ASIP+ITP: A Verification Tool Based on Algebraic Semantics

Manuel Clavel
Dept. Sistemas Informáticos y Programación
Univ. Complutense de Madrid
clavel@sip.ucm.es

Juan Santa-Cruz
Dept. Sistemas Informáticos y Programación
Univ. Complutense de Madrid
juansc@sip.ucm.es

Resumen

The ASIP+ITP tool is an experimental tool for verifying imperative programs based on the algebraic definition of the semantics of programs.

1. Introduction

Using the algebraic definition of the semantics of imperative programs as a formal foundation for software verification was first proposed by J. Goguen and G. Malcolm in [5]. In this approach the semantics of imperative programs is defined by specifying a class of abstract machines and giving equational axioms which specify the effect of programs on such machines

The semantics of imperative programs is specified in [5] in a formal, implemented notation, the language OBJ [3]. An OBJ 'program' is an equational theory, an every OBJ computation proves some theorem about such a theory. This means that the OBJ program `SEM` used in [5] for defining the semantics of programs already has a precise mathematical meaning. Moreover, standard techniques for mechanizing equational reasoning can be used for verifying the axioms in `SEM` that describe the effect of imperative programs on abstract machines, and these axioms can then be used in mechanical proofs of properties of programs.

OBJ was not originally designed as a theorem prover, but rather as a programming and specification language. Despise this, in [4] some techniques for proving theorems about OBJ programs with OBJ are stated, justified, and illustrated. However, as it is also warned in [4], "some things have to be done by hand that would be done automatically in a fully developed verification environment." For that reason, it is concluded in [4], "it would be useful to construct a verification interface for OBJ to generate proof scores that use only techniques already shown correct. For example, a score for an inductive proof could be automatically generated from knowing the constructors, what to prove, and what variable to do induction over. Such a system would guarantee the validity of any proof that it constructs, and could also support incremental proof management."

The techniques introduced [4] are used in [5] to prove the correctness of programs in OBJ. The general methodology is as follows: to check a property ϕ about a program P, we first formalized ϕ as a property about `SEM`, and then we prove in OBJ that ϕ holds in `SEM` using the techniques introduced [4]. However, the examples that illustrate in [5] this methodology also show its limitations: the amount of interaction required to complete a verification proof, and the expertise demanded to guarantee its validity, clearly hindered its applicability.

The ITP tool [1] is an experimental inductive theorem prover for proving properties of Maude [2] 'programs'. Like in OBJ, a Maude (functional) 'program' is an equational theory, an every Maude computation proves some theorem about such a theory. In the ITP tool the user introduces commands which are interpreted as actions that may change the state of the proof: that is, the set of goals that remain to be proved, with each goal consisting of a property to be proved and the Maude 'program' about which the property must be proved. The ITP is indeed a "verification interface" for Maude "to generate proof scores that use only techniques already shown correct," in such a way that it guarantees "the validity of any proof that it constructs", and also supports "incremental proof management." The ASIP+ITP tool is an extension of the ITP tool, that makes available the ITP "verification interface" to prove properties about a specific Maude 'program', named `ASIP`, which algebraically defines the semantics of programs in the spirit of the OBJ 'program' `SEM`.

Organization

2. The ITP Tool

The ITP [?] tool is an experimental interactive tool for proving properties of Maude 'programs', i.e., equational specifications with an initial algebra semantics. The ITP tool has been written entirely in Maude, and is in fact an *executable* specification of the formal inference system that it implements. The ITP inference system treats Maude 'programs' as *data* — for example, one inference may add to the 'program' an induction hypothesis as a new equational axiom. This makes a *reflective* design, in which Maude 'programs' become data at the metalevel, ideally suited for implementing the ITP. Using reflection to implement the ITP tool has one important additional advantage, namely, the ease to rapidly extend it by integrating other tools implemented in Maude using reflection, as it is the case of the ASIP+ITP tool.

In the ITP, the user introduces commands which are interpreted as actions that may change the state of the proof, specifically the set of goals that remain to be proved, with each goal consisting of a formula to be proved and the Maude 'program' in which the formula must be proved. After executing the action requested by the user, the tool reports the resulting state of the proof. The main module implementing the ITP is the `ITP-TOOL` module. In this module, states of proofs, sets of goals, goals and formulas are represented by terms of different sorts, and the actions interpreting the ITP commands are represented as different, equationally defined functions over those terms.

