The Development of a Computer Application that Identifies Reusable Components through Formal Specifications

Francisco Moreira Couto
Dept. of Mathematics, Technical University of Lisbon
Av. Rovisco Pais, 1049-001 Lisboa Codex, Portugal
E-mail: fcouto@math.ist.utl.pt

Abstract

Software reuse is certainly a way for increasing software productivity. In this paper we propose to use Formal Methods to develop one computer application, which will be able to automate the process of software reuse. We use single-sort algebraic specifications to specify the components functionality, and then we apply the mathematical framework of category theory to develop matching mechanisms, which identify reusable components.

Keywords: algebraic specification, category theory, software reuse, specification matching.

1 Introduction

This paper presents one solution to the problem of software reuse. The method proposed here uses formal specifications in the classification and retrieval of the components, with the ultimate goal of identifying reusable components. More precisely, the solution presented here consists of a computer application with the functionality represented in the figure 1. Generically, the user identifies the component's algebraic specification [TM87], which is given to the application. Afterwards, the application creates the respective categorical representation [Wal91, Pie93]. Finally, the application will try to identify reusable components using the categorical representations that are in the repository.

We selected algebraic specifications in the component's classification because the algebraic specification languages are closer to human reasoning than the category theory, so is more difficult to the user classify the components using the category theory. We selected category theory in the components retrieval because category theory is a powerful, language independent, mathematical framework and can be efficiently implemented in matching algorithms.

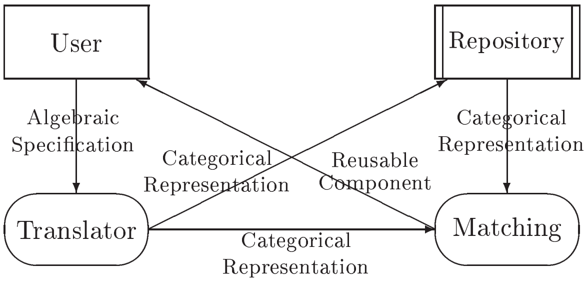

Figure 1: The generic architecture of the application proposed

The implementation of this application will be split in two phases. The first phase develops the process Translator of the figure 1, which translates components' algebraic specifications into their respective categorical representations. The second phase develops the process Matching of the figure 1, which uses matching algorithms to identify categorical representations that represent the same functionality behaviour.

1.1 First phase

This phase will be made in two stages. The first stage will be the revision and the feasibility study of the methods proposed in the literature [Cre98, Lai76]. The goal is to obtain a set of algorithms that can be directly implemented in a computer application. The second stage consists in implementing the algorithms identified in the first stage. The result of this stage will be a computer application that accomplishes the main objective of this phase.

1.2 Second phase

This phase consists in identifying reusable components through their categorical representation. Here, it'll be revised the algorithms presented in the paper [Cre98], which determine if two categorical representations match one with each other. Until now, only isomorphic matching algorithms had been used to identify reusable components. These algorithms do not cover every possible case of code reusability. Therefore, other non-isomorphic matching algorithms must be developed.

Another objective of this phase is to develop a computer application that implement the matching algorithms identified. This application will identify, automatically, possible reusable components.

1.3 Complexity

During the developing of this project it'll be created some component's algebraic specifications to test the functionality of the computer application developed. Therefore, it'll be possible to make studies about the complexity of the algorithms developed, and then see if they are practicable.

1.4 Programming language

The algorithms developed in this project will be expressed in ML [RB98], a functional programming language. The principal reason for this choice is because functional languages are closer to mathematical notation and the computer application will manipulate two types of specifications based in mathematics.

2 Basic definitions

This section gives a brief overview of single-sort algebraic logics and category theory.

2.1 Single-sort algebraic logics

Algebraic logic [Gin86] is a restriction of the classical logics in a way that the only acceptable terms are variables and constants.

The basic alphabet for one single-sort algebraic language is made of:

- A list of individual variables $v_1, v_2, \ldots$;
- A list of constants $c_1, c_2, \ldots$;
- Connectives $\neg, \vee, \wedge, \Rightarrow$;
- Universal and existencial quantifiers $\forall, \exists$;
- A list of predicates $P_1, P_2, \ldots$;

Terms: These are expressions denoting individuals and are variables or constants.

