Dynamic Semantics of Hobbes

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

17 September 2002

1 Introduction

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

This is an incomplete draft: the language is missing many of its important features. These will be added as the course proceeds.

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:

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

Week 6 begin

Week 6 end

The dynamic semantics is specified by defining a relation:

$$P \rightarrow Q$$

where:

- P and Q are Programs.
- P can perform one step of evaluation and become Q.

For example:

```
  thread Main { let x = 1+2; let y = x+3; return Nothing; }
→
  thread Main { [x:=3] let y = x+3; return Nothing; }
→
  thread Main { [x:=3,y:=6] return Nothing; }
```

Here we are writing $[X := V]B$ as the result of *substituting* value V for variable X in block B. This is formally defined in Section 4.

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.
- P and Q range over Program.
- 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 }

2.2 Structural equivalence

We work up to *structural equivalence* $\equiv$ on programs, which says that the order of declarations in a program do not matter:

Commutativity: $P_1\ P_2 \equiv P_2\ P_1$

Associativity: $(P_1\ P_2)\ P_3 \equiv P_1\ (P_2\ P_3)$

Unit: $P\ \varepsilon = P$

Idempotence: $P \equiv P$

Reflexivity: if $P_1 \equiv P_2$ then $P_2 \equiv P_1$

Transitivity: if $P_1 \equiv P_2$ and $P_2 \equiv P_3$ then $P_1 \equiv P_3$

(For the mathematically inclined, this definition says that $\equiv$ is an equivalence relation, and that programs up to $\equiv$ form a commutative monoid.)

3 Free variables

The *free variables* of a program are the identifiers of any variables which do not have binders. For example, the free variables of the block:

```
let x = 1+f; let z = g+x; return z;
```

are f and g. The variables x and z occur *bound* in the program, they are not free.

Week 8 begin

The free variables also include any type variables which do not have binders. For example, the free variables of the block:

```
let x : List[a] = new ListCons[a] { hd=y, tl=z };
```

are a, y and z. Week 8 end

Note that global values such as True or Nothing do not count as free variables. For example, the free variables of the block:

```
let x = Foo.bar (y); let z = baz.bar (y);
```

are y and baz, not Foo.

3.1 Identifiers

Define $\mathrm{id}(X)$ as the identifier of the variable X:

Id Untyped: $\mathrm{id}(x) = x$

Id Typed: $\mathrm{id}(x : T) = x$

Week 8 begin

Define $\mathrm{id}(A)$ as the identifier of the type variable A:

Id Type Var: $\mathrm{id}(\texttt{type}\ x) = x$

Week 8 end

3.2 Free variables of a value

Define the free variables $\mathrm{fv}(V)$ of a value V as the set of variable identifiers occurring in V:

Free Variables Val Local: $\mathrm{fv}(x) = \{x\}$ Week 8 begin

Free Variables Val Global: $\mathrm{fv}(a[T_1, \ldots, T_n]) = \mathrm{fv}(T_1) \cup \cdots \cup \mathrm{fv}(T_n)$ Week 8 end

Free Variables Val Integer: $\mathrm{fv}(i) = \emptyset$

Free Variables Val String: $\mathrm{fv}(\text{"}s\text{"}) = \emptyset$

A value V is *closed* if $\mathrm{fv}(V) = \emptyset$. For example 27 and True are closed, but x is not. Week 8 begin

3.3 Free variables of a type

Define the free variables $\mathrm{fv}(T)$ of a type T as the set of names occurring in T.

Free Variables Type Local: $\mathrm{fv}(x) = \{x\}$

Free Variables Type Global: $\mathrm{fv}(a[T_1,\ldots,T_n]) = \mathrm{fv}(T_1) \cup \cdots \cup \mathrm{fv}(T_n)$

A type T is *closed* if $\mathrm{fv}(T) = \emptyset$. For example Integer and Ref[Integer] are closed, but x and Ref[x] are not. Week 8 end

3.4 Free variables of an expression

Define the free variables $\mathrm{fv}(E)$ of an expression E as the set of variable identifiers occurring in E:

Free Variables Dynamic Call: $\mathrm{fv}(V_0.m\ (V_1,\ldots,V_n)) = \mathrm{fv}(V_0) \cup \cdots \cup \mathrm{fv}(V_n)$ Week 8 begin

Free Variables Static Call: $\mathrm{fv}(V_0::T.m\ (V_1,\ldots,V_n)) = \mathrm{fv}(T) \cup \mathrm{fv}(V_0) \cup \cdots \cup \mathrm{fv}(V_n)$ Week 8 end

Free Variables Field Access: $\mathrm{fv}(V.f) = \mathrm{fv}(V)$

Free Variables Field Update: $\mathrm{fv}(V_0.f := V_1) = \mathrm{fv}(V_0) \cup \mathrm{fv}(V_1)$ Week 8 begin

Free Variables New Object:

$$\mathrm{fv}(\texttt{new}\ T\ \{\ f_1 = V_1, \ldots, f_n = V_n\ \}) \ = \ \mathrm{fv}(T) \cup \mathrm{fv}(V_1) \cup \cdots \cup \mathrm{fv}(V_n)$$

Week 8 end

3.5 Free variables of a block

Define the free variables $\mathrm{fv}(B)$ of a block B as the set of variable identifiers occurring in B without matching binders: Week 8 begin

Free Variables Let: $\mathrm{fv}(\texttt{let}\ x : T = V;\ B) = \mathrm{fv}(V) \cup \mathrm{fv}(T) \cup (\mathrm{fv}(B) - x)$ Week 8 end

Free Variables If: $\mathrm{fv}(\texttt{if}\ (V)\ \{\ B_1\ \}\ \texttt{else}\ \{\ B_2\ \}) \ = \ \mathrm{fv}(V) \cup \mathrm{fv}(B_1) \cup \mathrm{fv}(B_2)$

Free Variables Return: $\mathrm{fv}(\texttt{return}\ V;) = \mathrm{fv}(V)$

For example:

$$\begin{aligned}
&\mathrm{fv}(\texttt{let x:Integer = 1+x; let y:Integer = z+x; return Nothing;})\\
&\ = \ \{\texttt{x}\} \cup (\mathrm{fv}(\texttt{let y:Integer = z+x; return Nothing;}) - \texttt{x})\\
&\ = \ \{\texttt{x}\} \cup (\{\texttt{z,x}\} \cup (\mathrm{fv}(\texttt{return Nothing;}) - \texttt{y}) - \texttt{x})
\end{aligned}$$

$$\begin{aligned} &= \{\mathsf{x}\} \cup (\{\mathsf{z},\mathsf{x}\} \cup (\emptyset - \mathsf{y}) - \mathsf{x}) \\ &= \{\mathsf{x},\mathsf{z}\} \end{aligned}$$

A block B is *closed* if $\mathrm{fv}(B) = \emptyset$. For example `return True;` and `let x = 37; return x;` are closed, but `let x = 37; return y;` is not.

