The Meta Package

Patrice Quinton

Version of December 29, 2007

Abstract

This document presents the `Meta` package of MMALPHA. This package allows one to define quickly translators for Mathematica expressions, in particular, MMALPHA Abstract Syntax Trees. Translators are specified by a set of rules which are translated as a Mathematica package. When loaded, together with a separate semantic file, one gets a translator. The `meta` package is used in MMALPHA to produce VHDL code.

1 Introduction

The `meta` package allows one to define a set of meta rules for the translation of a Mathematica expression, and to translate these rules as a Mathematica package. The functions of this package can then be applied to an expression, in order to translate this expression into something useful. In the following, we explain how this meta translator operates, and we illustrate this by means of a few examples. This package is used in various parts of MMALPHA.

Briefly speaking, creating a translator consists in:

1. writing a set of meta-rules in a `.meta` file,
2. writing a semantic file in a `.sem` file (not mandatory, but often convenient),
3. evaluating the `.meta` file with the `meta` function. This creates a `.m` Mathematica package which contains the translator,
4. load the translator which then becomes available.

The organization of this document is as follows. Section 2 presents the syntax of meta-rules. In Section 3, the use of the `meta` package is described. Options of the `meta` function are explained in Section 4. Section 5 explains how meta-rules are translated. An example of translator is shown in Section 6. The meta code and the semantic file of this translator are shown in Appendix A and Appendix B respectively.

Directory `$MMALPHA/doc/Packages/Meta` contains a notebook with examples. In particular, there is a nice translator of (a part of) Alpha to Mathematica.

You may access this notebook from the master notebook of MMALPHA[1] or by evaluating the expression

[1] Recall that the master notebook may be loaded by means of the command `on[]` from the Mathematica front-end.

```
docLink["Meta"]
```

in any notebook. This evaluation produces a button which opens the notebook.

2 How to define meta rules

The input file contains a list of meta-rules. Let `myTranslator.meta` be this file. A meta rule has the following syntax (in BNF form):

```
<metarule> ::= <rule name> "::=" <abstract node> ":>" <semantics>
                       | <rule name> "::=" <switch_rule> ":>" <semantics>
<rule name> := <MMAsymbol>
<abstract node> ::= <MMAsymbol> "[" <list of arguments> "]"
<switch_rule> ::= <MMAsymbol> ":>" <list of switch arguments>
<semantics> ::= a call to a semantic function (*)
<argument> ::= <symbol> ":" <pattern>
                        | <MMAsymbol> ":" <rule name>
                        | <MMAsymbol> ":" "{" <rule name> "}"
<switch argument> ::= "_" <MMAsymbol> "->" <rule name>
| <MMAsymbol> "->" "{" <rule name> "}"
<pattern> ::= any Mathematica pattern
```

A rule therefore describe a so-called *abstract node*, or a *switch rule*.

2.1 Abstract nodes

Abstract nodes represent expressions which correspond to an internal node of the abstract syntax tree. For example, the first level of an Alpha program has the form `system[n,p,id,od,ld,e]`, where `system` is the head, `n` is a string representing the name of the system, `p` is the parameter, `id` is the list of input variable declarations, `od` is the list of output variable declaration, `ld` the local declarations, and `e` the list of equations. Such a node is described by the following meta-rule:

```
SYSTEM ::=
  system[ n:_String, p:DOMAIN, id:{DECLARATION}, od:{DECLARATION},
  ld:{DECLARATION}, e:{EQUATION} ] :> semantics...
```

The left-hand-side of the rule, `system`, is the name of the rule. Following the `::=`, is the pattern which describes the form of the abstract node, followed, after the symbol `:>` by the semantics associated to this node.

The first argument of this pattern, `n:_String`, has the form `<symbol>:<pattern>`. This pattern will be used to check that the corresponding terminal element in the AST is a string. If not, an error message will be issued. The `n` symbol names the argument which matches this position, and can be used in the semantics of the rule as explained later.

The second argument, `p:DOMAIN`, has the form `<symbol>:<rule name>`. It generates recursively a call to the meta-rule `DOMAIN.` The subexpression which matches this second position is named p. The next argument, id:DECLARATION, is treated in a similar way, except that one expect a list of expressions matching the DECLARATION meta-rule.

