Detecting Implicit Indeterminates in Symbolic Computation

Stephen M. Watt
David R. Cheriton School of Computer Science
University of Waterloo
smwatt@uwaterloo.ca

***Abstract*—In the design of symbolic mathematical computation systems, it is a popular choice to use the same syntax for both mathematical indeterminates and programming variables. While mathematical indeterminates are to be used without specific values, programming variables must be initialized before being used in expressions. A problem occurs when mistakenly uninitialized programming variables are silently taken to be mathematical indeterminates. This article explores how this problem can arise and its consequences. An algorithm to analyze programs for this defect is shown along with a Maple implementation.**

I. Introduction

The notion of a "variable" has existed both in mathematics and in programming for almost as long as these subjects have existed.. Although there are similarities between the concepts in the two fields, there are also important differences. In symbolic mathematical computation, both ideas are in play simultaneously, leading to confusion of novices and errors of experts. Once the problems of novices are overcome, the problems of experts remain. One of the most frequent problems in the development of symbolic computation software is that of late detection of uninitialized variables. In some symbolic computing systems, names that are not programming variables with values are assumed to be mathematical indeterminates. We call these "implicit indeterminates". Sometimes these are indeed intended to be mathematical indeterminates. Often, however, they are misspellings of intended variable names. This can lead to silent errors that are difficult to trace or that go undetected while incurring huge computational resources. We present a solution to this problem.

This paper is organized as follows: Section II describes the concepts of variables in mathematics and programming, and how they differ. Section III explains the consequences of implicit indeterminates and how the problem of unassigned variables in symbolic mathematical computation differs from the problems in other types of computing. Section IV explores potential remedies to the problem of implicit indeterminates and Section V presents an algorithm to detect them. Section VI presents a Maple implementation that detects implicit indeterminates. Finally, Section VII presents some conclusions.

II. Variables in Mathematics and Computing

Letters have been used to stand for quantities in mathematics for centuries. They have been used in geometry for at least 2300 years [1] and in algebra for about 400 years [2]. In modern mathematics, symbols may be used for parameters, indices, indeterminates, known or unknown constants, values varying over some domain or unknowns to be solved for. They may also be used to represent expressions more readably or more compactly (giving an expression as a tree can require exponentially more space than with named intermediate parts).

Letters have been used to stand for storage locations in computer programs since the the early days of programming languages. Uses prior to 1960 include the Whirlwind [3] and Flow-matic [4] programming languages. While the natural language style of Flow-matic evolved to give Cobol, Whirlwind inspired the algebraic notation used later in in Fortran and Algol. In modern computing, symbolic names are used to represent storage locations containing constants, locations whose contents may be updated or values that exist only at the time of program translation. Each programming language has its own rules about what types of values these names may represent, and in which parts of a program the names may be used. These symbolic names may represent numbers, structured objects, procedures, function parameters and types, among other things, and the time-frames in which they are bound to storage locations may range from just a few machine instructions to the entire lifetime of a program or interactive session.

To complicate matters, symbolic names in all these uses may be called "variables" in different situations. Although it may be awkward, this paper will use the terms "mathematical variable" and "programming variable" when necessary for clarity. A very simple example of how the mathematical and programming notions differ is given by the following statements:

```
x = x + 6
y = y / 2.
```

Interpreted as mathematical algebraic statements, these are two equations in variables x and y and which may have solutions, depending on the algebraic domain. For example, interpreted over the real numbers, there is no value of x that satisfies the first equation and the second equation requires $y = 0$. On the other hand, interpreted as statements in the programming language Fortran or C, these are assignments that update the values stored in locations named `x` and `y`. To avoid confusion, some programming languages use other symbols for programming assignments, e.g. "`:=`" in the Algol family of languages, or "`:`" in Macsyma [5].

```
> newton_sqrt := proc (x, niters)
    local i, rold, rnew;
    Digits := max(4, 2^(niters-2));
    rold := x;
    for i to niters do
        rnew := (rold+x/rold)/2.0;
        rold := rnew
    end do;
    rold
  end proc;

> length(newton_sqrt(100, 15));

                8199
```

(a) Using underscores in names.

```
> newtonSqrt := proc (x, nIters)
    local i, rOld, rNew;
    Digits := max(4, 2^(nIters-2));
    rOld := x;
    for i to nIters do
        rNew := (rold+x/rOld)/2.0;
        rOld := rNew
    end do;
    rOld
  end proc;

> length(newtonSqrt(100, 15));

                246162
```

(b) Converted names, with a mistake.