4 Substitution

An example execution of a program is:

```
  thread Main { let x = 1+2; let y = x+3; return Nothing; }
→
  thread Main { [x:=3] let y = x+3; return Nothing; }
→
  thread Main { [x:=3,y:=6] return Nothing; }
```

Here we are writing $[x := V]B$ as the result of *substituting* value V for free variable x in block B.

Week 8 begin

When a method is called of a generic class, we also need to perform a *type substitution*, replacing the type parameters of the class declaration with the type arguments of the class usage. For example if *P* is:

```
class Ref[type a] {
  field contents : a;
  method this.set (x : a) { this.contents := a; }
  method this.get () : a { return this.contents; }
  method this.clone () : Ref[a] {
    let x : a = this.contents;
    let y : Ref[a] = new Ref[a] { contents=x };
    return y;
  }
}
```

then one possible execution is:

```
object R : Ref[Integer] { contents = 37 }
thread Main {
  let z : Ref[Integer] = R:Ref[Integer].clone ();
  }
```
P

$\longrightarrow$

```
  object R : Ref[Integer] { contents = 37 }
  thread Main {
    [ a:=Integer ]
    let x : a = this.contents;
    let y : Ref[a] = new Ref[a] { contents=x };
    let z : Ref[Integer] = y;
  }
```
P

$\longrightarrow$

```
  object R : Ref[Integer] { contents = 37 }
  thread Main {
    [ a:=Integer, x:=37 ]
    let y : Ref[a] = new Ref[a] { contents=x };
    let z : Ref[Integer] = y;
  }
```
P

$\longrightarrow$

```
  object R : Ref[Integer] { contents = 37 }
  object S : Ref[Integer] { contents = 37 }
  thread Main {
    [ a:=Integer, x:=37, y:=S ]
    let z : Ref[Integer] = y;
  }
  P
→
  object R : Ref[Integer] { contents = 37 }
  object S : Ref[Integer] { contents = 37 }
  thread Main {
    [ a:=Integer, x:=37, y:=S, z:=S ]
  }
  P
```

Note that when we call clone on object R : Ref[Integer] we need to replace the formal type parameter a by the actual type argument Integer. This is performed by the type substitution a:=Integer. Week 8 end

We now define this formally.

4.1 Definition of a substitution

A *substitution* is given by the grammar:

```
Substitution ::=
  Subst ","...","  Subst

Subst ::=
  LocalId ":=" Val |
  LocalId ":=" Type
```

Week 8 begin

Week 8 end

for example:

```
foo := 37, bar := fish, baz := "fred"
```

Week 8 begin

or:

```
a := Integer
```

Week 8 end

We let σ and ρ range over Substitution.

We will write $X := V$ for the substitution $\mathrm{id}(X) := V$, for example:

(foo:Integer := 37, bar:Bang := fish, baz:String := "fred") = (foo := 37, bar := fish, baz := "frec

Week 8 begin

We will write $A := T$ for the substitution $\mathrm{id}(A) := T$, for example:

(type a := Person) = (a := Person)

Week 8 end

4.2 Free variables of a substitution

We define fv(σ) as the free variables of all the values in σ:

Free Variables Subst Empty: $\mathrm{fv}() = \emptyset$

Free Variables Subst Var: $\mathrm{fv}(x := V) = \mathrm{fv}(V)$

Week 8 begin

Free Variables Subst TypeVar: $\mathrm{fv}(x := T) = \mathrm{fv}(T)$ Week 8 end

Free Variables Subst Concat: $\mathrm{fv}(\sigma, \rho) = \mathrm{fv}(\sigma) \cup \mathrm{fv}(\rho)$

For example:

$\mathrm{fv}(\textsf{foo} := \textsf{37}, \textsf{bar} := \textsf{fish}, \textsf{baz} := \textsf{"fred"}) = \{\ \textsf{fish}\ \}$

A substitution σ is *closed* if $\mathrm{fv}(\sigma) = \emptyset$. For example $(\textsf{x} := \textsf{5})$ and $(\textsf{y} := \textsf{True})$ are closed, but $(\textsf{z} := \textsf{x})$ is not.

4.3 Domain of a substitution

We define $\mathrm{dom}(\sigma)$ as the identifiers with definitions in σ:

Subst Domain Empty: $\mathrm{dom}() = \emptyset$

Subst Domain Var: $\mathrm{dom}(x := V) = \{x\}$ Week 8 begin

Subst Domain TypeVar: $\mathrm{dom}(x := T) = \{x\}$ Week 8 end

Subst Domain Concat: $\mathrm{dom}(\sigma, \rho) = \mathrm{dom}(\sigma) \cup \mathrm{dom}(\rho)$

For example:

$\mathrm{dom}(\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := \text{"fred"}) = \{ \text{foo}, \text{bar}, \text{baz} \}$

4.4 Removing a variable from a substitution

We define $\sigma - x$ to be the substitution σ with any bindings for x removed:

Subst Remove Var In Domain: $(\sigma, x := V, \rho) - x = (\sigma, \rho) - x$. Week 8 begin

Subst Remove TypeVar In Domain: $(\sigma, x := T, \rho) - x = (\sigma, \rho) - x$. Week 8 end

Subst Remove Not In Domain: $(\sigma) - x = \sigma$ if $x \notin \mathrm{dom}(\sigma)$

For example:

$(\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := \text{"fred"}) - \text{bar} = (\text{foo} := 37, \text{baz} := \text{"fred"})$
$(\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := \text{"fred"}) - \text{bang} = (\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := \text{"fred"})$

