Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 184 (2007) 63–80
www.elsevier.com/locate/entcs

Mechanised Refinement of Procedures

Manuela Xavier[1]

Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Ana Cavalcanti[2]

Department of Computing Science
University of York
York, United Kingdom

Abstract

Refine is a tool that supports the application of Morgan's refinement calculus. It was designed to support teaching and use by beginners; it is already in use. We describe here the extension of Refine to support the development of (possibly recursive) procedures in the algebraic style of the refinement calculus already adopted by Refine.

Keywords: Formal methods, refinement calculus, refinement tool.

1 Introduction

Refinement is a concept that supports the correct development of computer systems. Refinement techniques support incremental development: a specification is refined in a sequence of steps in a context that uses an uniform notation for specification and programming. A refinement relation, which models preservation of correctness, is kept between successive steps of refinement, guaranteeing that the final implementation satisfies the original specification.

Back [1], Morris [15] and Morgan [16] worked independently on refinement calculi. The refinement calculus presented here is based on Morgan's work; it is attractive, from a practical point of view, because all refinement steps are applications of algebraic laws.

[1] Supported by CNPq. Email: max@cin.ufpe.br.
[2] Partially supported by QinetiQ and the Royal Society, England. Email: Ana.Cavalcanti@cs.york.ac.uk.

doi:10.1016/j.entcs.2007.03.015

Using Morgan's refinement calculus, we can write software in a precise and consistent way. However, its application to problems of any size requires a lot of tedious manipulations of predicates, that can easily generate errors. This situation suggests the need to develop tools that support the refinement calculus. A tool that supports law applications allows the users concentrate in the important elements of a development, avoiding the monotonous work and possible errors.

Therefore, we developed Refine, an educational tool for refinement. The initial version of the tool was presented in [5], but it had few facilities of development management. Moreover, the supported language did not include procedures and recursions. In this paper, we present a new version of Refine that addresses these issues.

Morgan's refinement calculus presented a problem in its approach to procedures and parameters [7]. Therefore, a new set of refinement laws was presented in [6]. These laws address the existing difficulties, and allow an algebraic approach to the development of recursive programs. They are the basis for the implementation of Refine presented here. Most of the refinement tools cited in literature [4,12,13,23,2,21] do not make possible the development of procedures and recursion, or do not have a friendly interface to be used by beginners. Moreover, none of them considers the approach presented in [6].

In Section 2 we give a brief explanation of refinement calculus, and of the procedure and recursion laws that we consider. Section 3 presents Refine, and Section 4 describes the results of the integration of the procedure and recursion laws. Finally, in Section 5 we summarise related and future work.

2 Refinement Calculus

The refinement calculus is composed of an unified language of specification, design, and implementation, and of refinement laws. It is based on a refinement relation between programs (specifications, designs, or simply programs); the set of laws determines how refinements can be generated in an algebraic way. The development of programs, using this technique, consists of law applications over and over again, until an initial specification is transformed into an executable program.

In the refinement calculus, if a program p_2 is better than a program p_1, we write $p_1 \sqsubseteq p_2$. The relation $\sqsubseteq$ is called refinement: we say that p_2 refines p_1. The notion of improvement is based on the user point of view, and is formalised using a weakest precondition semantics.

For each step in the refinement process, the current program, or some subprogram of the current program, is transformed by the application of a refinement law. For some transformations, proof obligations are generated. If the proof obligation can be discharged, the correctness of the generated program is guaranteed.

A specification has the form $w : [pre, post]$. Its precondition (pre) describes the initial state in which execution of the program is well behaved. The postcondition ($post$) describes the final states that can be obtained if the precondition is satisfied. The frame (w) lists the variables whose values can change. The language used to

define the preconditions and postconditions is the predicate calculus. If the initial state satisfies the precondition, then the variables listed in the frame can be modified in such a way that the final state satisfies the postcondition. If the initial state does not satisfy the precondition, the result cannot be predicted. A precondition *true* can be omitted.

In a postcondition, a 0-subscripted variable can be used to represent the initial value of the corresponding variable. As an example, we have the specification $x : [0 \leq x, x^2 = x_0]$. The precondition indicates that the program has a well-defined result when the value of x is greater than or equal to zero. The postcondition indicates that at the end of the program execution, from a state where the precondition is satisfied, we have in the variable x the square root of its initial value.

Besides the specification statement, the language of Morgan's calculus includes all the constructors of Dijkstra's language [9]. Block constructs are also available to declare local variables, and logical constants.

Variable blocks have the form $[\![\mathbf{var}\ x : T \bullet\ p]\!]$, where x is the name of a new variable declared to be of type T, with a scope restricted to p. Similarly, logical constants c are declared in blocks $[\![\mathbf{con}\ c\ \bullet\ p]\!]$. A logical constant is a name that can be used to mention a value of interest during a development. Differently of a variable, a logical constant is not code, and then, in some moment during the refinement it must be removed.

In [6], a procedure block takes the form $[\![\mathbf{proc}\ name \mathrel{\widehat{=}} body\ \bullet\ main]\!]$. It introduces the procedure *name*, and a program fragment: the procedure *body*, which may declare parameters. Finally, we have the *main* program, which can call the procedure.

Parameters can be passed by value, by result, or by value-result. Procedures with parameters have as body a parameterized command, that is, the declaration of the parameters in a procedure is associated with the procedure body instead of with its name.

Instead of using or acting on specific variables, a parameterized command is generic: they need to be applied to arguments. The behavior of the resulting program depends on these arguments. An example is this program that increases the value of a variable x using a parameterized procedure Inc: $[\![\mathbf{proc}\ Inc \mathrel{\widehat{=}} (\mathbf{val\_res}\ n : \mathbb{N} \bullet n := n + 1) \bullet Inc(x)]\!]$. The procedure body is a parameterized command that defines a parameter n, passed by value-result, whose value is increased by 1. The main program is a call to the procedure, passing as parameter the variable x whose value we want to increase.

