Formal Methods: Lecture 14

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

The Infrastructure for Java: Continuations

Within continuations all the execution context is stored. This can be viewed as "the rest of the program" which needs to be executed. The two operators shown here are the ending point of an execution. Within the semantics we will define other operators with co-domain `Continuation` as needed.

```
fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
  op res : -> Continuation .
endfm
```

The Infrastructure for Java: Arrays

The arrays are represented this way. The array environment maps indices (of sort `Int`) to locations. An array is built as `a(Type, [0, Loc])` for example.

```
fmod ARRAY-ENV is ex INT . ex LOCATION . pr TYPE .
  sort ArrayEnv .
  op noArrayEnv : -> ArrayEnv .
  op [_,_] : Int Location -> ArrayEnv .
  op __ : ArrayEnv ArrayEnv -> ArrayEnv [assoc comm id: noArrayEnv] .
endfm

fmod ARRAY is ex ARRAY-ENV . ex VALUE .
  sort Array .
  subsort Array < Value .
  op a : Type ArrayEnv -> Array .
endfm
```

The Infrastructure for Java: Outputs

The outputs of our language are values, wrapped in specific operators and connected as below:

```
fmod OUTPUT is ex VALUE .
  ex STORE .
  sort Output .
  op O : Value -> Output .
  op O : Value Store -> Output .
  op noOutput : -> Output .
  op _,_ : Output Output -> Output [assoc id: noOutput] .
endfm
```

The Infrastructure for Java: State

The state is made up of state attributes which are the environment, store, output and a counter for the next free memory location each wrapped in some operator. It has the usual multiset union composition operator.

```
fmod STATE is extending ENVIRONMENT . extending STORE .
  extending CONTINUATION . ex OUTPUT .
  sorts StateAttribute MyState .
  subsort StateAttribute < MyState .
  op empty : -> MyState .
  op _,_ : MyState MyState -> MyState [assoc comm id: empty] .
  op e : Env -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
  op out : Output -> StateAttribute .
```

The Infrastructure for Java: State (II)

```
  sort SuperState .
  sort WrappedState .
  subsort WrappedState < SuperState .
  op noState : -> WrappedState .
  op state : MyState -> WrappedState .
  op k : Continuation -> SuperState .
  op _,_ : WrappedState WrappedState -> WrappedState [assoc comm id: noState] .
  op _,_ : SuperState SuperState -> SuperState [assoc comm id: noState] .
endfm
```

This is some more of the state which is needed because we did not want the context, i.e. the `Continuation`, to be part of the state but to be composable with it. So instead of having `e(..)`, `m(..)`, `n(..)`, `out(..)`, `k(..)` we now have `state(e(..), m(..), n(..), out(..))`, `k(..)` which is of sort `SuperState`.

The Infrastructure for Java: Helpers

There are a few helper functionalities which are needed that are not part of Java but which we use in our evaluation of Java code. This operator returns the value of a variable in a given state.

```
var MYS : MyState . var V : Value . var X : Name .
var L : Location . var Env : Env . var M : Store .
op _[_] : WrappedState Name -> Value .
eq state((MYS, e([X,L] Env), m([L,V] M)))[X]
   = V .
```

The Infrastructure for Java: Helpers (II)

These equations are needed to get non-ground term reduced in the decision procedure we use for proving properties of programs.

```
vars I J I' : Int .
eq 0 + I = I .
eq _-_(I, J) = I + (- J) .
eq I' * (I + J) = (I' * I) + (I' * J) .
```

The Infrastructure for Java: Helpers (III)

These operators extract the initial, guard and body parts of a loop respectively.

```
op extInit_ : BlockStatements -> BlockStatements .
op extGuard_ : BlockStatements -> Exp .
op extBody_ : BlockStatements -> BlockStatements .

var BS : BlockStatements . var E : Exp . var S : Statement .

eq extInit(while E S) = ; .
eq extInit(BS while E S) = BS .
eq extBody(while E S) = S .
eq extBody(BS while E S) = S .
eq extGuard(while E S) = E .
eq extGuard(BS while E S) = E .
```

