 \dots F_j \}$
and $(\text{method } m \dots) \notin \{ M_1, \dots, M_i \}$
and $\Gamma(U \leq [A_1 := T_1, \dots, A_n := T_n]T.m) = M$
then $\Gamma(U \leq a[T_1, \dots, T_n].m) = M.$

Context Apply Method Local Class: If $\Gamma(x) = \text{type } x < \# T;$

and $\Gamma(U \leq T.m) = M$
then $\Gamma(U \leq x.m) = M.$

Context Apply Field Superclass:

If $\Gamma(a) = \text{class } a [A_1, \dots, A_n] \text{ extends } T \cdots \{ M_1 \dots M_i F_1 \dots F_j \}$

and $(u \text{ field } f \dots) \notin \{F_1, \dots, F_j\}$
and $\Gamma(U \leq [A_1 := T_1, \dots, A_n := T_n]T.f) = F$
then $\Gamma(U \leq a[T_1, \dots, T_n].f) = F$.

Context Apply Field Local Class: If $\Gamma(x) = \text{type } x < \# T$;
and $\Gamma(U \leq T.f) = F$
then $\Gamma(U \leq x.f) = F$.

Context Apply Method Supertype:

If $\Gamma(a) = \text{interface } a[A_1, \dots, A_n] \dots \text{ extends } T_1, \dots, T_n \{ M_1 \dots M_i \}$
and $(\text{method } m \dots) \notin \{M_1, \dots, M_i\}$
and $\exists k . \Gamma(U \leq [A_1 := T_1, \dots, A_n := T_n]T_k.m) = M$
then $\Gamma(U \leq a[T_1, \dots, T_n].m) = M$.

Context Apply Method Initial: If $\Gamma(T \leq T.m) = M$
then $\Gamma(T.m) = M$.

Context Apply Field Initial: If $\Gamma(T \leq T.f) = F$
then $\Gamma(T.m) = F$.

3.5 Type context of a program

We write $\Gamma(P)$ for the type context given by program P .

Context Program Empty: $\Gamma() = ()$

Context Program Append: $\Gamma(P_1 P_2) = \Gamma(P_1), \Gamma(P_2)$

Week 8 begin

Context Program Class:

$\Gamma(\text{class } a[A_1, \dots, A_k] \text{ extends } T_0 \text{ implements } T_1, \dots, T_n \{ F_1 \dots F_i M_1 \dots M_j \})$
 $= (\text{class } a[A_1, \dots, A_k] \text{ extends } T_0 \text{ imple-}$
 $\text{ments } T_1, \dots, T_n \{ F_1 \dots F_i \text{ header}(M_1) \dots \text{header}(M_j) \})$

Context Program Interface:

$\Gamma(\text{interface } a[A_1, \dots, A_k] \text{ extends } T_1, \dots, T_n \{ M_1 \dots M_j \})$
 $= (\text{interface } a[A_1, \dots, A_k] \text{ extends } T_1, \dots, T_n \{ M_1 \dots M_j \})$

Context Program Object:

$\Gamma(\text{object } a[A_1, \dots, A_k] : T \{ F_1 \dots F_n \}) = (\text{object } a[A_1, \dots, A_k] : T)$

Week 8 end

Context Program Thread: $\Gamma(\text{thread } a \{ B \}) = (\text{thread } a)$

where

header(method $x.m (x_1 : T_1, \dots, x_n : T_n) : T \{ B \} = (\text{method } x.m (x_1 : T_1, \dots, x_n : T_n) : T)$). Week 6 begin

4 Subtyping and subclassing

4.1 Subclassing

When a type T is a subclass of U we write this as $\Gamma \vdash T <\# U$.

Subclass Idempotent: $\Gamma \vdash T <\# T$.

Subclass Transitive: If $\Gamma \vdash T_1 <\# T_2$ and $\Gamma \vdash T_2 <\# T_3$ then $\Gamma \vdash T_1 <\# T_3$. Week 8 begin

Subclass Class: If $\Gamma(a) = (\text{class } a[A_1, \dots, A_n] \text{ extends } T \dots)$
 then $\Gamma \vdash a[T_1, \dots, T_n] <\# [A_1 := T_1, \dots, A_n := T_n]T$. Week 9 begin

Subclass Local Class: If $T = x$

and $\Gamma(x) = (\text{type } x < \# S;)$
and $\Gamma \vdash S < \# U$
then $\Gamma \vdash T < \# U$

Week 9 end

4.2 Subtyping

When a type T is a subtype of U we write this as $\Gamma \vdash T <: U$.