Propositions: These are expressions of the form $P_i(t_1, \ldots, t_n)$ where P_i is a predicate of arity n and $t_1, \ldots, t_n$ are terms. The logic constants $true$ and $false$ are propositions too.

Formulae: These are built inductively by the rules

- Each proposition is a formula.
- If ϕ and ψ are formulae, then so are $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$ and $\phi \Rightarrow \psi$.
- If ϕ is a formula and v_i an individual variable, then $\forall v_i\, \phi$ and $\exists v_i\, \phi$ are formulae.

2.2 Normal forms

Normal forms [Hog90] are forms in which the formulas are replaced by others formulas that are equivalent, but with a more regular structure. Therefore, these formulas can be more efficiently manipulate.

In our work we adopted the clausal form, which is a particular case of a normal form.

Definition 2.1 *A* clausal representation *is a set of clauses that denotes its conjunction.*

Definition 2.2 *A* clause *is a formula with the form $N \Rightarrow P$, where N is a conjunction of propositions and P is a disjunction of propositions.*

Note 2.1 *The* quantifiers *are applied to a set of clauses.*

2.3 Category theory

Category theory [Wal91] is the algebra of functions in which the principal operation on functions is taken to be composition.

A category is an abstract structure: a collection of objects, together with a collection of arrows between them. For example, the objects could be sets of propositions representing their conjunction or disjunction, and the arrows the implications between them.

Definition 2.3 *A category **Cat** consists of a collection of* objects *(called obj **Cat**) and a collection of* morphisms *or* arrows *(arr **Cat**). The objects are denoted $A, B, C, \ldots$ and the arrows are denoted $f, g, h, \ldots$*

Further:

- *A arrow has a designed* domain *and* codomain *in obj **Cat**. When the domain of f is A and the codomain of f is B we write $f : A \to B$.*
- *Given arrows $f : A \to B$, $g : B \to C$, there is a designated* composite *arrow, $g \circ f : A \to C$.*
- *Given any object A, there is a designated* identity *arrow $1_A : A \to A$.*
- *The data above is required to satisfy the following:*

  Identity laws *If $f : A \to B$ then $1_B \circ f = f$ and $f \circ 1_A = f$*

  Associative law *If $f : A \to B$, $g : B \to C$ and $h : C \to D$ then*
  $h \circ (g \circ f) = (h \circ g) \circ f : A \to D$

Definition 2.4 *In a category **Cat** an object 0 is called* initial *if for any object X in **Cat** there is a unique arrow $0 \to X$.*
An object 1 is called terminal *if for any object X in **Cat** there is a unique arrow $X \to 1$.*

In the category described above, where the objects were sets of propositions, the initial object should represent the logical value $false$ and the terminal object should represent the logical value $true$.

3 Tools used

The stage of implementation began with the search of tools to implement Standard ML (SML) algorithms. The tools chosen are from the version 110.0.6 of Standard ML of New Jersey (SML/NJ) system. The SML/NJ system can be found on the Internet at
http://cm.bell-labs.com/cm/cs/what/smlnj

The reasons for choosing the SML/NJ system are: it's a freeware system; it works in various machine/OS combinations; it's supplied with a lexical analyzer and a parser generator.

4 Options made

This section describes the more important options and extensions made during the revision and the feasibility study of the methods proposed in the literature [Cre98, Lai76].

4.1 Quantifiers

The methods proposed restrict the application of quantifiers to a single predicate, and they couldn't be directly represented as algorithms because they have some ambiguous aspects. Therefore, in this work was developed a new method for treating quantifiers, based in the following result:

Result 4.1 *Given a predicate P and being the set Sort equal to $\{c_1, \ldots, c_n\}$, we have the following tautologies:*

- $\forall x\, P(x) \Leftrightarrow P(c_1) \wedge \ldots \wedge P(c_n)$
- $\exists x\, P(x) \Leftrightarrow P(c_1) \vee \ldots \vee P(c_n)$

For any formula, with quantifiers applied to a single predicate, we can use the result 4.1 to get a new equivalent formula without quantifiers.
This result works only with quantifiers applied to a single predicate. Therefore, we have to find new results that given any formula, with quantifiers applied to more than one predicate, identify a new equivalent formula without quantifiers. It was been made a study of such results, but it wasn't possible to found all of them until now.