4.5 Applying a substitution to an identifier

We define $\sigma(x)$ to tbe the value resulting from substituting σ into local identifier x, when $x \in \mathrm{dom}(\sigma)$:

Subst Apply Var: $(\sigma, x := V, \rho)(x) = V$ if $x \notin \mathrm{dom}(\rho)$ Week 8 begin

Subst Apply TypeVar: $(\sigma, x := T, \rho)(x) = T$ if $x \notin \mathrm{dom}(\rho)$ Week 8 end

For example:

(foo := 37, bar := fish, baz := "fred")(foo) = 37
(foo := 37, bar := fish, baz := "fred")(bar) = fish
(foo := 37, bar := fish, baz := "fred")(baz) = "fred"

Note that if a variable occurs more than once in a substitution, it is the right-most binding that counts, for example:

(foo := 37, foo := 5)(foo) = 5

4.6 Applying a substitution to a value

We define $[\sigma]V$ to be the value resulting from substituting σ into value V:

Subst Val Local: $[\sigma]x = \begin{cases} \sigma(x) & \text{if } x \in \text{dom}(\sigma) \\ x & \text{otherwise} \end{cases}$ Week 8 begin

Subst Val Global: $[\sigma](a[T_1, \ldots, T_n]) = a[[\sigma]T_1, \ldots, [\sigma]T_n]$ Week 8 end

Subst Val Integer: $[\sigma]i = i$

Subst Val String: $[\sigma]"s" = "s"$

For example:

$[\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := "\text{fred}"]\text{foo} = 37$
$[\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := "\text{fred}"]\text{bang} = \text{bang}$
$[\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := "\text{fred}"]\text{True} = \text{True}$
$[\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := "\text{fred}"]16 = 16$
$[\text{foo} := 37, \text{bar} := \text{fish}, \text{baz} := "\text{fred}"]"\text{wilma}" = "\text{wilma}"$

Week 8 begin

4.7 Applying a substitution to a type

We define $[\sigma]T$ to be the value resulting from substituting σ into type T:

Subst Type Local: $[\sigma]x = \begin{cases} \sigma(x) & \text{if } x \in \mathrm{dom}(\sigma) \\ x & \text{otherwise} \end{cases}$

Subst Type Global: $[\sigma](a[T_1, \ldots, T_n]) = a[[\sigma]T_1, \ldots, [\sigma]T_n]$

For example:

$[\mathsf{a} := \mathsf{Integer}]\mathsf{a} = \mathsf{Integer}$
$[\mathsf{a} := \mathsf{Integer}]\mathsf{Ref[a]} = \mathsf{Ref[Integer]}$
$[\mathsf{a} := \mathsf{Integer}]\mathsf{Ref[Ref[a]]} = \mathsf{Ref[Ref[Integer]]}$
$[\mathsf{a} := \mathsf{Integer}]\mathsf{Ref[Integer]} = \mathsf{Ref[Integer]}$

4.8 Applying a substitution to an abstract method

We define $[\sigma]M$ to be the value resulting from substituting σ into abstract method M:

Subst Method Abstract: $[\sigma](\textsf{method}\ x.m\ (x_1 : T_1, \ldots, x_n : T_n) : R)$
$= (\textsf{method}\ x.m\ (x_1 : [\sigma]T_1, \ldots, x_n : [\sigma]T_n) : [\sigma]R)$

4.9 Applying a substitution to an abstract field

We define $[\sigma]F$ to be the value resulting from substituting σ into abstract field F:

Subst Field Abstract: $[\sigma](u\ \textsf{field}\ f : T) = (u\ \textsf{field}\ f : [\sigma]T)$

Week 8 end

4.10 Applying a substitution to an expression

We define $[\sigma]E$ to be the block resulting from substituting a substitution σ into expression E:

Subst Dynamic Call: $[\sigma](V_0.m\ (V_1, \ldots, V_n)) = (([\sigma]V_0).m\ ([\sigma]V_1, \ldots, [\sigma]V_n))$ Week 8 begin

Subst Static Call: $[\sigma](V_0{::}T.m\ (V_1, \ldots, V_n)) = (([\sigma]V_0){::}([\sigma]T).m\ ([\sigma]V_1, \ldots, [\sigma]V_n))$ Week 8 end

Subst Field Access: $[\sigma](V.f) = (([\sigma]V).f)$

Subst Field Update: $[\sigma](V_0.f := V_1) = (([\sigma]V_0).f := ([\sigma]V_1))$ Week 8 begin

Subst New Object:
$\text{new } T \{ f_1 = V_1, \ldots, f_n = V_n \}) = (\text{new } ([\sigma]T) \{ f_1 = ([\sigma]V_1), \ldots, f_n = ([\sigma]V_n) \}$ Week 8 end

4.11 Applying a substitution to a block

We define $[\sigma]B$ to be the block resulting from substituting a closed substitution σ into block B: Week 8 begin

Subst Let: $[\sigma](\text{let } x : T = V;\ B) = (\text{let } x : ([\sigma]T) = ([\sigma]E);\ [\sigma - x]B)$ Week 8 end

Subst If: $[\sigma](\text{if } (V) \{ B_1 \} \text{ else } \{ B_2 \}) \ = \ (\text{if } ([\sigma]V) \{ [\sigma]B_1 \} \text{ else } \{ [\sigma]B_2 \})$

Subst Return: $[\sigma](\text{return } V;) = (\text{return } [\sigma]V;)$

For example:

```
 [foo := 37, bar := Fish, baz := "fred"](let a = foo+bar; return baz;)
   = let a = 37+Fish; [foo := 37, bar := fish, baz := "fred"](return baz;)
   = let a = 37+Fish; return "fred";
```

4.12 Comments on variable scope

Note that in Subst Let (and similarly in other cases) we remove *x* from σ before substituting into *B*. This is to avoid substituting into a bound variable; if we did not make this restriction then we would have:

```
 thread Main { let x = 1; let x = 2; let y = x; }
→
 thread Main { [x:=1] let x = 2; let y = x; }
=
 thread Main { let x = 2; [x:=1] let y = x; }
   (wrong!)
=
 thread Main { let x = 2; let y = 1; }
→
 thread Main { [x:=2] let y = 1; }
```

```
→
  thread Main { let y = 1; }
```