The Semantics of Java: Generic Expressions

The first two operators allow for expression and value lists on the top of a continuation. The `int` operator creates a value of an integer. The semantics of an integer expression is to become an integer value. This module is the foundation upon which all the following semantic equations rely. Most equations here transform non-Java syntax intermediate continuations which are created by the rules for each feature.

```
fmod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op _->_ : Exps Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op int : Int -> Value .
  op str : String -> Value .
```

The Semantics of Java: Generic Expressions (II)

Integer and string expressions become integer, respectively string, values.

```
vars E E' : Exp .  var El : Exps .
vars V V' : Value .  var Vl Vl' : ValueList .
var L : Location .  var Ll : LocationList .
var I : Int .  var K : Continuation .
var M : Store .  vars Env Env' : Env .
var str : String . var MYS : MyState .

eq k(#i(I) -> K) = k(int(I) -> K) .
eq k(#s(str) -> K) = k(str(str) -> K) .
```

The Semantics of Java: Generic Expressions (III)

This makes sure that a list of expressions `E1, E2, E3, ...` is evaluated in the given order and has the result `V1, V2, V3, ...` where `Vi` is the result of `Ei`'s evaluation. This is useful to have arguments of a function evaluated. We also always focus only on the first elements of the continuation which is enough because of the way we defined it.

```
op [_|_] -> _ : Exps ValueList Continuation -> Continuation .
eq k((E,E',El) -> K) = k(E -> [E',El | noVal] -> K) .
eq k(V -> [() | Vl] -> K) = k((Vl,V) -> K) .
eq k(V -> [E,El | Vl] -> K) = k(E -> [El | Vl,V] -> K) .
eq k(() -> K) = k(noVal -> K) .
```

The Semantics of Java: Generic Expressions (IV)

Lists of locations can also be on the top of a continuation, as well as the `change` operator which will change the value at the given location to the given value. This will be helpful to define assignments when no result is needed.

```
op _->_ : LocationList Continuation -> Continuation .
op change (_,_) -> _ : Value Location Continuation -> Continuation .
eq k(noVal -> noLoc -> K), state(m(M), MYS) = k(K), state(m(M), MYS) .
eq k((V, Vl) -> (L, Ll) -> K), state(m([L, V'] M), MYS)
     = k(change(V, L) -> (Vl -> Ll -> K)), state(m([L, V'] M), MYS) .

eq k(change(V, L) -> K), state(m([L, V'] M), MYS)
     = k(K), state(m([L, V] M), MYS) .
```

The Semantics of Java: Generic Expressions (V)

Here `set&fetch` creates a second copy of the values that are assigned. Once all the values of the value list are assigned to the locations of the location list the only thing left (outside of the rest `K`) will be exactly the copy of the value list. This allows assignments where the result is also directly used afterwards.

```
op set&fetch_->_ : LocationList Continuation -> Continuation .

eq k(Vl -> set&fetch(Ll) -> K) = k(Vl -> Ll -> Vl -> K) .
```

The Semantics of Java: Generic Expressions (VI)

Restoring environments is important. It can be done while keeping some values on top of the continuation or without such values. This will be used for the handling of blocks.

```
  op _->_ : Env Continuation -> Continuation .
  eq k(Vl -> Env -> K), state(e(Env'), MYS) = k(Vl -> K), state(e(Env), MYS) .
  eq k(Env -> K), state(e(Env'), MYS) = k(K), state(e(Env), MYS) .
endfm
```

The Semantics of Java: Locations

This gives the semantics for locations on the continuation and will be used to get the location of a variable.

```
fmod LOCATION-SEMANTICS is ex GENERIC-EXP-SEMANTICS . pr ARRAY-SYNTAX .
  pr ARRAY .
  op loc_ -> _ : Exp Continuation -> Continuation .
  op loc -> _ : Continuation -> Continuation .
  vars E E' : Exp . var L : Location . var X : Name .
  var I : Int .  var K : Continuation . var Env : Env .
  var T : Type . var aEnv : ArrayEnv . var MYS : MyState .
  eq k(loc(X) -> K), state(e([X, L] Env), MYS)
       = k(L -> K), state(e([X, L] Env), MYS) .
  eq k(loc(E [ E' ]) -> K) = k((E, E') -> loc -> K) .
  eq k((a(T, [I,L] aEnv), int(I)) -> loc -> K) = k(L -> K) .
endfm
```