Fig. 1: Newton iteration to approximate square root.

```
> newtonSqrt(100, 5);
```

$$.50000000\,rold + \frac{50.000000}{.50000000\,rold + \frac{50.000000}{.50000000\,rold + \frac{50.000000}{.50000000\,rold + \frac{50.000000}{.50000000\,rold + .50000000}}}}$$

Fig. 2: A toy computation with an implicit indeterminate, *rold*. The cut-off of 5 iterations allows the output to be displayed.

While the syntactic confusion of programming assignments versus equations is easily dispensed with, the problem of unassigned programming variables being taken as implicit indeterminates is more difficult, as is explained next.

III. The Problem

Symbolic mathematical computation systems produce and operate on mathematical objects of various sorts, including expressions involving symbols for mathematical variables. Each of these systems typically provides an imperative programming language that is used to extend its built-in algebraic capabilities.

These systems typically have an interactive interface that allows statements such as the following,

```
p := x^2 + x + 1
```

If the symbol `x` has previously been established as a programming variable with a value, then the statement above uses that value to compute a value for the programming variable `p`. Otherwise `x` is taken to be the indeterminate x, and the programming variable `p` is assigned a polynomial in x as its value. This is a desirable behaviour for an interactive interface. A simple user model is that names that appear on the left hand side of an assignment are variables, and otherwise "a name is just a name" [7]. This is more satisfying than "it means just what I choose it to mean — neither more nor less" [8].

The system designer must decide whether the language used for the interactive interface will be the same as the programming language used for system extension or whether it will be different. If different, then it places a burden on users of knowing two languages. If the same, then the some aspects that are desirable for interactive use cause difficulties when writing libraries of programs. This treatment of unrecognized names as implicit indeterminates is such a feature.

We give an example showing the problem of unintentional use of an unassigned name. Consider the Maple session shown in Figure 1a. This session defines a function to compute an approximation to the square root of a number `x` using `niters` Newton iteration steps. It then computes the approximation to $\sqrt{100}$ after 15 iteration steps. The Maple length of the result is 8199, consisting of 2^{13} digits plus some overhead.

Now suppose the programmer wishes to add this function to a code base that uses capitalization to separate words in a name ("camel casing"), rather than underscores. The programmer modifies the program as follows and obtains the code shown in Figure 1b. The result is 30 times larger than expected! What happened? We don't want to print such a large expression, so let's examine the output after just a few iterations. This is shown in Figure 2.

We have discovered that one of the uses of `rold` was not converted to `rOld` and what has happened with `newtonSqrt(100, 15)` is that a large continued fraction was constructed with many large floating point coefficients. This example shows what can happen when unassigned names stand for themselves in a symbolic computation environment.

When writing programs, it happens frequently that editing errors or logical errors introduce mistakes in a program. If unassigned names stand for symbolic indeterminates in a program, then these errors go uncaught.

```
Algorithm check-proc-for-implicit-symbols.
Input: procedure p.
Output: set of implicit indeterminates
Method:
    used, def, sym :=
       check-proc(p, {}, globalDefNames,
                         globalSymNames)
    return used \ def \ sym


Procedure check-expr.
Input:  expr -- expression or statement
        used, def, sym
Output: used, def, sym updated
Method:
   if expr is a name then
      add name expr to the used set
   else if expr is a constant then
       /* Do nothing */
   else if expr is a procedure then
       used, def, sym :=
           check-proc(expr, used, def, sym)
   else if expr is a definition then
       used, def, sym :=
           check-def(expr, used, def, sym)
   else
       for each part p of expr do
           used, def, sym :=
               check-expr(p, used, def, sym)
   return used, def, sym
```

```
Procedure check-def.
Input:  expr -- definition expression
        used, def, sym
Output: used, def, sym updated
Method:
   for each part p of expr not being defined do
      used, def, sym :=
         check-expr(p, used, def, sym)
   for each name n defined in expr
      add the name n to the def set
   return used, def, sym

Procedure check-proc.
Input:  proc -- procedure to be checked
        used, def, sym
Output: used, def, sym updated
Method:
    loc := parameters and locals of proc
    def := union(def, parameters of proc)
    sym := union(sym, symbolics  of proc)

    for each statement s in body of proc do
        used, def, sym :=
            check-expr(s, used, def, sym)
    optionally-report-inner-proc(used, def, sym)

    used := used \ loc
    def  := def  \ loc
    sym  := sym  \ loc
    return used, def, sym
```