Procedure blocks can also declare recursive procedures, but in their development we use variant blocks. Besides the procedure and the main program, a variant block declares a variant expression e of name v. It has the form $[\![\mathbf{proc}\ name \mathrel{\widehat{=}} body\ \mathbf{variant}\ v\ \mathbf{is}\ e \bullet main]\!]$. The variant is part of the argument for the termination of the procedure.

Example 2.1 As an example, we consider a program that computes the square root of the inverse of a positive number. We adopt $x : [0 < x, x^2 = 1/x_0]$ as the initial specification of this program. Each law used in the development can be found

in Appendix A.

At first, we introduce a block that defines a procedure that computes the square root of a number. The name of the procedure is *Sqrts*; its parameters are a, passed by value, and b, passed by result; the procedure body is the specification $b : [0 < a, b^2 = a_0]$. The main program is the initial specification. The procedure block results from the application of the *Parameterized Procedure Introduction* law.

$$\begin{array}{l}[\![\ \textbf{proc}\ Sqrts \mathrel{\widehat{=}} (\textbf{res}\ b; \textbf{val}\ a \bullet b : [0 < a, b^2 = a_0])\bullet \\ \quad x : [0 < x, x^2 = 1/x_0] \qquad (1) \\ ]\!]\end{array}$$

Continuing the program development, we split the main program into the sequence of two specifications: the first computes the inverse of x, and the second computes the square root of the resulting value of x. For this, we apply to (1) the *Sequential Composition with Constants* law. We get the following result.

$$\begin{array}{l}[\![\textbf{con}\ X : \mathbb{R}\bullet \\ \quad x : [0 < x, 0 < x \wedge x = 1/x_0]; \qquad (2) \\ \quad x : [0 < x \wedge x = 1/X, x^2 = 1/X] \qquad (3) \\ ]\!]\end{array}$$

A logical constant X is declared and (1) is split with basis on an intermediate state definition $0 < x \wedge x^2 = 1/x_0$; it is used as the postcondition of the first resulting specification, and the precondition of the second one. Moreover, all references to the initial variables in the second specification are substituted by the declared constant X.

Now, we can refine the specification (2) to $x := 1/x$ using the *Assignment* law. The proof obligation is $x = x_0 \wedge 0 < x \Rightarrow 1/x = 1/x_0 \wedge 0 < 1/x$. This follows from properties of equality and basic arithmetic; so we can conclude that the law application generates a refinement of (2).

Next, we rewrite (3) to allow its refinement to a call to the procedure *Sqrts*. For this, we apply the *Strengthen Postcondition* law, and afterwards, the *Weaken Precondition* law. We obtain as result: $x : [0 < x, x^2 = x_0]$. The proof obligations are $0 < x_0 \wedge x_0 = 1/X \wedge x^2 = x_0 \Rightarrow x^2 = 1/X$, which is follows by a property of equality, and $0 < x \wedge x = 1/X \Rightarrow 0 < x$, which holds because the consequent is part of the antecedent of the implication.

We continue the development of this program in the Section 4, where we discuss the application of the laws that handle parameterized programs and procedure calls. In the following section, we illustrate the use of Refine using the part of the example presented so far.

3 Refine

Refine is based on the Windows standard; its interface is composed of four windows presented in Figure 1. The refinement window, which presents the program development step by step; the proof-obligations window, which lists the proof obligations generated by all law applications; the laws window, which lists the refinement laws

supported by the tool; and, finally, the code window, which presents the currently developed program. Moreover, Refine has menus and buttons that allow the access to its main functionalities.

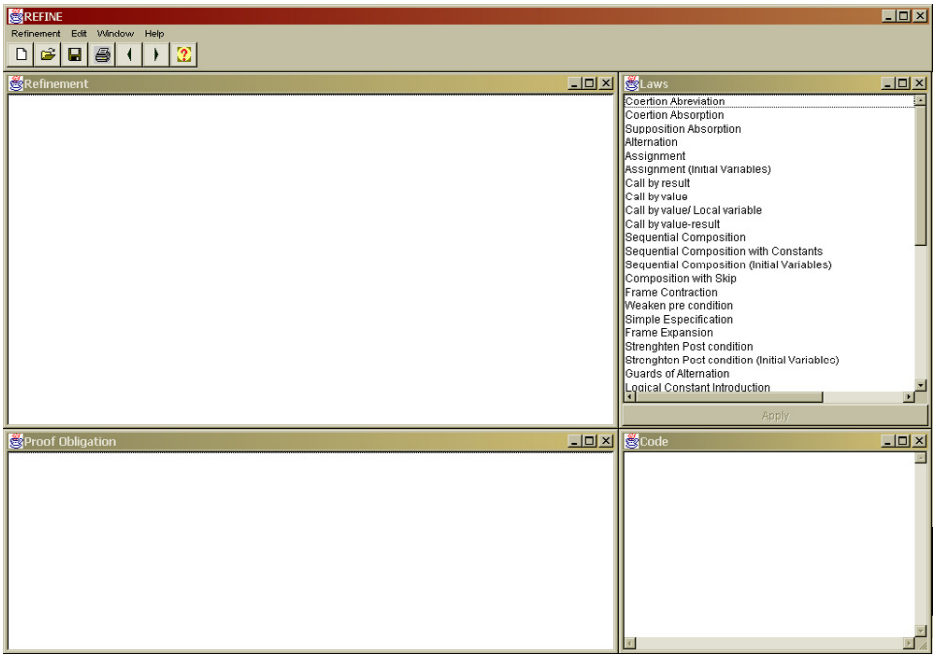

Fig. 1. Refine's windows

Refinement Window. The refinement window shows the steps of the development: transformations to an initial program. Each transformation is justified by the application of a law.

Starting a program development, we type the specification; the window that is shown when we press the start new development button is presented in Figure 2. We have to type the initial specification in accordance with the format supported by Refine; we type `x:[0<x,x**2=1/x0]` for our example in Section 2. There is a mapping of the symbols of the predicate calculus to ASCII symbols that are supported by Refine. The user can type these symbols or use a symbol keyboard provided by Refine.

If the specification is well-formed, the tool displays it in the refinement window. At the end of the program development, the refinement window shows the complete refinement of the program (see Figure 3). In this way, we have the same that we would have in paper, with the additional advantage of the management facilities that we explain later.