Subclass Idempotent: $\Gamma \vdash T <: T$.

Subtype Transitive: If $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_2 <: T_3$ then $\Gamma \vdash T_1 <: T_3$.

Subtype Empty: $\Gamma \vdash \text{Empty} <: T$.

Week 8 begin

Subtype Class:

If $\Gamma(a) = (\text{class } a[A_1, \dots, A_n] \text{ extends } T_0 \text{ implements } T_1, \dots, T_n \dots)$
then $\Gamma \vdash a[T_1, \dots, T_n] <: [A_1 := T_1, \dots, A_n := T_n]T_i$.

Subtype Interface: If $\Gamma(a) = (\text{interface } a[A_1, \dots, A_n] \text{ extends } T_1, \dots, T_n \dots)$
then $\Gamma \vdash a[T_1, \dots, T_n] <: [A_1 := T_1, \dots, A_n := T_n]T_i$.

Week 9 begin

Subtype Local Class: If $T = x$
and $\Gamma(x) = (\text{type } x < \# S;)$
and $\Gamma \vdash S <: U$
then $\Gamma \vdash T <: U$

Week 9 end

4.3 Subtyping fields

When an abstract field F is a subtype of G we write this as $\Gamma \vdash F <: G$.

Subtype Immutable Field: If $\Gamma \vdash T <: U$
then $\Gamma \vdash (\text{immutable field } f : T) <: (\text{immutable field } f : U)$

Subtype Mutable Field: If $\Gamma \vdash T : \text{type}$
then $\Gamma \vdash (\text{mutable field } f : T) <: (\text{mutable field } f : T)$

4.4 Subtyping methods

When an abstract method M is a subtype of N we write this as $\Gamma \vdash M <: N$.

Subtype Method: If $\Gamma \vdash U_1 <: T_1 \dots \Gamma \vdash U_n <: T_n$
and $\Gamma \vdash T <: U$
then $\Gamma \vdash (\text{method } x_0.m (x_1 : T_1, \dots, x_n : T_n): T) <: (\text{method } y_0.m (y_1 : U_1, \dots, y_n : U_n): U)$

Week 6 end

5 Static semantics of user code

In this section we provide the static semantics for Hobbes user code.

The type judgements require that all code is explicitly typed, in that all variables have types. Type inference (which deals with untyped variables) is discussed in Section 7.

5.1 Static semantics of programs

The static semantics of a program P is given by the typing judgement $\Gamma \vdash P : \text{program}$.

Static Program:

If $\Gamma, \Gamma(D_1 \dots D_n) \vdash D_1 : \text{declaration} \dots \Gamma, \Gamma(D_1 \dots D_n) \vdash D_n : \text{declaration}$
and $\Gamma, \Gamma(D_1 \dots D_n)$ is disjoint
and $\Gamma, \Gamma(D_1 \dots D_n)$ is acyclic
then $\Gamma \vdash (D_1 \dots D_n) : \text{program}$

Week 6 begin

Week 6 end

where:

$(\gamma_1, \dots, \gamma_n)$ is *disjoint* whenever if $\text{id}(\gamma_i) = \text{id}(\gamma_j)$ then $i = j$.

Week 6 begin

Γ is *acyclic* whenever if $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_2 <: T_1$ then $T_1 = T_2$.

Week 6 end

5.2 Static semantics of declarations

The static semantics of a declaration D is given by the typing judgement $\Gamma \vdash D : \text{declaration}$.

Week 9 begin

Static Dec Class: If $\Gamma' = (\Gamma, A_1, \dots, A_k)$ and $\Gamma'' = (\Gamma', \text{type } y < \# T;)$
and $T = a[\text{id}(A_1), \dots, \text{id}(A_k)]$