Fig. 3: Algorithm to detect names that are used but neither defined nor declared symbolic.
The entry point is `check-proc-for-implied-symbols`. The other procedures are helpers to recursively traverse the program. The variables `globalDefNames` and `globalSymNames` are initialized with the defined and symbolic names in the environment. The symbol "\" denotes set difference.

Unintended interpretation of names as mathematical indeterminates can have several consequences:

- Expressions containing unintended indeterminates can flow through the program execution, causing errors far from where they occur. The error can manifest itself, for example, when an expression with variables is used when a numeric value is expected. These errors are difficult to debug, because it is hard to determine where the unwanted behaviour has occurred.
- What is worse, is that expressions containing unintended indeterminates may not raise an execution error at all. They may simply lead to wrong results.
- In both cases, programs usually generate huge intermediate expressions unintentionally. This leads to significant performance penalties, even when the program produces correct results (as may occur through cancellation, late evaluation of when requiring only a sub-expression).

This kind of bug occurs frequently in developing Maple programs, so we ask what remedies might be brought to bear.

IV. Potential Remedies

There are several potential ways to address the problem of unassigned names, including misspellings, being taken as implicit indeterminates. These include:

- *Have a distinct syntax for mathematical variables.* For example, in Lisp systems one can use a quote to create a symbol data object, *e.g.* `(QUOTE X)` or `'X`. Programming variables intended to be used as mathematical indeterminates could then be initialized with symbol data objects as values. The use of an unassigned variable would then be an error in exactly the same way as in other programming languages.
  This solution can be suitable if creating a system *ab initio*. This is the approach taken in Axiom and the Aldor libraries. In Axiom, however, the top-level interactive language is different than the library programming language. This is not a solution that can be retrofit to existing systems such as Maple, however, as it is an incompatible change.
- *Require all variables to have a syntactic assignment.* This is subtly different than requiring them to have values. For example, to make a name unassigned in Maple one uses the idiom `a := 'a'`. This could be used to indicate that a name was intended to be a mathematical indeterminate. It would then be straightforward for the language processor to detect variables are used but have no assignment. This would be burdensome for interactive use, however.
- *Declare all names that are intended to be mathematical variables.* Then unassigned names that had been declared as mathematical variables would be allowed. Other unassigned names would be errors in the same way as in other

```
> read "Symbol.mpl";
> with(UndeclaredSymbolFinder);
               [CheckUnassigned]

> Eg0 := proc(p1, p2)
    local l1, l2;
    l1 := p1 + l2 + g
  end proc:
> CheckUnassigned(Eg0, 'Eg0');

Analyzed Eg0 of [p1, p2]
Used:                   g, l2, p1
Assigned or symbolic: l1, p1, p2
Used but neither assigned nor symbolic: g

> Eg1 := proc(p)
    local l;
    l := 100;
    proc(q) local m; m := p+l+m+g end proc
  end proc:
> CheckUnassigned(Eg1, 'Eg1');

Analyzed anonymous of [q]
Used:                   g, l, m, p
Assigned or symbolic: l, m, p, q

Analyzed Eg1 of [p]
Used:                   g, l, p
Assigned or symbolic: l, p
Used but neither assigned nor symbolic: g

> Eg2 := proc(p) local l;
    l := x; (q -> r -> p+q+r+l+g)(3)(2)
  end proc:
> CheckUnassigned(Eg2, 'Eg2');

Analyzed anonymous of [r]
Used:                   g, l, p, q, r, x
Assigned or symbolic: l, p, q, r

Analyzed anonymous of [q]
Used:                   g, l, p, q, x
Assigned or symbolic: l, p, q

Analyzed Eg2 of [p]
Used:                   g, l, p, x
Assigned or symbolic: l, p
Used but neither assigned nor symbolic: g, x

> Eg3 := proc(x, y, z)
    local a, b, f, v, c;
    option remember, symbolic(u, v);
    f := proc(u) u + v end proc;
    v := 21;
    v := sin(x) + cos(y) + tan(z) + u + v + w;
    for a in 1 .. 2 do v := v + a + b end do;
    for c to 2 do v := v + a + b end do;
    v
  end proc:
> CheckUnassigned(Eg3, 'Eg3');

Analyzed f of [u]
Used:                   u, v
Assigned or symbolic: u, v

Analyzed Eg3 of [x, y, z]
Used:                   a, b, cos, sin, tan, true,
                        u, v, w, x, y, z
Assigned or symbolic: a, c, u, v
Used but neither assigned nor symbolic: w
> quit
```

