UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Universitat Politècnica de València
Departamento de Sistemas Informáticos y Computación

Translating PVS recursive functions into Java iterations

Candidate: Marcelo Cordini Moreno
Supervisor: Santiago Escobar Román
NASA Supervisor: Alwyn Goodloe

Thesis developed at NASA Langley Research Center and National Institute of Aerospace

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Valencia, 10 de Febrero de 2012

Contents

1. Introduction

The Prototype Verification System (PVS) developed at SRI International is a tool for the development and analysis of formal specifications including a theorem prover. PVS does not only provide a theorem prover, it can also be used to declare specifications of digital systems. These declarations contain parts that can be automatically translated into executable code, reducing the errors typically introduced through manual coding.

A prototype code generator (or translator) has been developed during the last years and presented in [1]. It translates PVS functional specifications (the input to the translator) into an intermediate language, called Why, and from there to multiple target programming languages such as Java, the output of the translator.

A problem of PVS is that it does not support one of the most commonly and desirable features in programming languages, iteration. The alternative of iteration is recursion; so the users that want to define a loop are forced to use recursion. However, recursion has many drawbacks for execution such as function-call overhead and memory-management issues. The problem arises when the translator generates the Java code of the recursive functions, because the final code will suffer these problems and a potential user would like to avoid them or, at least, reduce their impact.

Therefore, it is desirable to modify the prototype code generator to be able to translate the PVS recursive functions into iterations in the target programming language. This is going to solve the problem mentioned before but also could improve any recursive call.

1.1 Plan of the Thesis

The thesis is organized as follows. In Chapter 2, we introduce the problem to be solved and in Chapter 3 we explain how we made the solution to work and the followed steps. In Chapter 4, the preliminaries of this thesis are presented, i.e., the main concepts that are important to know to understand what has been done. Then, Chapter 5 is the most important, where the development and the different decisions made to that development are explained. In Chapter 6, a tool demo is provided showing the translation in action. In this chapter, the reader can compare the translator before the changes were made and after them. Also, different cases to see how the translator behaves in diverse situations are shown.

2. Problem

PVS does not support one of the most useful features of imperative programming, iteration. The PVS to Java prototype simply translates recursive PVS functions into recursive Java functions. Yet recursion has drawbacks that affect execution performance and program safety. For instance, the function-call overhead that results from the fact that each recursive call requires the return address and to operate on automatic variables. In addition, another issue with recursion is memory management due to the fact that each function call requires saving information on the call stack.

For the reasons mentioned above, it is desirable to modify the prototype code generator to translate PVS recursive functions into iterations. Translating any recursive function into an iteration is an extremely difficult task that is of active interest to researchers and developers in both academia and industry. Consequently, we focus on the difficult, but more tractable problem of translating tail-recursive PVS functions into iterative Java procedures. In addition, to simplify translating the tail-recursion to iteration, we generate information that allows the translator to produce invariants and assertions to ease the verification of the generated code.

Another problem that we have is how we are going to apply the solution in the existing translator code. The translator was built in such a way that when it analyzes the input theory it is also translating and printing in the new java file. We could take the final java code and then delete the tail recursive or we could modify directly the translator to have the final result that we want.

3. Methodology

This thesis has been developed as part of a research stay in the NASA Langley Research Center and the National Institute of Aerospace (NIA) in 100 Exploration Way Hampton, VA 23666 from July 2011 to October 2011.

The methodology that we have followed is:

1. Reading documentation on the LISP functional programming language and the PVS theorem prover developed at SRI International.
2. Reading documentation on the PVS to Java translator and running test cases to understand how the translator works.
3. Analyzing the code of the PVS to Java translator, since this is the main target for changes.
4. Investigating how recursion is translated into iteration in the literature and how tail recursive could be replaced by a loop.
5. Devising how the translation from recursion to iteration can be included into the PVS to Java translator.
6. Implementation of a first prototype of tail recursion elimination in the translator.
7. Testing the prototype with several case studies, ensuring that the changes are working fine and the previous solution is not affected by these new changes.
8. Reporting to the NASA and NIA supervisor and presentation of the tool to NIA fellows.

4. Preliminaries

In this chapter we have collected from different articles the state of the art of the main concepts used in this work.

4.1 PVS

The Prototype Verification System (PVS) is a tool for the development and analysis of formal specifications. In this section we have merged information from [1] , [2] and [6].

The PVS system consists of a specification language, a parser, a type-checker, a prover, specification libraries, and various browsing tools.

PVS provides an integrated environment for the development and analysis of formal specifications, and supports a wide range of activities involved in creating, analyzing, modifying, managing, and documenting theories and proofs.

The specification language is based on classical higher order logic, augmented with a sophisticated type system that uses predicate subtypes and dependent types. It also has the capability to define algebraic data types. All functions that are defined in the specification language must be total, i.e., functions must be defined for all the input values. However, partial functions can be defined by restricting the domain of the function to a subtype using predicate subtyping. The many features of the PVS type system make it very powerful, but also make type checking in general undecidable. The theorem prover generates type correctness conditions (TCC's) for the undecidable parts of the type checking process. In practice, most of the TCC's are automatically discharged by the system.

The PVS Environment

PVS runs on SUN 4 workstations using Solaris 2, higher and PC systems running Linux or OS X. PVS is implemented in Common Lisp and the Emacs (Gnu Emacs or XEmacs) editors provide the interface to PVS.

The PVS Language

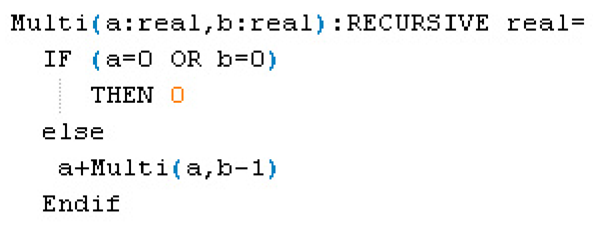
The specification language of PVS is built on higher-order logic; i.e., functions can take functions as arguments and return them as values, and quantification can be applied to function variables. There is a rich set of built-in types and type constructors, as well as a powerful notion of subtype. Specifications can be constructed using definitions or axioms, or a mixture of the two.

Specification Files and the PVS Context

PVS specifications are ordinary ASCII text prepared and modified using a text editor-usually the Emacs editor that acts as the interface to PVS. A PVS specification consists of any number of such files, each of which containing one or more theories or datatypes. PVS specification files have the .pvs extension.

Typechecking

The PVS typechecker analyzes theories for semantic consistency and adds semantic information to the internal representation built by the parser. The type system of PVS is not algorithmically decidable; theorem proving may be required to establish the type-consistency of a PVS specification. The theorems that need to be proved are called type-correctness conditions (TCCs). TCCs are attached to the internal representation of the theory and displayed on request.



```
Multi(a:real,b:real):RECURSIVE real=
  IF (a=0 OR b=0)
     THEN 0
  else
   a+Multi(a,b-1)
  Endif
```

Figure 1 - PVS Code Multiplication example

4.2 Why

“Why” is a software verification platform. In this section we have collected information from [1] and [7].

Basically, the Why tool takes annotated programs written in a very simple imperative programming language of its own, produces verification conditions and sends them to existing provers (proof assistants such as Coq, PVS, etc. or automatic provers such as Simplify, CVC Lite, etc.).

Why builds a functional interpretation of the imperative program given as input, containing both a computational and a logical part. Using this information, the tool applies a Hoare logic and Dijkstra's calculus of weakest preconditions to generate proof obligations. Why's input language, which is also called Why, is based on Milner's ML programming language and has imperative features, such as references and exceptions, and functional features, such as higher-order functions.

The “Why” tool is used as the back-end of verification condition generators. Indeed, the same team that develops Why also develops the tools Krakatoa and Caduceus, which are front-ends for Java and C verification condition generators, respectively.

4.3 Translator from PVS to Java – Code generator

The translator described in [1] which is a code generator prototype to translate from PVS code to multiple target languages, is presented and the translation process is described.

The input to the code generator is a declarative specification written in PVS. Since this work aims at a wide range of applications, the target language is not fixed. Indeed, the tool first generates code in Why, an intermediary language for program verification.

The current prototype generates Java annotated code from Why code. In the future, the generator may be extended to support other functional and imperative programming languages.

In order to increase confidence in the generated code, the generator annotates the code with logical assertions such as pre-conditions, post-conditions, and invariants. These assertions are extracted from the declarations, definitions, and lemmas in the formal model. Therefore, the generated code can be the input of a verification condition generator such as Krakatoa. Krakatoa generates proof obligations for several theorem provers, including PVS. The generated PVS proof obligations are different from the original PVS specification. However, if the original specification has been shown to be correct, discharging the proof obligations is a relatively easy task. The annotated code is also amenable to static analysis, software model checking, and automated test generation.

In Figure 2 the scheme that represents the process of the translator is presented.