2.2 Switch rules

Switch rules allow alternate abstract rules to be called, depending on a pattern. For example, the following meta-rule is used to select the various forms of Alpha expressions:

```
EXPRESSION ::=
  {_binop -> BINOP,
   _var -> VAR,
   _affine -> AFFINE,
   _restrict -> RESTRICT
  }
```

The arguments of this rule are particular cases of Mathematica patterns, namely, <typed_blanks>. What we call here a typed_blank is a Mathematica expression of the form _type, which matches any Mathematica expression whose head is type.

This rule can be read as follows: if the head of the current expression is binop, then apply meta-rule BINOP, if it is var, then apply meta-rule VAR, etc.

2.3 Arguments

2.3.1 Arguments of abstract nodes

The form of arguments depends on the type of subtree which is expected in the AST. If the subtree simply consists of a terminal element, a simple pattern is used, for example, name:_String.

For non terminal subtrees, arguments of the form <symbol>:<symbol> or <symbol>:<symbol> are used. The latter form allows a list of expressions to be described. Finally, an argument of the form <symbol>:List generates a call to a particular function which expects a (Mathematica) list of objects, and returns this list unchanged.

2.3.2 Switch arguments

A switch argument has the form _Symbol -> <rule name> or _Symbol -> <rule name> . The <rule name> is the name of a rule which is selected by the head of the node, as specified by the pattern. As an example: _var -> varexpression describes an alternative where the varexpression rule is called, when the head of the node is var.

2.4 Semantic expressions

The semantic part of a meta-rule is a Mathematica expression, involving symbols which name either arguments in the right-hand side part of the meta-rule, or results of the translation of these arguments by the semantic functions associated to the meta-rules.

Let us explain this on an example. Consider the meta-rule:

```
EQUATION::= equation[c:_String, e:EXPRESSION] :>
    semanticFunction[ equation, {c, tre} ]
```

The above rule describes the abstract node `EQUATION`. We expect this expression to contain a string, then an `EXPRESSION`, the latter being described by the `EXPRESSION` meta-rule. The string, when found, will be named `c`. The equation is named `e`. By convention, the result of the call to function `translateEXPRESSION` applied to `e` is named `tre` and can be used in the semantic part. The semantic expression, `semanticFunction[ equation, c, tre ]`, is a call to a function defined in a separate file, called the semantic file. There are absolutely no constraints in the way the semantic expression is defined. In this example, I choosed to name the functions I use `semanticFunction`, whose first parameter is the head of the expression, and the second one is the list formed by `c` and by the translation of the `EXPRESSION`.

Warning

Although one can put any Mathematica expression as the semantic part of a meta-rule, calling a semantic function is a good way to avoid trouble. Indeed, when the meta translator reads the meta-rules, it uses the Mathematica interpreter which evaluates as many expressions as possible. Therefore, the result of the semantic expression might be very surprising. For example, lets assume that you write the semantic part `c === "blabla"` expecting the semantic of your rule to be `True` if `c` is equal to the string `"blabla"`, and `False` otherwise. When the Mathematica parser reads this expression, it evaluates it, and the result of this evaluation is `False`, as the symbol `c` is different from the string `"blabla"`. The problem comes from the fact that we expected this evaluation to happen later... There are probably ways to avoid this kind of problem using complex Mathematica tricks (avoiding Mathematica to evaluate an expression is sometimes hard...)

3 Using Meta

Assume you want to write a translator called `myTranslator`.

1. Write the meta-rules in a file named `myTranslator.meta`.
2. Write semantic functions (if needed) in a separate package named `myTranslator.sem` . This package should follow some rules which are described in appendix B.
3. Load the `Meta.m` package (if it is not already loaded).

4. Evaluate the expression
   `meta[ "myTranslator", debug->True ]`
5. Load the file `myTranslator.m` which was created by `meta`.
6. To translate an expression `expr` of type `EXPRESSION`, evaluate `myTranslatorTranslateEXPRESSION[expr]`. Another better way is to introduce in the semantic package the function which you want to call to activate your translator on a tree. See for example the semantic file given as an example in section B: the semantic file contains the definition of a `checkCell` function.

4 Options of `meta`

- `verbose`: if `True`, gives some information. Default value is `False`.
- `debug`: if `True`, gives a lot of information. Default value is `False`.
- `check`: if `True`, calls to functions are generated inside a `Check` statement. This may help debugging the translator. Default value is `False`.
- `directory`: gives the directory where the semantic file is to be found and read. If `Null`, this directory is `$MMALPHA/lib/Packages/Alpha`. Default value is `Null`.