The Semantics of Java: Arithmetic Expressions

This gives the semantics for arithmetic expressions. Take for example addition, here first the two arguments are evaluated and then, if they are both integers, they are added together. All other operators are defined similarly.

```
fmod ARITH-EXP-SEMANTICS is protecting ARITH-EXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op + -> _ : Continuation -> Continuation .
  op - -> _ : Continuation -> Continuation .
  op * -> _ : Continuation -> Continuation .
  op / -> _ : Continuation -> Continuation .
  op % -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .  vars I I' : Int .
  vars str str' : String .
  eq k((E + E') -> K) = k((E,E') -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
```

The Semantics of Java: Arithmetic Expressions (II)

```
  eq k((E - E') -> K) = k((E,E') -> - -> K) .
  eq k((int(I), int(I')) -> - -> K) = k(int(I - I') -> K) .
  eq k((- E) -> K) = k(E -> - -> K) .
  eq k(int(I) -> - -> K) = k(int(- I) -> K) .
  eq k((E * E') -> K) = k((E,E') -> * -> K) .
  eq k((int(I), int(I')) -> * -> K) = k(int(I * I') -> K) .
  eq k((E / E') -> K) = k((E,E') -> / -> K) .
  eq k((int(I), int(I')) -> / -> K) = k(int(I quo I') -> K) .
  eq k((E % E') -> K) = k((E,E') -> % -> K) .
  eq k((int(I), int(I')) -> % -> K) = k(int(I rem I') -> K) .
endfm
```

The Semantics of Java: Comparisons

The semantics of comparisons are defined similarly to those of arithmetic expressions. First evaluate the subexpressions and then compare the results. They are all defined similarly.

```
fmod REXP-SEMANTICS is pr REXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  op == -> _ : Continuation -> Continuation .
  op != -> _ : Continuation -> Continuation .
  op < -> _  : Continuation -> Continuation .
  op <= -> _ : Continuation -> Continuation .
  op > -> _  : Continuation -> Continuation .
  op >= -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .
  eq k(#b(B) -> K) = k(bool(B) -> K) .
```

The Semantics of Java: Comparisons (II)

```
  eq k((E == E') -> K) = k((E,E') -> == -> K) .
  eq k((int(I),int(I')) -> == -> K) = k(bool(I == I') -> K) .
  eq k((E != E') -> K) = k((E,E') -> != -> K) .
  eq k((int(I),int(I')) -> != -> K) = k(bool(not (I == I')) -> K) .
  eq k((E >= E') -> K) = k((E,E') -> >= -> K) .
  eq k((int(I),int(I')) -> >= -> K) = k(bool(I >= I') -> K) .
  eq k((E > E') -> K) = k((E,E') -> > -> K) .
  eq k((int(I),int(I')) -> > -> K) = k(bool(I > I') -> K) .
  eq k((E <= E') -> K) = k((E,E') -> <= -> K) .
  eq k((int(I),int(I')) -> <= -> K) = k(bool(I <= I') -> K) .
  eq k((E < E') -> K) = k((E,E') -> < -> K) .
  eq k((int(I),int(I')) -> < -> K) = k(bool(I < I') -> K) .
endfm
```

The Semantics of Java: Boolean Expressions

The evaluation of boolean expressions is a bit more complicated, the simple case, negation, can be seen here, conjunction and disjunction follow.

```
fmod BEXP-SEMANTICS is pr BEXP-SYNTAX . ex REXP-SEMANTICS .
  op ! -> _  : Continuation -> Continuation .
  op && -> _ : Continuation -> Continuation .
  op || -> _ : Continuation -> Continuation .

  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .

  eq k((! E) -> K) = k(E -> ! -> K) .
  eq k(bool(B) -> ! -> K) = k(bool(not B) -> K) .
```

The Semantics of Java: Boolean Expressions (II)