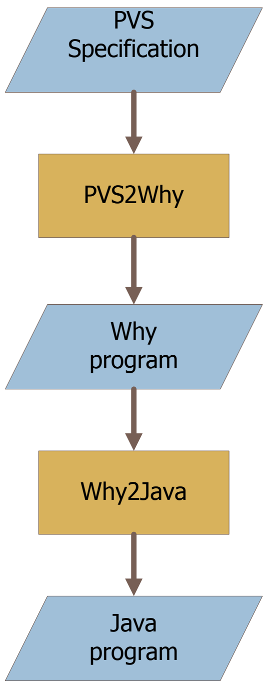


Figure 2 - Translator scheme

4.3.1 Code generation

For the most part, the translation from a declarative PVS specification into the Why language is straightforward. Each language construct in the functional subset of the PVS specification language has an almost immediate counterpart in the Why language. Indeed, like PVS, Why can be used as a purely functional programming language.

In order to ease the translation, the Why language has been extended with several features such as records, tuples, and a simple notion of modules. Although records and tuples could be defined in the logic part of the language, they have been added as syntactic sugar and treated similarly to arrays. Modules provide a naming scope for a set of Why declarations. They correspond directly to the parameterized theories in PVS and allow for modularity in the generated programs. A more general notion of module that includes the notion of interface is currently being added to the Why core language.

4.3.2 Interface

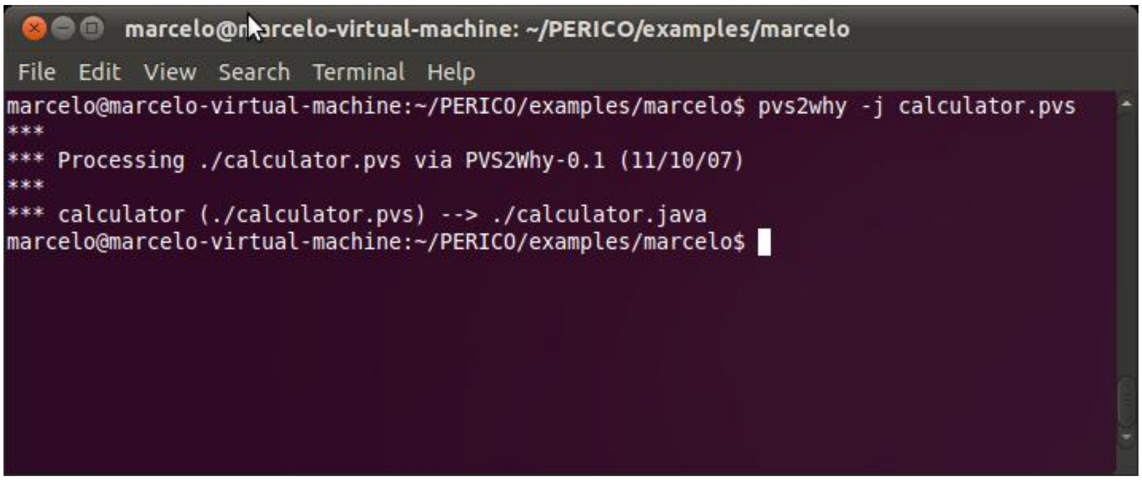
In this section, we are going to show how the translator can be executed.

The translator is a console application, so the interface is a command-line interface. To execute the translator we need to write the following command:

- **pvs2why** –j *file.pvs*
    - *pvs2why* is the translator command to execute it.
    - *–j* is to select the option to translate to Java.

- *file.pvs* is the input file that contains the PVS theory

In Figure 3 an execution example is shown.


marcelo@marcelo-virtual-machine: ~/PERICO/examples/marcelo
File Edit View Search Terminal Help

```
marcelo@marcelo-virtual-machine:~/PERICO/examples/marcelo$ pvs2why -j calculator.pvs
***
*** Processing ./calculator.pvs via PVS2Why-0.1 (11/10/07)
***
*** calculator (./calculator.pvs) --> ./calculator.java
marcelo@marcelo-virtual-machine:~/PERICO/examples/marcelo$
```

Figure 3 - Interface example

When we execute the translator, if there is not any problem, we are going to receive a notification that the new java file is created.

If there are errors in the PVS file, the tool will report the error provided by the PVS system.

4.4 Tail recursion

First of all, to define a tail recursion we need to recall a tail call from [4] and [5].

A procedure call F is a tail call if its caller C does not do any additional processing after F. C must return the same value than F (if there is any). Tail calls are significant because C no longer needs any of the data on the stack (except the return address) and could be discarded or reused by F.

A call is tail recursive if it is a tail call and is directly recursive. Therefore, there is tail recursion when a recursive call is the last thing a function does.

4.5 Tail Recursion removal - Recursive vs Iteration

The main focus of this thesis is tail recursion removal. That is, transforming a tail recursive function into an iterative procedure. To achieve this goal, we modify the prototype PVS to Java translator so that as it transforms tail recursive functions into java code. We have included details from [5].

In [5] the removal of tail recursion is described as an optimization that often is done by hand at source level. It describes an implementation in a C compiler and its benefits and compares them with removing by hand at source level.

When the callee returns from a tail call, the caller will immediately return as well, just passing the return value of the tail call, if there is any. For that reason, the caller just needs the return address. Any other environment data stored in the stack frame is not useful and could be deleted.

If it is not necessary to return the control to the caller, the procedure call could be replaced with a simple jump. The argument passing is going to be replaced with assignments and the stack frame can be overwritten and reused.

The steps proposed in [5] are:

a) insert assignments where needed to simulate argument passing,

b) add a label at the beginning of the function body,

c) replace the tail-recursive call with a goto to that label.

In [5], they are thinking in the implementation in a C code and that is why they use the GOTO label. Our work is focused on Java and there is no GOTO label, so we need to find an alternative to this label and we do not have another alternative than a loop.

4.5.1 Performance

The original function has stack space linear in the length of the input string. Each call required pushing a return address, as well as saved registers, onto the stack. Also, on return the function had to pop each stack frame in turn, and execute one return for each call. In the transformed version, space usage is constant and the call and return sequence is executed only once.

The use of tail recursion and recursion in general, can make life significantly easier for the programmer. Many problems are solved most readily with recursive algorithms, and recursive algorithms are often shorter and clearer than their iterative counterparts. While some functions can be converted to iteration at the source level fairly cleanly, in other cases this conversion is not intuitive or reduces the readability of the code.

With the benefit of automatic tail-recursion removal, both the tail-recursive and hand-optimized, iterative version will compile to equivalent machine code. This optimization allows us to preserve elegance at the source level, without sacrificing efficiency in the compiled program.

The qualitative observation that recursive code is often clearer is backed up by some quantitative data. Benander, Benander, and Sang in [8] performed a study of debugging performance in undergraduates, using recursive and iterative solutions for the same problems. They presented students with logically incorrect C programs for searching and copying linked lists, and asked them to debug them by inspection, which they argue remains the primary method of debugging once the problem has been isolated to a small segment of code. They found that students were significantly more likely to correctly fix the bugs in the recursive programs than in the iterative program. Their study lends support to the claim that some programs are more elegantly expressed in a recursive form, even in C, where iteration is traditionally favored.

In a tail recursive function, stack space grows in O(n), where n is the depth of recursion. However, this optimization removes the procedure call and recycles the same stack frame, which ensures constant use of stack space. Programs which otherwise might have run out of stack space can, after optimization, execute to completion.

While the abundance of memory on desktop PC and servers makes this a minor concern, in embedded environments, where memory usage is a significant constraint, this may prove to be a greater benefit.

5. Solution

In this section we present the solution to the problem described in Section 2, to remove the tail recursion. First, we identify and present an algorithm to remove tail recursion. The presentation illustrates how to manually remove tail recursion so as to convey the idea of the algorithm. Second, we discuss the implementation including the need to modify the existing PVS to Java translator code base without modifying its behavior in undesirable ways.

It is really important to understand the translator input and how the translator manages it. The input is a PVS file that contains a theory. This theory contains a function defined in PVS that is translated to Java. But the translator does not treat this input theory function as text, it parses it and creates an appropriate data structure representation. Then in the translator we are going to query this data structure that is composed by fields.

5.1 Algorithm

In [4] and [5] there is a description of how to replace tail recursive by iteration. Based on that the steps that we have to follow to replace the tail recursive with iteration are:

1. Surround the function body with a “while(true)” loop
2. Delete the tail recursive call
3. Set parameters to their new values where the tail recursive was. We have to insert assignments where needed to simulate argument passing

Neither in [4] nor [5] there is any mention to what happened in step 3 when the value of a parameter “a” is used to set the new value of other parameter “b” and the value of “a” was changed before. For example

```
someFunction(a,b,c){
    …
    return  someFunction(a+1, a+b, c)]
}
```

Following the 3 steps mentioned before the result of the function translation is going to be:

```
someFunction(a,b,c){
    …
    a=a+1
    b= a+b
    c= c
}
```

Therefore, before step 3, it is necessary to backup the values in temporary variables because one parameter could be used in more than one input parameter and its value must be always the one at the moment the recursive call is done.

We are replacing the value of “a” and we need the previous value. So, we should do that in any of these 2 ways:

```
someFunction(a,b,c)
…
a’=a+1
b’= a+b
c’= c
a=a’
b=b’
c=c’
```

or

```
someFunction(a,b,c)
…
a’=a
b’=b
c’=c
a=a’+1
b= a’+b
c= c
```

Also, we should add another step (at the beginning) to identify the tail recursive call.

