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:

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

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 ⊢ 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: Week 8 begin

- A ranges over TypeVar. Week 8 end
- B and C range over Block.
- D and E range over Exp.
- 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 $X = V.m\ (W_1, \ldots, W_n)$; 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 { reutrn y } : Integer
```

It also contains information about class, interface, local type, native class, object and thread declarations. Week 7 begin
Week 8 begin
Week 8 end

3.1 Definition of a typing context

A *typing context* is given by the grammar:

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

ContextAtomic ::=
  VarTyped |                                                    Week 8 begin
  TypeVar |                                                     Week 9 begin
  ClassVar |                                                    Week 9 end
  ContextClass |                                                Week 6 begin
  ContextInterface |                                            Week 6 end
  ContextNative |
  ContextObject |
  ContextThread

                                                                Week 9 begin
ClassVar ::=
  "type" LocalId "<#" TypeGlobal ";"
                                                                Week 9 end
ContextClass ::=
  "class" GlobalId                                              Week 8 begin
  TypeParams                                                    Week 8 end
  ExtendsClass
  ImplementsInterfaces                                          Week 9 begin
  ExtendedClass                                                 Week 9 end
  "{"
    FieldAbstract ... FieldAbstract                             Week 5 begin
    MethodAbstract ... MethodAbstract                           Week 5 end
  "}"

                                                                Week 6 begin
ContextInterface ::=
  "interface" GlobalId                                          Week 8 begin
  TypeParams                                                    Week 8 end
  ExtendsInterfaces
 "{"
    MethodAbstract ... MethodAbstract
  "}"

                                                                Week 6 end
ContextNative ::=
  "native" "class" GlobalId                                     Week 8 begin
  TypeParams                                                    Week 8 end
  "{"
    MethodAbstract ... MethodAbstract
  "}"

ContextObject ::=
  "object" GlobalId                                             Week 8 begin
  TypeParams                                                    Week 8 end
  ":" Type

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 $\mathrm{id}(\gamma)$ that it provides a definition for: Week 9 begin

Identifier Class Var: $\mathrm{id}(\text{type } a \ <\# \ T;) \ = \ a$ Week 9 end

Identifier Class: $\mathrm{id}(\text{class } a \cdots) \ = \ a$ Week 6 begin

Identifier Interface: $\mathrm{id}(\text{interface } a \cdots) \ = \ a$ Week 6 end

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

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

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

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

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

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

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

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

3.3 Domain of a typing context

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

Domain Context: $\mathrm{dom}(\gamma_1, \ldots, \gamma_n) = \{\mathrm{id}(\gamma_1), \ldots, \mathrm{id}(\gamma_n)\}$

3.4 Applying a typing context to an identifier

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

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

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

Week 9 begin

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. Week 9 end

Week 9 begin

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

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

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

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

Context Apply Method Local Class: If $\Gamma(x) = \mathsf{type}\ x <\# T$; and $\Gamma(U \leq T.m) = M$
then $\Gamma(U \leq x.m) = M$.

Context Apply Field Superclass: If $\Gamma(a) = \mathsf{class}\ a\ [A_1,\ldots,A_n]\ \mathsf{extends}\ T\ \cdots\ \{\ M_1\ \ldots\ M_i\ F_1\ \ldots\ F_j\ \}$
and $(u\ \mathsf{field}\ f\ \ldots) \notin \{\ F_1,\ldots,F_j\ \}$ and $\Gamma(U \leq [A_1 := T_1,\ldots,A_n := T_n]T.f) = F$
then $\Gamma(U \leq a[T_1,\ldots,T_n].f) = F$.

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

Context Apply Method Supertype: If $\Gamma(a) = \mathsf{interface}\ a[A_1,\ldots,A_n]\ \cdots\ \mathsf{extends}\ T_1,\ldots,T_n\ \{\ M_1\ \ldots\ M_i\ \}$
and $(\mathsf{method}\ m\ \ldots) \notin \{\ M_1,\ldots,M_i\ \}$ and $\exists\ k\ .\ \Gamma(U \leq [A_1 := T_1,\ldots,A_n := T_n]T_k.m) = M$
then $\Gamma(U \leq a[T_1,\ldots,T_n].m) = M$.