Fig. 4: Sample session showing the detection of implicit indeterminates. The output is edited for spacing.

programming languages.
This would be burdensome for systems such as Maple, where the interactive language and the library extension language are the same. In this case, the rule could be softened to the following,

- *Declare all names that are intended to be mathematical variables in procedures.* One could do this either with a special declarative form, or by requiring a special assignment, such as `a := 'a'` discussed above.

Of these, the last is the most promising. The other approaches have distinct drawbacks to user experience in existing systems.

V. Implicit Indeterminate Detection Algorithm

We assume a block-structured procedural language that allows nested procedures with lexical scoping. Adopting the standard terminology used in compiler construction [6], we call any statement that writes to a programming variable a *definition* and any statement that reads a value from a programming variable to be a *use*.

The algorithm presented in Figure 3 detects names that are used but neither assigned nor declared symbolic. It is applied to a single procedure, `p`, within a global environment. Whether global names have values may be determined either by

- analysis of the global statement sequence, possibly embedding it as a procedure body and applying this algorithm, or
- consulting the current environment for values associated to the names.

The first of these is robust and will give sensible results when multiple mutually recursive global procedures are defined. The second of these is more suitable for incremental use in an interactive top level (and is the approach we have taken).

For simplicity, we assume that the local variables and parameters have been renamed to make them all unique. The algorithm proceeds by structural induction. At each stage, three sets are available,

- `used`, names in scope and used so far
- `def`, names in scope and defined so far
- `sym`, names in scope and declared symbolic so far

Algorithm `check-proc-for-implicit-symbols` specializes a general recursive method, applying it to a single top-level procedure body. The general recursive method `check-expr` is by syntactic cases. The step `check-def` separates the defined names from the expressions used in computing the definition. For example, the defined names include `a` and `b` in `a, b := minmax(l)` or `i` in `for i in m..M`. The names used to determine definition are added to `used`. For example, this would include `i` in `a[i] := 7`.

The case of procedures is handled by `check-proc`. The parameters are added to the set of defined names `def` and the names declared to be symbolic are added to the set `sym`. The local variables and the procedure parameters are removed from the sets returned since they will have no meaning outside the body of the procedure. It is possible to report on unused local names or names defined but not used on a per-procedure basis, and this is indicated by the invocation of `optionally-report-inner-proc`.

VI. An Implementation

This section describes a Maple 2022 [9] implementation of the algorithm presented in Section V. The code itself is given in the Appendix. The implementation differs from the algorithm presented in a few ways. First, it combines the `def` and `sym` sets since, in finding implicit indeterminates, a symbolic declaration is as good as an assignment. Secondly, it allows scoped variable names to shadow each other.

Names are declared symbolic by including an option $\texttt{symbolic}(\texttt{n}_1, ...\texttt{n}_k)$ in a procedure, for example

```
diffFunction := proc(f)
    local x;
    option symbolic(x);
    unapply(diff(f(x), x), x)
end proc:
```

This method of declaring variables intended to be symbolic allows backward compatibility with earlier releases of Maple.