and $\Gamma' \vdash T_0 : \text{class}$ and $\Gamma' \vdash T_1 : \text{interface} \dots \Gamma' \vdash T_n : \text{interface}$
 and $\Gamma'' \vdash F_1 : \text{field} \dots \Gamma'' \vdash F_i : \text{field}$ and (F_1, \dots, F_i) is disjoint
 and $\Gamma'' \vdash M_1 : \text{method in } a \dots \Gamma'' \vdash M_j : \text{method in } T$
 and (M_1, \dots, M_j) is disjoint
 and $\forall f . \Gamma'' \vdash \Gamma''(y \setminus \text{le } T.f) <: \Gamma''(y \setminus \text{le } T_0.f)$
 and $\forall k . \forall m . \Gamma'' \vdash \Gamma''(y \setminus \text{le } T.m) <: \Gamma''(y \setminus \text{le } T_k.m)$
 then $\Gamma \vdash (\text{class } a[A_1, \dots, A_k] \text{ extends } T_0 \text{ implements } T_1, \dots, T_n \text{ extended } y \{ F_1 \dots F_i M_1 \dots M_j \}) : \text{declaration}$

Week 9 end

Static Dec Interface: If $\Gamma' = (\Gamma, A_1, \dots, A_k)$ and $T = a[\text{id}(A_1), \dots, \text{id}(A_k)]$
 and $\Gamma' \vdash T_0 : \text{class}$ and $\Gamma' \vdash T_1 : \text{interface} \dots \Gamma' \vdash T_n : \text{interface}$
 and $\Gamma' \vdash M_1 : \text{method} \dots \Gamma' \vdash M_j : \text{method}$ and (M_1, \dots, M_j) is disjoint
 and $\forall k . \forall m . \Gamma' \vdash \Gamma'(T.m) <: \Gamma'(T_k.m)$
 then $\Gamma \vdash (\text{interface } a[A_1, \dots, A_k] \text{ extends } T_1, \dots, T_n \{ M_1 \dots M_j \}) : \text{declaration}$

Static Dec Object: If $\Gamma' = (\Gamma, A_1, \dots, A_k)$ and $\Gamma' \vdash T : \text{class}$
 and $\forall f . \Gamma'(T.f) = F \Rightarrow \exists i . \Gamma' \vdash G_i : F$
 and $\exists f . \Gamma'(T.f) = (\text{mutable field } \dots) \Rightarrow k = 0$
 and (G_1, \dots, G_j) is disjoint

then $\Gamma \vdash (\text{object } a[A_1, \dots, A_k] : T \{ G_1 \dots G_j \}) : \text{declaration}$

Week 8 end

Static Thread: If $\Gamma \vdash B : \text{Unit}$

then $\Gamma \vdash (\text{thread } a \{ B \}) : \text{declaration}$

where:

(F_1, \dots, F_n) is *disjoint* whenever if $\text{id}(F_i) = \text{id}(F_j)$ then $i = j$.

Week 5 begin

(M_1, \dots, M_n) is *disjoint* whenever if $\text{id}(M_i) = \text{id}(M_j)$ then $i = j$.

Week 5 end

Week 5 begin

5.3 Static semantics of concrete methods

The static semantics of a concrete method N in class T with header given by the abstract method M is given by the typing judgement

$\Gamma \vdash N : \text{method in } T$.

Static Method Concrete: If $\Gamma \vdash T_1 : \text{type} \dots \Gamma \vdash T_n : \text{type}$

and $\Gamma \vdash T : \text{type}$

and $\Gamma, x_0 : U, x_1 : T_1, \dots, x_n : T_n \vdash B : T$

then $\Gamma \vdash (\text{method } x_0.m (x_1:T_1, \dots, x_n:T_n) : T \{ B \}) : \text{method in } U$

Week 6 begin

5.4 Static semantics of abstract methods

The static semantics of an abstract method M is given by the typing judgement $\Gamma \vdash M : \text{method}$.

Week 8 begin

Static Method Abstract: If $\text{neg}(\Gamma) \vdash T_1 : \text{type} \dots \text{neg}(\Gamma) \vdash T_n : \text{type}$

and $\Gamma \vdash T : \text{type}$

then $\Gamma \vdash (\text{method } x_0.m (x_1:T_1, \dots, x_n:T_n) : T;) : \text{method}$

Week 8 end

Week 6 end

5.5 Static semantics of abstract fields

The static semantics of an abstract field F is given by the typing judgement $\Gamma \vdash F : \text{field}$.

Static Field Abstract: If $\Gamma \vdash T : \text{type}$
then $\Gamma \vdash (u \text{ field } f : T;) : \text{field}$

5.6 Static semantics of concrete fields

The static semantics of a concrete field G is given by the typing judgement $\Gamma \vdash G : F$.