The evaluation of boolean expressions needs a bit more subtlety. According to the Java Language Specification the result of a conjunction is `false` whenever the first argument evaluates to `false` and in that case the second argument is not evaluated! The definition for disjunction works similarly.

```
  eq k((E && E') -> K) = k(E -> && -> (E' -> K)) .
  eq k(bool(false) -> && -> (E' -> K)) = k(bool(false) -> K) .
  eq k(bool(true) -> && -> (E' -> K)) = k(E' -> && -> (bool(true) -> K)) .
  eq k(bool(B) -> && -> (bool(B') -> K)) = k(bool(B' and B) -> K) .

  eq k((E || E') -> K) = k(E -> || -> (E' -> K)) .
  eq k(bool(true) -> || -> (E' -> K)) = k(bool(true) -> K) .
  eq k(bool(false) -> || -> (E' -> K)) = k(E' -> || -> (bool(false) -> K)) .
  eq k(bool(B) -> || -> (bool(B') -> K)) = k(bool(B' or B) -> K) .
endfm
```

The Semantics of Java: Assignment

The definition of assignment makes use of the previously defined `set&fetch` and `loc` operators and is therefore straightforward.

```
fmod ASSIGNM-EXP-SEMANTICS is ex ASSIGNM-EXP-SYNTAX .
  ex GENERIC-EXP-SEMANTICS .
  ex LOCATION-SEMANTICS .
  pr ARITH-EXP-SEMANTICS .
  op =_ -> _ : Exp Continuation -> Continuation .
  var E : Exp .  var K : Continuation .
  var L : Location .
  var X : Var .
  eq k((X = E) -> K) = k(loc(X) -> = (E) -> K) .
  eq k(L -> = (E) -> K) = k(E -> set&fetch(L) -> K) .
endfm
```

The Semantics of Java: Arrays

Arrays are accessed in the obvious way. Giving an integer as an argument to the array, if that index is defined then the corresponding memory location is accessed and the value returned.

```
fmod ARRAY-SEMANTICS is ex ARRAY-SYNTAX  .
  ex GENERIC-EXP-SEMANTICS .
  ex ARRAY .
  op [] -> _ : Continuation -> Continuation .
  vars E E' : Exp . var T : Type . var aEnv : ArrayEnv . var M : Store .
  var I : Int . var L : Location . var V : Value . var K : Continuation .
  var MYS : MyState .
  eq k((E [ E' ]) -> K) = k((E, E') -> [] -> K) .
  eq k((a(T, [I,L] aEnv), int(I)) -> [] -> K), state(m([L,V] M), MYS)
       = k(V -> K), state(m([L,V] M), MYS) .
endfm
```

The Semantics of Java: Field Access

In our subset of Java field access boils down to retrieving the value a (local) variable has, as arrays are handled separately. This also uses a previously defined helper operator.

```
fmod FIELD-SEMANTICS is ex GENERIC-EXP-SEMANTICS .
  ex LOCATION-SEMANTICS .
  ex HELPING-OPERATORS .
  var X : Name . var K : Continuation .
  var MYS : MyState .
  eq k(X -> K), state(MYS)  = k((state(MYS)[X]) -> K), state(MYS) .
endfm
```

The Semantics of Java: Types

The types of declarations and values are defined this way. These are also helper operators and not part of the Java syntax.

```
fmod GET-TYPES is ex TYPE . ex GENERIC-EXP-SEMANTICS .
  ex ARRAY-SEMANTICS . ex DECLARATION-SYNTAX .
  op GetType : Declaration -> Type .
  var qid : Qid . vars T : Type . var dn : Name . var E : Exp .
  var I : Int . var str : String . var aEnv : ArrayEnv .
  eq GetType(T qid) = T .
  eq GetType(T dn[]) = GetType((T []) dn) .
  eq GetType(T dn = E) = GetType(T dn) .
  op GetType : Value -> Type .
  eq GetType(int(I)) = int .
  eq GetType(str(str)) = String .
  eq GetType(a(T, aEnv)) = T .
endfm
```

The Semantics of Java: Blocks

The semantics of blocks is that all changes to the environment that happen within the block are discarded at the end of the block. Otherwise the execution proceeds as usual.

```
fmod BLOCK-SEMANTICS is ex BLOCK-SYNTAX .
  ex GENERIC-EXP-SEMANTICS .
  op _ -> _ : BlockStatements Continuation -> Continuation .
  var K : Continuation . var bs : BlockStatements . var Env : Env .
  var MYS : MyState .
  eq k( ; -> K) = k(K) .
  eq k({bs} -> K), state(e(Env), MYS)
       = k(bs -> Env -> K), state(e(Env), MYS) .
endfm
```

