Program Verification: Lecture 14

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign
We are now ready to consider a general methodology for verifying declarative programs. We will present the ideas in the context of verifying Maude functional modules, which are based on equational logic. The first key observation is that there are three viewpoints involved:

- the customer’s viewpoint, expressed in the form of requirements that the desired software should satisfy;
- the implementor’s viewpoint, whose job is to write a program meeting the customer’s requirements; and
- the verifier’s viewpoint, whose responsibility is to verify that the implementation does indeed meet the customer’s requirements.
The Customer’s Requirements and Specification

The customer’s requirements may generally be informal. Furthermore, they may involve other concerns beyond correctness, such as user-friendliness, a good graphical user interface, performance requirements, requirements about the underlying hardware and systems software, interoperability requirements, and so on.

Program verification focuses primarily on correctness requirements, which are always important, but may be crucial for safety-critical applications, were incorrect software may cause loss of human lives and/or other important damages.
To make possible the high assurance of correctness afforded by mathematical verification, such correctness requirements must be formalized, typically in the form of a logical theory $T_{spec}$, stating precisely the customer’s (correctness) specification.

This capture of the informal correctness requirements into a formal specification can be done by the customer himself, or by an expert aiding the customer in this task. It is of course very important to make sure that the formal specification captures faithfully the informal requirements.
In the context of Maude functional modules, it is reasonable to assume that such a formal specification will take the form of a theory,

\[ T_{spec} = (\Delta, E_0 \cup Q) \]

where:

- \((\Delta_0, E_0)\), with \(\Delta_0 \subseteq \Delta\), is an equational theory, that could be called the framework theory, specifying things such as key data structures and functions, including auxiliary functions needed to state key properties, and

- \(Q\) is a collection of sentences in first-order logic, specifying the actual correctness properties that the software must satisfy within the \((\Delta, E_0)\) framework.
Customer Specification: A Sorting Example

We can illustrate these ideas with a simple example, namely a customer who wants a sorting program to sort lists of integers.

As already mentioned, the customer’s requirements may involve other important considerations, such as reasonable efficiency; for example, that it returns answers in time at most quadratic on the size of the input list.

Informally, the correctness requirement seems both obvious and tautological, namely, the program should return the input list in sorted form.
Customer Specification: A Sorting Example (II)

However, in order to prove that a given implementation satisfies such a requirement, we need to capture such an informal requirement in a formalized way as a theory $T_{spec}$.

We must specify two things:

1. the data, namely lists of integers, and some auxiliary functions, and

2. the sorting function and its properties.
Specifying the properties of the sorting function is not entirely trivial:

- first of all, we need to make precise what we mean by a list being sorted;

- but it is not enough to just require that the result is sorted: a function returning always the empty list will satisfy such a requirement! The original list and the sorted list should have the same elements.

All this can be stated precisely in three equational theories:
First, the data, say lists of numbers, is specified in a module such as the following INT-LIST module

```plaintext
fmod INT-LIST is protecting INT .
  sorts List .
  op nil : -> List [ctor] .
endfm
```
Then, a framework theory importing INT-LIST,

fmod FRAME-SORTING-REQUIREMENTS is protecting INT-LIST .
    sort Multiset .
    subsort Int < Multiset .
    op sorted : List -> Bool .
    op null : -> Multiset .
    op mset : List -> Multiset .
    vars N M : Int .
    var L : List .
    eq sorted(nil) = true .
    eq sorted(N : nil) = true .
    ceq sorted(N : M : L) = sorted(M : L) if (N <= M) = true .
    ceq sorted(N : M : L) = false if N <= M = false .
    eq mset(nil) = null .
    eq mset(N : L) = N mset(L) .
endfm
Finally, a functional theory specifying two key requirements for the sort function:

```
fth SORTING-REQUIREMENTS is
  protecting FRAME-SORTING-REQUIREMENTS .
  op sort : List -> List .
  var L : List .
  eq sorted(sort(L)) = true .
  eq mset(sort(L)) = mset(L) .
endfth
```
This is an instance of our general methodology, where the correctness specification has the form, $T_{\text{spec}} = (\Delta, E_0 \cup Q)$. Here, the framework theory $(\Delta_0, E_0)$ is the module FRAME-SORTING-REQUIREMENTS, and the theory $T_{\text{spec}}$ itself is SORTING-REQUIREMENTS.