5 How meta-rules are translated

A meta rule named `RULE` is translated as a function `myTranslatorTranslateRULE`. This function is put in a package called `Alpha‘myTranslator‘`. In such a way, it is not possible to create conflicting names, provided the name of the translator is different from the names of the MMAlpha packages.

The package is equipped with a symbol called `myTranslatorDebug`, whose initial value is `False`. This symbol can be used as a debug flag by the semantic functions, in order to debug the translator.

The package `myTranslator.m` loads automatically the semantic file `myTranslator.sem` when it is itself loaded.

6 An example

Appendix A shows the meta-rules for a program, named `checkCell`, which analyzes an Alpha program which can be interpreted as a `AlphHard` ”cell”, and reports errors if not. A so-called cell is basically an Alpha program which describes a Register-Trans fer-Level description of a piece of hardware (see [**?**] for details).

The meta-rules read an AST and check that this AST represents a cell. For example, such a program cannot contain use statements, etc. etc.

Appendix **??** presents the semantic file CheckCell.sem which is associated with the previous file.

A Meta-rules for the checkCell translator

```
{
(*
This file contains the syntax of AlpHard Cell's Abstract Syntax Tree. Once
processed with the meta function, it produces a file named CheckCell.m which
allows one to analyze an AST and check that this AST has the syntax of a
Cell.
*)

(*
System declaration
*)
  SYSTEMDECLARATION ::=
    system[ systemName: _String,
            paramDecl: DOMAINPARAM,
            inputDeclList: {DECLARATION},
            outputDeclList: {DECLARATION},
            localDeclList: {DECLARATION},
            equationBlock: {EQUATION}
          ]
          :> semanticFunc[ system, trparamDecl,
               trinputDeclList, troutputDeclList, trlocalDeclList,
               trequationBlock, debug -> CheckCellDebug ]
,

  DECLARATION ::=
    decl[ varName: _String,
          varType: _Symbol,
          domain: DOMAIN
        ]
        :> semanticFunc[ decl, decl[ varName, varType, domain ], debug -> CheckCellDebug ]
(*
The semantics has to check that the dimension of this
declaration is 2. Returns True if so, False otherwise.
*)
,

  DOMAINPARAM ::=
    domain[ dimension: _Integer,
            indexList: {___String},
            polyedronList: {POLYHEDRON}
          ]
          :> semanticFunc[ domain, dimension , debug -> CheckCellDebug ]
(* The semantics of this construct is to set the global variable CheckCellParam *)
```

```
,
  DOMAIN ::=
    domain[ dimension: _Integer,
            indexList: {___String},
            polyedronList: {POLYHEDRON}
          ]
          :> dimension
(*
The semantics of this construct is its dimension.
*)
,

  POLYHEDRON ::=
    pol[ constraintsNum:_Integer,
         generatorsNum:_Integer,
         equationsNum:_Integer,
         linesNum:_Integer,
         constraintList: {CONSTRAINT},
         generatorList: {GENERATOR}
       ]
       :> Null
(*
Null result. We assume that the polyhedron is correct.
*)
,

  CONSTRAINT ::= {_List -> List} (* OK *)
,

  GENERATOR ::= {_List -> List} (* OK *)
,

(*
An equation is either an assignment or a use
*)
  EQUATION ::=
    { _equation -> ASSIGNMENT,
      _use -> USESTATEMENT  (* Not sure that we can have a use in a cell *)
    }
(*  OK *)
,

(*
An assignment
*)
```

```
  ASSIGNMENT ::=
    equation[ leftHandSide: _String,
                rightHandSide: ELEMENTS
            ]
            :> semanticFunc[ assignment, trrightHandSide,
                              equation[ leftHandSide, rightHandSide], debug -> CheckCellDe
(*
Semantics: returns the result of the rhs evaluation, which should be
a boolean, and the equation itself, in order to issue an error message.
*)
,

(*
Here, the syntax differs from a standard Alpha program. Only the following
types of assignments are allowed: multiplexers either in form of case statements
or if statements, affine expressions (X[t-k], where k is a constant or cons[]),
        variables, constants, binary and unary expression.
*)
  ELEMENTS ::=
    {
      _case-> CASEMUX,
      _if -> IFMUX,
      _affine -> AFFEXP,
      _binop -> BINEXP,
      _unop -> UNEXP,
      _var -> VAREXP,