So, we have five identified steps in our algorithm:

- Step 1. Identify the tail recursive call
- Step 2. Surround the function body with a “while(true)” loop
- Step 3. Delete the tail recursive call
- Step 4. Backup the parameters values in temporary variables
- Step 5. Set parameters to their new values

This describes how to perform the transformation manually while the remainder of this section describes our effort to automate this approach by modifying an existing translator prototype.

5.2 Implementation

In this section we are going to describe how we have implemented the five steps described before.

It is important to understand that we have not developed the tail recursive elimination as a separate algorithm, but we have modified the existing translator code in different places. The main reason of this is because the translator was built in such a way that when it analyzes the input theory it is also translating and printing in the new java file. So if we had created a separate algorithm, we would have taken the final java file and then apply the tail recursive elimination. The problem of this is that we are losing important information that the PVS theory provides and also it is more complicated because we need to create a Java parser.

5.2.1 Running the program

The translator interface, presented in Section 4.3.1, was modified to give the user the option to remove the tail recursive or not. We add a new flag “t”, that indicates if the user wants to delete the tail recursive calls. This will set the variable *tail-recursive-flag* and start to run the translator on tail recursive remove mode.

The modified code is displayed in Chapter 9.

5.2.2 Step 1 - Identify the tail recursive call

We need to identify if the function has a tail recursive call. If there is not any tail recursive call, the function will not be modified and the translator is going to be executed as always.

As it is mentioned in Section 4.4 a call is tail recursive if:

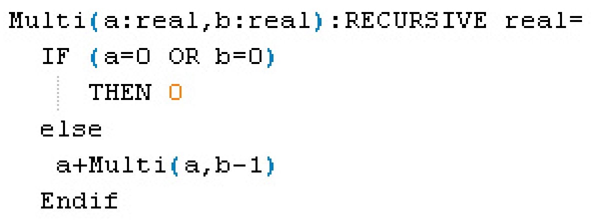
- it is directly recursive
- it is a tail call

We must check both issues to identify a tail recursive call.

5.2.2.1 Check if it is recursive

To check if it is recursive, we are going to do it in two places in the input code: in the function signature and in the call declaration.

In the function signature

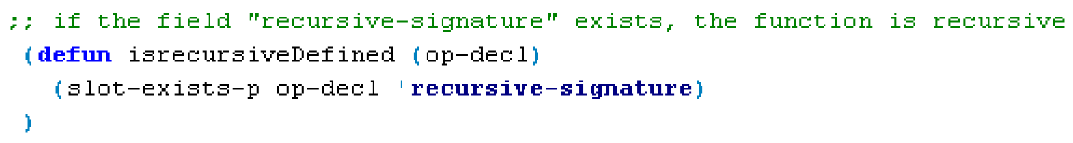


```
Multi(a:real,b:real):RECURSIVE real=
  IF (a=0 OR b=0)
     THEN 0
  else
   a+Multi(a,b-1)
  Endif
```

Figure 4- PVS Code Recursive and non-tail call example

In PVS, if you want to define a recursive function it must be explicitly indicated in its signature with the term RECURSIVE. In Figure 4 we can see an example of this, where the Multiplication is defined as a recursive function.

As we said before, we are not looking to the function as text, but the function is represented by a structure that contains different fields. There is more than one field in the function definition that we can use to deduce that the function is recursive. The field "recursive-signature" is the only one that implies it directly, so we are going to use that one, because probably is going to avoid some misunderstanding in the future. The function *isrecursiveDefined* was created in *pvs2why.lisp* to check this field as it is shown in Figure 5.



```
;; if the field "recursive-signature" exists, the function is recursive
 (defun isrecursiveDefined (op-decl)
   (slot-exists-p op-decl 'recursive-signature)
 )
```

Figure 5 – LISP Code for isrecursiveDefined function

When the translator found a function in the input, it is declared creating an instance of the class *why-function*. In this project, we have created two new fields in the structure and we have added a new constructor just to store the information if we found that the function is recursive.

```lisp
(defcl why-function (why-def)
  (parameters :type list) ; List of why-binding
  (return-type :type why-type)
  (type :type why-type)
  (body :type why-expr)
  (precondition :type why-expr)
  (postcondition :type why-expr)
  isRecursive
  (measure :type why-expr))
```

Figure 6 – LISP code for why-function

As it is shown in Figure 6, we have created two new fields (the last two) in the structure *why-function*:

- isRecursive: a Boolean that declares if the function is recursive
- measure: store the measure defined in PVS for the recursive function.

We have to set these new fields when the object is created. Just to be compatible with the previous solution we are going to keep the same input parameters in the constructor, called *mk-why-function*, and create a new constructor for the recursive functions, called *mk-why-function-recursive*.

In the call declaration

The other place to check if the function is recursive is in the call declaration. For example in Figure 4, we can see that the call "Multi(a,b-1)" is calling a function with the same signature and parameters of the function where we are.

To check this, we create a new method *isRecursiveCall* showed in Figure 7.

```lisp
;;check if the call of a function in lexpr is recursive
(defmethod isRecursiveCall (lexpr)
  (and
       (why-function-application? lexpr)
       (and (equal (identifier(why-identifier *function-definition*))
                   (identifier(operator lexpr))
            )
            (equal (length(parameters *function-definition*))
                   (length(arguments lexpr)))
            )
  )
)
```

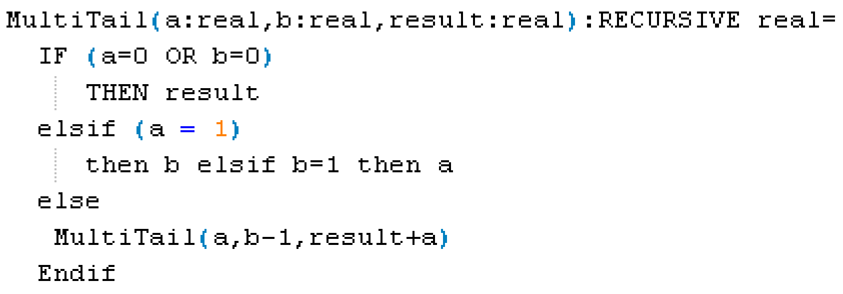
Figure 7 – LISP code for isRecursiveCall method

We just check that the identifiers and the length of the arguments are the same.

5.2.2.2 Check if it is a tail call

We check if a call is a tail call in a simple way. How we are translating functions? If the *return* statement is the function call, then it is a tail call. If the return statement is something else, it means that it is going to do something else before return and that is not a tail call. So, we changed the translator where the *return* sentence is translated and if we found a call there, it is a tail call. The modified code can be found in Figure 9.

For example, PVS code in the Figure 4 is not going to be a tail call because the return sentence is *a+Multi(a,b-1).*



```
MultiTail(a:real,b:real,result:real):RECURSIVE real=
  IF (a=0 OR b=0)
     THEN result
  elsif (a = 1)
     then b elsif b=1 then a
  else
   MultiTail(a,b-1,result+a)
  Endif
```

Figure 8 – PVS Code for recursive and tail call example

In Figure 8 there is an example of a tail call. The translator is going to take the return statement *MultiTail(a,b-1,result+a)* and realize that is a function call.

5.2.3 Step 2 - Surround the function body with a while loop

If in Step 1 we identify a tail recursive call, we are going to continue with Step 2. If not, the translator must behave as before these changes.

As we explained before to convert the recursive function to iteration we are going to surround the function body with a *while(true)* loop.