Static Field Concrete: If $\Gamma \vdash V : T$
then $\Gamma \vdash (f = V) : (u \text{ field } f : T;)$

5.7 Static semantics of values

The static semantics of a value V is given by the typing judgement $\Gamma \vdash V : T$.

Static Val Local: If $\Gamma(x) = (x : T)$
then $\Gamma \vdash x : T$.

Static Val Global Object: If $\Gamma(a) = (\text{object } a[A_1, \dots, A_n] : T)$
and $\Gamma \vdash T_1 : \text{type} \dots \Gamma \vdash T_n : \text{type}$
then $\Gamma \vdash a[T_1, \dots, T_n] : [A_1 := T_1, \dots, A_n := T_n]T$.

Week 8 end

Static Val Global Thread: If $\Gamma(a) = (\text{thread } a)$
then $\Gamma \vdash a : \text{Thread}$.

Static Val Integer: $\Gamma \vdash i : \text{Integer}$

Static Val String: $\Gamma \vdash "s" : \text{String}$

Week 6 begin

Static Val Subsum: If $\Gamma \vdash V : T$ and $\Gamma \vdash T <: U$ then $\Gamma \vdash V : U$

Week 6 end

5.8 Static semantics of expressions

The static semantics of an expression E is given by the typing judgement $\Gamma \vdash E : T$.

Static Exp Dynamic Call: If $\Gamma \vdash V_0 : T_0 \dots \Gamma \vdash V_n : T_n$
and $\Gamma(T_0.m) = (\text{method } X_0.m(x_1:T_1, \dots, x_n:T_n) : T)$

then $\Gamma \vdash V_0.m (V_1, \dots, V_n) : T$

Static Exp Static Call: If $\Gamma \vdash V_0 : T_0 \dots \Gamma \vdash V_n : T_n$
and $\Gamma(T_0.m) = (\text{method } X_{0.m} (x_1:T_1, \dots, x_n:T_n) : T)$
then $\Gamma \vdash V_0:T_0.m (V_1, \dots, V_n) : T$

Static Exp Field Access: If $\Gamma \vdash V_0 : T_0$ and $\Gamma(T_0.f) = (u \text{ field } f : T)$
then $\Gamma \vdash V_0.f : T$

Static Exp Field Update: If $\Gamma \vdash V_0 : T_0$ and $\Gamma(T_0.f) = (\text{mutable field } f : T)$
and $\Gamma \vdash V_1 : T$
then $\Gamma \vdash V_0.f := V_1 : T$

Week 8 begin

Static Exp New Object: If $\Gamma(a) = \text{class } a [] \{ M_1 \dots M_m F_1 \dots F_n \}$
and $\Gamma \vdash G_1 : F_1 \dots \Gamma \vdash G_n : F_n$
then $\Gamma \vdash \text{new } a \{ G_1, \dots, G_n \} : a$

Week 8 end

Static Exp Subsum: If $\Gamma \vdash E : T$ and $\Gamma \vdash T <: U$ then $\Gamma \vdash E : U$

Week 6 end

5.9 Static semantics of blocks

The static semantics of a block B is given by the typing judgement $\Gamma \vdash B : T$.

Static Block Let: If $\Gamma \vdash E : U$ and $\Gamma, x:U \vdash B : T$
then $\Gamma \vdash (\text{let } x:U = E; B) : T$

Static Block If: If $\Gamma \vdash V : \text{Boolean}$ and $\Gamma \vdash B_1 : T$ and $\Gamma \vdash B_2 : T$
then $\Gamma \vdash (\text{if } (V) \{ B_1 \} \text{ else } \{ B_2 \}) : T$

Static Block Return: If $\Gamma \vdash V : T$
then $\Gamma \vdash (\text{return } V;) : T$

5.10 Static semantics of types

The static semantics of a type T is given by the typing judgement $\Gamma \vdash T : \text{type}$.