      _const -> CONSTEXP
 }
,

(* This represents a rhs of the form V[t] or V[t-1] or 1[] (constant) *)
  AFFEXP ::=
    affine[
      affExpression: VAREXPORCONST,
      affineFunction: MATRIX
          ]
          :> semanticFunc[ affexp, traffExpression,
                            affineFunction,
                            debug -> CheckCellDebug ]
(*
The semantics checks that the matrix is a translation. It does not
check yet that the translation is negative.
*)
,
```

```
  MATRIX ::=
    matrix[
      d1:_Integer,
      d2:_Integer,
      indexes:{___String},
      mmaMatrix:MATRIXNUM
          ]
          :> Null
(* Semantics: Null
OK  *)
,

  MATRIXNUM ::= {_List -> {List}}
,

  BINEXP ::=
    binop[
          binaryOp: ( add | sub |  mul | div | or | and | xor |
                      min | max | eg | ne | le | lt | gt |
      minus | div | mod ),
    (* "mul" ou "mult"   et les operateurs : + | - | * | / | ...... ???  *)
          operand1: SUBEXPRESSION,
          operand2: SUBEXPRESSION
         ]
         :> semanticFunc[ binop, troperand1, troperand2, debug -> CheckCellDebug ]
,

  UNEXP ::=
    unop[
          unaryOp: ( neg | not | sqrt ),
          operand: SUBEXPRESSION
         ]
         :> troperand
,

  VAREXPORCONST ::=
{ _var -> VAREXPE,
  _const -> CONSTEXPE
} (* OK *)

,
  VAREXPE ::=
    var[
         identifier: _String
       ]
```

```
    :> var[ identifier ]
(* Semantics: var *)
,

  CONSTEXPE ::=
    const[
         constant: ( _Integer | _Real | True | False )
         ]
         :> const[ constant ]
(* Semantics: const *)

,

VAREXP ::=
    var[
         identifier: _String
       ]
    :> True
(* Semantics: var *)
,

  CONSTEXP ::=
    const[
         constant: ( _Integer | _Real | True | False )
         ]
         :> True
(* Semantics: const *)
,

(*
Expression. Used in the binary or unary expressions.
*)
  SUBEXPRESSION ::=
    {
      _affine -> AFFEXP,
      _binop -> BINEXP,
      _unop -> UNEXP,
      _var -> VAREXP,
      _const -> CONSTEXP
 }
,

(*  First type of mux *)
  CASEMUX ::=
    case[
```

```
          expressionList: {RESTEXP}
         ]
         :> semanticFunc[ case , trexpressionList, debug -> CheckCellDebug ]
(* Here, I assume a very restricted case of multiplexer: only two branches *)
,
  RESTEXP ::=
    restrict[
          domain: DOMAIN,
          ifExpression: IFEXP
          ]
          :> semanticFunc[ restrict, trdomain, trifExpression, debug -> CheckCellDebug ]
(* Semantics: the dimension of the domain, plus the ifExpression *)
,

  IFEXP ::=
    if[
       ifCondition: CONTROLEXPRESSION,
       alt1: SUBEXPRESSION,
       alt2: SUBEXPRESSION
      ]
      :> semanticFunc[ if, trifCondition, ifCondition, tralt1, alt1, tralt2, alt2, debug
(* The semantics is that alt1 should be a control signal *)
,

(*  Control expression *)
  CONTROLEXPRESSION ::=
    affine[
  expression: VAREXP,
  affineFunction: MATRIX
           ]
           :> semanticFunc[ controlexpression, expression,
                            affineFunction, debug -> CheckCellDebug ]
(*
    The semantics
*)
,
(* Second type of mux: an If and a case inside. Not checked  *)
  IFMUX ::=
    if[
       ifCondition: AFFEXP,
       alt1: SUBMUX2,
       alt2: SUBMUX2
      ]
      :> semanticFunc[ if , { ifCondition, alt1, alt2 },
                       debug -> CheckCellDebug]
```

```
,

  SUBMUX2 ::=
    {
      (* A case expression is not mandatory *)
      _case -> CASEXP,
      _affine -> AFFEXP,
      _binop -> BINEXP,
      _unop -> UNEXP,
      _var -> VAREXP,
      _const -> CONSTEXP
 }
,

(*
This is a case inside a If multiplexer. Only a restricted expression
can appear
*)
  CASEXP ::=
    case[
         expressionList: {RESTEXPMUX2}
        ]
        :> expressionList
,

  RESTEXPMUX2 ::=
    restrict[
         domain: DOMAIN,
         restExpression: SUBEXPRESSION
         ]
         :> { domain, restExpression }

(*
For the moment, there are no use in cells... I wonder if this is
correct
,

  USESTATEMENT ::=
    use[ id: _String,
         domainExtension: DOMAIN,
         paramAssign: MATRIX,
         inputList: {SOUSEXPRESSION},
 idList: {___String}
       ]
       :> semanticFunc[ use , {id, domExtension, paramAssign, expList, idList},
```

```
                        debug -> CheckCellDebug ]
*)

}
```