To give a taste of the ITP tool, consider the following Maude 'program' which specifies lists of integers:

```
fmod LIST is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> List [ctor] .
  op _:_ : Int List -> NeList [ctor] .
  op _++_ : List List -> List .

  var I : Int .
  vars L L' : List .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .
endfm
```

An obvious property that `LIST` should satisfy is that concatenation of lists (`_++_`) is associative. To prove it, once the `LIST` module has been added to Maude's database, we load the ITP with the Maude command `in itp-tool` and initialize its own database with `loop init-itp .`; then, the property can be presented to the ITP using the command `goal`:

```
(goal list-assoc : LIST |-
    A{L1:List ; L2:List ; L3:List}
     (((L1:List ++ L2:List) ++ L3:List) =
        (L1:List ++ (L2:List ++ L3:List))) .)
```

The ITP then outputs

```
=================================
label-sel: list-assoc@0
=================================
A{L1:List ; L2:List ; L3:List}
    ((L1:List ++ L2:List)++ L3:List =
       L1:List ++(L2:List ++ L3:List))
+++++++++++++++++++++++++++++++++
```

indicating that the goal has been correctly processed, has been internally labelled as `list-assoc$0` and is ready to be worked upon.

Now we can try to prove the property by structural induction on the first variable, using the `ind` command.

```
(ind on L1:List .)
```

The ITP then generates a corresponding subgoal for each operator with codomain `List` that has been declared with the attribute `ctor` (taking subsorts into account), and selects one of them as the working subgoal; in this case the first one, corresponding to the empty list `[]`:

```
================================
label-sel: list-assoc@1.0
================================
A{L2:List ; L3:List}
    (([]++ L2:List)++ L3:List =
      []++(L2:List ++ L3:List))

================================
label: list-assoc@2.0
================================
A{V0#0:Int ; V0#1:List}
((A{L2:List ; L3:List}
   ((V0#1:List ++ L2:List)++ L3:List =
     V0#1:List ++(L2:List ++ L3:List)))==>
 (A{L2:List ; L3:List}
   (((V0#0:Int : V0#1:List)++ L2:List)++
                                  L3:List =
     (V0#0:Int : V0#1:List)++
                  (L2:List ++ L3:List))))
++++++++++++++++++++++++++++++++
```

At this point, we can try to automatically prove the selected subgoal with the command `auto`, that first transforms all variables into fresh constants and then rewrites the terms in both sides of the equality as much as possible by using the equations in the module as rewrite rules.

```
(auto .)
```

The command succeeds, the subgoal is discharged, and the ITP presents us with the remaining subgoal generated by the induction; note how `label` has been replaced by `label-sel`.

```
================================
label-sel: list-assoc@2.0
================================
A{V0#0:Int ; V0#1:List}((A{L2:List ; L3:List}
   ((V0#1:List ++ L2:List)++ L3:List =
     V0#1:List ++(L2:List ++ L3:List)))==>
 (A{L2:List ; L3:List}
   (((V0#0:Int : V0#1:List)++ L2:List)++
                                  L3:List =
     (V0#0:Int : V0#1:List)++
                  (L2:List ++ L3:List))))
++++++++++++++++++++++++++++++++
```

We can also try to prove this subgoal automatically and, again, the ITP succeeds and this completes the proof.

```
(auto .)

q.e.d

++++++++++++++++++++++++++++++++
```

3. The ASIP+ITP Tool

The 'program' `ASIP` introduces

- a sort `Store` to represent stores s, that is, (abstract) entities which associate values with program variables;
- a sort `Pgm` to represent programs P; and
- an operation `_;_`(s, P) (also written as s`;`P) that returns the store that results of executing a program P on a store s.

The ASIP tool is then based on the following

Theorem 1 *For all Hoare triples $(\!|\phi|\!)\ P\ (\!|\psi|\!)$,*

$$\vdash_{\text{hoa}} (\!|\phi|\!)\ P\ (\!|\psi|\!) \iff \models_{\text{asip}} \forall X \forall\{s\}(\overline{\phi}(s) \rightarrow \overline{\psi}(s/s;\overline{P}))\ ,$$

where X is the set of *specification constants*, s is a variable of type `Store`, $\overline{P}$ is a term of sort `Pgm` that represents the program P in `ASIP`, $\overline{\phi}(s)$ and $\overline{\psi}(s)$ are first-order formulas over the signature of `ASIP`, with free variable s, that formalize the Hoare pre- and postconditions ϕ and ψ, and, finally, $\overline{\psi}(s/s;\overline{P})$ is the substitution of the variable s by the term $s;\overline{P}$ in the formula $\overline{\psi}$.

