Formal Methods: Lecture 13

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Verification of Imperative Sequential Programs

We are now ready to consider the verification of sequential imperative programs. We will do so using a subset of the Java programming language.

Of course, for the formal verification of some properties Q about a program P in a sequential imperative language $\mathcal{L}$ to be meaningful at all, our first and most crucial task is to make sure that the programming language $\mathcal{L}$ has a clear and precise mathematical semantics, since only then can we settle mathematically whether a program P satisfies some properties Q.

Verification of Imperative Sequential Programs (II)

The issue of giving a mathematical semantics to a programming language $\mathcal{L}$ is actually nontrivial, particularly for imperative languages; it is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

For example, for a Maude functional module, its mathematical semantics is given by the initial algebra of its equational theory, whereas its operational semantics is based on equational simplification with its equations, which are assumed confluent and terminating.

Some imperative languages have never been given a precise semantics; their only precise documentation may be the different compilers, perhaps inconsistent with each other.

Verification of Imperative Sequential Programs (III)

Of course, there is no reason why we cannot, if we try, give a precise mathematical semantics to a programming language. If the language is very baroque, this may of course be a big task; and if some dark corners are undocumented, a rational reconstruction of the language, making precise what nobody made clear before and perhaps got the compiler writers into trouble, may be needed. But it can be done.

Those who think otherwise, or who would prefer to live with ambiguous programming languages engage in a form of anti-mathematical irrationalism. A quite different question is whether a specific programming language is worth the effort of giving it a mathematical semantics.

Verification of Imperative Sequential Programs (IV)

In the end, giving mathematical semantics to a programming language $\mathcal{L}$ amounts to giving a mathematical model of the language. This is typically done using some mathematical formalism: either the language of set theory, which is a de-facto universal formalism for mathematics, or some other well-defined formalism.

For sequential imperative languages equational formalisms are quite well-suited to the task. In traditional denotational semantics, a higher-order equational logic, namely the lambda calculus, is used. However, it was pointed out by a number of authors, including Joseph Goguen, that first-order equational logic is perfectly adequate for the task, and has some specific advantages.

Algebraic Semantics of Sequential Languages

The choice of first-order equational logic leads to a form of algebraic semantics of sequential imperative languages in which:

- the semantics of a programming language $\mathcal{L}$ is axiomatized as an equational theory $T_{\mathcal{L}}$;
- the mathematical semantics of the language is given by the initial algebra $T_{T_{\mathcal{L}}}$;
- if the equations in $T_{\mathcal{L}}$ are confluent, this also gives an operational semantics to the language, expressed in terms of equational simplification.

Algebraic Semantics of Sequential Languages (II)

In this setting, the program correctness question can be formulated as follows: given a program P in a sequential imperative language $\mathcal{L}$, and given some properties Q about P (where Q typically involves the text of P) we say that P satisfies Q iff,

$$T_{T_{\mathcal{L}}} \models Q.$$

Proof-theoretically, we use an inductive inference system, to try to prove,

$$T_{\mathcal{L}} \vdash_{ind} Q.$$

The Syntax of Java: Types

We illustrate these ideas with a fragment of Java. This fragment includes arithmetic expressions, assignments, sequential composition and loops.

We start with defining types for this language.

```
fmod TYPE is
  sort Type .
  sort TypeWithOutArr .
  subsort TypeWithOutArr < Type .
  ops int boolean String : -> TypeWithOutArr .
endfm
```

The Syntax of Java: Names

Next we introduce names which will be the basis of variables. Please note that the `equalName` operator is not part of the actual Java syntax but rather an extra operator.

```
fmod NAME is ex QID .
  pr INT .
  pr STRING .
  pr BOOL .
  sorts Name NameList .
  subsort Qid < Name < NameList .
  op _[] : Name -> Name [prec 50] .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: ()] .

  op equalName : Name Name -> Bool .
endfm
```

The Syntax of Java: Variables

Variables are defined this way:

```
fmod VAR-SYNTAX is ex NAME .
  sorts Var Vars .
  subsort Name < Var < Vars .
  subsort NameList < Vars .
  op _,_ : Vars Vars -> Vars [ditto] .
endfm
```

The Syntax of Java: Generic Expressions

This is our definition of generic expressions. Note the lists of expressions.

```
fmod GENERIC-EXP-SYNTAX is ex VAR-SYNTAX .
  sorts Exp Exps StExp .
  subsort Var StExp < Exp < Exps .
  subsort Vars < Exps .
  op #i : Int -> Exp .
  op #s : String -> Exp .
  op #b : Bool -> Exp .
  op _,_ : Exps Exps -> Exps [ditto] .
endfm
```