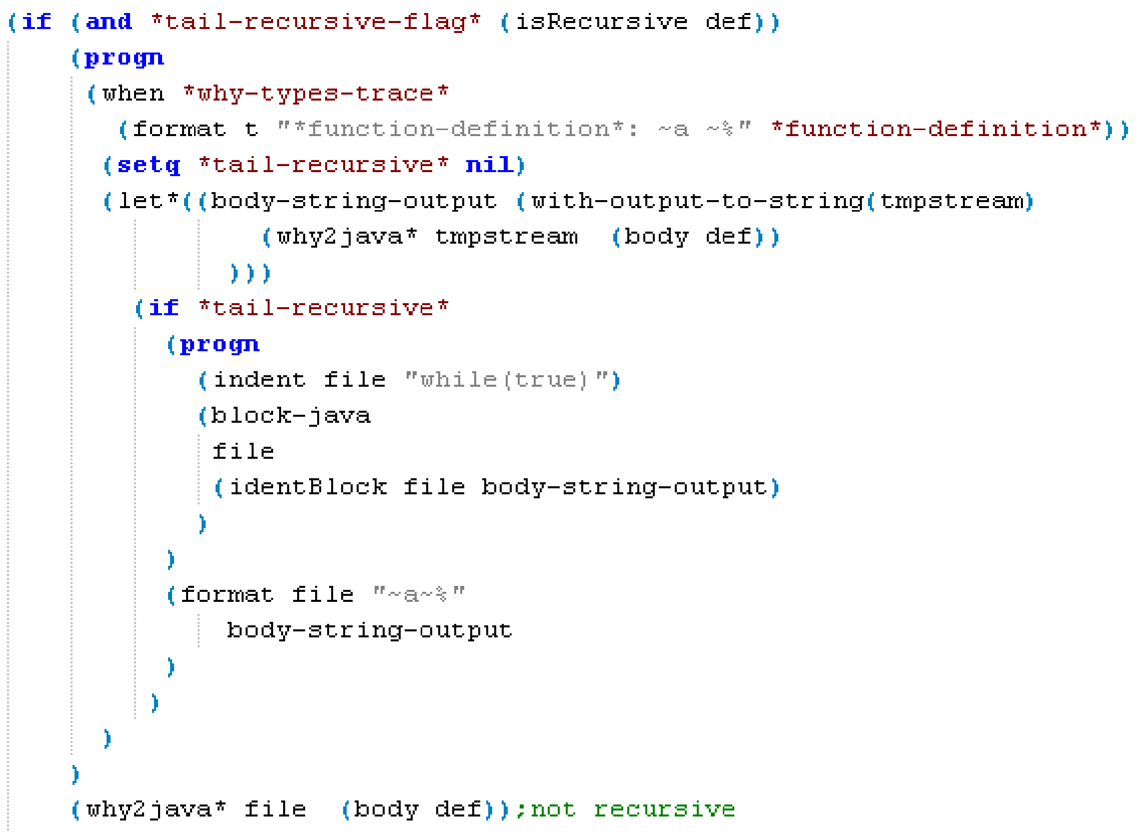
We found in *write-java-function* the place the translator starts to translate the body, so if we are going to insert the *while(true)* block, if it must be there.

The translator was built in such a way that when it analyzes the input theory it is also translating and printing in the new java file. If we are going to delete the tail recursive we must insert the *while(true)* block at the function beginning, so we need to know if it is tail recursive or not before start printing in the new file. We could do this in 2 ways:

1. Make an analysis of the function before starting the translation to know if it is tail recursive and then start the translation knowing whether it is tail recursive or not

2. Store the translation result of the body in a temporal stream. If we found a tail recursive call, delete that and replace it what we saw before. Then print in the Java file the "while" block and then copy from the temporal file the result of the translation. We must correct the indentation because there is a new block. If it is not tail recursive we copy directly the body without adding the "while" block.

We took the second option because it is less complicate to do and requires less encoding time.



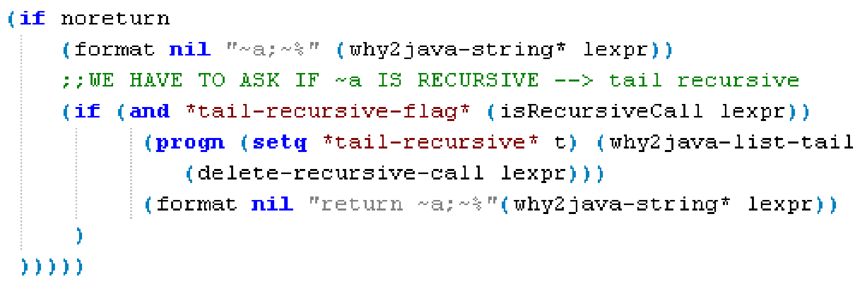
```
(if (and *tail-recursive-flag* (isRecursive def))
    (progn
     (when *why-types-trace*
       (format t "*function-definition*: ~a ~%" *function-definition*))
      (setq *tail-recursive* nil)
      (let*((body-string-output (with-output-to-string(tmpstream)
                    (why2java* tmpstream  (body def))
                  )))
        (if *tail-recursive*
          (progn
            (indent file "while(true)")
            (block-java
             file
             (identBlock file body-string-output)
            )
          )
          (format file "~a~%"
             body-string-output
          )
        )
      )
    )
    (why2java* file  (body def));not recursive
```

Figure 9 – LISP Code for write-java-function surrounding the body function

In Figure 9 we can see how we store the translation result in the variable *body-string-output*. If in the process we identify a tail recursive call (**tail-recursive** variable) we are going to insert the *while* loop. If not, the translator is going to behave as before the changes.

5.2.4 Step 3 - Delete the tail recursive call

This step is the easiest one. In the same place that we have explain in Step 1 how a tail recursive call is detected we can delete the recursive call. As it is shown in Figure 10, if we detect a tail recursive call we are not going to write the *return* sentence and besides of that we call the method *delete-recursive-call.* This method is explained and developed in the next two steps.



```
(if noreturn
    (format nil "~a;~%" (why2java-string* lexpr))
    ;;WE HAVE TO ASK IF ~a IS RECURSIVE --> tail recursive
    (if (and *tail-recursive-flag* (isRecursiveCall lexpr))
          (progn (setq *tail-recursive* t) (why2java-list-tail
             (delete-recursive-call lexpr)))
          (format nil "return ~a;~%"(why2java-string* lexpr))
      )
 )))))
```

Figure 10 – LISP Code for how we identify the tail recursive call

5.2.5 Step 4 - Backup the parameters values in temporary variables

As we explain in section 5.1, we need to backup the parameters values in temporary variables because they can be used to set the new parameters values (in Step 5) and we need the previous value.

In the method *delete-recursive-call*, showed in Figure 11, we have made this step and the next one. First we collect the parameters with its identifier and type and the function call arguments values that are going to match these parameters in the collection *loopvar*. Therefore, the collection *loopvar* it is going to have the following tuple:

(parameter-type, parameter-identifier, new-value)

As we have explained in Section 5.1, we have two ways to do the backup but we choose to backup the new values besides to backup the old variables because it was easier to implement.

The thing to do in the backup is to store the new values that are passed in the function call in temporary variables. Taking the elements in the collection we are going to use the following template: a b = c. "a" is going to be parameter type and can be found in *parameter-type*. Variable "b" is the temporary variable identifier and is going to be formed with the *parameter-identifier* plus "temp". Variable"c" is the new variable value, found in *new-value*. In Lisp, to replace a value with some variable we use the symbol "~a". So in the code the template is going to be like this: ~a ~atemp = ~a

Owing to some indentation issues we separate the first element of the collection in *carloopvar* and the rest of the collection in *cdrloopvar*, because the first one not need indentation.

```
;;replace the recursive call expr with the new values of the variables
(defmethod delete-recursive-call (expr)
  (append
   (let* ((loopvar (loop for x in (parameters *function-definition*)
                         for y in (arguments expr)
                         collect (list
                            (why2java-type* (type x))
                            (identifier (why-identifier x))
                            (why2java-string* y))
                         )
                   )
          (carloopvar (car loopvar))
          (cdrloopvar (cdr loopvar))
          )
     (append
       (list (format nil "~a ~atemp = ~a" (car carloopvar) (cadr carloopvar) (caddr carloopvar)))
       (loop for z in
         cdrloopvar
         collect (format nil "~a~a ~atemp = ~a" (make-string (get-indent) :initial-element #\Space)
            (car z) (cadr z) (caddr z))
            )
       )
   )
   (loop for x in (parameters *function-definition*)
         collect (format nil "~a~a = ~atemp" (make-string (get-indent) :initial-element #\Space)
         (identifier (why-identifier x)) (identifier (why-identifier x)))
         )
   )
)
```

Figure 11 – LISP Code for delete-recursive-call method

5.2.6 Step 5 - Set parameters to their new values

We need to set the parameters with their new values to restart the loop and simulate the recursive function.

In Step 4, we backed up the new parameter values in temporary variables. Now we need to use these temporary variables to replace the parameters values.

As it is shown in Figure 11, the template used in this case is ~a~a = ~atemp, where the first element is just an indentation space, the second and the last one the *parameter-identifier*.

6. Tool demo

In this chapter, it is possible to see the translator in action with the new function of tail recursive elimination.