4.2 Normal forms

In the original algorithms, the formulas weren't represented in a normal form. Thus, the algorithms were modified in order to represent the formulas in the clausal form. Therefore, the user must introduce an algebraic specification with all formulas in clausal form.

The formulas used in the computer application can only have quantifiers applied to a single predicate. Therefore for representing these formulas in the clausal representation it was necessary to extend the proposition definition.

Definition 4.1 *A* pseudo-proposition *is a proposition or a quantifier applied to a proposition.*

Note 4.1 *Clauses that have pseudo-propositions are called* pseudo-clauses.

The result 4.1, for dealing with the quantifiers, give formulas that could not be in the clausal form. Thus, it's necessary to represent these formulas into the clausal form. For doing this, we use the following result:

Result 4.2 *Given the propositions a, b, c, d, then we have the following tautologies:*

- $[a \wedge (b \vee c) \Rightarrow d] \Leftrightarrow [(a \wedge b \Rightarrow d) \wedge (a \wedge c \Rightarrow d)]$
- $[a \Rightarrow b \vee (c \wedge d)] \Leftrightarrow [(a \Rightarrow b \vee c) \wedge (a \Rightarrow b \vee d)]$

4.3 Predicate arity

In the originals methods predicates have only one argument, i.e., their arity is one. In this work, we didn't remove such limitation because it's been given more importance to the restrictions on the quantifiers. Therefore, in the computer application presented here, predicates still have only one argument.

4.4 Predicate implication

The set formulae were extended with the predicate implication formula. This new type of formula is represented in the form $P_1 \rightarrow P_2$, where P_1 and P_2 are any predicates.
The meaning of this new type formula can be described as $P_1 \rightarrow P_2 \Leftrightarrow \forall x \, [P_1(x) \Rightarrow P_2(x)]$

Note 4.2 *The notion of* pseudo-clause *is extended with the predicate implication formula.*

The reasons for using this new type of formula are: it isn't possible to apply explicitly a quantifier to a formula; it's a very useful formula; and its meaning will be very helpful in matching categorical representations.

4.5 Equality predicate

The use of the equality predicate (=) is very important to specify a component's functionality. The equality predicate arity is two, but in this project we only have predicates of arity one. Therefore was developed the following solution to this problem. The computer application replaces the proposition =(Variable,Constant) by the proposition =Variable(Constant), therefore, there is a distinct equality predicate for each variable although they have all the same meaning, which is to verify if the variable value is equal to the constant.

The following propositions =(Var1,Var2) and =(Var2,Var1) are equivalent, but the computer application represents these propositions, respectively, by =Var1(Var2) and =Var2(Var1). Therefore, it'll be impossible to match =Var1(Var2) with =Var2(Var1) because they use two different predicates although they represent equivalent propositions. Consequentially, in this project the arguments of an equality predicate shouldn't be two variables.

4.6 Specials predicates

In the original proposals, all the predicates were used in the same way, but when the user apply the equality predicate he implicitly gives a special meaning to this predicate, which is the same in all specifications. Therefore, we shouldn't match an equality predicate with any other. In this project, the equality predicate is treated as special predicate so it can't be matched with others normal predicates.

It could be used much more special predicates like the major (>), minor (<), ...
All of these predicates holds its own meaning. In this project, we only used the equality predicate, but the methods implemented can support more special predicates.