This is the wrong result! Instead, we remove *x* out of the substitution to get:

```
  thread Main { let x = 1; let x = 2; let y = x; }
→
  thread Main { [x:=1] let x = 2; let y = x; }
=
  thread Main { let x = 2; [ ] let y = x; }
    (right!)
=
  thread Main { let x = 2; let y = x; }
→
  thread Main { [x:=2] let y = x; }
→
  thread Main { let y = 2; }
```

We also require that in any substitution $[\sigma]B$ that σ is closed. This is to ensure that free variables do not become captured; if we did not make this restriction then we would have:

```
 thread Main { let x = y; let y = 17; let z = x; }
→
 thread Main { [x:=y] let y = 17; let z = x; }
   (not allowed, since [x:=y] is not closed)
=
 thread Main { let y = 17; [x:=y] let z = x; }
=
 thread Main { let y = 17; let z = y; }
→
 thread Main { [y:=17] let z = y; }
=
 thread Main { let z = 17; }
```

Languages which do not make this requirement are called *dynamically scoped*. Hobbes is *statically scoped*, so we make this requirement on substitutions. Week 5 begin

5 Continuations

Most programming languages have a notion of the *call stack*: this keeps track of the call history of the program. In Hobbes, the behaviour of the call stack is specified using *continuations*. A continuation gives an abstract definition of the behaviour of the call stack.

Continuations are introduced when a method is called, for example:

```
class C {
  method this.foo (n : Integer) : Integer {
    let tmp = n + 1;
    return tmp;
  }
}
object O : C {}
thread Main {
  let x = O:C.foo (5);
  return x*2;
}
```

$\longrightarrow$

```
class C {
  method this.foo (n : Integer) : Integer {
    let tmp = n + 1;
    return tmp;
  }
}
object O : C {}
thread Main {
  [ this := O, n := 5 ]
  let tmp = n + 1;
  return tmp;
  continuation (x) {
    return x*2;
  }
}
```

$=$

```
class C {
  method this.foo (n : Integer) : Integer {
    let tmp = n + 1;
    return tmp;
  }
}
object O : C {}
thread Main {
  [ this := O, n := 5 ]
  let tmp = n + 1;
  let x = tmp;
  return x*2;
}
```

Informally, the behaviour of a continuation is:

(B; return V;) continuation (X) { C }
  $=B$; let X = V;C

This is made formal in this section.

5.1 Definition of a continuation

A continuation is just a binding for a variable:

```
Continuation ::=
  "(" Var ")" "{" Block "}"
```

For example:

```
(x:Integer) { let y = x*2; return y+5; }
```

We let K range over Continuation.

5.2 Free variables of a continuation

The free variables of a continuation are the free variables of the body, not including the binding variable:

$$\mathrm{fv}((x : T)\ \{\ B\ \}) = \mathrm{fv}(B) - x$$

For example:

fv((x:Integer) { let y = x*2; return y+5; })

$=$ fv(let y = x*2; return y+5;)$-$x
$= \{$ x $\}-$x
$= \emptyset$

A continuation is *closed* if $\text{fv}(K) = \emptyset$. For example (x) { let y = x*2; return y+5; } is closed. but (x) { return y; } is not.

5.3 Applying a continuation to a block

We apply a closed continuation K to a block B to produce the block B continuation K, defined:

Cont Let: ((let X = E; B) continuation K) $=$ (let X = E; (B continuation K))

Cont If: ((if (V) { B_1 } else { B_2 }) continuation K)
$=$ (if (V) { B_1 continuation K } else { B_2 continuation K })

Cont Return: ((return V;) continuation (X){ B }) $=$ (let X = V; B)

5.4 Comments on continuations

In Hobbes, continuations are not first-class citizens: there are no continuation variables, no continuation types, and there is no way for a program to find the current continuation object. This is different from LISP-like languages, where continuations can be treated like any other data, and the current continuation can be found using call-cc.

The continuation operation is only defined on closed continuations, otherwise we would introduce dynamic binding, for example:

```
 class C { method foo () { let x = 5; return Nothing; } }
 object O : C { }
 thread Main {
  let tmp = O:C.foo ();
  return x;
 }
→
 class C { method foo () { let x = 5; return Nothing; } }
 object O : C { }
 thread Main {
    let x = 5;
    return Nothing;
    continuation (tmp) { return x; }
  }
  (not allowed, since continuation (tmp) { return x; } is not closed)
=
  class C { method foo () { let x = 5; return Nothing; } }
  object O : C { }
  thread Main {
    let x = 5;
    let tmp = Nothing;
    return x;
  }
```

Since Hobbes is statically scoped, we require continuations to be closed. Week 5 end

6 Dynamic semantics of user code

This is the most important section of this document! It defines the dynamic semantics of Hobbes programs, by defining a *reduction relation* $P \rightarrow Q$ where P and Q are closed programs. Native classes are considered in the next section, in this section we discuss user code.

6.1 Definition of the dynamic semantics

The dynamic semantics is defined as a relation $P \rightarrow Q$ given by the following rules:

Dynamic Structural Equivalence: If $P_1 \equiv P_2$, $P_2 \rightarrow Q_2$ and $Q_2 \equiv Q_1$ then $P_1 \rightarrow Q_1$.

Week 8 begin

Dynamic Dynamic Call: $($thread a { let X = $b[T_1, \ldots, T_j]$.m $(V_1, \ldots, V_n)$; B } $P) \rightarrow$
$($thread a { let X = $b[T_1, \ldots, T_j] : ([A_1 := T_1, \ldots, A_j := T_j]T)$.$m$ $(V_1, \ldots, V_n)$; B } $P)$
where object $b[A_1, \ldots, A_j] : T$ { $\cdots$ } $\in P$

Week 9 begin

Dynamic Static Call:

$(\text{thread } a \{ \text{ let } X = b[U_1, \backslash ldots, U_k] : c[T_1, \ldots, T_j].m\ (V_1, \ldots, V_n);\ B_1 \}\ P)$
$\rightarrow\ (\text{thread } a \{ [y := [B_1 := U_1, \ldots, B_k := U_k]T, A_1 := T_1, \ldots, A_j := T_j, X_0 :=$
$V_0, \ldots, X_n := V_n]B_0 \text{ continuation } (X) \{ B_1 \} \}\ P)$
where class $c[A_1, \ldots, A_j] \cdots$ ex-
ended $y \{ \cdots \text{ method } X_0.m\ (X_1, \ldots, X_n) : T \{ B_0 \} \cdots \} \in P$
and object $b[B_1, \ldots, B_k] : T \cdots \in P$