3.1. Stores, Variables, Values, and Assignment

The single feature that characterizes imperative programming languages, namely, the assignment of values to variables, is formulated in the algebraic semantics approach using the concept of a *store*: an abstract entity which associates integer values with the variables of the programming language. Following sections present the syntax and semantics of various other features found in programming languages, but the semantics of each of these features is based on the semantics of assignment introduced here.

We declare first the syntax of assignment, as an operator `_:=_` that takes two arguments, a program variable and an expression, and it returns a program. We also declare a boolean function `equal` that checks whether two program variables are equal; for the sake of space limitations, we omit here the specification of this function.

```
--- program variables
sort PVar .
op v : Char -> PVar .
op equal : PVar PVar -> Bool .
--- expressions
sort Exp .
subsort PVar < Exp .
subsort Int < Exp .
op _+'_ : Exp Exp -> Exp [prec 10] .
op _*'_ : Exp Exp -> Exp [prec 8] .
op _-'_ : Exp Exp -> Exp [prec 10] .
op -'_ : Exp -> Exp [prec 5] .
--- assignments
sort BPgm .
op _:=_ : PVar Exp -> Pgm [prec 20] .
```

Next, we define a function `valExp` that specifies how the value of an expression depends on the values of the program variables that occur in that expression, and how assignments modify the values that stores associate with variables. The store `initial` represents the initial state of our abstract computing machine; we arbitrarly specify that in this initial state the value associated with each program variable is 0.

```
--- store
sort Store .
op initial : -> Store .
op _;_ : Store Pgm -> Store [gather(E e)].
--- valExp
op valExp : Store PVar -> Int .
eq valExp(S, I) = I .
eq valExp(S, (E1 +' E2))
   = valExp(S, E1) + valExp(S, E2) .
eq valExp(S, (E1 *' E2))
   = valExp(S, E1) * valExp(S, E2) .
eq valExp(S, (E1 -' E2))
   = valExp(S, E1) - valExp(S, E2) .
eq valExp(S, (-' E1))
   = - valExp(S, E1) .
eq valExp(initial, X) = 0 .
ceq valExp((S ; X := E), Y)
    = valExp(S, E)
    if equal(X, Y) = true .
ceq valExp((S ; X := E), Y)
    = valExp(S, Y)
    if equal(X, Y) = false .
```

3.2. Conditional

Consider, for example the following program *swap*, written in a generic Pascal-like programming language, that swaps the values of two program variables x and y, assigning the value of x to y and the value of y to x.

```
VAR X, Y, T : Int ;
BEGIN
   T := X ; X := Y ; Y := T
END
```

Of course, for this program to be correct, it must satisfies the following specification in Hoare logic: $(\!|X = N \wedge Y = M|\!)$ *swap* $(\!|X = M \wedge Y = N|\!)$, where N and M are specification constants. By Theorem **??**, this is equivalent to proving the following goal:

```
(asip SWAPXY-ASIP :
--- specification constants
(N:Int ; M:Int)
--- precondition
((valExp(S:Store, v("X"))) = (N:Int)
 & (valExp(S:Store, v("Y"))) = (M:Int))
--- program
swapxy
--- postcondition
((valExp(S:Store, v("X"))) = (M:Int)
  & (valExp(S:Store, v("Y"))) = (N:Int))
.)
```

where `SWAPXY-ASIP` is a Maude ‘program’ that extends `ASIP` with a constant `swapxy` that is equal to the representation of the program *swap*.

```
fmod SWAPXY-ASIP is
including ASIP .
op swapxy : -> Pgm .
eq swapxy  =
   v("T") := v("X") ;
   v("X") := v("Y") ;
   v("Y") := v("T") .
endfm
```

The following ASIP+ITP script automatically proves the goal.

```
(auto .)
(cnj .)
(auto .)
(auto .)
```

3.3. Conditional

4. Conclusion and Future Work

Referencias

[1] M. Clavel. The ITP tool’s home page. http://maude.sip.ucm.es/itp, 2005.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[3] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 1988.

[4] J. A. Goguen. OBJ as a theorem prover with applications to hardware verification. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer-Verlag, 1989.

[5] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* The MIT Press, 1996.