Refinement-Laws Window. The refinement-laws window shows the list of all the laws supported by the tool: all of the Morgan's calculus plus the laws proposed in [6]. Through this window, we can select the law that we want to apply to a specific program by clicking the left-button of the mouse on the law name.

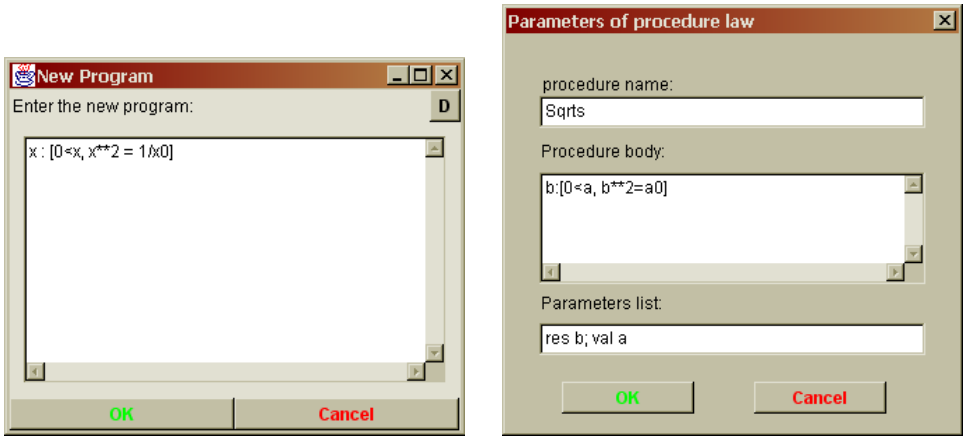

Fig. 2. New Program and Parameter Windows

Some laws, to be applied, need some parameters supplied by the user. The entrance of these parameters is made through a parameters window, which is specific for each type of law.

In our example, we must select the initial specification in the refinement window, choose the *Parameterized Procedure Introduction* law, and press the button APPLY in the refinement-laws window. We are asked to type the parameters of this law application; the input window is presented in Figure 2. The parameters are the name of the procedure, its body, and its parameters, which are all checked for syntactic correctness. After the application of the law, the refinement, proof obligations, and code windows are updated.

We can use the same idea to continue the development of our example in the tool. The final result in the refinement window is presented in Figure 3. As we can observe in this figure, Refine uses the symbol [= to represent a refinement relation ($\sqsubseteq$).

Proof-Obligations Window. Our example of refinement generates some proof obligations. In Refine, these are displayed in the proof-obligations window, as presented in Figure 3. In this version of Refine, the user is responsible for verifying if the generated proof obligations are true. If not, the tool assists in the correction of the program development.

Code Window. The code window displays the collected code that results from the current development. In this way, when we enter an initial specification, this is the program that is displayed in the code window. When refinement laws are applied, the collected code is displayed in the code window, as presented in Figure 4.

Refinement Management. It is possible to save developments into a file and to recover the development saved in a file. When a development is saved, information about the proof obligations and the generated code is saved too. Therefore, when the development is recovered, the additional information is readily available as well. The undo and redo operations help in the correction of

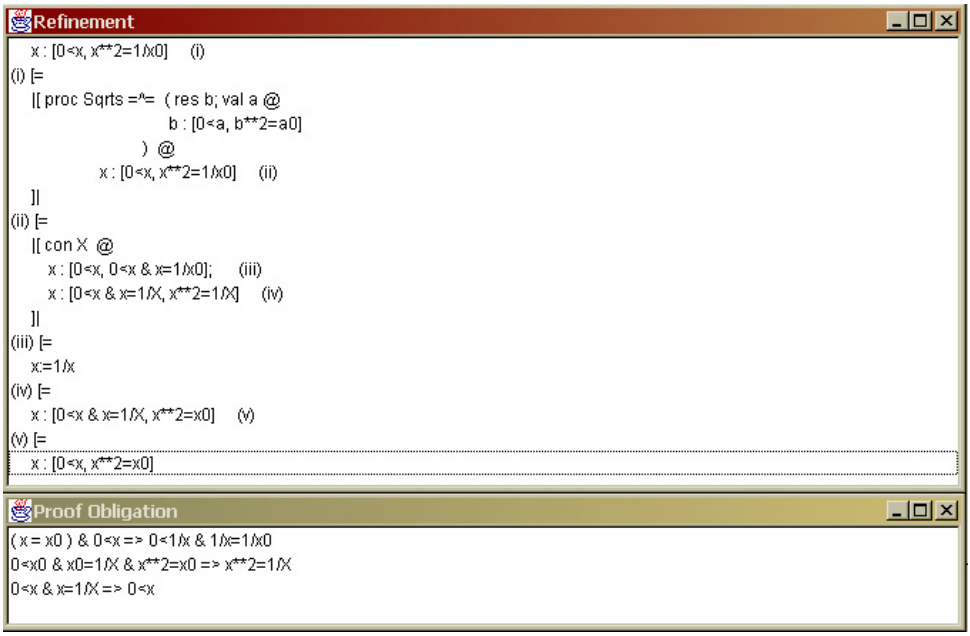

Fig. 3. Refinement and Proof Obligations Windows

developments.

Refine associates the proof obligations and the laws that generated them. To find out which law application generated a specific proof obligation, we must select the proof obligation in the proof-obligations window. The tool will automatically select the part of the refinement, in the refinement window, that generated the proof obligation selected.

It is possible to insert comments during the program development through the refinement window. A comment is associated with a specific part of the development. By clicking with the right button on a part of the development, and selecting the insert comments option, we get a window (Figure 4) where we can include or edit comments. The comments are usually hidden; they are displayed only when the user requests. It is possible to save comments when the development is saved; and recover the comments, when the development is openned.