The Semantics of Java: Declarations

For declarations we first evaluate the expression that is assigned to the newly declared variable, if there is any. Then the variable is mapped to the next free location in the environment and that location is mapped to the initial value of the appropriate type in the store. If there is an assignment to the variable it is processed next, see next slide.

```
fmod DECLARATION-SEMANTICS is ex DECLARATION-STATEMENT-SYNTAX .
  ex GENERIC-EXP-SEMANTICS . ex GET-TYPES . ex BLOCK-SEMANTICS .
  op _ -> _ : Declaration Continuation -> Continuation .
  var T : Type . var K : Continuation . var dn : Name .
  var E : Exp . var Env : Env . var V : Value . var N : Int .
  var M : Store . var qid : Qid . var dc : Declaration .
  var bs : BlockStatements . var MYS : MyState .
  eq k((dc ;) -> K) = k(dc -> K) .
  eq k(((dc ;) bs) -> K) = k(dc -> bs -> K) .
```

The Semantics of Java: Declarations (II)

For this we also need to be able to get the name out of a declaration and define the initial values of any type, as in Java, unlike C/C++, a '0' value is assigned to each newly declared variable. `GetName` and `alloc` will be defined next.

```
eq k((T dn = E) -> K) = k(E -> (T dn) -> K) .
eq k((T dn) -> K), state(e(Env), n(N), m(M), MYS) =
   k(K), state(e(Env[GetName(dn) <- l(N)]), n(N + 1),
               m(M[l(N), alloc(GetType(T dn))]), MYS) .
eq k(V -> (T dn) -> K), state(e(Env), n(N), m(M), MYS) =
   k(change(V, l(N)) -> K),
   state(e(Env[GetName(dn) <- l(N)]), n(N + 1),
         m(M[l(N), alloc(GetType(T dn))]), MYS) .
```

The Semantics of Java: Declarations (III)

This defines the helpers `GetName` and `alloc`.

```
  op GetName : Name -> Qid .
  eq GetName(qid) = qid .
  eq GetName(dn[]) = GetName(dn) .
  op alloc : Type -> Value .
  eq alloc(int) = int(0) .
  eq alloc(String) = str("") .
  eq alloc(T[]) = a(T[], noArrayEnv) .
endfm
```

The Semantics of Java: Output

Outputs are gathered in the output state attribute in the order in which they are printed.

```
fmod OUTPUT-METHOD is ex GENERIC-EXP-SEMANTICS .
  pr OUTPUT .
  op print -> _ : Continuation -> Continuation .
  var E : Exp . var V : Value . var K : Continuation .
  var O : Output . var MYS : MyState .
  op System.out.print : Exp -> Exp .
  eq k((System.out.print(E)) -> K) = k(E -> print -> K) .
  eq k(V -> print -> K), state(out(O), MYS)
       = k(K), state(out(O,O(V)), MYS) .
endfm
```

The Semantics of Java: New

We only have arrays to worry about for the `new` operator, as we do not use objects. Here we define some helping operators.

```
fmod NEW-SEMANTICS is ex NEW-SYNTAX . pr GENERIC-EXP-SEMANTICS .
  ex GET-TYPES . ex DECLARATION-SEMANTICS . ex BLOCK-SEMANTICS .
  op GetArrayType : Type -> Type .
  op GetExps : Type -> Exps .
  var K : Continuation . var Vl : ValueList . var array : Array .
  var aEnv : ArrayEnv . vars I N : Int . var V : Value .
  var M : Store . var MYS : MyState . var Inotzero : NzInt .
  var TnoArr : TypeWithOutArr . var T : Type . var E : Exp .
  eq GetArrayType(T[E]) = GetArrayType(T) [] .
  eq GetArrayType(T[]) = GetArrayType(T) [] .
  eq GetArrayType(TnoArr) = TnoArr .
  eq GetExps(T[E]) = GetExps(T), E .
  eq GetExps(T[]) = GetExps(T) .
  eq GetExps(TnoArr) = () .
```

The Semantics of Java: New (II)