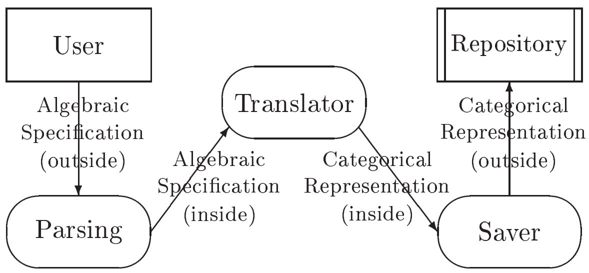

Figure 2: The generic architecture of the application developed

4.7 The set Sort

The proposed methods consider the set Sort as a discrete and finite set. This isn't a very serious limitation because almost all programming languages have its variables restricted to a maximum and a minimum value. Therefore, the computer application developed uses a discrete and finite set to represent the set Sort.

5 Results

This section gives a brief overview of the results obtained in this work. These results have permitted to implement a computer application that translates a component's algebraic specification into its categorical representation.

The generic architecture of this application is described in the figure 2. The user gives a single-sort algebraic specification in the outside representation, which is transformed in its inside representation by the Parsing process. Afterwards, the Translator creates the respective inside categorical representation, which is transformed in its outside representation and saved in the repository by the Saver process.

5.1 Algebraic specifications

In this work is considered a code component as the basic unit of reusability. The user must describe the behaviour of a code component by a formal specification that he writes in a file. This specification is called an algebraic specification (outside), whose syntax in EBNF notation is depicted next:

```
<specification>::=Component<id>
                  Sort:{<constants>}
                  Variables:<vars>
                  Invariant:<pseudo-clauses>
                  {<method>}
<constants>::=<constant>{,<constant>}
<vars>::=<var>{,<var>}
<method>::=Method:<id>
           Interface:<interfaces>
           Requires:<preconditions>
           Ensures:<postconditions>
<interfaces>::=[<interface>{,<interface>}]
<interface>::=?<var>|!<var>
<preconditions>::=<disjunction>
                  {,<disjunction>}
<postconditions>::={<postcondition>;}
<postcondition>::=<antecedent>-><consequent>
<antecedent>::=<conjunction>
<consequent>::=<pseudo-clauses>
<pseudo-clauses>::=<pseudo-clause>
                   {,<pseudo-clause>}
<pseudo-clause>::=
                <conjunction>=><disjunction>|
                <predicate>-><predicate>
<predicate>::=<id>|<variable>=
<conjunction>::=<pseudo-proposition>
                {&<pseudo-proposition>}
<disjunction>::=<pseudo-proposition>
                {|<pseudo-proposition>}
<pseudo-proposition>::=><var>:<proposition>|
                       <<var>:<proposition>|
                       <proposition>
<proposition>::=<id>(<term>)|
                <variable>=<term>|
                true|false
<term>::=<variable>|<constant>
<variable>::=<var>|?<var>|!<var>
<var>::=<id>
<constant>::=<id>
<id>::=<character>{<character>|<digit>}
```

The set Sort is defined by a set of constants, which define the possible values to any variable. The Invariant's list of pseudo-clauses depicts constraints on the system behavior. State variables are presented in the Variables section.

A component contains a list of methods. For every method the inputs and outputs variables are depicted in the Interface section. Variable names with question and exclamation marks represent, respectively, input and output parameters. The Requires preconditions describe restrictions on the arguments, which define how the method may be invoked. The Ensures postconditions depict constraints on the method behavior. The postcondition antecedent selects the initial states and may only contain input and state variables.

The specification must be deterministic. Therefore, in every method specification the initial set of states described by two different antecedents must be disjoint.
The specification must also be robust: for every operation, the specification must describe the resulting properties when the system starts holding any possible preconditions, namely, normal and error conditions. Therefore, the union of the initial set of states described by the disjunction of the antecedents must be equal to the set of states described by the precondition.
In this specification language there is no guarantee that, if not referred in the consequent part of the postcondition, the value of state variables will remain unchanged.

An implicit conjunction of conditions is assumed in the pre and postconditions. Likewise, a list of pseudo-clauses represents an implicit conjunction of pseudo-clauses.

Some of the symbols, from the basic alphabet for one single-sort algebraic language, were replaced by others symbols, so it can be possible to save a specification in a text file. These replacements are depicted next: $>$ replaced $\forall$, $<$ replaced $\exists$, & replaced $\wedge$ and $|$ replaced $\vee$.