A. The representation functions

A.1. The representation of programs

$$\begin{array}{rcl}
\overline{x\ :=\ E} & \triangleq & \overline{x}\,\texttt{:=}\,\overline{E} \\
\overline{C_1\ ;\ C_2} & \triangleq & \overline{C_1}\ \texttt{;}\ \overline{C_2} \\
\overline{\texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}} & \triangleq & \texttt{if}\ \overline{B}\ \texttt{\{}\overline{C_1}\texttt{\}}\ \texttt{else}\ \texttt{\{}\overline{C_2}\texttt{\}} \\
\overline{\texttt{while}\ B\ \{C\}} & \triangleq & \texttt{while}\ \overline{B}\ \texttt{do}\ \overline{C}\ \texttt{enddo}
\end{array}$$

A.2. The representation of program expressions

$$\begin{array}{rcl}
\bar{i} & \triangleq & i\text{, for } i \text{ an integer number} \\
\overline{x} & \triangleq & \texttt{v("x")}\text{, for } x \text{ a program variable} \\
\overline{E_1 + E_2} & \triangleq & \overline{E_1}\ \texttt{+'}\ \overline{E_2} \\
\overline{E_1 \times E_2} & \triangleq & \overline{E_1}\ \texttt{*'}\ \overline{E_2} \\
\overline{E_1 - E_2} & \triangleq & \overline{E_1}\ \texttt{-'}\ \overline{E_2} \\
\overline{-E_1} & \triangleq & \texttt{-'}\ \overline{E_1}
\end{array}$$

A.3. The representation of program boolean tests

$$\begin{array}{rcl}
\overline{E_1 \leq E_2} & \triangleq & \overline{E_1}\ \texttt{<='}\ \overline{E_2} \\
\overline{E_1 \geq E_2} & \triangleq & \overline{E_1}\ \texttt{>='}\ \overline{E_2} \\
\overline{E_1 = E_2} & \triangleq & \overline{E_1}\ \texttt{='}\ \overline{E_2} \\
\overline{E_1 < E_2} & \triangleq & \overline{E_1}\ \texttt{<'}\ \overline{E_2} \\
\overline{E_1 > E_2} & \triangleq & \overline{E_1}\ \texttt{>'}\ \overline{E_2} \\
\overline{\neg(B)} & \triangleq & \texttt{not'(}\overline{B}\texttt{)} \\
\overline{B_1 \wedge B_2} & \triangleq & \overline{B_1}\ \texttt{and'}\ \overline{B_2} \\
\overline{B_1 \vee B_2} & \triangleq & \overline{B_1}\ \texttt{or'}\ \overline{B_2}
\end{array}$$

A.4. The representation of verification conditions

$\overline{\phi \wedge \psi}(s)$	$\triangleq$	$\overline{\phi}(s)$ `&` $\overline{\psi}(s)$
$\overline{\neg \phi}(s)$	$\triangleq$	`~` $\overline{\phi}(s)$
$\overline{\bot}(s)$	$\triangleq$	`falseFormula`
$\overline{\top}(s)$	$\triangleq$	`trueFormula`
$\overline{E_1 = E_2}(s)$	$\triangleq$	$\overline{E_1}(s)$ `=` $\overline{E_2}(s)$
$\overline{E_1 \leq E_2}(s)$	$\triangleq$	$\overline{E_1}(s)$ `<=` $\overline{E_2}(s)$
$\overline{E_1 \geq E_2}(s)$	$\triangleq$	$\overline{E_1}(s)$ `>=` $\overline{E_2}(s)$
$\overline{E_1 < E_2}(s)$	$\triangleq$	$\overline{E_1}(s)$ `<` $\overline{E_2}(s)$
$\overline{E_1 > E_2}(s)$	$\triangleq$	$\overline{E_1}(s)$ `>` $\overline{E_2}(s)$
$\overline{E_1 + E_2}(s)$	$\triangleq$	$\overline{E_1}(s) + \overline{E_2}(s)$
$\overline{E_1 \times E_2}(s)$	$\triangleq$	$\overline{E_1}(s) * \overline{E_2}(s)$
$\overline{E_1 - E_2}(s)$	$\triangleq$	$\overline{E_1}(s) - \overline{E_2}(s)$
$\overline{-E_1}(s)$	$\triangleq$	$-\overline{E_1}(s)$
$\overline{i}(s)$	$\triangleq$	i, for i an integer number or a specification constant
$\overline{x}(s)$	$\triangleq$	`valExp(`s`,`x`)`, for x a program variable