Static Type Class: If $\Gamma(a) = (\text{class } a[A_1, \dots, A_n] \dots)$
and $\Gamma \vdash T_1 : \text{type} \dots \Gamma \vdash T_n : \text{type}$
then $\Gamma \vdash a[T_1, \dots, T_n] : \text{type}$

Static Type Interface: If $\Gamma(a) = (\text{interface } a[A_1, \dots, A_n] \dots)$
and $\Gamma \vdash T_1 : \text{type} \dots \Gamma \vdash T_n : \text{type}$
then $\Gamma \vdash a[T_1, \dots, T_n] : \text{type}$

Static Type Native: If $\Gamma(a) = (\text{native class } a[A_1, \dots, A_n] \dots)$
and $\Gamma \vdash T_1 : \text{type} \dots \Gamma \vdash T_n : \text{type}$
then $\Gamma \vdash a[T_1, \dots, T_n] : \text{type}$

Static Type Local Type: If $\Gamma(x) = (\text{type } x)$
then $\Gamma \vdash x : \text{type}$

Static Type Local Class: If $\Gamma(x) = (\text{type } x < \# S;)$
then $\Gamma \vdash x : \text{type}$

Define $\Gamma \vdash T : \text{class}$ when type T is a class, and similarly for interfaces.

Week 9 begin

Week 9 end

Week 6 begin

Week 8 begin

Static Kind Class If $\Gamma(a) = \text{class } \dots$
and $\Gamma \vdash a[T_1, \dots, T_n] : \text{type}$
then $\Gamma \vdash a[T_1, \dots, T_n] : \text{class}$.

Static Kind Interface If $\Gamma(a) = \text{interface } \dots$
and $\Gamma \vdash a[T_1, \dots, T_n] : \text{type}$
then $\Gamma \vdash a[T_1, \dots, T_n] : \text{interface}$.

Week 8 end

Week 6 end

6 Type checking of native classes

Hobbes does not treat native classes any differently than user-defined classes (except that it provides special syntax for `ValInteger` and `ValString` constants). Native classes and objects are type checked in the same way as user-defined classes and object.

The additional typechecking rules required for native classes are:

Context Program Native Class:

$$\Gamma(\text{native class } a \{ M_1 \dots M_j \}) = (\text{native class } a \{ M_1 \dots M_j \})$$

Context Program Native Object: $\Gamma(\text{native object } a : T;) = (\text{object } a : T)$

Static Dec Native Class: $\Gamma \vdash (\text{native class } a \{ M_1 \dots M_n \}) : \text{declaration}$

Static Dec Native Class: $\Gamma \vdash (\text{native object } a : T;) : \text{declaration}$

Note that no typechecking is carried out on native classes or objects, it is the programmer's responsibility to ensure type safety for them!

7 Type inference

Hobbes is mostly an explicitly typed language: all variables have to be provided with types. There are two exceptions to this rule (both of which are required by the translation of the surface language into the core language):

- In a let-expression $\text{let } x = E; B$ x does not have to be explicitly typed.

- In a method declaration `method x.m (y:U) : T`, the self variable `x` does not have to be explicitly typed, although the parameters do.

In each case, we can infer the types of untyped variables using a simple type inference system.

We define a simple type inference algorithm which rewrites a program P (which may contain some untyped variables) into a program Q (where all variables are explicitly typed). For example:

```
class Factorial {
  method fact (n : Integer) : Integer {
    if (n <= 1) {
      return 1;
    } else {
      return n * this.fact (n-1);
    }
  }
}
object F : Factorial {
}
```

```
thread Main {  
  F.fact (5);  
}
```

is translated into the core language as:

```
class Factorial {  
  method this.fact(n : Integer) : Integer {  
    let tmp1 = n <= 1;  
    if (tmp1) {  
      return 1;  
    } else {  
      let tmp2 = n - 1;  
      let tmp3 = this.fact(tmp2);  
      let tmp4 = n * tmp3;  
      return tmp4;  
    }  
  }  
}  
object F : Factorial { }  
thread Main {
```

```
let tmp5 = F.fact(5);  
return Nothing;  
}
```

We then run the type inference algorithm on this program to get:

```
class Factorial {  
  method this:Factorial.fact(n : Integer) : Integer {  
    let tmp1 : Boolean = n <= 1;  
    if (tmp1) {  
      return 1;  
    } else {  
      let tmp2 : Integer = n - 1;  
      let tmp3 : Integer = this.fact(tmp2);  
      let tmp4 : Integer = n * tmp3;  
      return tmp4;  
    }  
  }  
}  
object F : Factorial { }  
thread Main {
```



```
let tmp5 : Integer = F.fact(5);  
return Nothing;  
}
```

7.1 Type inference for programs

The type inference algorithm for programs is specified as a judgement $\Gamma \vdash \text{infer program } P = Q$.