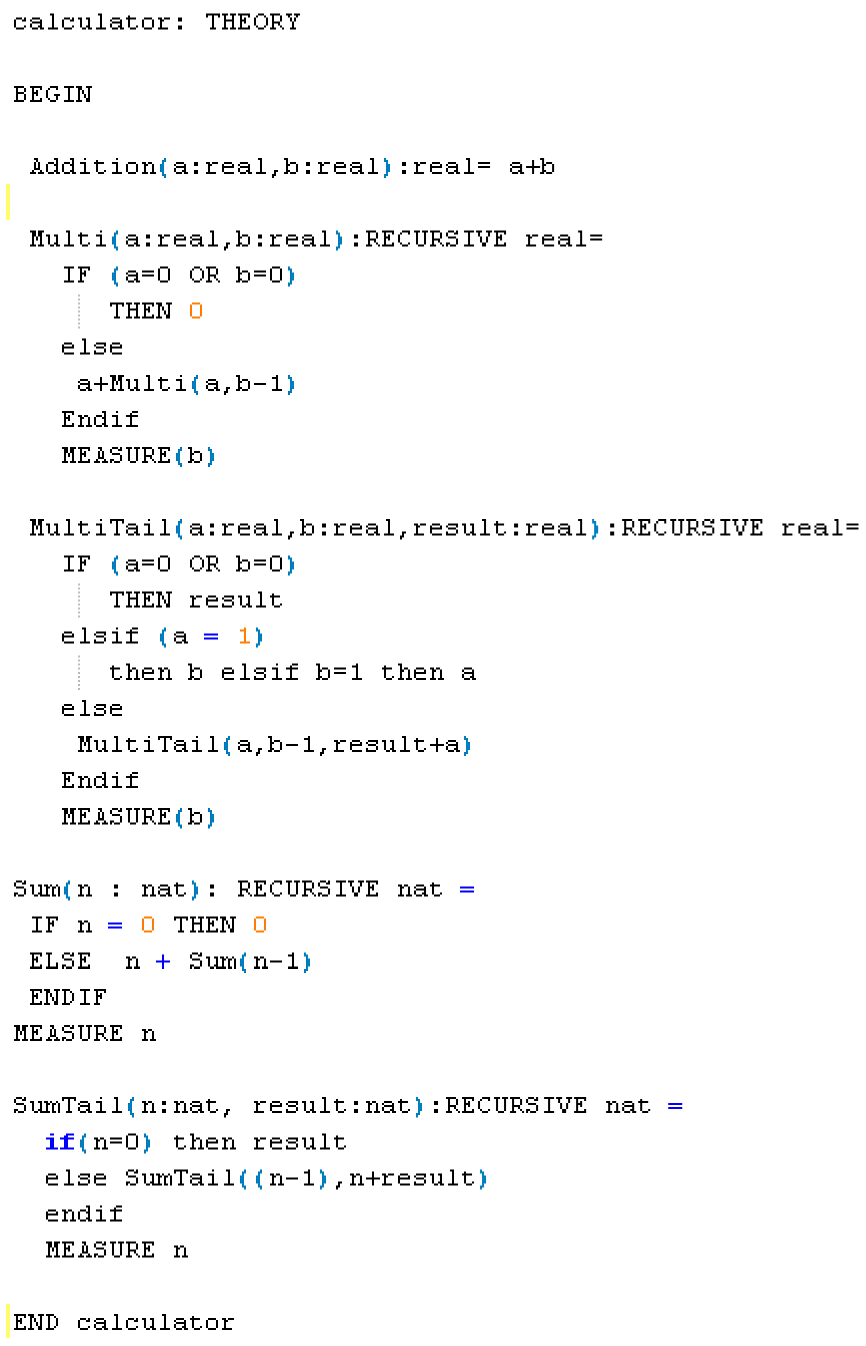
We can compare the translator before the changes that were made and after that. Actually, the translator was modified in such a way that you can choose if you want to delete the tail recursive or not, so if you do not want to eliminate the tail recursive the translator is going to behave as before this project.

Also in this chapter, several case studies are shown to see how the translator behaves in different situations.

6.1 Input

The input of this example execution is a PVS theory called *calculator*. This theory is written in PVS language, as it is explained in Section 4.1, PVS is a system that contains its own language. The theory is located in one file called *calculator.pvs* and actually this file is the real input. This theory is showed in Figure 12.

The *calculator* theory is a specification of a simple calculator with three simple operations: Additional, Summation and Multiplication. We have one more case of each Summation and Multiplication, one tail recursive and other not. This is just to prove that the translation is working fine with different cases. So, in total we have 5 functions in this theory and that is what we are going to translate to Java.



```
calculator: THEORY

BEGIN

 Addition(a:real,b:real):real= a+b

 Multi(a:real,b:real):RECURSIVE real=
   IF (a=0 OR b=0)
      THEN 0
   else
    a+Multi(a,b-1)
   Endif
   MEASURE(b)

 MultiTail(a:real,b:real,result:real):RECURSIVE real=
   IF (a=0 OR b=0)
      THEN result
   elsif (a = 1)
      then b elsif b=1 then a
   else
    MultiTail(a,b-1,result+a)
   Endif
   MEASURE(b)

Sum(n : nat): RECURSIVE nat =
 IF n = 0 THEN 0
 ELSE  n + Sum(n-1)
 ENDIF
MEASURE n

SumTail(n:nat, result:nat):RECURSIVE nat =
  if(n=0) then result
  else SumTail((n-1),n+result)
  endif
  MEASURE n

END calculator
```

Figure 12 – PVS Code Input example: Calculator

6.2 Execution & Output

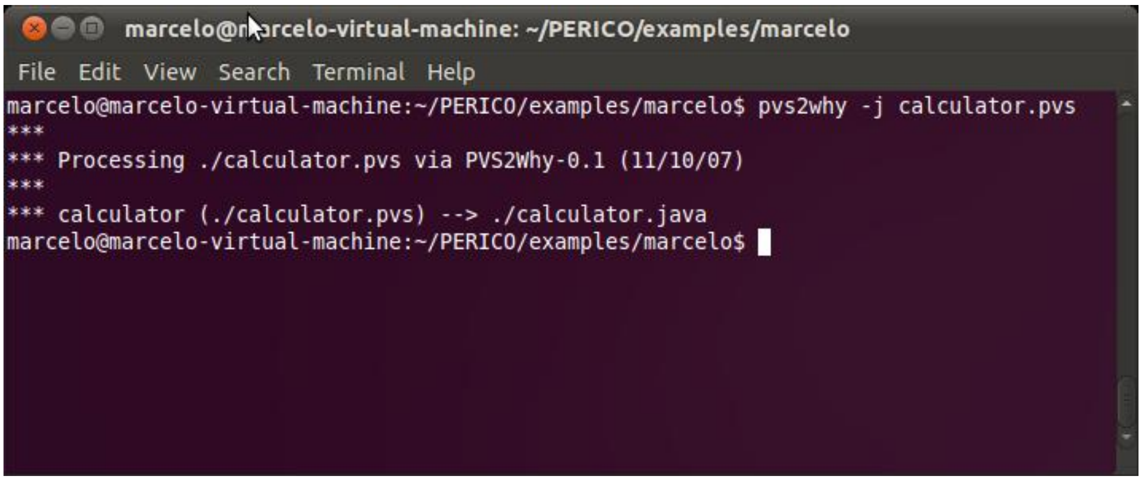
In this section we are going to execute the translator with the input presented in the previous section. First we are going to translate without the tail recursive elimination, just to know how it works. Then we are going to execute with the new functionality and delete the tail recursive.

6.2.1 Translator demo without the tail recursive elimination

In this section, we translate the input theory presented in Figure 12 without the tail recursive elimination, just to know how the translator works without the new functionality that we made. Also, this test is useful to prove that after the changes are made, the translator without the tail recursive elimination still works fine.

As is shown in Figure 13, to execute the translator the command needed is:

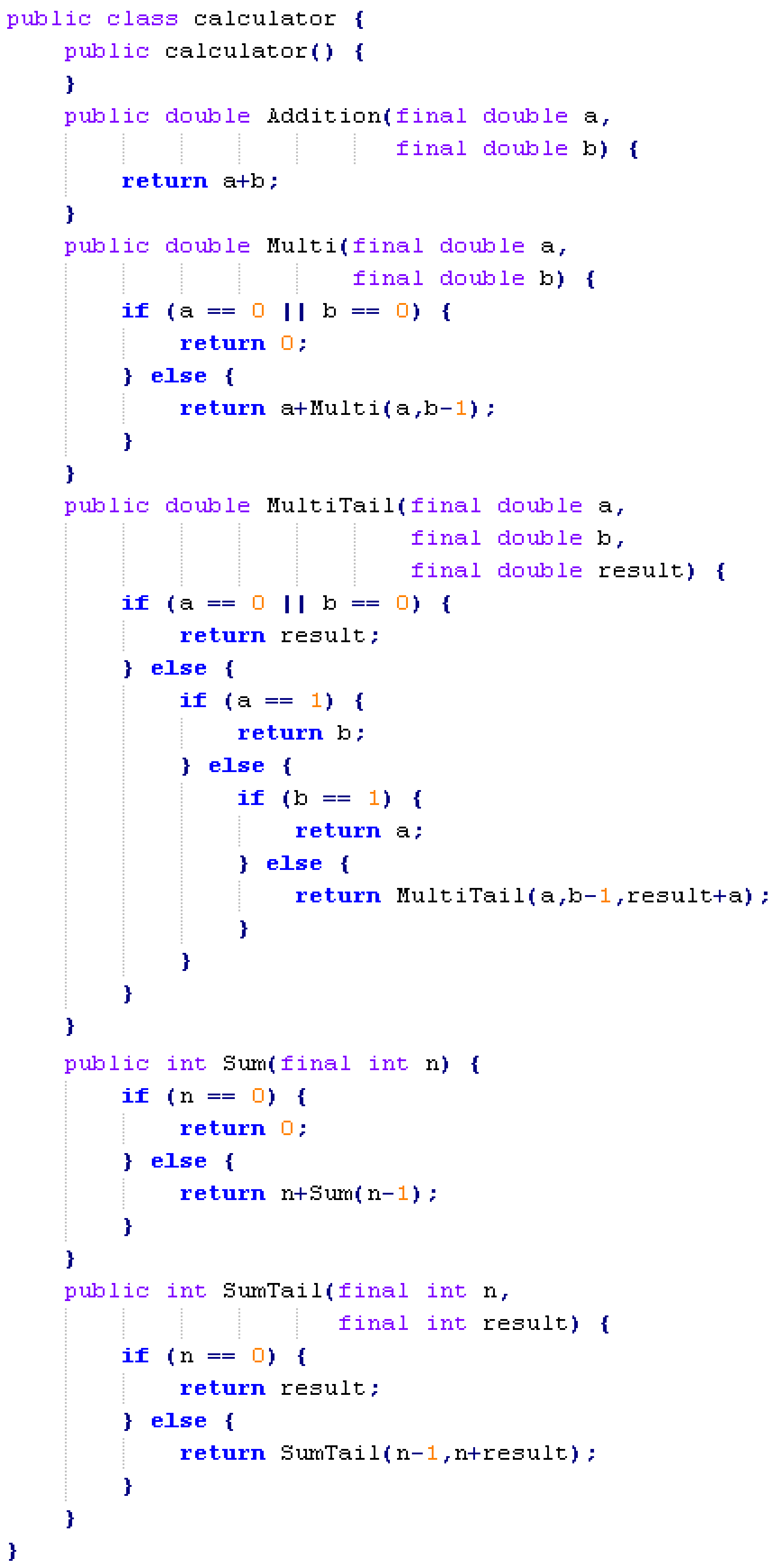
- **pvs2why** –j *calculator.pvs*


marcelo@marcelo-virtual-machine: ~/PERICO/examples/marcelo
File Edit View Search Terminal Help

```
marcelo@marcelo-virtual-machine:~/PERICO/examples/marcelo$ pvs2why -j calculator.pvs
***
*** Processing ./calculator.pvs via PVS2Why-0.1 (11/10/07)
***
*** calculator (./calculator.pvs) --> ./calculator.java
marcelo@marcelo-virtual-machine:~/PERICO/examples/marcelo$
```

Figure 13 - executing translator without tail recursive elimination

As it is shown in Figure 13 the execution output is stored in *calculator.java*. This file contains the PVS to Java translation, where the PVS theory is represented by a Java class showed in Figure 14.



```
public class calculator {
    public calculator() {
    }
    public double Addition(final double a,
                           final double b) {
        return a+b;
    }
    public double Multi(final double a,
                        final double b) {
        if (a == 0 || b == 0) {
            return 0;
        } else {
            return a+Multi(a,b-1);
        }
    }
    public double MultiTail(final double a,
                            final double b,
                            final double result) {
        if (a == 0 || b == 0) {
            return result;
        } else {
            if (a == 1) {
                return b;
            } else {
                if (b == 1) {
                    return a;
                } else {
                    return MultiTail(a,b-1,result+a);
                }
            }
        }
    }
    public int Sum(final int n) {
        if (n == 0) {
            return 0;
        } else {
            return n+Sum(n-1);
        }
    }
    public int SumTail(final int n,
                       final int result) {
        if (n == 0) {
            return result;
        } else {
            return SumTail(n-1,n+result);
        }
    }
}
```

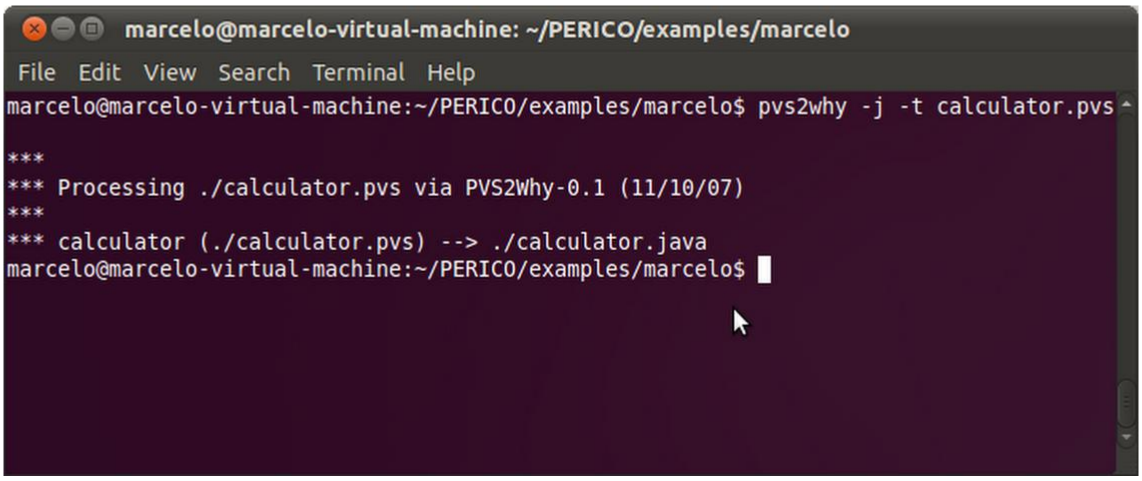
Figure 14 – Java Code Output example without tail recursive elimination

6.2.2 Translator demo with the tail recursive elimination

This section is the most important of the tool demo. We are going to execute the translation with the functionality of tail recursive elimination. This means that if in the PVS theory there is any tail recursion, it should be eliminated and replaced by an iteration.

As it is shown in Figure 15, to execute the translator the command needed is:

- **pvs2why** –j –t *calculator.pvs*

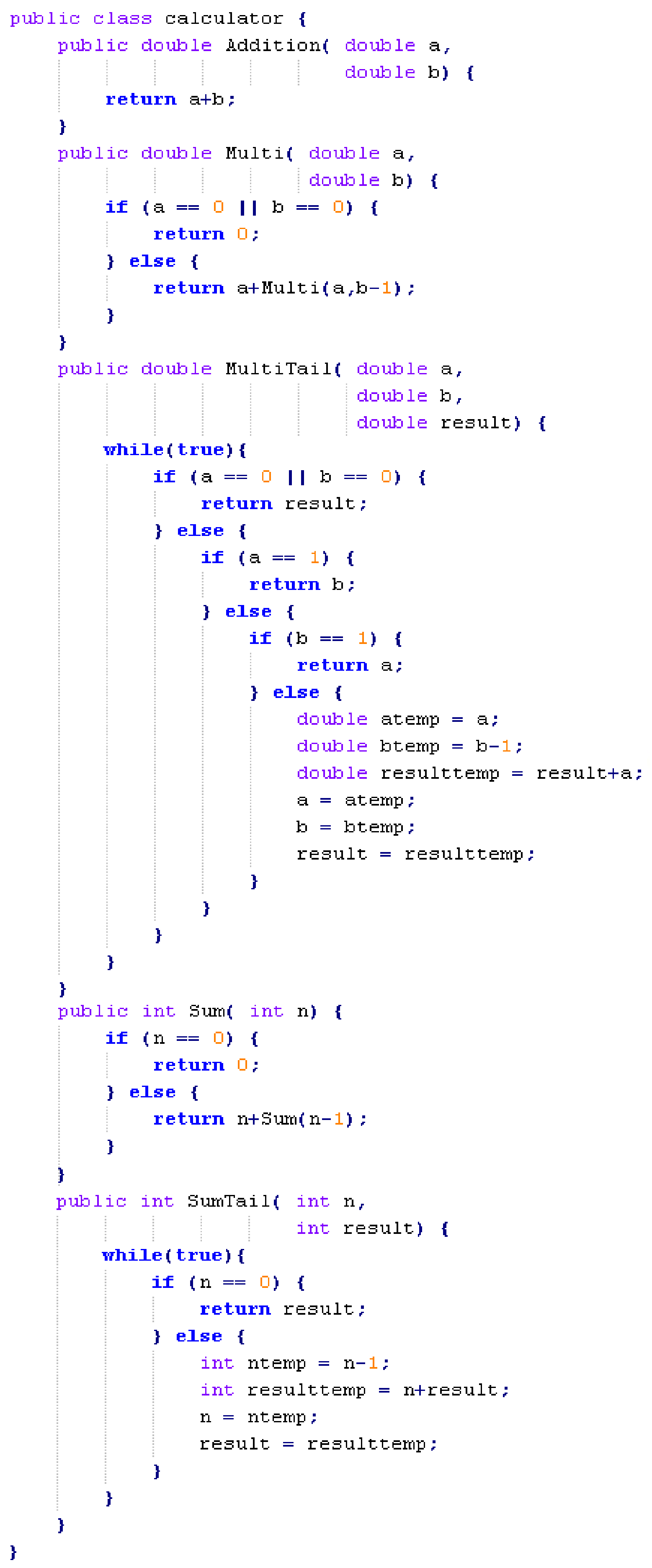

marcelo@marcelo-virtual-machine: ~/PERICO/examples/marcelo
File Edit View Search Terminal Help

```
marcelo@marcelo-virtual-machine:~/PERICO/examples/marcelo$ pvs2why -j -t calculator.pvs

***
*** Processing ./calculator.pvs via PVS2Why-0.1 (11/10/07)
***
*** calculator (./calculator.pvs) --> ./calculator.java
marcelo@marcelo-virtual-machine:~/PERICO/examples/marcelo$ 
```