We can print the program development, the proof obligations generated, the

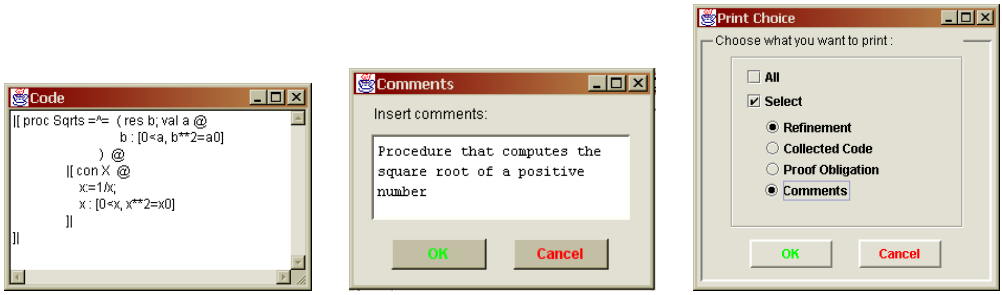

Fig. 4. Code, Comment and Print Windows

final code, and the comments associated with the development. It is possible to select what we want to print through a print window (Figure 4). The refinement steps are printed with the names of the applied laws. The lines are numbered, and each comment and proof obligation refers to the line of the program that generated it.

The main menu also offers a help for users who are not familiar with the tool or the refinement calculus. A documentation of the refinement laws supported is provided.

4 Refine: Procedures and Recursion

To support the application of the laws related to procedures, we implemented a facility for code collection. As we explain in detail later on, some of these laws are applied to parts of the developed program that are not necessarily available in the development window. By collecting the code, we record an updated view of the developed program obtained so far, and we can consider the refinement of its constituent commands. To illustrate this functionality and others, we will continue the development of the square root procedure (Example 4.1) and start the development of a recursive program that evaluates the factorial of any number (Example 4.2).

Example 4.1 To finish our development, we must refine: `x:[0<x,x**2=x0]` (see Figure 3) to introduce a procedure call, by converting that specification to a program like the procedure body. At first, we apply the *Call by Result* law; Refine shows a parameter window where we type the parameters that we want to declare ($b : \mathbb{R}$) and the arguments we want to pass (x). This results in the following fragment of program.

```
(res b @ b:[0<x, b**2=x0])(x)
```

To introduce a call by value, we refine the specification in the body of the parameterized command above so that we can apply the *Call by Value* law. First, we apply the *Strengthen Postcondition* law (using as argument the predicate `(b**2=a0)[a,a0\b,b0]`), and afterwards, the *Weaken Precondition* law (using as argument the predicate `(0<a)[a\x]`) in order to obtain a specification which matches the pattern required by the *Call by Value* law. We obtain the following result.

```
b:[(0<a)[a\x], (b**2=a0)[a0\x0]]
```

The application of the *Call by Value* law does not require parameters and we obtain the following result.

```
(val a @ b:[0<a, b**2=a0])(x)
```

Now we need to collect the code, because we want to join the two parameter declarations to get a single parameterized command. This is achieved by applying a law to the parameterized command that declares `b` and includes in its body the parameterized command that declares `a`. The result of the code collection is presented in Figure 5.

At this stage, we can remove the constant block, since the declared constant occurs nowhere in the program. We apply the *Logical Constant Removal* law, and as result we have the program below.

```
x :=1/x;
(res b @ (val a @ b:[0<a, b**2=a0])(x))(x)
```

The parameterized command above is almost equal to the body of the procedure `Sqrts`. By applying the *Multiple Parameters* law, we obtain the command: `(res b; val a @ b:[0<a,b**2=a0])(x,x)`, as main program of the procedure block `Sqrts`.

Again, we can collect the code in order to get the whole procedure block. This gives us the following procedure block in the refinement window.

```
|[proc Sqrts =^= (res b; val a @ b:[0<a, b**2=a0]) @
      x := 1/x;
      (res b; val a @ b:[0<a, b**2=a0])(x,x)
]|
```

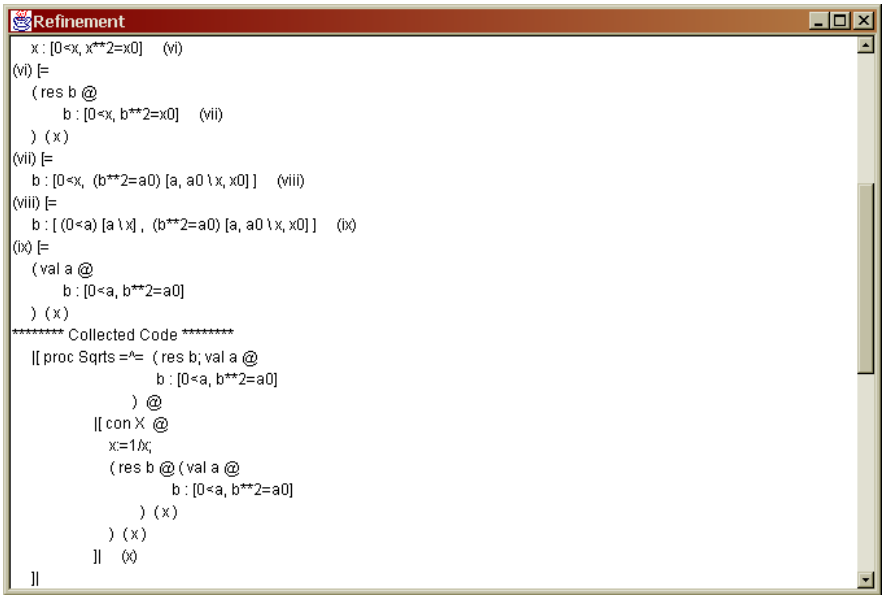

Fig. 5. Refinement Window after the Code Collection

Then, we apply the *Procedure Call Introduction with Parameters* law to the whole procedure block. It replaces the occurrences of the procedure body in the main program by procedure calls; this is done automatically by the tool, and involves the following steps:

(i) Verification if the selected program is a procedure block;

(ii) Identification of all occurrences of the parameterized command applications in the main program;

(iii) For each parameterized command application found:

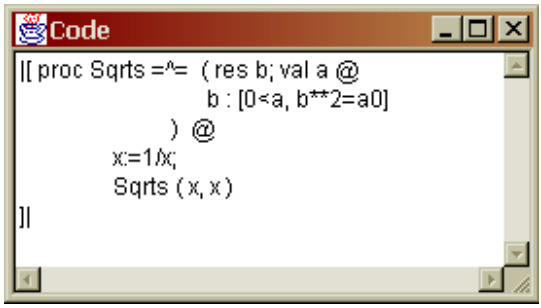

Fig. 6. Generated Code

(a) Verification of the procedure name scope.
We need to check if the parameterized command to be substituted by the procedure call is not inside another block (of procedure, variant, variables or constants) that declares again the procedure name. For example, if we try to apply the *Procedure Call Introduction with Parameters* law to the outer procedure block of the program below, no procedure calls will be introduced.

$$
\begin{array}{l}
[\![\ \textbf{proc}\ Sqrts \mathrel{\widehat=} (\textbf{res}\ b; \textbf{val}\ a \bullet \\
\qquad\qquad\qquad\qquad b : [0 < a, b^2 = a_0] \\
\qquad\qquad\qquad\qquad) \bullet \\
\quad x := 1/x; \\
\quad [\![\ \textbf{proc}\ Sqrts \mathrel{\widehat=}\ x : [x^2 = x_0] \bullet \\
\qquad\quad (\textbf{res}\ b; \textbf{val}\ a \bullet\ b : [0 < a, b^2 = a_0])(x, x) \\
\quad]\!] \\
]\!]
\end{array}
$$

The parameterized command application in the main program matches the body of the outer $Sqrts$, but it is hidden by the second declaration of $Sqrts$.

(b) Verification of the variables scope.
This is an issue in the presence of variable redeclarations. For example, it is not possible to introduce a call to Inc in the program below, because its global variable x is being redeclared. The assignment to x inside the variable block updates the local x, and a call to Inc at that point would update the global x. In such situations, Refine does not perform any substitution.

$$[\![\ \textbf{proc}\ Inc \mathrel{\widehat=} x := x + 1 \bullet [\![\ \textbf{var}\ x : \mathbb{Z} \bullet x := x + 1\ ]\!]]\!]$$

In our simple example, as our procedure block does not have scope problems, we can successfully introduce the procedure calls, and the generated code is presented in Figure 6.

Example 4.2 Now we will develop a program that computes the factorial of a number. The main goal in this example is to show how we can refine a recursive program using the tool. The initial specification is: `f:[f=fat(n)]`.
As we want to develop a recursive procedure for this program, the first step is to apply the *Variant Introduction with Parameters* law in the initial specification. This law application requires some parameters: the procedure name (`Fact`), the

specification of the procedure body (`f:[f=fat(x)]`), the variant name (`V`), the variant expression (`x`), and the procedure parameters (`val x`). The result of this law application is presented below. Refine includes automatically the predicate *variant name* = *expression variant* (`V=x`) in the precondition of the procedure body specification.

```
|[proc Fact =^= (val x @
                        f:[V=x, f=fat(x)]
                    ) variant V is x @
      f:[f=fat(n)]
  ]|
```

The first step of the development is to transform the main program of the block into a program similar to the procedure body, in order to replace it with a procedure call. For this, we first apply the *Strengthen Postcondition* law (using the predicate `(f=fat(x))[x,x0\n,n0]` as argument). This give us the following specification.

```
f:[(f=fat(x))[x,x0\n,n0]]
```

Now, we can apply the *Call by Value* law, and as result we have the following parameterized command application.

```
(val x @ f:[f=fat(x)])(n)
```

Collecting the code, we observe that the main program is similar to the program in the procedure body (see Figure 7). The variant block is very similar to a standard procedure block; the presence of the variant is relevant only for the refinement of procedure body, which is discussed in the sequel.

At this stage, we can apply the *Procedure Call Introduction in the Main Program of a Variant Block (with Parameters)* law, which results in the replacement of the main program with a procedure call. Verifications related to the scope of names in procedure blocks are necessary here in the same way as they were necessary when

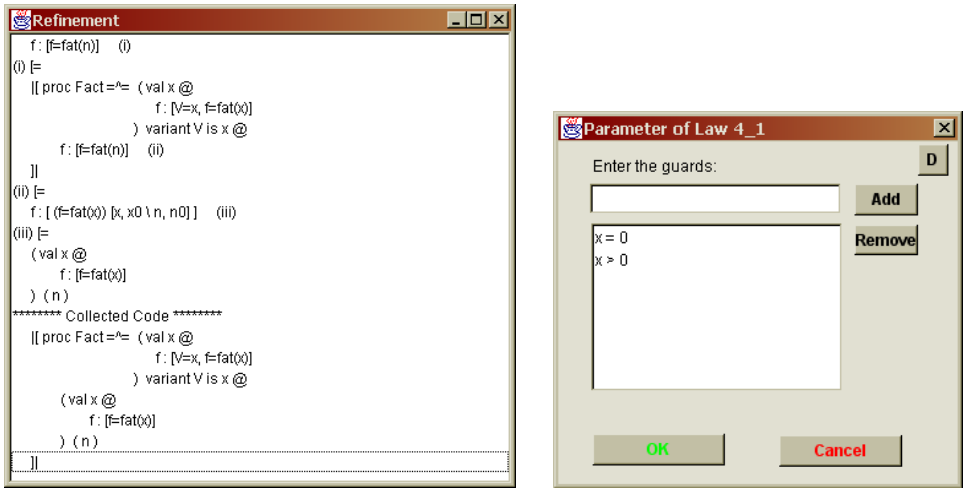

Fig. 7. Refinement Window after Code Collection and Parameter Window of Alternation Law

the *Procedure Call Introduction (with Parameters)* law was applied in the previous example. Since in our example the variant block does not have problems related to scope, the law application give us the following result.

```
|[proc Fact =^= (val x @
                        f:[V=x, f=fat(x)]
                    ) variant V is x @
         Fact(n)
]|
```

The second step of the development is to refine the procedure body to obtain a program that computes the factorial in a recursive way. For this, we split the specification of the procedure body into two development flows: the first concludes the execution of the recursion; and the second one stores, in a recursive way, the factorial calculation. For this, we apply the *Alternation* law. In Refine, this law application requires us to give as argument the guards `x=0` and `x>0`, which we provide through the parameter window presented in Figure 7. The result is presented below.

```
if x=0 -> f:[V=x & x=0, f=fat(x)]
[] x>0 -> f:[V=x & x>0, f=fat(x)]
fi
```

In the refinement of first specification, we apply the *Assignment* law to implement the base case of the recursion; the resulting program is `f:=1`.