$T_{\text{spec}}$ is a Maude theory (see Section 8.3.1 of “All About Maude”) introduced with the keywords \texttt{fth} $T_{\text{spec}}$ \texttt{endth}. This means that it has a loose semantics; that is, we do not require its models to be initial. However, because of the keyword protecting FRAME-SORTING-REQUIREMENTS, the functional submodule FRAME-SORTING-REQUIREMENTS is imported with its initial semantics.
Mathematically, what this means is that, for $\mathcal{A}$ to be an acceptable model of $T_{spec}$, besides having to satisfy the axioms in $T_{spec}$, the data types of lists, multisets, integers, and of booleans, as well as all the functions defined on them by initial algebra semantics (with keywords fmodFRAME-SORTING-REQUIREMENTSendfm) must be respected. In short, we must have an isomorphism,

$$\mathcal{A}|_{\Sigma_{\text{FRAME-SORTING-REQUIREMENTS}}} \cong T_{\text{FRAME-SORTING-REQUIREMENTS}}.$$
One possible Maude implementation is `insert-sort`,

```maude
fmod INSERT-SORT is
  protecting INT-LIST .
  op ins : Int List -> List .
  op sort : List -> List .
  vars N M : Int .
  var L : List .
  eq ins(N, nil) = N : nil .
  eq sort(nil) = nil .
  eq sort(N : L) = ins(N, sort(L)) .
endfm
```
The Implementation: Insert-Sort (II)

The module $T_{imp}$ in our general methodology, becomes in this case the Maude module INSERT-SORT.

Note that the auxiliary function $\text{ins}$ defined in INSERT-SORT for implementation purposes is completely different from the auxiliary functions, $\text{sorted}$, $\_\_\_$, and $\text{msset}$ defined in FRAME-SORTING-REQUIREMENTS for specification purposes, to formally capture the customer’s correctness requirements.

Therefore, the signatures of $T_{spec}$ and $T_{imp}$ do not necessarily coincide. However, for verification purposes, they are both included as subsignatures in $T_{ver}$. 
For verification purposes we typically need to use the auxiliary functions defined in both the framework theory \((\Delta_0, E_0)\) and in \(T_{imp} = (\Sigma, E)\). This means that the theory \(T_{ver} = (\Sigma', E')\) will typically have theory inclusions,

- \((\flat)\) \((\Delta_0, E_0) \subseteq (\Sigma', E')\), and
- \((\dagger)\) \((\Sigma, E) \subseteq (\Sigma', E')\).

The main goal of the verification effort is then to establish,

\[ T_{ver} \vdash_{\text{ind}} Q. \]

But for this inductive inference to be applicable to the implementation theory \(T_{imp} = (\Sigma, E)\), we need to require that \((\dagger)\) is a protecting inclusion, so that we have an isomorphism, \(T_{\Sigma'/E'}|_{\Sigma} \cong T_{\Sigma/E}\).
The Theory $T_{ver}$ and its Verification

In this example, the theory $T_{ver}$ must contain both the framework theory and the implementation theory:

\[
\text{fmod INSERT-SORT-VERIFICATION is} \\
\text{protecting INSERT-SORT} . \\
\text{protecting FRAME-SORTING-REQUIREMENTS} . \\
\text{endfm}
\]

We can then give this theory to the ITP and prove in it as theorems the two equations in $T_{spec}$, namely,

\[
\text{eq sorted(sort(L)) = true} . \\
\text{eq mset(sort(L)) = mset(L)} .
\]
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 $Q$. 
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.
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 $\mathcal{T}_{T_{\mathcal{L}}}$;
- if the equations in $T_{\mathcal{L}}$ are ground confluent and sort-decreasing, this also gives an operational semantics to the language, expressed in terms of equational simplification.
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,

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

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

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

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

```plaintext
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:

```plaintext
fmod VAR-SYNTAX is ex NAME .
  sorts Var Vars .
  subsort Name < Var < Vars .
  subsort NameList < Vars .
  op _,_ : Vars Vars -> Vars [ditto] .
endfm
```
This is our definition of generic expressions. Note the lists of expressions.