Week 9 end

Dynamic Static Call Inherit:

$(\text{thread } a \{ \text{ let } X = V_0 : c[T_1, \ldots, T_j].m\ (V_1, \ldots, V_n);\ B \}\ P)$
$\rightarrow\ (\text{thread } a \{ \text{ let } X = V_0 : ([A_1 := T_1, \ldots, A_j := T_j]T).m\ (V_1, \ldots, V_n);\ B \}\ P)$
where class $c[A_1, \ldots, A_j]$ extends $T \cdots \{ M_1 \ldots M_i\ F_1 \ldots F_j \} \in P$
and $(\text{method } m \cdots) \notin \{M_1 \ldots M_i\}$

Dynamic Field Access: $(\text{thread } a \{ \text{ let } X = b[T_1, \ldots, T_j].f;\ B \}\ P)$
$\rightarrow\ (\text{thread } a \{ [X := [A_1 := T_1, \ldots, A_j := T_j]V]B \} \}\ P)$
where object $b[A_1, \ldots, A_j] : T \{ \cdots f = V \cdots \} \in P$

Week 8 end

Dynamic Field Update:

$(\text{thread } a \{ \text{ let } X = b.f_0 := V;\ B \} \text{ object } b : T \{ f_0 = V_0, f_1 = V_1, \ldots, f_n = V_n \}\ P)$

$\rightarrow$ (thread a { $[X := V_0]B$ } object $b : T$ { $f_0 = V, f_1 = V_1, \ldots, f_n = V_n$ } P)

Dynamic New Object: (thread a { let X = new T { $f_1 = V_1, \ldots, f_n = V_n$ }; B } P)
$\rightarrow$ (object $b : T$ { $f_1 = V_1, \ldots, f_n = V_n$ } thread a { $[X := b]B$ } P)
where b is a fresh global identifier

Dynamic Let: (thread a { let X = V; B } P) $\rightarrow$ (thread a { $[X := V]B$ } P)

Dynamic If True: (thread a { if (True) { B_1 } else { B_2 } } P) $\rightarrow$ (thread a { B_1 } P)

Dynamic If False: (thread a { if (False) { B_1 } else { B_2 } } P) $\rightarrow$ (thread a { B_2 } P)

Week 5 begin

For example if we define:

$P =$
```
mutable class IntRef {
  field contents : Integer;
  method this.inc () : Integer {
    let tmp1 = this.contents;
    let tmp2 = tmp1+1;
    let tmp3 = this.contents := tmp2;
```

```
      return tmp2;
    }
  }
```

then we have:

```
 P
 object R : IntRef { contents = 5; }
 thread Main {
   let x = R.inc ();
   let y = new IntRef { contents = x; }
   return y;
 }
```

$\rightarrow$ (Using Dynamic Dynamic Call)

```
 P
 object R : IntRef { contents = 5; }
 thread Main {
   let x = R : IntRef.inc ();
   let y = new IntRef { contents = x; }
   return y;
 }
```

$\longrightarrow$ (Using Dynamic Static Call)

```
 P
 object R : IntRef { contents = 5; }
 thread Main {
   [ this := R ]
   let tmp1 = this.contents;
   let tmp2 = tmp1+1;
   let tmp3 = this.contents := tmp2;
   return tmp2;
   continuation (x) {
     let y = new IntRef { contents = x; }
     return y;
   }
 }
```

$=$

```
 P
 object R : IntRef { contents = 5; }
 thread Main {
   let tmp1 = R.contents;
   let tmp2 = tmp1+1;
   let tmp3 = R.contents := tmp2;
```

```
    let y = new IntRef { contents = tmp2; }
    return y;
  }
```
$\longrightarrow$ (Using Dynamic Field Access)

P
```
  object R : IntRef { contents = 5; }
  thread Main {
    [ tmp1 := 5 ]
    let tmp2 = tmp1+1;
    let tmp3 = R.contents := tmp2;
    let y = new IntRef { contents = tmp2; }
    return y;
  }
```
$\longrightarrow$ (Using Dynamic Integer infix +)

P
```
  object R : IntRef { contents = 5; }
  thread Main {
    let tmp3 = R.contents := 6;
    let y = new IntRef { contents = 6; }
    return y;
  }
```

$\longrightarrow$ (Using Dynamic Field Update)

```
P
object R : IntRef { contents = 6; }
thread Main {
  [ tmp3 := 5 ]
  let y = new IntRef { contents = 6; }
  return y;
}
```

$\longrightarrow$ (Using Dynamic New Object)

```
P
object R : IntRef { contents = 6; }
object S : IntRef { contents = 6; }
thread Main {
  return S;
}
```

Week 5 end

6.2 Comments on the dynamic semantics

This semantics is based on a number of sources, principally Abadi and Cardelli's object calculus, Gordon and Hankin's concurrent object calculus, and Bruce's LOOM.

Note that the only difference between the dynamic semantics of dynamic and static method dispatch is when the run-time class of the object is resolved: for dynamic method dispatch it is determined at run-time, where for static method dispatch it is determined before run-time. Week 5 begin

The only rule which makes use of continuations is Dynamic Static Call, which introduces a new continuation to the call history. Week 5 end

7 Dynamic semantics of native classes

There are five native classes in Hobbes: Boolean, Integer, String, Thread and Void. Their dynamic semantics is given here.

7.1 Boolean

The native Boolean class has no native methods, so requires no dynamic semantics.