To develop the factorial calculation for numbers greater that 0, we apply the *Next Assignment* law to divide the second specification above into two development flows again: the first will be implemented with a recursive call with a decreased value of x as argument; the second stores the temporary value of the factorial. The result is presented below.

```
f:[V=x & x>0, f*x = fat(x)];
f:=f*x
```

Analyzing the specification above, we can observe that it requires that the final value of `f` multiplied by `x` is the factorial of `x`. So, the value of `f` has to be the factorial of `x - 1`, which can be calculated with a recursive call.

The introduction of recursive calls is based on occurrences of programs that are similar to the initial specification of the procedure, but that decrease the variant. Therefore, recursive calls that lead to non-termination cannot be introduced.

We transform the specification above into a program that is similar to the initial specification of `Fact`, with the variant being decreased. Then, we apply the *Strengthen Postcondition* and *Weaken Precondition* laws, using the predicates `(f=fat(x))[x,x0\x-1,x0-1]` and `(0<=x & x<V)[x\x-1]` as arguments, respectively. This results in the following specification.

```
f:[(0<=x & x<V)[x\x-1], (f=fat(x))[x,x0\x-1,x0-1]]
```

The specification above has the appropriate format for the application of the *Call by Value* law. As a result, we obtain the following code fragment.

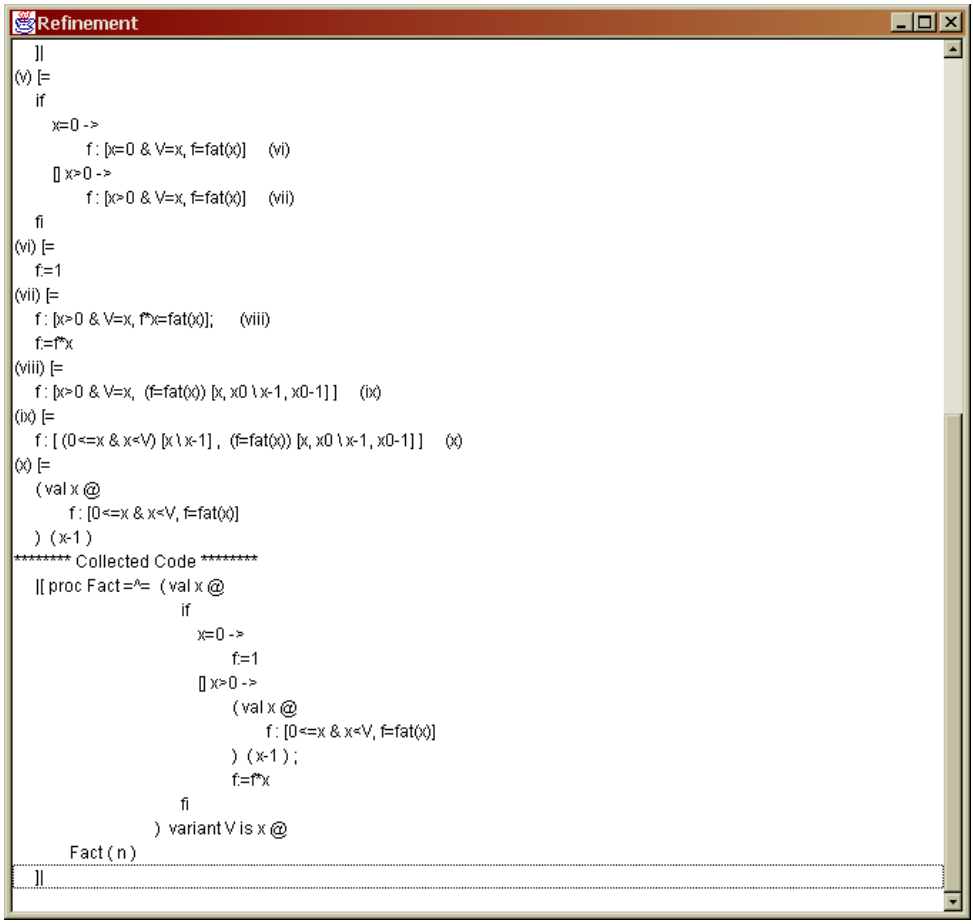

Fig. 8. Refinement Window after Code Collection

```
(val x @ f:[0<=x & x<V, f=fat(x)])(x-1)
```

Collecting the code again, we can observe the current state of the program, as presented in Figure 8. Now, we can select the whole variant block and apply the *Recursive Call Introduction (with Parameters)* law. To execute this law, Refine carries out the following steps:

(i) It verifies if the selected program is a variant block;

(ii) It verifies if the procedure body is a parameterized command;

(iii) It extracts from the procedure body all occurences of parameterized command applications;

(iv) For each parameterized command application found:
   (a) It verifies if its body is a specification statement;
   (b) It verifies if the precondition is the predicate:
   $$0 \leq (\textit{variant expression}) < (\textit{variant name})$$
   (c) It verifies the procedure name scope and the variables scope (as explained in the Example 4.1).