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 $\Gamma(T.m) = F$. Week 9 end

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: $\Gamma(\mathsf{class}\ a[A_1,\ldots,A_k]\ \mathsf{extends}\ T_0\ \mathsf{implements}\ T_1,\ldots,T_n\ \{\ F_1\ \ldots\ F_i\ M_1\ \ldots\ M_j\ \})$
$= (\mathsf{class}\ a[A_1,\ldots,A_k]\ \mathsf{extends}\ T_0\ \mathsf{implements}\ T_1,\ldots,T_n\ \{\ F_1\ \ldots\ F_i\ \mathsf{header}(M_1)\ \ldots\ \mathsf{header}(M_j)\ \})$

Context Program Interface: $\Gamma(\mathsf{interface}\ a[A_1,\ldots,A_k]\ \mathsf{extends}\ T_1,\ldots,T_n\ \{\ M_1\ \ldots\ M_j\ \})$
$= (\mathsf{interface}\ a[A_1,\ldots,A_k]\ \mathsf{extends}\ T_1,\ldots,T_n\ \{\ M_1\ \ldots\ M_j\ \})$

Context Program Object: $\Gamma(\mathsf{object}\ a[A_1,\ldots,A_k] : T\ \{\ F_1\ \ldots\ F_n\ \}) = (\mathsf{object}\ a[A_1,\ldots,A_k] : T)$ Week 8 end

Context Program Thread: $\Gamma(\mathsf{thread}\ a\ \{\ B\ \}) = (\mathsf{thread}\ a)$
Week 5 begin

where $\mathsf{header}(\mathsf{method}\ x.m\ (x_1 : T_1,\ldots,x_n : T_n) : T\ \{\ B\ \} = (\mathsf{method}\ x.m\ (x_1 : T_1,\ldots,x_n : T_n) : T)$. 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 $\Gamma(a) = (\text{class } a[A_1,\ldots,A_n] \text{ extends } T \ldots)$ then $\Gamma \vdash a[T_1,\ldots,T_n] <\# [A_1 := T_1,\ldots,A_n := T_n]T$. 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 $\Gamma(a) = (\text{class } a[A_1,\ldots,A_n] \text{ extends } T_0 \text{ implements } T_1,\ldots,T_n \cdots)$
then $\Gamma \vdash a[T_1,\ldots,T_n] <: [A_1 := T_1,\ldots,A_n := T_n]T_i$.

Subtype Interface: If $\Gamma(a) = (\text{interface } a[A_1,\ldots,A_n] \text{ extends } T_1,\ldots,T_n \cdots)$
then $\Gamma \vdash a[T_1,\ldots,T_n] <: [A_1 := T_1,\ldots,A_n := T_n]T_i$. 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 $\Gamma \vdash U_1 <: T_1 \ldots \Gamma \vdash U_n <: T_n$
and $\Gamma \vdash T <: U$
then $\Gamma \vdash (\text{method } x_0.m\ (x_1 : T_1,\ldots,x_n : T_n)\colon T) <: (\text{method } y_0.m\ (y_1 : U_1,\ldots,y_n : U_n)\colon U)$
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 $\Gamma \vdash P$: program.