Figure 15 - executing translator WITH tail recursive elimination

As it is shown in Figure 15 the execution output is stored in *calculator.java*. This file contains the PVS to Java translation with the tail recursive elimination where the PVS theory is represented by a java class showed in Figure 16.



```java
public class calculator {
    public double Addition( double a,
                            double b) {
        return a+b;
    }
    public double Multi( double a,
                         double b) {
        if (a == 0 || b == 0) {
            return 0;
        } else {
            return a+Multi(a,b-1);
        }
    }
    public double MultiTail( double a,
                             double b,
                             double result) {
        while(true){
            if (a == 0 || b == 0) {
                return result;
            } else {
                if (a == 1) {
                    return b;
                } else {
                    if (b == 1) {
                        return a;
                    } else {
                        double atemp = a;
                        double btemp = b-1;
                        double resulttemp = result+a;
                        a = atemp;
                        b = btemp;
                        result = resulttemp;
                    }
                }
            }
        }
    }
    public int Sum( int n) {
        if (n == 0) {
            return 0;
        } else {
            return n+Sum(n-1);
        }
    }
    public int SumTail( int n,
                        int result) {
        while(true){
            if (n == 0) {
                return result;
            } else {
                int ntemp = n-1;
                int resulttemp = n+result;
                n = ntemp;
                result = resulttemp;
            }
        }
    }
}
```

Figure 16 - Java Code Output example with tail recursive elimination

7. Conclusion and Future Work

We have presented the work developed at NASA Langley Research Center and National Institute of Aerospace with Alwyn Goodloe as the NASA Supervisor from July 2011 to October 2011. The internship was very satisfactory both personally and academically. The work requested was finished on time and the problem described was solved. We executed many test cases not only to know if the new changes were working fine but also to ensure that the features developed before were not affected and still worked fine. The test cases were showed to the NASA staff and they approved the final result.

Future work has been identified to be investigated. One of the most important issues and with a lot of interest in the NASA staff is to transform regular recursive functions to tail recursive function. This is useful because then we could apply the solution offered in this work to many more PVS models use by NASA. So, we could transform any regular recursion to iteration.

Another desirable feature is that the user could choose which recursive definition must be transformed into iteration. Right now, the translator transforms all the tail recursive functions in the theory. Some labels could be available to the user to mark which recursive definition has to be translated into iteration.

This work is also an important help to the project described in [1]. In that project, the translator uses the information of PVS declarations to extract pre-conditions and pos-conditions and put them in the Java code to be verifiable. In the PVS recursive declarations the measure of a recursive function must be declared. In our work when we analyze some function to know if it is recursive, we are storing the measure information and it could be used to generate the annotated code.

8. References

1. Lensink, L., Muñoz, C., & Goodloe, A.: *From Verified Models to Verifiable Code*, June 2009
2. Owre, S., & Shankar, N.: *PVS System Guide*, November 2001
3. Filliâtre, J. C.: *Why: a multi-language multi-prover verification tool*, Research Report 1366, LRI, Université Paris Sud http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz
4. Chappel, Glenn G: *Recursion vs. Iteration, Eliminating Recursion*. 2009.
5. M. W. Bailey, N. C. Weston: *Performance benefits of tail recursion removal in procedural languages*, Technical Report TR-2001-2, Computer Science Department, Hamilton College, June 2001
6. Owre, S., & Shankar, N.: *PVS Language Reference*,Version 2.4, November 2001
7. The why verification tool. http://why.lri.fr/
8. Benander, A., Benander, B., & Sang, J.: *An empirical analysis of debugging performance - differences between iterative and recursive constructs*, Journal of Systems and Software, 2000.

9. Code

In this chapter, the developed code is presented. We are going to show only the relevant functions and not the entire code. The whole code is available from http://research.nianet.org/fm-at-nia/PERICO/

9.1 pvs2why-init.lisp

```
…
 (in-package :pvs)

(defvar *pvs2why-version* "")
(defvar *why-decls-trace* nil)
(defvar *why-types-trace* nil)
(defvar *pvs2why-trace* nil)
(defvar *pvs2why-unique-names* nil)
(defvar *why-krakatoa* nil)
(defvar *tail-recursive* nil)
(defvar *tail-recursive-flag* nil)
(defvar *function-definition* nil)


(let* ((pvs2why-dir (environment-variable "PVS2WHYPATH"))
       (filename   (environment-variable "PVS2WHYFILENAME"))
       (filedir    (environment-variable "PVS2WHYFILEDIR"))
       (pvsfile    (format nil "~a/~a.pvs" filedir filename))
       (packlist   (read-from-string (environment-variable "PVS2WHYPACK")))
       (debug      (read-from-string (environment-variable "PVS2WHYDEBUG")))
       (krakatoa   (read-from-string (environment-variable "PVS2WHYKRAKATOA")))
       (xml        (read-from-string (environment-variable "PVS2WHYXML")))
       (java       (read-from-string (environment-variable "PVS2WHYJAVA")))
       (c          (read-from-string (environment-variable "PVS2WHYC")))
       (attachment-file (environment-variable "PVS2WHYATTACHMENTS"))
       (*pvs2why-version* (environment-variable "PVS2WHYVERSIONMSG"))
       (*pvs2why-trace* debug)
       (*why-decls-trace* debug)
       (*why-types-trace* debug)
       (*why-krakatoa* krakatoa)
       (*noninteractive* t)
       (*tail-recursive-flag* (read-from-string (environment-variable
           "TAILRECURSIVEELIMINATION")))
       (current-prefix-arg t))
  (multiple-value-bind
    (val err)
    (ignore-errors
     (change-context (probe-file filedir))
     (dolist (pack packlist) (load-prelude-library pack))
     (typecheck-file filename nil nil nil t)
     (init-semantic-attachments pvs2why-dir "Prelude-Attachments.txt")
    (init-semantic-attachments filedir attachment-file)
    (let* ((theories (theories-in-file filename)))
      (format t "~%***~%*** Processing ~a via ~a~%***~%" pvsfile
              *pvs2why-version*)
      (dolist (theory theories)
        (when (or xml (and (not java) (not c)))
          (pvs2xml filename theory *pvs2why-version*)
          (format t "*** ~a (~a) --> ~a/~a.xml~%"
                  theory pvsfile filedir theory))
        (when c
          (pvs2c filename theory *pvs2why-version*)
          (format t "*** ~a (~a) --> ~a/~a.c~%"
                  theory pvsfile filedir theory))
        (when java
          (pvs2java filename theory *pvs2why-version*)
          (format t "*** ~a (~a) --> ~a/~a.java~%"
                  theory pvsfile filedir theory))))
    t)
   (when err
     (format t "~%*** ~a (~a)~%" err pvsfile)
     (bye -1)))
  (fresh-line)
  (bye))
```

9.2 pvs2why.lisp

Changes in: pvs2why-resolution-destructive

```
;; destructive variant. Main difference is the analysis of the *output-vars*
(defun pvs2why-resolution-destructive (op-decl formals body range-type)
  (when *pvs2why-trace*
    (format t "Function: pvs2why-resolution-destructive ~a ~{ ~a ~} ~a~%" op-decl formals range-
type))
  (let* ((*destructive?* t)
         (*output-vars* nil)
         (bind-ids (pvs2why-make-bindings formals nil))
         (declared-type range-type)
         ;   (dummy (format t "### Startnig with body"))
         (cl-type ;(pvs2why-type (type op-decl)))
         (let ((domain
                (loop for var in formals
                      collect
                      (if (assoc (declaration var)
                                 *output-vars* :key #'declaration)
                          (pvs2why-type (type var)) ; unique!
                          (pvs2why-type (type var)))))
               (range (pvs2why-type range-type)))
           (mk-why-function-type domain range)))
    (cl-body (if (not body)
                 nil ; abstract function
                 (pvs2why* body (pairlis formals bind-ids) nil declared-type)))
    ;       (dummy (format t "Done with body"))
    (hash-entry (gethash op-decl *why-destructive-hash*))
    (precondition (if *why-krakatoa*
                   (pvs2why-preconditions formals (pairlis formals bind-ids))
                   nil))
    (postcondition (if *why-krakatoa*
                    (pvs2why-postcondition (declared-type op-decl)) ; range-type)
                  nil))
  (cl-defn  (if (isrecursiveDefined op-decl)
              (mk-why-function-recursive (why-info-id hash-entry) bind-ids cl-body cl-type
precondition postcondition t (measure op-decl))
              (mk-why-function (why-info-id hash-entry) bind-ids cl-body cl-type precondition
postcondition)
              )
           )
  ;          (cl-defn  (mk-why-letrec (id op-decl) bind-ids cl-body cl-type))
  (old-output-vars (why-info-analysis hash-entry)))
    ;        (format t "~%Defining (destructively) ~a with ~%type ~a ~%as ~a" (id op-decl) cl-type
cl-defn)
    ;   (when *pvs2why-trace*
    ;      (format t "Defining XML: ~a~%" (why2String* cl-defn)))
    (setf (why-info-type hash-entry) cl-type
         (why-info-definition hash-entry) cl-defn
         (why-info-analysis hash-entry) *output-vars*)
    cl-defn
    ))
;    (unless (equalp old-output-vars *output-vars*)
;      (pvs2why-resolution-destructive op-decl formals body range-type))))

;; if the field "recursive-signature" exists, the function is recursive
;; but it could be done in other different ways.
(defun isrecursiveDefined (op-decl)
  (when t
     (format t "Function: isrecursiveDefined: ~a trans: ~a resultado: ~a recursive-signature: ~a
            ~%" op-decl (format nil "~a ~%" op-decl) (eq (search "#<def-decl" (format nil "~a ~%"
            op-decl)) 0) (slot-exists-p op-decl 'recursive-signature)))
;  (eq (search "#<def-decl" (format nil "~a ~%" op-decl)) 0)
   (slot-exists-p op-decl 'recursive-signature)
)
```