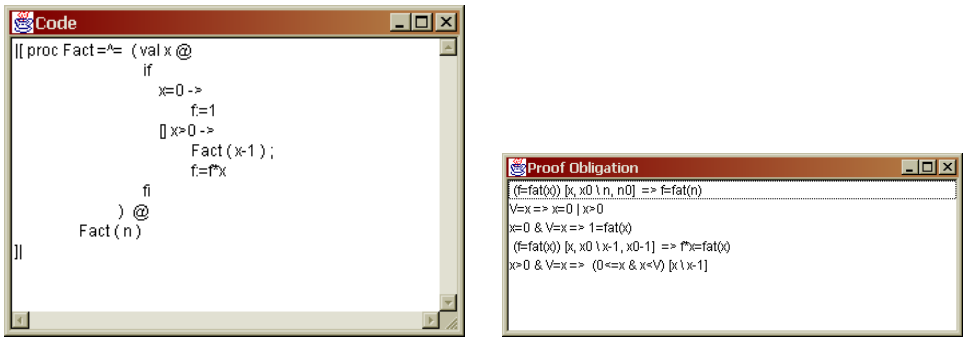

Fig. 9. Factorial Code and Proof Obligations

(d) If there is no problem, Refine searches in the development of the variant block for an occurrence of a specification in the body of the procedure that is exactly like that in the body of the parameterized command, except that the precondition $0 \leq (\textit{variant expression}) < (\textit{variant name})$ is replaced by $(\textit{variant name}) = (\textit{variant expression})$. The goal is to determine if, at some point of the development, this has been the specification of the procedure. If this is true, its refinement has lead to a procedure body that includes a copy of itself, but in a context in which the variant has been decreased.

If we apply the *Recursive Call Introduction (with Parameters)* law to the variant block in the collected code for our example, we successfully introduce a recursive call. In the procedure body, there is an occurrence of the parameterized command `(val x @ f:[0<=x&x<V, f=fat(x)])`, as required; actually this is the initial specification of `Fact`. The result of the application concludes the program development. Figure 9 presents the code and the proof obligations generated.

Checking the development history is a requirement that is peculiar to the law for introduction of recursive calls. Formally, it includes a proviso that requires us to prove that the obtained specification refines the initial specification of the recursive procedure; see Appendix A. In practice, however, this is a consequence of the development of the body of recursive procedure itself. Refine discharges this proof obligation automatically.

5 Conclusions

In this paper, we presented a tool that supports the use of the refinement calculus. Refine includes facilities to manage developments and, most importantly, supports the use of (possibly recursive) procedures in accordance with [6]. As we saw, Refine allows and helps the process of development navigation; moreover, it automates the recursion law applications, what demands a complete knowledge of the development description.

Several existing tools provide support for the refinement calculus. The Proxac system [21] is a transformation editor that supports the application of a sequence of transformation steps based on algebraic rules of one or more theories. The re-

finement calculator presented in [2] consists of a graphical user interface that uses a theorem proving system, HOL [11]. The works described in [13,4] use the HOL System to formalise a refinement support system too.

Some tools [14,22] support the use of procedures in the context of a refinement calculus. However, the calculational approach adopted in Morgan's Refinement Calculus [16], in which parameters and procedures are introduced and treated independently, is not supported by these tools. Furthermore, it is not possible to deal with procedure and variant blocks using the tools presented in [3,25,24].

Refine has been used successfully in teaching for almost four years and it has already proved to be useful as an educational tool. It was developed using Java, and amounts to about 45000 lines of code, in 203 classes. Details of Refine are available in its site [26], where we can also find UML documentation of the design, a tutorial, and development examples.

Refine was used as the starting to the development of another tool that supports a refinement calculus for Z based on Morgan's calculus [10]. In that work, support for procedures was used to in laws that implement operations defined using the Z promotion technique.

Besides, Refine was also extended with a tactic tool, called Gabriel [19,20]. It allows the definition of refinement tactics which document routine law applications, as for example, tactics for iteration developments, and for introduction of procedure calls. Using Gabriel, these tactics can be used as ordinary laws; an extra window lists the available tactics, and provides facilities to define and edit new tactics.

The tools developed from Refine, as cited above, have contributed sufficiently for its validation. Besides, the use in teaching has increased the tests and improved its benefits. Now, we believe that Refine is a robust tool.

We plan to integrate a theorem prover with Refine in order to mechanize the process of verification of proof obligations. We also plan to adapt Refine to support yet another formalism: *Circus* [8], which combines Z and CSP.

References

[1] R.J.R. Back. Correctness Preserving Program Refinements: Proof Theory and Applications. *Tract 131, Mathematisch Centrum*, Amsterdam, 1980.

[2] M.J. Butler, T. Langbacka. Program Derivation Using the Refinement Calculator. In von Wright, et.al. (eds). *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of Lecture Notes in Computer Science, Turku, Finalnd, August 1996. Springer-Verlag. pages 93-108.

[3] R. J. R. Back and J. von Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247-274, 1990.

[4] D. A. Carrington, I; J. Hayes, R. Nickson, G. Watson, J. Welsh: A Program Refinement Tool. *Formal Aspects of Computing* 10(2): 97-124 (1998)

[5] S. L. Coutinho, T. P. C. Reis, and A. L. C. Cavalcanti. A Tool for Teaching Refinement. In *13th Brazilian Symposium on Software Engineering*, pages 61 - 64, 1999. Tool Session. In Portuguese.

[6] A. L. C. Cavalcanti, A. C. A. Sampaio and J. C. P. Woodcock. Procedures and Recursion in the Refinement Calculus. *Journal of the Brazilian Computer Society*, 5(1): 1 - 15, 1998.

[7] A. L. C. Cavalcanti, A. C. A. Sampaio and J. C. P. Woodcock. An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus. *Science of Computer Programminh*, 33(1): 87 - 96, 1999.

[8] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in Circus. In L Eriksson and PA Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451-470. Springer-Verlag, 2002.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prantice Hall, 1976.

[10] A. F. Freitas, C. M. P. Nascimento, and A. L.C.Cavalcanti. A Refinement Tool for Z. In J. S. Dong and J. C. P. Woodcock, editors, *Formal Methods and Software Engineering: 5th International Conference on Formal Engineering Methods, ICFEM 2003*, volume 2885 of Lecture Notes in Computer Science, pages 396 - 415. Springer-Verlag, January 2003.

[11] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[12] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272 - 297. Springer-Verlag, 1992.

[13] J. Grundy. A Window Inference Tool for Refinement. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshop in Computing, pages 230 - 254. Spronger-Verlag, 1992.

[14] L. Laibinis. *Mechanised Formal Reasoning about Modular Programs*. PhD thesis, Turku Centre for Computer Science, 2000.

[15] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3): 298 - 306, 1987.

[16] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[17] M. V. M. Oliveira. ArcAngel: a Tactic Language for Refinement and its Tool Support. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Brazil, http://www.cin.ufpe.br/~mvmo, December 2002.