B Semantic file for the checkCell translator (part of)

```
BeginPackage["Alpha`CheckCell`",{"Alpha`",
"Alpha`Domlib`",
"Alpha`Tables`",
"Alpha`Matrix`",
"Alpha`Options`",
"Alpha`Static`"}];

(*
Semantics of the CheckCell parser
*)

CheckCellParam::usage =
"CheckCellParam is a global variable of Alpha`CheckCell which contains the
number of parameters of a system being checked";

CheckCellDebug::usage = "CheckCellDebug is the value of the debug option for
the CheckCell function";

CheckCellDebug = False;

checkCell::usage = "checkCell[] checks whether $result is an AlpHard Cell";

semanticFunc::usage = "";

Options[ semanticFunc ] = { debug -> False, verbose -> False };

Begin["`Private`"];

Clear[checkCell];
checkCell[opts:___Rule]:=
Module[{msg, error, dbg},
  CheckCellDebug = debug/.{opts}/.{debug->False};
  Catch[
    error = CheckCellTranslateSYSTEMDECLARATION[$result];
    If[ CheckCellDebug,
        If[ error, Print["There is a Mistake somewhere..."],
```

```
                    Print["This program seems to be an AlpHard Cell..."]
         ]
     ];
   !error
   ]
]
checkCell[___]:=Message[checkCell::params];

Clear[ semanticFunc ];

(*
System
*)
semanticFunc[ system , param:_, input:_, output:_, local:_, eq:_, opts:___Rule] :=
Module[{ dbg, errorparam, errorinput, erroroutput, errorlocal, erroreq },
  errorparam = param;
  dbg = debug/.{opts}/.Options[ semanticFunc ];
  errorinput = Not[ Apply[ And, input ]];
  erroroutput = Not[ Apply[ And, output ]];
  errorlocal = Not[ Apply[ And, local ]];
  erroreq = Not[ Apply[ And, eq ]];
  errorinput || erroroutput || errorlocal || erroreq || errorparam
];

(*
domain
*)
checkCell::paramerror =
"In a cell, the number of parameters should be greater than or equal to 1";
semanticFunc[ domain, dim:_Integer, opts:___Rule]:=
Module[{error},
  If[dim >= 1, error = False, If[dbg,Message[checkCell::paramerror]];error = True];
(* Store parameter value for future use *)
  CheckCellParam = dim;
  error
];

(*
assignment
*)
checkCell::assignment = "Error in equation\n ‘1‘\n(check dimensions of variable or consta
semanticFunc[ assignment , rhs:_, eq:_, opts:___Rule] :=
Module[{ dbg },
  dbg = debug/.{opts}/.{debug->False};
  (* rhs is supposed to be true of false, except if this is a simple
```

```
     var or const *)
  Which[
    Head[rhs] === var || Head[rhs] === const, True,
    !rhs&&dbg, Message[ checkCell::assignment, show[ eq, silent->True ] ]; False,
    !rhs, False,
    True, True
  ]
];

(*
binop
*)
semanticFunc[ binop, trop1:_, trop2:_, opts:___Rule] :=
Module[{ dbg  },
  dbg = debug/.{opts}/.{debug->False};
(* Return the and of both operands *)
  trop1 && trop2
];

(* Declaration *)
checkCell::decldim = "In :\n `1`\nthe dimension should be one plus the number of parameter
semanticFunc[ decl , d:_ , opts:___Rule] :=
Module[{ dbg  },
  dbg = debug/.{opts}/.{debug->False};
  If[ d[[3]][[1]] === CheckCellParam+1, True,
    If[dbg, Message[checkCell::decldim, show[d,silent->True]]]; False]
];

(*
use statement
*)
semanticFunc[ use , {id:_, extension:_, paramAssign:_, expList:_,
              idList:_}, opts:___Rule] :=
Module[{ dbg  },
  dbg = debug/.{opts}/.{debug->False};
  If[ dbg, Print["Semantic function called on use statement"]];
];

(*
case expression
*)
checkCell::case = "In a case expression, the number of branches should be 2.";
semanticFunc[ case , expressionList:_, opts:___Rule ] :=
Module[{ dbg  },
  dbg = debug/.{opts}/.{debug->False};
```

```
  If[Length[expressionList]=!=2&&dbg,
     Message[checkCell::case]
  ];