The implementation relies on the `ToInert` function of Maple that provides a traversable form of all Maple objects. In particular, it allows the examination of procedure bodies, including the lists of parameters, local variables, options, remember table, executable statements, *etc*. The implementation uses a package `InertTools` to abstract the details of the representation returned by `ToInert`. Complexities in the implementation arise from dealing with Maple's representation of lexically scoped names. This requires adding and removing lexical levels to the internal representation of parameters and local variables in the used/defined/symbolic variable sets as procedure bodies are traversed recursively.

A sample session is in Figure 4.

The output of used and assigned or symbolic names allows tracing the use of names and may be turned off. The appearance of `true` in the last list of used names arises from the internal representation of Maple's `do` loops having a default `while true`.

VII. Conclusions

We have seen how the notions of variables arising in mathematics and computing are different and how the differences in the meanings give rise to problems in symbolic mathematical computation. In particular, when unassigned programming variables are interpreted by a system as mathematical indeterminates, simple programming errors can give rise to hard to detect bugs in correctness or efficiency. We have presented an algorithm that detects programming variables that are neither assigned nor declared symbolic and a Maple 2022 implementation that does not require any modification of existing libraries.

Acknowledgement We thank one of the anonymous referees for particularly thorough comments.

References

[1] Euclid, $\Sigma\tau o\iota\chi\epsilon\ddot{\iota}\alpha$ (The Elements), c. 300 BCE.

[2] François Viète, *Opera Mathematica, in unum volumen congesta ac recognita, opera atque studio Francisci a Schooten*, (Mathematical works, collected and revised in one volume, the works and study of Francis Schooten), Officine de Bonaventure et Abraham Elzevier, Leyde, 1646. (http://gallica.bnf.fr/ark:/12148/bpt6k107597d.pdf retrieved July 1, 2022).

[3] J.H. Laning Jr and N. Zierler, *A Program for the Translation of Mathematical Equations for Whirlwind I*, Engineering Memorandum E-364, Instrumentation Laboratory, Massachusetts Institute of Technology, January 1954.

[4] Remmington Rand Univac, Flow-matic Programming System, Sperry Rand Corporation, 1958.

[5] The Mathlab Group, *MACSYMA Reference Manual, Version nine*, Laboratory for Computer Science, Massachusetts Institute of Technology, 1977.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools 2nd ed*, Pearson Education, 2006.

[7] D. Wardle, *Sometimes a name is just a name: Tacitus' Use of 'Augustus'*, Acta Classica LVIII (2015), 166-190.

[8] Lewis Carrol, *Through the Looking Glass*, Macmillan, 1871.

[9] Maple 2022. Maplesoft, a division of Waterloo Maple Inc, 2022.