[18] M. V. M. Oliveira and A. L. C. Cavalcanti. Tactics of refinement. In *14th Brazilian Symposium on Software Engineering*, pages 117 - 132, October 2000.

[19] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a Tactic Language For Refinement. *Formal Aspects of Computing*, 15(1):28 - 47, 2003.

[20] M. V. M. Oliveira, M. A. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In J. R. Cuellar and Z. Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310 - 319. IEEE Computer Society Press, September 2004.

[21] Jan L. A. van de Snepscheut. Mechanized support for stepwise refinement. In Jrg Gutknecht, editor, *Programming Languages and System Architecures*, volume 782 de *Lecture Notes in Computer Science*, pages 35–48. Spinger-Verlag, 1994.

[22] M. Staples. *A Mechanised Theory of Refinement*. PhD thesis, Cambridge University, 1999.

[23] T. Vickers. An Overview of a Refinement Editor. In *Proceedings of the 5th Australian Software Engineering Conference*, pages 39 - 44, Sydney - Australian, May 1990.

[24] J. Wright, J. Hekanaho, P. Luostarinen, and T. Langbacka. Mechanizing Some Advanced Refinement Concepts. *Formal Methods in System Design*, 3:49 - 81, 1993.

[25] J. Wright. Program Refinement by Theorem Prover. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing, pages 121 - 150, London - UK, 1994. Springer-Verlag.

[26] M. A. Xavier. *Refine Project Page*, 2004. At http://www.cin.ufpe.br/~max/Refine/.

A Refinement Laws

Law *Strengthen Postcondition*: if $pos' \Rightarrow post$, then

$$w : [pre, pos] \quad \sqsubseteq \quad w : [pre, pos']$$

Law *Weaken Precondition*: if $pre \Rightarrow pre'$, then

$$w : [pre, post] \quad \sqsubseteq \quad w : [pre', post]$$

Law *Assignment*: if $pre \Rightarrow post[w\backslash E]$, then
$w, x : [pre, post] \sqsubseteq w := E$

Law *Next Assignment*: For any term E,
$w, x : [pre, post] \sqsubseteq \ w, x : [pre, post[x\backslash E]]; w := E$

Law *Alternation*: if $pre \Rightarrow GG$, where $GG = G_0 \wedge G_1 \wedge ... \wedge G_n$, then
$w : [pre, post] \quad \sqsubseteq \quad \textbf{if } ([]i \bullet G_i \rightarrow w : [G_i \wedge pre, post])\textbf{fi}$

Law *Strengthen Postcondition with Initial Variables*: if $pre[w\backslash w_0] \wedge post' \Rightarrow post$, then
$w : [pre, post] \ \sqsubseteq w : [pre, post']$

Law *Assignment with Initial Variables*: if $(w = w_0) \wedge pre \Rightarrow post[w\backslash E]$, then
$w, x : [pre, post] \ \sqsubseteq w := E$

Law *Logical Constant Removal*: if c does not happen int the program p, then
$[\![\textbf{con } c : T \ \bullet \ p]\!] \sqsubseteq p$

Law *Sequential Composition with Constants*: for new constants X,
$w, x : [pre, post] \sqsubseteq [\![\textbf{con } X \ \bullet \ x : [pre, mid]; w, x : [mid[x_0\backslash X], post[x_0\backslash X]]]\!]$
The formula mid does not have to contain another initial variables beyond x_0.

Law *Parameterized Procedure Introduction*: if p_n is not free in p_2,
$p_2 = [\![\textbf{proc } p_n = (par \bullet \ p_1) \bullet \ p_2]\!]$

Law *Procedure Call Introduction with Parameters*:
$[\![\textbf{proc } pn = (par \bullet p_1) \bullet p_2[(par \bullet p_1)(a)]]\!] =$
$[\![\textbf{proc } p_n = (par \bullet p_1) \bullet p_2[p_n(a)]]\!]$

Law *Call by Value*: since that f is not in w e w is not free in a,
$w : [pre[f\backslash a], post[f, f_0\backslash a, a_0]] = (\textbf{val } f \bullet w : [pre, post]) \ (a)$

Law *Call by Result*: since that f is not in w and is not free in pre or $post$, and f_0 is not free in $post$,
$w, a : [pre, post] = (\textbf{res } f \bullet \ w, f : [pre, post[a\backslash f]])(a)$

Law *Multiple Parameters*: since that f_1 is not free in a_2 ,
$(par_1 \ f_1 \bullet \ (par_2 \ f_2 \bullet \ p) \ (a_2)) \ (a_1) = (par_1 \ f_1; par_2 \ f_2 \bullet \ p) \ (a_1, a_2)$

Law *Variant Introduction with Parameters*: if nm and n are not free in e and p_2,

$p_2 = [\![\mathbf{proc}\ nm = (par \bullet\ w : [n = e \wedge pre, post])\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2]\!]$

Law *Procedure Call Introduction in the Main Program of a Variant Block (with Parameters)*: since that nm is not recursive and n is not free in e and $w : [pre, post]$,

$[\![\mathbf{proc}\ nm = (par \bullet w : [n = e \wedge pre, post])\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet$
$1\!\!\widehat{}\ p_2[(par\ \bullet\ w : [pre, post])(a)]]\!]$
$=$
$[\![\mathbf{proc}\ nm = (par\ \bullet\ w : [n = e \wedge pre, post])\ \mathbf{variant}\ n\ \mathbf{is}\ e\ \bullet\ p_2[nm(a)]]\!]$

Law *Recursive Call Introduction (with Parameters)*: since that n is not free in $w : [pre, post]$ and $p_1[nm(a)]$, and $w : [n = e \wedge pre, post] \sqsubseteq p_1$ then

$[\![\mathbf{proc}\ nm = (par \bullet p_1[(par \bullet w : [0 \leq e \leq n \wedge pre, post])(a)])$
$1\!\!\widehat{}\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2]\!]$
$=$
$[\![\mathbf{proc}\ nm = (par \bullet p_1[nm(a)]) \bullet p_2]\!]$