The Syntax of Java: Arithmetic Expressions

The arithmetic expressions are defined in the following way. In Java they are left-associative which is what the gather pattern assures.

```
fmod ARITH-EXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  ops _+_ _-_ : Exp Exp -> Exp [prec 40 gather(E e)] .
  ops _*_ _/_ _%_ : Exp Exp -> Exp [prec 35 gather(E e)] .
  op -_ : Exp -> Exp [prec 33] .
endfm
```

The Syntax of Java: Examples

Program variables, of sort `Var`, are terms of the form `'Q` with `Q` a character string such as `"foo"`. They are, of course, completely different from the mathematical variables that we use in equations.

Arithmetic expressions, of sort `Exp`, are then expressions formed from variables and integer constants using the usual arithmetic operations. For example,

```
'X

#i(99)

'X + (#i(7) * 'Y)

'Z - (- 'X + (#i(12) * 'W))
```

The Syntax of Java: Comparison and Bool Expressions

This defines the syntax of comparisons and boolean connectives.

```
fmod REXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  ops _==_ _!=_ _<_  _<=_ _>_ _>=_ : Exp Exp -> Exp [prec 45] .
endfm

fmod BEXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  op !_ : Exp -> Exp [prec 47] .
  op _&&_ : Exp Exp -> Exp [prec 49] .
  op _||_ : Exp Exp -> Exp [prec 51] .
endfm
```

For example, `'X == #i(7)` is using a comparison and a boolean connective is used in `'X == #i(7) && B`. Both are of sort `Exp`.

The Syntax of Java: Array and New

Arrays are composed in this way and the syntax of new is straightforward.

```
fmod ARRAY-SYNTAX is ex TYPE .
  ex GENERIC-EXP-SYNTAX .
  op _[] : Type -> Type [prec 20] .
  op _[_] : Type Exp -> Type [prec 20] .
  op _[_] : Exp Exp -> Var [prec 20] .
endfm

fmod NEW-SYNTAX is ex ARRAY-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op new_ : Type -> Exp [prec 25 gather(E)] .
endfm
```

As an example, `int[]` and `int[#i(5)]` are of the type integer array, and `a[b]` is the variable at position `b` in the array `a`. A new integer array of length 5 is defined by `new int[#i(5)]`.

The Syntax of Java: Assignment

Next let us present assignments.

```
fmod ASSIGNM-EXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  op _=_ : Var Exp -> StExp [prec 60] .
endfm
```

An example `StExp` is `'X = 'Y + #i(7)`.

The Syntax of Java: Declaration

This leads us to declarations.

```
fmod DECLARATION-SYNTAX is pr TYPE .
  ex ASSIGNM-EXP-SYNTAX .
  sort Declaration Declarators .
  subsort Name < Declarators .
  subsort StExp < Declarators .
  op __ : Type Declarators -> Declaration [prec 70] .
endfm
```

A `Declarator` is for example `'X` or `'X = #i(0)` and so a `Declaration` would be for example `int 'X = #i(0)`.

The Syntax of Java: Blocks

Next let us present blocks.

```
fmod BLOCK-SYNTAX is
  sorts Block BlockStatements .
  subsort Block < BlockStatements .
  op ; : -> Block .
  op {_} : BlockStatements -> Block .
  op __ : BlockStatements BlockStatements -> BlockStatements [assoc prec 125] .
endfm
```

Any code (i.e. any `BlockStatements`) can be put into a `Block` by means of the {_} operator.

The Syntax of Java: Statements

Let us now look at statements:

```
fmod STATEMENT-SYNTAX is ex BLOCK-SYNTAX .
  pr QID .
  sort Statement .
  subsort Block < Statement < BlockStatements .
endfm

fmod EXP-STATEMENT-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op _; : StExp -> Statement [prec 110] .
endfm

fmod DECLARATION-STATEMENT-SYNTAX is ex DECLARATION-SYNTAX .
  ex STATEMENT-SYNTAX .
  op _; : Declaration -> Statement [prec 75] .
endfm
```

A `BlockStatements` example is: `'X = #i(0) ; int 'Y ;.`

The Syntax of Java: Conditional

Here is the definition of conditionals.

```
fmod IF-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op if__else_ : Exp Statement Statement -> Statement [prec 110] .
  op if__ : Exp Statement -> Statement [prec 115] .
endfm
```

An example for a Statement of this type is:

```
if ('X <= 'Y) 'X = #i(0) ;
              else {'X = 'Y - #i(1) ; Y = Y + #i(1) ;}
```