9.3 why.lisp

(defcl **why-function** (why-def)
```
  (parameters :type list) ; List of why-binding
  (return-type :type why-type)
  (type :type why-type)
  (body :type why-expr)
  (precondition :type why-expr)
  (postcondition :type why-expr)
  isRecursive
  (measure :type why-expr))
```

(defun **mk-why-function** (identifier parameters expr type precondition postcondition)

```
  (when *why-decls-trace*
    (format t "mk-why-function ~a ~a ~a ~%" identifier parameters expr type))
  (make-instance
   'why-function
   :why-identifier (mk-why-identifier identifier :context type)
   :parameters (list-bindings parameters (domain type))
   :return-type (range type)
   :type type
   :precondition precondition
   :postcondition postcondition
   :body expr
   :isRecursive nil
   ))
;
;** COMMENT-MCORDINI is the same that mk-why-function but for recursive functions
(defun **mk-why-function-recursive** (identifier parameters expr type precondition postcondition
            isRecursive measure)
  (when *why-decls-trace*
    (format t "mk-why-function-recursive ~a ~a ~a ~a isRecursive: ~a measure: ~a ~%" identifier
            parameters expr type isRecursive measure))
  (make-instance
   'why-function
   :why-identifier (mk-why-identifier identifier :context type)
   :parameters (list-bindings parameters (domain type))
   :return-type (range type)
   :type type
   :precondition precondition
   :postcondition postcondition
   :body expr
   :isRecursive isRecursive
   :measure measure
   ))
```

9.4 why2java.lisp

```
  (defun write-java-function (file def)
    (when *pvs2why-trace*
      (format t "Function: write-java-function ~%" ))
    (if (and (and (and *why-krakatoa*
                       (eq (identifier (return-type def)) 'int))
                  (body def))
             (not (parameters def)))
        (indent file (format nil "~%"))
        (progn

          (when (or (precondition def) (postcondition def))
            (indent file (format nil "/*@~%")))
          (when (precondition def)
            (indent file (format nil "  @ requires ~a;~%" (why2java-predicate-string* (precondition
              def)))))
          (when (postcondition def)
        (indent file (format nil "  @ ensures ~a;~%" (why2java-predicate-string* (postcondition
        def)))))
      (when (or (precondition def) (postcondition def))
        (indent file (format nil "  @*/~%")))))
(let ((header
       (if (and (and (and *why-krakatoa*
                     (eq (identifier (return-type def)) 'int))
                (body def))
           (not (parameters def)))
         (format nil "public static final ~a ~a"
                (why2java-type* (return-type def))
                (identifier def))
         (format nil "public ~:[abstract ~;~]~a ~a("
                                           (body def)
                                           (why2java-type* (return-type def))
                                           (identifier def)))))
      (indent file header)
      (when (parameters def)
        (open-block (length header))
        (why2java* file (car (parameters def)))
        (dolist (param (cdr (parameters def)))
          (format file ",~%")
          (indent file)
          (why2java* file param))
        (close-block))
      (cond ((body def)
             (if (and (and *why-krakatoa*
                          (eq (identifier (return-type def)) 'int))
                     (not (parameters def)))
               (format file " = ~a;"
                      (why2java-string* (body def)))
               (progn
                 (format file ") ")
               (block-java
                file
                (let* ((*function-definition* def))
                  (if (and *tail-recursive-flag* (isRecursive def))
                      (progn
                       (when *why-types-trace*
                         (format t "*function-definition*: ~a ~%" *function-definition*))
                        (setq *tail-recursive* nil)
                        (let*((body-string-output (with-output-to-string(tmpstream)
                                  (why2java* tmpstream  (body def))
                                )))
                          (if *tail-recursive*
                            (progn
                              ;if we are going to put some invariants to the loop, should be here
                              ;(when (measure def)
                                ;(indent file (format nil "/*@~%  @ maintaining ~a;~% @*/~%" (measure
        def))))
                              (indent file "while(true)")
                              (block-java
                               file
                              (identBlock file body-string-output)
                             )
                            )
                            (format file "~a~%"
                                body-string-output
                            )
                           )
                         )
                        )
                        (why2java* file  (body def));not recursive -> like always
                      )
                    )
                  )
                 )))
        (t (format file ");~%")))))

(defmethod identBlock (file stringToIdent)
  (with-input-from-string (tmps stringToIdent)
   (loop for line = (read-line tmps nil)
          while line do
             (format file "~a~a~%" (make-string 4 :initial-element #\Space) line)
      )
   )
)

(defun why2java-list-tail (l)
  (format nil "~{~a~#[~:;;~%~]~};~%"  l))

(defmethod  why2java* ((file stream) expr &optional noreturn)
  (when *pvs2why-trace*
    (format t "Function: why2java*-optional ~%" ))
  (let* ((dummy (format t "Enter lifted expr: ~a~%" (why2java-string* expr)))
         (lifted-expr (lift-let* expr))
         (dummy (format t "Leave lifted-expr : ~a~%" lifted-expr))
         (lexpr (car lifted-expr))
         (dummy (format t "Translated lifted-expr: ~a~%" (why2java-string* lexpr)))
         (seq (cdr lifted-expr))
         (dummy (format t "Prefix: ~a~%" seq)))
    (progn
      (when seq
        (why2java* file (if (eq (length seq) 1)
                        (car seq)
                        (mk-why-sequential-composition seq)) t ))
      (indent
       file
       (if noreturn
          (format nil "~a;~%" (why2java-string* lexpr))
          (progn(format t "this is the return ~a ~a ~a ~a ~a b~ab ~%"
                    lexpr
                    (identifier(why-identifier *function-definition*))
                    (when (why-function-application? lexpr) (identifier (operator lexpr)))
                    (when (why-function-application? lexpr) (length(parameters *function-
            definition*)))(when (why-function-application? lexpr)(length(arguments lexpr)))
```

```
                (when (why-function-application? lexpr)(progn (setq xuno (identifier(why-
            identifier *function-definition*)))
                                                       (setq xdos (identifier(operator lexpr)))
                                                       (eql xuno xdos)
                                                    ))
               )
          ;;WE HAVE TO ASK IF ~a IS RECURSIVE --> tail recursive
           (if (and *tail-recursive-flag* (isRecursiveCall lexpr))
             (progn (setq *tail-recursive* t) (why2java-list-tail(delete-recursive-call lexpr)))
             (format nil "return ~a;~%"(why2java-string* lexpr))

           )
          )))))) ;lexpr ;)

;;check if the call of a function in lexpr is recursive
(defmethod isRecursiveCall (lexpr)
  (and
     (why-function-application? lexpr)
     (and (equal (identifier(why-identifier *function-definition*))
               (identifier(operator lexpr))
        )
        (equal (length(parameters *function-definition*))
               (length(arguments lexpr)))
        )
  )
)

;;replace the recursive call expr with the new values of the variables
(defmethod delete-recursive-call (expr)
  (append
   (let* ((loopvar (loop for x in (parameters *function-definition*)
                     for y in (arguments expr)
                     collect (list (why2java-type* (type x)) (identifier (why-identifier x)) (why2java-
            string* y))
                     )
                 )
         (carloopvar (car loopvar))
         (cdrloopvar (cdr loopvar))
         )
     (append
      (list (format nil "~a ~atemp = ~a" (car carloopvar) (cadr carloopvar) (caddr carloopvar)))
      (loop for z in
        cdrloopvar
        collect (format nil "~a~a ~atemp = ~a" (make-string (get-indent) :initial-element #\Space)
           (car z) (cadr z) (caddr z))
           )
      )
   )
   #|
   (loop for z in
        (loop for x in (parameters *function-definition*)
              for y in (arguments expr)
              collect (list (identifier (why-identifier (type x))) (identifier (why-identifier x)) (why2java-
             string* y))
              )
        collect (format nil "~a~a ~atemp = ~a" (make-string (get-indent) :initial-element #\Space)
           (car z) (cadr z) (caddr z))
   )
   |#
   (loop for x in (parameters *function-definition*)
        collect (format nil "~a~a = ~atemp" (make-string (get-indent) :initial-element
           #\Space)(identifier (why-identifier x)) (identifier (why-identifier x)))
        )
   )
  )
```