7.2 Integer

The native Integer class has native methods for arithmetic and comparisons. Their dynamic semantics is given:

Integer prefix -: (thread a { let X = -i; B } P) $\rightarrow$ (thread a { $[X := (-i)]B$ } P)

Integer prefix \$: (thread a { let X = \$$i$; B } P) $\rightarrow$ (thread a { $[X := (\text{"}i\text{"})]B$ } P)

Integer infix +: (thread a { let X = (i + j); B } P) $\rightarrow$ (thread a { $[X := (i + j)]B$ } P)

Integer infix -: (thread a { let X = (i - j); B } P) $\rightarrow$ (thread a { $[X := (i - j)]B$ } P)

Integer infix *: (thread a { let X = (i * j); B } P) $\rightarrow$ (thread a { $[X := (i \times j)]B$ } P)

Integer infix <: $(\text{thread } a\ \{ \text{ let } X = (i < j); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (i\ <\ j)]B\ \}\ P)$

Integer infix <=:
$(\text{thread } a\ \{ \text{ let } X = (i \text{ <= } j); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (i\ \leq\ j)]B\ \}\ P)$

Integer infix >: $(\text{thread } a\ \{ \text{ let } X = (i > j); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (i\ >\ j)]B\ \}\ P)$

Integer infix >=:
$(\text{thread } a\ \{ \text{ let } X = (i \text{ >= } j); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (i\ \geq\ j)]B\ \}\ P)$

Integer infix ==:
$(\text{thread } a\ \{ \text{ let } X = (i \text{ == } j); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (i\ =\ j)]B\ \}\ P)$

Integer infix !=:
$(\text{thread } a\ \{ \text{ let } X = (i \text{ != } j); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (i\ \neq\ j)]B\ \}\ P)$

Thread infix ==:
$(\text{thread } a\ \{ \text{ let } X = (a \text{ == } b); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (a\ =\ b)]B\ \}\ P)$

Thread infix !=:
$(\text{thread } a\ \{ \text{ let } X = (a \text{ != } b); \text{B } \}\ P) \rightarrow (\text{thread } a\ \{\ [X := (a\ \neq\ b)]B\ \}\ P)$

For the boolean operations, the result of a equality comparison $i = j$ is the value True (if $i = j$) or the value False (if $i \neq j$), and similarly for the other comparisons.

7.3 String

The native String class has a native method for string concatenation. Its dynamic semantics is given:

String infix +: (thread a { let X = ("s" + "t"); B } P) $\rightarrow$ (thread a { $[X :=$"st"$]B$ } P)

7.4 Thread

The native Thread class has no native methods, so requires no dynamic semantics.

7.5 Void

The native Void class has no native methods, so requires no dynamic semantics.

8 Examples

8.1 Simple arithmetic

```
// Before type inference
import "Base.hob";
thread Main {
  let tmp4 = 1 + 2;
  let tmp3 = $tmp4;
  let tmp2 = "1 + 2 = " + tmp3;
  let tmp1 = Out.println(tmp2);
  return Nothing;
}
```

```
// Initial state typechecked OK!

// After type inference
// Step 1
import "Base.hob";
thread Main {
  let tmp4 : Integer = 1 + 2;
  let tmp3 : String = $tmp4;
  let tmp2 : String = "1 + 2 = " + tmp3;
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}


——>

// Step 2
import "Base.hob";
thread Main {
  [ tmp4 := 3 ]
  let tmp3 : String = $tmp4;
  let tmp2 : String = "1 + 2 = " + tmp3;
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}

—–>

// Step 3
import "Base.hob";
thread Main {
  [ tmp3 := "3", tmp4 := 3 ]
  let tmp2 : String = "1 + 2 = " + tmp3;
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}

—–>

// Step 4
import "Base.hob";
```

```
thread Main {
  [ tmp2 := "1 + 2 = 3", tmp3 := "3", tmp4 := 3 ]
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}

——>

// Step 5
import "Base.hob";
thread Main {
  return Nothing;
}

–/–>

// Final state typechecked OK!
```

8.2 Variable rebinding

```
// Before type inference
import "Base.hob";
thread Main {
  let x : Integer = 1 + 2;
  let x : Integer = x + 3;
  let tmp3 = $x;
  let tmp2 = "x = " + tmp3;
  let tmp1 = Out.println(tmp2);
  return Nothing;
}

// Initial state typechecked OK!

// After type inference
// Step 1
import "Base.hob";
thread Main {
  let x : Integer = 1 + 2;
  let x : Integer = x + 3;
  let tmp3 : String = $x;
  let tmp2 : String = "x = " + tmp3;
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}
```

—–>

```
// Step 2
import "Base.hob";
thread Main {
  [ x := 3 ]
  let x : Integer = x + 3;
  let tmp3 : String = $x;
  let tmp2 : String = "x = " + tmp3;
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}
```

—–>

```
// Step 3
import "Base.hob";
thread Main {
  [ x := 6 ]
  let tmp3 : String = $x;
  let tmp2 : String = "x = " + tmp3;
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}
```

——>

```
// Step 4
import "Base.hob";
thread Main {
  [ tmp3 := "6", x := 6 ]
  let tmp2 : String = "x = " + tmp3;
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}
```

```
——>

// Step 5
import "Base.hob";
thread Main {
  [ tmp2 := "x = 6", tmp3 := "6", x := 6 ]
  let tmp1 : Void = Out.println(tmp2);
  return Nothing;
}

——>

// Step 6
import "Base.hob";
thread Main {
  return Nothing;
}

–/–>
```

```
// Final state typechecked OK!
```

Week 5 begin

8.3 Integer references

```
// Before type inference
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 = this.contents := n;
    return Nothing;
  }
}
```

```
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  let x : IntRef = Factory.build(17);
  let tmp14 = *x;
  let tmp13 = $tmp14;
  let tmp12 = "Before: *x = " + tmp13;
  let tmp11 = Out.println(tmp12);
  let tmp10 = *x;
  let tmp9 = tmp10 + 5;
  let tmp8 = x <- tmp9;
  let tmp7 = *x;
  let tmp6 = $tmp7;
  let tmp5 = "After: *x = " + tmp6;
  let tmp4 = Out.println(tmp5);
  return Nothing;
}
```

```
// Initial state typechecked OK!

// After type inference
// Step 1
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  let x : IntRef = Factory.build(17);
  let tmp14 : Integer = *x;
  let tmp13 : String = $tmp14;
  let tmp12 : String = "Before: *x = " + tmp13;
  let tmp11 : Void = Out.println(tmp12);
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
```

——>

```
// Step 2
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
```

```
thread Main {
  let x : IntRef = Factory::IntRefFactory.build(17);
  let tmp14 : Integer = *x;
  let tmp13 : String = $tmp14;
  let tmp12 : String = "Before: *x = " + tmp13;
  let tmp11 : Void = Out.println(tmp12);
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
```