```plaintext
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 .
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 ensures.

```
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)] .
endfm
```
Program variables, of sort Var, are terms of the form ’Q with Q a character string such as X of abc. 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))
This defines the syntax of comparisons and boolean connectives.

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

fmod BEXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
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 arrays and new is straightforward.

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

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

As examples, int[] and int[#i(5)] are integer array types. Instead, new int[#i(5)] is an expression for a new integer array of type int[#i(5)].
Next let us present assignments.

```plaintext
next let us present assignments.

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

As examples StExp we can consider 'X = 'Y + #i(7), and 'a = new int[#i(5)].
```
This leads us to **declarations**.

\[
\text{This leads us to **declarations**.}
\]

\[
fmod DECLARATION-SYNTAX is pr TYPE .
\]

\[
ex \text{ ASSIGNM-EXP-SYNTAX .}
\]

\[
sort Declaration Declarators .
\]

\[
subsort Name < Declarators .
\]

\[
subsort StExp < Declarators .
\]

\[
op \_ : Type Declarators \to Declaration [prec 70] .
\]

\[
endfm
\]

\[
'X, 'X = \#i(0), and 'a = new int[\#i(5)] are Declarators.
\]

\[
Two examples of Declaration would be int 'X = \#i(0), and int[\#i(5)] 'a = new int[\#i(5)].
\]
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.
Let us now look at statements:

```plaintext
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 ;
```
Here is the definition of conditionals.

```plaintext
fmod IF-SYNTAX is ex STATEMENT-SYNTAX .
ex GENERIC-EXP-SYNTAX .
endfm
```

An example for a Statement of this type is:

```plaintext
if ('X <= 'Y) 'X = #i(0) ;
else {'X = 'Y - #i(1) ; Y = Y + #i(1) ;}
```
Now we want to show the **while** and **do while** loops.

```plaintext
fmod WHILE-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
endfm

fmod DO-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
endfm

A loop example is:
while ('X < #i(10)) 'X = 'X + #i(1) ;
There are also for loops in our language.

```plaintext
fmod FOR-SYNTAX is ex STATEMENT-SYNTAX .
    ex GENERIC-EXP-SYNTAX .
    ex DECLARATION-SYNTAX .
endfm
```
This puts all parts of the syntax together:

```plaintext
fmod JAVA-SYNTAX is
  ex ARITH-EXP-SYNTAX .
  ex REXP-SYNTAX .
  ex BEXP-SYNTAX .
  ex ARRAY-SYNTAX .
  ex NEW-SYNTAX .
  ex ASSGNM-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
```
We can now take a look at a few examples and let Maude parse them using the syntax we defined.

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

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

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

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

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

```plaintext
parse int 'n = #i(5) ; int 'c = #i(0) ; int 'x = #i(1) ;
while ('c < 'n) { 'c = 'c + #i(1) ; 'x = 'x * 'c ; } .
```
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.
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 1(17).

```plaintext
fmod LOCATION is
    protecting INT .
    sorts Location LocationList .
    subsort Location < LocationList .
    op noLoc : -> LocationList .
    op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
    op 1 : Nat -> Location .
endfm
```
This module defines what an environment is and also gives equations on how to change it.

```markdown
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)

\[
eq ([X,L] \text{Env})[X <- L'] = ([X,L'] \text{Env}) .
\]

\[
\text{ceq} ([Y, L] \text{Env})[X <- L'] = [Y, L] (\text{Env} [X <- L']) \\
\text{if equalName}(Y, X) = \text{false} .
\]

\[
\text{eq noEnv} [X <- L'] = [X,L'] .
\]

endfm

The environment \([’X, 1(1)] [’Y, 1(2)] [’X,’Y,’Z <- 1(3),1(4),1(5)]\) evaluates thus to \([’X,1(3)] [’Y,1(4)] [’Z,1(5)]\).
Values and stores are defined like this. No equations are given for the store though (unlike for the environment).

```plaintext
fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
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 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.