The Syntax of Java: Loops

Now we want to show the while and do while loops.

```
fmod WHILE-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op while__ : Exp Statement -> Statement [prec 110] .
endfm

fmod DO-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op do_while_; : Statement Exp -> Statement [prec 110] .
endfm
```

A loop example is:

```
while ('X < #i(10)) 'X = 'X + #i(1) ;
```

The Syntax of Java: Loops (II)

There are also for loops in our language.

```
fmod FOR-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  ex DECLARATION-SYNTAX .
  op for(_;_;_)_ : Exps Exp Exps Statement -> Statement [prec 110] .
  op for(_;_;_)_ : Declaration Exp Exps Statement -> Statement [prec 110] .
endfm
```

The Syntax of Java

This puts all parts of the syntax together:

```
fmod JAVA-SYNTAX is
  ex ARITH-EXP-SYNTAX .
  ex REXP-SYNTAX .
  ex BEXP-SYNTAX .
  ex ARRAY-SYNTAX .
  ex NEW-SYNTAX .
  ex ASSIGNM-EXP-SYNTAX .
  ex DECLARATION-SYNTAX .
  ex DECLARATION-STATEMENT-SYNTAX .
  ex EXP-STATEMENT-SYNTAX .
  ex IF-SYNTAX .
  ex WHILE-SYNTAX .
  ex DO-SYNTAX .
  ex FOR-SYNTAX .
  sort Pgm .
  subsort BlockStatements < Pgm .
endfm
```

The Syntax of Java: Parsing examples

We can now take a look at a few examples and let Maude parse them using the syntax we defined.

```
parse int 'x = #i(5) ; int 'y = #i(4) ; {'x = #i(42) ; } 'x = 'x + 'y ; .

parse int 'x = #i(5) ; int 'y = #i(4) ; {int 'x = #i(42) ; } 'x = 'x + 'y ;  .

parse int 'x = #i(0) ; if #b(false)
                          if #b(true) 'x = #i(1) ;
                        else 'x = #i(2) ;  .

parse int 'a = #i(3) ; int 'b = #i(2) ; int 'c = #i(1) ;
        int 'd = 'a - 'b + 'c ; .

parse int 'x = #i(7) ; int 'y = #i(5) ; int 't = 'x ; 'x = 'y ; 'y = 't ; .

parse int 'n = #i(5) ; int 'c = #i(0) ; int 'x = #i(1) ;
                  while ('c < 'n) { 'c = 'c + #i(1) ; 'x = 'x * 'c ; } .
```

The Infrastructure for Java: State Infrastructure

So far, we have only described the syntax of our language. To describe its semantics we must specify how the execution of such programs affects the state infrastructure containing the values for the program variables (among other information).

The state infrastructure is defined by the following modules, where we separately present the locations, environments, values, stores and continuations that make up the state.

The Infrastructure for Java: Location

A program variable will not be directly mapped to its value but to a location in the store. This leads to a two level mapping of variables to locations on the one hand and locations to values on the other. This just shows what a location is, an example location is `l(17)`.

```
fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
endfm
```

The Infrastructure for Java: Environment

This module defines what an environment is and also gives equations on how to change it.

```
fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .

  vars X Y : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  op _[_<-_] : Env NameList LocationList -> Env .
  op _[_<-_] : Env Name Location -> Env .
  eq Env[() <- noLoc] = Env .
  eq Env[X,Y,Xl <- L,L',Ll] = (Env [X <- L]) [Y,Xl <- L',Ll] .
```

The Infrastructure for Java: Environment (II)

```
  eq ([X,L] Env)[X <- L'] = ([X,L'] Env) .
  ceq ([Y, L] Env)[X <- L'] = [Y, L] (Env [X <- L'])
    if equalName(Y, X) = false .
  eq noEnv [X <- L'] = [X,L'] .
endfm
```

The environment `(['X, 1(1)] ['Y, 1(2)])['X,'Y,'Z <- 1(3),1(4),1(5)]` evaluates thus to `['X,1(3)] ['Y,1(4)] ['Z,1(5)]`.

The Infrastructure for Java: Value and Store

Values and stores are defined like this. No equations are given for the store though (unlike for the environment).

```
fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
endfm
```

The Infrastructure for Java: Environments and Stores

The environments and stores are defined in a pretty concrete way for this language. Using a more abstract environment/store concept would have its advantages from the point of view of program verification.

A more abstract concept of environment/stores does not work nicely with the side-effects and hiding that are possible in our language though, so we settled with the concrete variant. This is also done so we can extend this subset of Java to a more complete version of Java in the future.