Now this defines how a new array is created and initialized.

```
  op newArray _ -> _ : Type Continuation -> Continuation .
  op newArray (_,_,_) -> _ : Array ValueList Int Continuation -> Continuation .
  eq k((new T) -> K) = k(GetExps(T) -> newArray (GetArrayType(T)) -> K) .
  eq k((int(I), Vl) -> newArray (T) -> K)
       = k(newArray(a(T, noArrayEnv), Vl, I) -> K) .
  eq k(noVal -> newArray(int) -> K) = k(int(0) -> K) .
  eq k(noVal -> newArray(String) -> K) = k(str("") -> K) .
  eq k(noVal -> newArray(T[]) -> K) = k(a(T[], noArrayEnv) -> K) .
  eq k(newArray(array, Vl, 0) -> K) = k(array -> K) .
  eq k(newArray(a(T[], aEnv), Vl, Inotzero) -> K) =
     k(Vl -> newArray(T) -> (newArray(a(T[], aEnv), Vl, (Inotzero - 1)) -> K)) .
  eq k(V -> newArray(a(T, aEnv), Vl, I) -> K), state(m(M), n(N), MYS) =
     k(newArray(a(T, ([I, l(N)] aEnv)), Vl, I) -> K),
     state(m(M[l(N), V]), n(N + 1), MYS) .
endfm
```

The Semantics of Java: Statements

A list of statements, called `BlockStatements` is split apart into the first statement and the rest of the list of statements. With this and the definition of declarations we never need to consider sequential compositions of statements but can focus on one statement at a time.

```
fmod STATEMENT-SEMANTICS is ex STATEMENT-SYNTAX .
  ex BLOCK-SEMANTICS .
  var st : Statement . var bs : BlockStatements .
  var K : Continuation .
  eq k((st bs) -> K) = k(st -> bs -> K) .
endfm
```

An example of this is `'x = 'x + 'Y ; 'y = 'y * 'y ; -> K` which is transformed to `'x = 'x + 'Y ; -> 'y = 'y * 'y ; -> K`

The Semantics of Java: Statements (II)

An assignment as a statement is executed and afterwards the resulting value, which is the one of the right-hand side expression, is dropped.

```
fmod EXP-STATEMENT-SEMANTICS is ex EXP-STATEMENT-SYNTAX .
  ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS .
  var SE : StExp . var K : Continuation . var Vl : ValueList .
  eq k((SE ; ) -> K) = k(SE -> ; -> K) .
  eq k(Vl -> ; -> K) = k(K) .
endfm
```

The Semantics of Java: Conditionals

Conditionals first evaluate their condition and then execute the correct branch.

```
fmod IF-SEMANTICS is ex IF-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex BEXP-SEMANTICS .
  op ? (_,_) -> _ : Statement Statement Continuation -> Continuation .
  var E : Exp . vars St St' : Statement . var K : Continuation .
  eq k((if E St else St') -> K) = k(E -> ? (St, St') -> K) .
  eq k((if E St) -> K) = k(E -> ? (St, ;) -> K) .
  eq k(bool(true) -> ? (St, St') -> K) = k(St -> K) .
  eq k(bool(false) -> ? (St, St') -> K) = k(St' -> K) .
endfm
```

An example for this is `if #b(true) 'x = 'x + #i(1) ; -> K` becomes `#b(true) -> ? ('x = 'x + #i(1) ;, ;) -> K` which in turn becomes `'x = 'x + #i(1) ; -> K`.

The Semantics of Java: Loops

While loops make use of the conditional to decide whether the loop needs to be executed once more.

```
fmod WHILE-SEMANTICS is ex WHILE-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex IF-SEMANTICS .
  var E : Exp . var St : Statement . var K : Continuation .
  eq k((while E St) -> K) = k(E -> ?({St while E St}, ;) -> K) .
endfm
```

The Semantics of Java: Loops (II)

The `do_while_` and `for` loops are transformed into `while` loops and handled accordingly.

```
fmod DO-SEMANTICS is ex DO-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex WHILE-SEMANTICS .
  var E : Exp . var St : Statement . var K : Continuation .
  eq k((do St while E ;) -> K) = k(St -> (while E St) -> K) .
endfm

fmod FOR-SEMANTICS is ex FOR-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex DECLARATION-SEMANTICS . ex STATEMENT-SEMANTICS .
  ex WHILE-SEMANTICS . ex EXP-STATEMENT-SYNTAX .
  var E : Exp . vars Se Se' : StExp . var St : Statement .
  var K : Continuation . var dc : Declaration .
  eq k((for (Se ; E ; Se') St) -> K)
      = k((Se ; while (E) { St Se' ; }) -> K) .
  eq k((for (dc ; E ; Se') St) -> K)
      = k((dc ; while (E) { St Se' ; }) -> K) .
endfm
```