  (Length[expressionList]===2)&&Apply[And,expressionList]
];

(*
restriction expression
*)
semanticFunc[ restrict , trdomain:_, trifExpression:_, opts:___Rule ] :=
Module[{ dbg  },
  dbg = debug/.{opts}/.{debug->False};
  (* It is useless to check the dimension of the domain, it is done by analyze *)
  trifExpression
];

(*
if expression
*)
checkCell::ifcondition = "In expression\n`1`\nthe control signal is wrong";
checkCell::ifalt1 = "In expression\n`1`\nthe first alternative is wrong";
checkCell::ifalt2 = "In expression\n`1`\nthe second alternative is wrong";
semanticFunc[ if , trifCondition:_, condition:_, tralt1:_, alt1:_,
              tralt2:_, alt2:_, opts:___Rule] :=
Module[{ dbg },
  dbg = debug/.{opts}/.{debug->False};
  If[ !trifCondition&&dbg,
      (Message[checkCell::ifcondition,
              show[ if[ condition, alt1, alt2 ], silent->True]
              ]; False),
      False
  ];
  If[ !tralt1&&dbg,
      (Message[checkCell::ifalt1,
               show[ if[ condition, alt1, alt2 ], silent->True]
              ]; False),
      False
  ];
  If[ !tralt2&&dbg,
      (Message[checkCell::ifalt2, show[ if[ condition, alt1, alt2 ],
                            silent->True]]; False),
      False];
  trifCondition&&tralt1&&tralt2
];
```

```
(*
Control expression
*)
checkCell::controlexpression1 = "Control expression\n`1`\nis not boolean";
checkCell::controlexpression2 =
"Control expression\n`1`\nshould have an identity dependence";
semanticFunc[ controlexpression, expression:_, affineFunction:_,
              opts:___Rule] :=
Module[{ dbg, decl },
  dbg = debug/.{opts}/.{debug->False};
  (* The type of this expression must be boolean. We look for this
     variable *)
  decl = getDeclaration[expression[[1]]];
  If[ decl[[2]]=!=boolean&&dbg,
      Message[checkCell::controlexpression1,show[affine[expression,affineFunction],silent-
  If[ !identityQ[affineFunction]&&dbg,
      Message[checkCell::controlexpression2,show[affine[expression,affineFunction]silent-
  (decl[[2]]===boolean)&&(identityQ[affineFunction])
];

(*
affine expression
*)
checkCell::affexp1 = "In `1`, the matrix should be a translation matrix";
checkCell::affexp2 = "I am lost...";
checkCell::affexp3 = "In `1`, the translation should not be positive";
semanticFunc[ affexp, v:_, aff:_, opts:___Rule] :=
Module[{ dbg  },
  dbg = debug/.{opts}/.{debug->False};
(*  Two cases: var.[t-k], const[] *)
  Which[
    Head[v] === const, True,  (* We do not need to check the affine part, as this is done
   by analyze *)
    (* For a var, we need to check that the matrix is a translation,
       and also that the translation is positive *)
    Head[v] === var,
    Catch[
      Module[{tr},
        (* Is it a translation ? *)
        If[ translationQ[aff], True,
          If[dbg,
             Throw[
               (Message[checkCell::affexp1, show[aff,silent->True]];
               False)
             ],
```

```
              Throw[False]
            ]
          ];
          (* Get the translation vector *)
          tr = Last[aff[[4]][[1]]];
          If[ tr<=0, Throw[True],
            If[ dbg,
                Throw[Message[checkCell::affexp3, show[aff,silent->True]];
                False],
              Throw[False]
            ]
          ]
        ]
      ],
      True, Throw[If[dbg, Message[checkCell::affexp, aff];False, False]]
  ]
];

semanticFunc::params = "parameter error while calling a semantic function.";
(* Error case *)
semanticFunc[x:___] :=
Module[{ dbg },
  Throw[Message[ semanticFunc::params];Print["Parameters were: ", {x} ]];
];

End[];
EndPackage[];
```