Static Program: If $\Gamma,\Gamma(D_1 \ \ldots \ D_n) \vdash D_1$: declaration $\ldots$ $\Gamma,\Gamma(D_1 \ \ldots \ D_n) \vdash D_n$: declaration
and $\Gamma,\Gamma(D_1 \ \ldots \ D_n)$ is disjoint Week 6 begin
and $\Gamma,\Gamma(D_1 \ \ldots \ D_n)$ is acyclic Week 6 end
then $\Gamma \vdash (D_1 \ \ldots \ D_n)$: program

where:

$(\gamma_1,\ldots,\gamma_n)$ is *disjoint* whenever if $\mathrm{id}(\gamma_i) = \mathrm{id}(\gamma_j)$ 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 $\Gamma \vdash D$: declaration. Week 9 begin

Static Dec Class: If $\Gamma\prime = (\Gamma,A_1,\ldots,A_k)$ and $\Gamma\prime\prime = (\Gamma\prime,$ type $y <: \# \ T;)$ and $T = a[\mathrm{id}(A_1),\ldots,\mathrm{id}(A_k)]$
and $\Gamma\prime \vdash T_0$: class and $\Gamma\prime \vdash T_1$: interface $\ldots$ $\Gamma\prime \vdash T_n$: interface
and $\Gamma\prime\prime \vdash F_1$: field $\ldots$ $\Gamma\prime\prime \vdash F_i$: field and $(F_1,\ldots,F_i)$ is disjoint
and $\Gamma\prime\prime \vdash M_1$: method in a $\ldots$ $\Gamma\prime\prime \vdash M_j$: method in T and $(M_1,\ldots,M_j)$ is disjoint
and $\forall \ f \ . \ \Gamma\prime\prime \vdash \Gamma\prime\prime(y \setminus le \ T.f) <: \Gamma\prime\prime(y \setminus le \ T_0.f)$
and $\forall \ k \ . \ \forall \ m \ . \ \Gamma\prime\prime \vdash \Gamma\prime\prime(y \setminus le \ T.m) <: \Gamma;;(y \setminus le \ T_k.m)$
then $\Gamma \vdash$ (class $a[A_1,\ldots,A_k]$ extends T_0 implements $T_1,\ldots,T_n$ extended y { $F_1 \ \ldots \ F_i \ M_1 \ \ldots \ M_j$ }) : declaration Week 9 end

Static Dec Interface: If $\Gamma\prime = (\Gamma,A_1,\ldots,A_k)$ and $T = a[\mathrm{id}(A_1),\ldots,\mathrm{id}(A_k)]$
and $\Gamma\prime \vdash T_0$: class and $\Gamma\prime \vdash T_1$: interface $\ldots$ $\Gamma\prime \vdash T_n$: interface
and $\Gamma\prime \vdash M_1$: method $\ldots$ $\Gamma\prime \vdash M_j$: method and $(M_1,\ldots,M_j)$ is disjoint
and $\forall \ k \ . \ \forall \ m \ . \ \Gamma\prime \vdash \Gamma\prime(T.m) <: \Gamma\prime(T_k.m)$
then $\Gamma \vdash$ (interface $a[A_1,\ldots,A_k]$ extends $T_1,\ldots,T_n$ { $M_1 \ \ldots \ M_j$ }) : declaration

Static Dec Object: If $\Gamma\prime = (\Gamma,A_1,\ldots,A_k)$ and $\Gamma\prime \vdash T$: class
and $\forall \ f \ . \ \Gamma\prime(T.f) = F \Rightarrow \exists \ i \ . \ \Gamma\prime \vdash G_i : F$
and $\exists \ f \ . \ \Gamma\prime(T.f) =$ (mutable field $\cdots$) $\Rightarrow \ k = 0$
and $(G_1,\ldots,G_j)$ is disjoint
then $\Gamma \vdash$ (object $a[A_1,\ldots,A_k] : T$ { $G_1 \ \ldots \ G_j$ }) : declaration Week 8 end

Static Thread: If $\Gamma \vdash B$: Unit
then $\Gamma \vdash$ (thread a { B }) : declaration

where:

$(F_1,\ldots,F_n)$ is *disjoint* whenever if $\mathrm{id}(F_i) = \mathrm{id}(F_j)$ then $i = j$. Week 5 begin

$(M_1,\ldots,M_n)$ is *disjoint* whenever if $\mathrm{id}(M_i) = \mathrm{id}(M_j)$ 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 $\Gamma \vdash N$: method in T.

Static Method Concrete: If $\Gamma \vdash T_1$: type ... $\Gamma \vdash T_n$: type
and $\Gamma \vdash T$: type
and $\Gamma, x_0 : U, x_1 : T_1, \ldots, x_n : T_n \vdash B : T$
then $\Gamma \vdash$ (method $x_0.m$ (x_1:T_1,...,x_n:T_n) : T { B }) : 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 $\Gamma \vdash M$: method.

Week 8 begin

Static Method Abstract: If neg(Γ) $\vdash T_1$: type ... neg(Γ) $\vdash T_n$: type
and $\Gamma \vdash T$: type
then $\Gamma \vdash$ (method $x_0.m$ (x_1:T_1,...,x_n:T_n) : T;) : 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 $\Gamma \vdash F$: field.

Static Field Abstract: If $\Gamma \vdash T$: type
then $\Gamma \vdash$ (u field $f : T$;) : 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 $\Gamma \vdash (f = V) : (u$ 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 $\Gamma \vdash x : T$.