—–>

```
// Step 3
import "Base.hob";
object Factory : IntRefFactory {  }
```

```
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ n := 17, this := Factory, myType := IntRefFactory ]
  let tmp3 : IntRef = new IntRef{ contents=n };
  return tmp3;
  continuation (x : IntRef) {
    let tmp14 : Integer = *x;
    let tmp13 : String = $tmp14;
    let tmp12 : String = "Before: *x = " + tmp13;
    let tmp11 : Void = Out.println(tmp12);
    let tmp10 : Integer = *x;
    let tmp9 : Integer = tmp10 + 5;
    let tmp8 : Void = x <- tmp9;
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}

——>

// Step 4
import "Base.hob";
object Factory : IntRefFactory {  }
```

```
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  return Obj1;
  continuation (x : IntRef) {
    let tmp14 : Integer = *x;
    let tmp13 : String = $tmp14;
    let tmp12 : String = "Before: *x = " + tmp13;
    let tmp11 : Void = Out.println(tmp12);
    let tmp10 : Integer = *x;
    let tmp9 : Integer = tmp10 + 5;
    let tmp8 : Void = x <- tmp9;
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=17 }

—–>

// Step 5
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ x := Obj1 ]
  let tmp14 : Integer = *x;
  let tmp13 : String = $tmp14;
  let tmp12 : String = "Before: *x = " + tmp13;
  let tmp11 : Void = Out.println(tmp12);
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }
```

—–>

```
// Step 6
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  let tmp14 : Integer = *Obj1;
  [ x := Obj1 ]
  let tmp13 : String = $tmp14;
  let tmp12 : String = "Before: *x = " + tmp13;
  let tmp11 : Void = Out.println(tmp12);
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }

—–>

// Step 7
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ this := Obj1, myType := IntRef ]
  let tmp1 : Integer = this.contents;
  return tmp1;
  continuation (tmp14 : Integer) {
    [ x := Obj1 ]
    let tmp13 : String = $tmp14;
    let tmp12 : String = "Before: *x = " + tmp13;
    let tmp11 : Void = Out.println(tmp12);
    let tmp10 : Integer = *x;
    let tmp9 : Integer = tmp10 + 5;
    let tmp8 : Void = x <- tmp9;
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=17 }

—–>

// Step 8
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  return 17;
  continuation (tmp14 : Integer) {
    [ x := Obj1 ]
    let tmp13 : String = $tmp14;
    let tmp12 : String = "Before: *x = " + tmp13;
    let tmp11 : Void = Out.println(tmp12);
    let tmp10 : Integer = *x;
    let tmp9 : Integer = tmp10 + 5;
    let tmp8 : Void = x <- tmp9;
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=17 }

——>

// Step 9
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ tmp14 := 17, x := Obj1 ]
  let tmp13 : String = $tmp14;
  let tmp12 : String = "Before: *x = " + tmp13;
  let tmp11 : Void = Out.println(tmp12);
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }

—–>

// Step 10
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
```

```
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ tmp13 := "17", tmp14 := 17, x := Obj1 ]
  let tmp12 : String = "Before: *x = " + tmp13;
  let tmp11 : Void = Out.println(tmp12);
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }
```

——>

```
// Step 11
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
```

```
}
thread Main {
  [ tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, x := Obj1 ]
  let tmp11 : Void = Out.println(tmp12);
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }

——>

// Step 12
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
 [ tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, x := Obj1 ]
  let tmp10 : Integer = *x;
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }
```

—>

```
// Step 13
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  let tmp10 : Integer = *Obj1;
 [ tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, x := Obj1 ]
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }
```

—>

```
// Step 14
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ this := Obj1, myType := IntRef ]
  let tmp1 : Integer = this.contents;
  return tmp1;
  continuation (tmp10 : Integer) {
   [ tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, x := Obj1 ]
    let tmp9 : Integer = tmp10 + 5;
    let tmp8 : Void = x <- tmp9;
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=17 }

——>
```

```
// Step 15
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  return 17;
  continuation (tmp10 : Integer) {
   [ tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, x := Obj1 ]
    let tmp9 : Integer = tmp10 + 5;
    let tmp8 : Void = x <- tmp9;
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=17 }

——>

// Step 16
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
 [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, x := O
  let tmp9 : Integer = tmp10 + 5;
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }
```

——>

```
// Step 17
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
```

```
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp9
  let tmp8 : Void = x <- tmp9;
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }

——>

// Step 18
```

```
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  let tmp8 : Void = Obj1 <- 22;
  [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp9
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=17 }

—–>

// Step 19
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ n := 22, this := Obj1, myType := IntRef ]
  let tmp2 : Integer = this.contents := n;
  return Nothing;
  continuation (tmp8 : Void) {
   [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp9
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=17 }

—–>

// Step 20
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  return Nothing;
  continuation (tmp8 : Void) {
   [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp9
    let tmp7 : Integer = *x;
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=22 }

——>

// Step 21
```

```
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
 [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp8
```

```
ing, tmp9 := 22, x := Obj1 ]
  let tmp7 : Integer = *x;
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=22 }

——>

// Step 22
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  let tmp7 : Integer = *Obj1;
  [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp8 
ing, tmp9 := 22, x := Obj1 ]
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=22 }
```

——>

```
// Step 23
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ this := Obj1, myType := IntRef ]
  let tmp1 : Integer = this.contents;
  return tmp1;
  continuation (tmp7 : Integer) {
   [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp8
ing, tmp9 := 22, x := Obj1 ]
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=22 }

––>

// Step 24
import "Base.hob";
object Factory : IntRefFactory {  }
```

```
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  return 22;
  continuation (tmp7 : Integer) {
   [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp8
ing, tmp9 := 22, x := Obj1 ]
    let tmp6 : String = $tmp7;
    let tmp5 : String = "After: *x = " + tmp6;
    let tmp4 : Void = Out.println(tmp5);
    return Nothing;
  }
}
object Obj1 : IntRef { contents=22 }

——>

// Step 25
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp7
ing, tmp9 := 22, x := Obj1 ]
  let tmp6 : String = $tmp7;
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=22 }

——>
```

```
// Step 26
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
```

```
thread Main {
 [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp6
ing, tmp9 := 22, x := Obj1 ]
  let tmp5 : String = "After: *x = " + tmp6;
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=22 }
```