Appendix: Code of Maple Implementation

```
###############################################################################
#
# Package to analyze procedures for variables that are used but
# are neither assigned nor declared symbolic.
#
# (C) Copyright 2022, Stephen M. Watt.
#
###############################################################################

UndeclaredSymbolFinder := module ()
    export CheckUnassigned;

    local  DefUse_RHS, DefUse_LHS, DefUse_Proc,
           getDeclaredSymbolicNames, asName,
           convertToOuter, convertToInner,
           isStringOfAssigned, isStringOfProtected,
           showFlag, perProcFlag, noName, noProcInfo;

    uses InertTools;
           # Provides  isInert  and
           # * tests for specific primitive structures:  isInertXxx      Xxx = type
           # * constructors for new inert objects:       newInertXxx     Xxx = type
           # * extractors to get parts of inert objects: getInertXxxYyyy Xxx = type, Yyy= part

    noName      := 'anonymous';                              # When no LHS is being defined.
    noProcInfo  := getInertProcInfo(ToInert(proc() end));  # When not analyzing any proc.
    showFlag    := false;                                    # Set to get more info.
    perProcFlag := true;                                     # Give info about each proc.


    ###########################################################################
    # Main entry point to check a procedure.
    ###########################################################################
    #
    # p0          a procedure to be analyzed.
    # assignedTo  the name a proc p0 is being assigned to or 'anonymous'.

    CheckUnassigned := proc(p0, assignedTo)
        local p, nmUsed, nmDefOrSym, nmUsedUnassigned;
        if type(p0, 'name') and assigned(p0) then
            p := op('p0')
        elif type(p0, 'procedure') then
            p := p0
        else
            error "CheckUnassigned requires a procedure as argument"
        end if;

        # Environment variable to communicate downward required information
        # about the procedure being analyzed, e.g. params, locals, lexicals.
        _EnvProcInfo := noProcInfo;

        nmUsed, nmDefOrSym := DefUse_RHS(ToInert(op('p')), {}, {}, assignedTo);

        nmUsedUnassigned := nmUsed minus nmDefOrSym;
        nmUsedUnassigned := remove(isStringOfAssigned,  nmUsedUnassigned);
        nmUsedUnassigned := remove(isStringOfProtected, nmUsedUnassigned);

        if nmUsedUnassigned <> {} then
            printf("Used but neither assigned nor symbolic: %q\n",
                   op(map(asName, nmUsedUnassigned, noProcInfo)))
        end if
    end proc:


    ###########################################################################
    # Recursive descent to collect sets of names according to use.
    ###########################################################################
    #
    # ob          the statement or expression to be handled.
    # nmUsed0     is the set of symbols (names) used so far in this scope.
    # nmDefOrSym0 is the set of symbols (names) assigned (Def) or declared symbolic (Sym).
    # assignedTo  gives the name the RHS will be assigned to, if any.
```

```
DefUse_RHS := proc(ob, nmUsed0, nmDefOrSym0, assignedTo)
    global showFlag;
    local frees, bounds, nmUsed, nmDefOrSym, i;

    nmUsed     := nmUsed0;
    nmDefOrSym := nmDefOrSym0;

    if isInertVariable(ob) then
        nmUsed := nmUsed union { ob }
    elif isInertConstant(ob) then
        # Skip
    elif isInertProc(ob) then
        nmUsed, nmDefOrSym := DefUse_Proc(ob, nmUsed, nmDefOrSym, assignedTo)
    elif isInertAssign(ob) then
        nmUsed, nmDefOrSym := DefUse_LHS(op(1, ob), nmUsed, nmDefOrSym);
        nmUsed, nmDefOrSym := DefUse_RHS(op(2, ob), nmUsed, nmDefOrSym, asName(op(1, ob)))
    elif isInertFor(ob) then
        nmUsed, nmDefOrSym := DefUse_LHS(op(1, ob), nmUsed, nmDefOrSym);
        for i from 2 to nops(ob) do
            nmUsed, nmDefOrSym := DefUse_RHS(op(i, ob), nmUsed, nmDefOrSym, noName)
        end do
    elif isInert(ob) then
        for i from 1 to nops(ob) do
            nmUsed, nmDefOrSym := DefUse_RHS(op(i, ob), nmUsed, nmDefOrSym, noName)
        end do
    else
        error "Unhandled object kind in DefUse: ", ob
    end if;
    nmUsed, nmDefOrSym    # Return pair of sets
end proc:

DefUse_LHS := proc(ob, nmUsed0, nmDefOrSym0)
    local nmUsed, nmDefOrSym, opi;

    nmUsed       := nmUsed0;