Infer Program: If $\Gamma(D_1 \dots D_n) = \Delta$ and (Γ, Δ) is disjoint
and $\Gamma, \Delta \vdash \text{infer declaration } D_1 = E_1 \dots \Gamma, \Delta \vdash \text{infer declaration } D_n = E_n$
then $\Gamma \vdash \text{infer program } (D_1 \dots D_n) = (E_1 \dots E_n)$

7.2 Type inference for declarations

The type inference algorithm for declarations is specified as a judgement $\Gamma \vdash \text{infer declaration } D = E$.

Infer Dec Class: If $\Delta = (\Delta, A_1, \dots, A_k)$
 and $T = a[id(A_1), \dots, id(A_k)]$
 and $\Delta \vdash \text{infer } T \text{ method } M_1 = N_1 \dots \Delta \vdash \text{infer } T \text{ method } M_i = N_i$
 then $\Gamma \vdash \text{infer declaration (class } a[A_1, \dots, A_k] \text{ extends } T_0 \text{ imple-}$
 $\text{ments } T_1, \dots, T_n \{ M_1 \dots M_i F_1 \dots F_j \}) = (\text{class } a[A_1, \dots, A_k] \text{ extends } T_0 \text{ imple-}$
 $\text{ments } T_1, \dots, T_n \{ N_1 \dots N_i F_1 \dots F_j \})$

Infer Dec Interface: $\Gamma \vdash \text{infer declaration (interface } a[A_1, \dots, A_k] \text{ ex-}$
 $\text{tends } T_1, \dots, T_n \{ M_1 \dots M_i \}) = (\text{interface } a[A_1, \dots, A_k] \text{ ex-}$
 $\text{tends } T_1, \dots, T_n \{ M_1 \dots M_i \})$

Infer Dec Object:

$\Gamma \vdash \text{infer declaration (object } a[A_1, \dots, A_k] \{ G_1 \dots G_n \}) = (\text{object } a[A_1, \dots, A_k] \{ G_1 \dots G_n \})$

Infer Thread: If $\Gamma \vdash \text{infer block } B = C$

then $\Gamma \vdash \text{infer declaration (thread } a \{ B \}) = (\text{thread } a \{ C \})$

7.3 Type inference for concrete methods

The type inference algorithm for concrete methods is specified as a judgement $\Gamma \vdash \text{infer } T \text{ method } M = N$.

Infer Method Concrete Untyped:

If $\Gamma, x_0 : T, x_1 : T_1, \dots, x_n : T_n \vdash \text{infer block } B = C$

then $\Gamma \vdash \text{infer } T \text{ method } (\text{method } x_0.m(x_1:T_1, \dots, x_n:T_n) : T \{ B \}) = (\text{method } x_0:T.m(x_1:T_1,$

Infer Method Concrete Typed: If $\Gamma, x_0 : T_0, x_1 : T_1, \dots, x_n : T_n \vdash \text{infer block } B = C$

then $\Gamma \vdash \text{infer } T \text{ method } (\text{method } x_0:T_0.m(x_1:T_1, \dots, x_n:T_n) : T \{ B \}) = (\text{method } x_0:T_0.m(x_1:$

Note that Infer Method Concrete Untyped gives a type to the self variable x_0 .

Week 5 end

7.4 Type inference for blocks

The type inference algorithm for concrete methods is specified as a judgement $\Gamma \vdash \text{infer block } B = C$.

Infer Block Let Typed: If $\Gamma, x : T \vdash \text{infer block } B = C$
then $\Gamma \vdash \text{infer block } (\text{let } x : T = E; B) = (\text{let } x : T = E; C)$

Infer Block Let Untyped: If $\Gamma \vdash E : T$ and $\Gamma, x : T \vdash \text{infer block } B = C$
then $\Gamma \vdash \text{infer block } (\text{let } x = E; B) = (\text{let } x : T = E; C)$

Infer Block If: If $\Gamma \vdash \text{infer block } B_1 = C_1$ and $\Gamma \vdash \text{infer block } B_2 = C_2$
then $\Gamma \vdash \text{infer block } (\text{if } (V) \{ B_1 \} \text{ else } \{ B_2 \}) = (\text{if } (V) \{ C_1 \} \text{ else } \{ C_2 \})$

Infer Block Return: $\Gamma \vdash \text{infer block } (\text{return } V;) = (\text{return } V;)$