The Semantics of Java: Completion

The full Java subset we are defining is constituted by all the given submodules.

```
fmod JAVAX is ex JAVA-SYNTAX .
  ex ARITH-EXP-SEMANTICS . ex REXP-SEMANTICS .
  ex BEXP-SEMANTICS . ex ASSIGNM-EXP-SEMANTICS .
  ex ARRAY-SEMANTICS . ex FIELD-SEMANTICS .
  ex BLOCK-SEMANTICS . ex DECLARATION-SEMANTICS .
  ex NEW-SEMANTICS . ex STATEMENT-SEMANTICS .
  ex EXP-STATEMENT-SEMANTICS . ex IF-SEMANTICS .
  ex WHILE-SEMANTICS . ex DO-SEMANTICS .
  ex FOR-SEMANTICS . ex OUTPUT .
```

The Semantics of Java: Completion (II)

The initial state is defined as a convenience as follows.

```
op initial : -> WrappedState .
eq initial = state(e(noEnv), m(noStore), n(0), out(noOutput)) .

var MYS : MyState . var M : Store . var N : Nat . var X : Name .
var I : Int . var BS : BlockStatements . var E : Exp . var V : Value .
var B : Bool . var Env : Env . var O : Output .
```

The Semantics of Java: Completion (III)

From a state and an expression (`Exp`) or code (`BlockStatements`) the following executable `SuperState` variants are created.

```
op _|_ : WrappedState BlockStatements -> SuperState [gather(E e)] .
op _|_ : WrappedState Exp -> SuperState [gather(E e)] .
eq (state(MYS) | BS)
     = (state(MYS), k(BS -> stop)) .
eq (state(MYS) | E)
     = (state(MYS), k(E -> stop)) .

eq (state(MYS), k(V -> stop))
     = state(MYS) .
eq (state(MYS), k(stop))
     = state(MYS) .
```

The Semantics of Java: Completion (IV)

Evaluating a variable in a given state is done in this way. Boolean expressions can also be evaluated and their result be returned.

```
op evalInt : WrappedState Var -> Int .
eq evalInt(state(MYS), X) = int-val(state(MYS)[X]) .

op evalTst : SuperState -> Bool .
op evalTst : WrappedState Exp -> Bool .
eq evalTst(state(e(Env), m(M), n(N), out(O)), E)
     = evalTst(state(e(Env), m(M), n(N), out(O)), k(E -> res)) .
eq evalTst(state(MYS), k(bool(B) -> res)) = B .
```

The Semantics of Java: Completion (V)

The following is necessary because we need specific starting states for proving properties with the ITP. It creates a starting state with some known variables which are mapped to generic values, representative of whatever is there (only `int` currently).

```
op any-int : Int -> Int .
op any-loc : Int -> Location .
op ctxState : BlockStatements WrappedState -> WrappedState .
op ctxState : BlockStatements Int WrappedState -> WrappedState .
op ctxState : BlockStatements -> WrappedState .
eq ctxState(B:BlockStatements)
     = ctxState(B:BlockStatements,
                state(e(noEnv), n(0), m(noStore), out(noOutput))) .
eq ctxState((B:BlockStatements), state(e(Env), n(N), m(M), out(O)))
     = ctxState((B:BlockStatements), 0,
                 state(e(Env), n(N), m(M), out(O))) .
```

The Semantics of Java: Completion (VI)

This finalizes the starting state creation for use with the ITP.

```
  eq ctxState((int X ; B:BlockStatements), I,
                state(e(Env), n(N), out(O), m(M)))
       = ctxState(B:BlockStatements, I + 1,
           state(e([X, any-loc(I)] Env), n(N), out(O),
                 m([any-loc(I), int(any-int(I))] M))) .

  eq ctxState((int X ;), I, state(e(Env), n(N), m(M), out(O)))
       = state(e([X, any-loc(I)] Env), n(N), out(O),
               m([any-loc(I), int(any-int(I))] M)) .

  op int-val : Value -> Int .
  eq int-val(int(I)) = I .
endfm
```