    nmDefOrSym   := nmDefOrSym0;

    if isInertVariable(ob) then
        nmDefOrSym := { ob } union nmDefOrSym
    elif isInertExpseq(ob) then
        for opi in ob do
            nmUsed, nmDefOrSym := DefUse_LHS(opi, nmUsed, nmDefOrSym)
        end do
    elif isInertTableref(ob) then
        nmUsed, nmDefOrSym := DefUse_LHS(getInertTablerefTable(ob), nmUsed, nmDefOrSym);
        nmUsed, nmDefOrSym := DefUse    (getInertTablerefIndex(ob), nmUsed, nmDefOrSym)
    else
        error "Unexpected LHS type"
    end if;
    nmUsed, nmDefOrSym    # Return pair of sets
end proc:

DefUse_Proc := proc(ob, nmUsed0, nmDefOrSym0, assignedTo)
    local s, nmUsed, nmDefOrSym;

    _EnvProcInfo := getInertProcInfo(ob);
    nmUsed       := convertToInner(nmUsed0,      _EnvProcInfo);
    nmDefOrSym   := convertToInner(nmDefOrSym0,  _EnvProcInfo)
                    union { op(newInertParams(nops(_EnvProcInfo:-paramSeq))) }
                    union getDeclaredSymbolicNames(_EnvProcInfo); # option symbolic(...) names

    if  showFlag = true then _EnvProcInfo:-show() end if;

    for s in _EnvProcInfo:-statSeq do
        nmUsed, nmDefOrSym := DefUse_RHS(s, nmUsed, nmDefOrSym, noName);
    end do;

    if perProcFlag = true then
        printf("\n");
        printf("Analyzed %a of %a\n", assignedTo, [op(map(asName, _EnvProcInfo:-paramSeq))]);
        printf("Used:                 %q\n", op(map(asName, nmUsed)));
        printf("Assigned or symbolic: %q\n", op(map(asName, nmDefOrSym)))
    end if;

    # Return pair of sets
    convertToOuter(nmUsed, _EnvProcInfo), convertToOuter(nmDefOrSym, _EnvProcInfo)
end proc:
```

```
    ##########################################################################
    # Utility functions.
    ##########################################################################

    # Get the declared symbolic names from an option sequence.
    getDeclaredSymbolicNames := proc(procInfo)
        local r;
        r := { op(procInfo:-optionSeq) };
        r := select(isInertFunctionNamed("symbolic"), r);
        r := map(e -> op(getInertFunctionArgs(e)), r);
        r
    end proc:

    # Get the name of the various sorts of inert variables.
    asName := proc(ob)
        convert(proc()
            if   isInertParam(ob)        then getInertParamString       (ob, _EnvProcInfo)
            elif isInertLocal(ob)        then getInertLocalString       (ob, _EnvProcInfo)
            elif isInertLexical(ob)      then getInertLexicalString     (ob, _EnvProcInfo)
            elif isInertName(ob)         then getInertNameString        (ob)
            elif isInertAssignedName(ob) then getInertAssignedNameString(ob)
            else ob
            end if
        end proc(), name)
    end proc:

    # Keep names visible in outer scope, converting lexicals to how known there.
    convertToOuter := proc(obSet, procInfo)
        map(proc(ob)
                if   isInertParam(ob)   then NULL
                elif isInertLocal(ob)   then NULL
                elif isInertLexical(ob) then getInertLexicalOuterBinding(ob, procInfo)
                else                         ob
                end if
            end proc,
            obSet)
    end proc:

    # Keep names used in inner scope. Convert lexicals to how used in inner scope.
    convertToInner := proc(obSet, procInfo)
        local newSet, i, ob, nob;
        # Keep only globals
        newSet := MutableSet(select(isInertName, obSet));
        for i to nops(procInfo:-lexicalSeq) do
            ob := getInertProcNthLexicalOuterBinding(i, procInfo);
            if has(obSet, ob) then
                if isInertLocal(ob) or isInertLexicalLocal(ob) then
                    nob := newInertLexicalLocal(i)
                elif isInertParam(ob) or isInertLexicalParam(ob) then
                    nob := newInertLexicalParam(i)
                else
                    error "Unknown lexicaltype."
                end if;
                insert(newSet, nob);
                # newSet := newSet union { nob }
            end if
        end do;
        convert(newSet, set)
    end proc:

    # Is s the name string of a currently assigned name?
    isStringOfAssigned := proc(s)
        local n, nval;
        n    := convert(asName(s, noProcInfo), 'name');
        nval := op(n);
        evalb(n <> nval);
    end proc:

    # Is s the name string of a currently protected name?
    isStringOfProtected := proc(s)
        isProtectedName(convert(asName(s, noProcInfo), 'name'))
    end proc:
end module:
```