——>

```
// Step 27
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  [ tmp10 := 17, tmp11 := Nothing, tmp12 := "Before: *x = 17", tmp13 := "17", tmp14 := 17, tmp5
ter: *x = 22", tmp6 := "22", tmp7 := 22, tmp8 := Nothing, tmp9 := 22, x := Obj1 ]
  let tmp4 : Void = Out.println(tmp5);
  return Nothing;
}
object Obj1 : IntRef { contents=22 }

——>

// Step 28
```

```
import "Base.hob";
object Factory : IntRefFactory {  }
class IntRef {
  mutable field contents : Integer;
  method prefix *() : Integer {
    let tmp1 : Integer = this.contents;
    return tmp1;
  }
  method infix <-(n : Integer) : Void {
    let tmp2 : Integer = this.contents := n;
    return Nothing;
  }
}
class IntRefFactory {
  method build(n : Integer) : IntRef {
    let tmp3 : IntRef = new IntRef{ contents=n };
    return tmp3;
  }
}
thread Main {
  return Nothing;
}
object Obj1 : IntRef { contents=22 }

–/–>

// Final state typechecked OK!
```

Week 8 begin

8.4 Generic references

```
// Before type inference
import "Base.hob";
thread Main {
  let x : Ref[Integer] = new Ref[Integer]{ contents=37 };
  let y : Ref[Integer] = x.clone();
  let tmp4 = y.set(45);
  let z : Integer = y.get();
  return Nothing;
}
```

```
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 = new Ref[a]{ contents=x };
    return tmp3;
  }
}

// Initial state typechecked OK!

// After type inference
// Step 1
```

```
import "Base.hob";
thread Main {
  let x : Ref[Integer] = new Ref[Integer]{ contents=37 };
  let y : Ref[Integer] = x.clone();
  let tmp4 : Void = y.set(45);
  let z : Integer = y.get();
  return Nothing;
}
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}


——>

// Step 2
import "Base.hob";
thread Main {
  [ x := Obj1 ]
  let y : Ref[Integer] = x.clone();
  let tmp4 : Void = y.set(45);
  let z : Integer = y.get();
  return Nothing;
}
object Obj1 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

——>

// Step 3
import "Base.hob";
thread Main {
  let y : Ref[Integer] = Obj1::Ref[Integer].clone();
  [ x := Obj1 ]
  let tmp4 : Void = y.set(45);
  let z : Integer = y.get();
  return Nothing;
}
object Obj1 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}
```

—–>

```
// Step 4
import "Base.hob";
thread Main {
  [ this := Obj1, a := Integer, myType := Ref[Integer] ]
  let x : a = this.contents;
  let tmp3 : Ref[a] = new Ref[a]{ contents=x };
  return tmp3;
  continuation (y : Ref[Integer]) {
    [ x := Obj1 ]
    let tmp4 : Void = y.set(45);
    let z : Integer = y.get();
    return Nothing;
  }
}
object Obj1 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

——>

// Step 5
import "Base.hob";
thread Main {
  [ this := Obj1, x := 37, a := Integer, myType := Ref[Integer] ]
  let tmp3 : Ref[a] = new Ref[a]{ contents=x };
  return tmp3;
  continuation (y : Ref[Integer]) {
    [ x := Obj1 ]
    let tmp4 : Void = y.set(45);
    let z : Integer = y.get();
    return Nothing;
  }
}
object Obj1 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

——>

// Step 6
import "Base.hob";
thread Main {
  return Obj2;
  continuation (y : Ref[Integer]) {
    [ x := Obj1 ]
    let tmp4 : Void = y.set(45);
    let z : Integer = y.get();
    return Nothing;
  }
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

---->

// Step 7
import "Base.hob";
thread Main {
  [ x := Obj1, y := Obj2 ]
  let tmp4 : Void = y.set(45);
  let z : Integer = y.get();
  return Nothing;
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
```

```
}

——>

// Step 8
import "Base.hob";
thread Main {
  let tmp4 : Void = Obj2::Ref[Integer].set(45);
  [ x := Obj1, y := Obj2 ]
  let z : Integer = y.get();
  return Nothing;
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

——>

// Step 9
import "Base.hob";
thread Main {
  [ n := 45, this := Obj2, a := Integer, myType := Ref[Integer] ]
  let tmp2 : a = this.contents := n;
  return Nothing;
  continuation (tmp4 : Void) {
    [ x := Obj1, y := Obj2 ]
    let z : Integer = y.get();
    return Nothing;
  }
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=37 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}
```

——>

```
// Step 10
import "Base.hob";
thread Main {
  return Nothing;
  continuation (tmp4 : Void) {
    [ x := Obj1, y := Obj2 ]
    let z : Integer = y.get();
    return Nothing;
  }
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=45 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}
```

$\longrightarrow$

```
// Step 11
import "Base.hob";
thread Main {
  [ tmp4 := Nothing, x := Obj1, y := Obj2 ]
  let z : Integer = y.get();
  return Nothing;
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=45 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

—–>

// Step 12
```

```
import "Base.hob";
thread Main {
  let z : Integer = Obj2::Ref[Integer].get();
  return Nothing;
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=45 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

——>

// Step 13
import "Base.hob";
thread Main {
  [ this := Obj2, a := Integer, myType := Ref[Integer] ]
  let tmp1 : a = this.contents;
  return tmp1;
  continuation (z : Integer) {
    return Nothing;
  }
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=45 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

––>

// Step 14
import "Base.hob";
thread Main {
  return 45;
  continuation (z : Integer) {
    return Nothing;
  }
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=45 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}
```

—>

```
// Step 15
import "Base.hob";
thread Main {
  return Nothing;
}
object Obj1 : Ref[Integer] { contents=37 }
object Obj2 : Ref[Integer] { contents=45 }
class Ref[type a] {
  mutable field contents : a;
  method get() : a {
    let tmp1 : a = this.contents;
    return tmp1;
  }
  method set(n : a) : Void {
    let tmp2 : a = this.contents := n;
    return Nothing;
  }
  method clone() : Ref[a] {
    let x : a = this.contents;
    let tmp3 : Ref[a] = new Ref[a]{ contents=x };
    return tmp3;
  }
}

–/–>

// Final state typechecked OK!
```

Week 8 end