Week 8 begin

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

Week 8 end

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

Static Val Integer: $\Gamma \vdash i$: Integer

Static Val String: $\Gamma \vdash$ "s" : 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 $\Gamma \vdash V_0 : T_0 \ldots \Gamma \vdash V_n : T_n$ and $\Gamma(T_0.m) = (\mathsf{method}\ X_0.m\ (x_1{:}T_1,\ldots,x_n{:}T_n) : T)$
then $\Gamma \vdash V_0.m\ (V_1,\ldots,V_n) : T$

Static Exp Static Call: If $\Gamma \vdash V_0 : T_0 \ldots \Gamma \vdash V_n : T_n$ and $\Gamma(T_0.m) = (\mathsf{method}\ X_0.m\ (x_1{:}T_1,\ldots,x_n{:}T_n) : T)$
then $\Gamma \vdash V_0{:}T_0.m\ (V_1,\ldots,V_n) : T$

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

Static Exp Field Update: If $\Gamma \vdash V_0 : T_0$ and $\Gamma(T_0.f) = (\mathsf{mutable\ field}\ f : T)$ and $\Gamma \vdash V_1 : T$
then $\Gamma \vdash V_0.f := V_1 : T$ Week 8 begin

Static Exp New Object: If $\Gamma(a) = \mathsf{class}\ a\ []\ \{\ M_1\ \ldots\ M_m\ F_1\ \ldots\ F_n\ \}$ and $\Gamma \vdash G_1 : F_1 \ldots \Gamma \vdash G_n : F_n$
then $\Gamma \vdash \mathsf{new}\ a\ \{\ G_1,\ldots,G_n\ \} : a$ 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 $\Gamma \vdash (\mathsf{let}\ x{:}U = E; B) : T$

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

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

5.10 Static semantics of types

The static semantics of a type T is given by the typing judgement $\Gamma \vdash T$: type. Week 8 begin

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

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

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

Static Type Local Type: If $\Gamma(x) = (\mathsf{type}\ x)$
then $\Gamma \vdash x$: type Week 9 begin

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

Week 9 end

Week 6 begin

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

Week 8 begin

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

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

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

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

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

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 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 $\Gamma \vdash$ infer program $P = Q$.

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

7.2 Type inference for declarations

The type inference algorithm for declarations is specified as a judgement $\Gamma \vdash$ infer declaration $D = E$. Week 8 begin

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

Infer Dec Interface: $\Gamma \vdash$ infer declaration (interface $a[A_1, \ldots, A_k]$ extends $T_1, \ldots, T_n$ { $M_1 \ \ldots \ M_i$ }) = (interface $a[A_1, \ldots, A_k]$ extends $T_1, \ldots, T_n$ { $M_1 \ \ldots \ M_i$ })

Infer Dec Object: $\Gamma \vdash$ infer declaration (object $a[A_1, \ldots, A_k]$ { $G_1 \ \ldots \ G_n$ }) = (object $a[A_1, \ldots, A_k]$ { $G_1 \ \ldots \ G_n$ }) Week 8 end

Infer Thread: If $\Gamma \vdash$ infer block $B = C$ then $\Gamma \vdash$ infer declaration (thread a { B }) = (thread a { C })

Week 5 begin

7.3 Type inference for concrete methods

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

Infer Method Concrete Untyped: If $\Gamma, \ x_0 : T, x_1 : T_1, \ldots, x_n : T_n \vdash$ infer block $B = C$
then $\Gamma \vdash$ infer T method (method $x_0.m$ (x_1:T_1,...,x_n:T_n) : T { B }) = (method x_0:$T.m$ (x_1:T_1,...,x_n:T_n) : T { C })

Infer Method Concrete Typed: If $\Gamma, \ x_0 : T_0, x_1 : T_1, \ldots, x_n : T_n \vdash$ infer block $B = C$
then $\Gamma \vdash$ infer T method (method x_0:$T_0.m$ (x_1:T_1,...,x_n:T_n) : T { B }) = (method x_0:$T_0.m$ (x_1:T_1,...,x_n:T_n) : T { C })

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 $\Gamma \vdash$ infer block $B = C$.

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

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

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

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