Example 5.1 *Next is presented an example of a component's algebraic specification, which represents the functionality of three special switches.*

```
Component Switches
Sort: {On, Off, Usv}
Variables: switch1, switch2, switch3
Invariant:
% One(x) indicates if any
% switch has the value x.
 switch1= -> One,
 switch2= -> One,
 switch3= -> One,
 One(On) => switch1=On | switch2=On |
            switch3=On,
 One(Off) => switch1=Off | switch2=Off |
             switch3=Off,
 One(Usv) => switch1=Usv | switch2=Usv |
             switch3=Usv,
% All(x) indicates if all the
% switches have the value x.
% All(Usv) is always false.
 switch1=On &
 switch2=On &
 switch3=On => All(On),
 switch1=Off &
 switch2=Off &
 switch3=Off => All(Off),
 All -> switch1=,
 All -> switch2=,
 All -> switch3=,
 All -> One

 % If the switch2 is On then all
 % the switches stay with the value
 % of the ?InitSwitch variable.
Method: Init
Interface: ?InitSwitch
Requires:
  ?InitSwitch=On | ?InitSwitch=Off,
  switch2=On
Ensures:
  ?InitSwitch=On & switch2=On ->
     true => switch1=On,
     true => switch2=On,
     true => switch3=On;
  ?InitSwitch=Off & switch2=On ->
     true => switch1=Off,
     true => switch2=Off,
     true => switch3=Off;

 % If all the switches have the same value
 % then !Value stay with this value
 % else !Value stay with the value Usv.
Method: DisplayAll
Interface: !Value
Requires: <x:All(x) | >x:One(x)
Ensures:
  All(On) -> true => !value=On;
  All(Off) -> true => !value=Off;
  >x:One(x) -> true => !value=Usv;

 % Replace the switch2 value to the
 % value of ?s2, !s1 stay with
 % the value of swicth1, and all the
 % switches stay with different values.
Method: PutAllTypes
Interface: ?s2, !s1
Requires: <x:All(x)
Ensures:
  <x:All(x) ->
     switch1=On => !s1=On,
     switch1=Off => !s1=Off,
     switch1=Usv => !s1=Usv,
     ?s2=On => switch2=On,
```

```
?s2=Off => switch2=Off,
?s2=Usv => switch2=Usv,
true => >x:One(x);
```

An outside representation of an algebraic specification is transformed in its inside representation, so the manipulation of an algebraic specification could be more efficient. This inside representation has the same information using lists and structures. It's possible to put comments in an outside specification using the character %, but this comments are not present in an inside specification.

5.2 Categorical representations

The computer application creates a categorical representation of a component's algebraic specification. This categorical representation (outside) is written in a file, and its syntax in EBNF notation is depicted next[1]:

```
<categ.rep.>::=Categorical Representation
               Component<id>
               {<method>}
<method>::=Method:<id>
           <precategory>
           {<postcategory>}
<precategory>::=PreCategory:<categories>
<postcategory>::=PostCategory:
                 <C-object>⇒<categories>
<categories>::=<C-category><P-category>
<C-category>::=Category C:
               <C-objects><C-arrows>
<C-objects>::=Objects:{<C-object>}
<C-arrows>::=Arrows:{<C-arrow>}
<C-arrow>::=<C-object>→<C-object>
<C-object>::=<singular>|<minimum>|
             <maximum>
<singular>::=#{<assertion>}
<minimum>::=&{<assertions>}
<maximum>::=|{<assertions>}
<assertions>::=<assertion>{,<assertion>}
<assertion>::=0|1|<predicate>(<term>)
<P-category>::=Category P:
               <P-objects><P-arrows>
<P-objects>::=Objects:{<P-object>}
<P-arrows >::=Arrows:{<P-arrow>}
<P-arrow>::=<P-object>→<P-object>
<P-object>::=<predicate>
```

[1] <predicate>, <variable>, <term> and <id> are depicted in the section 5.1.

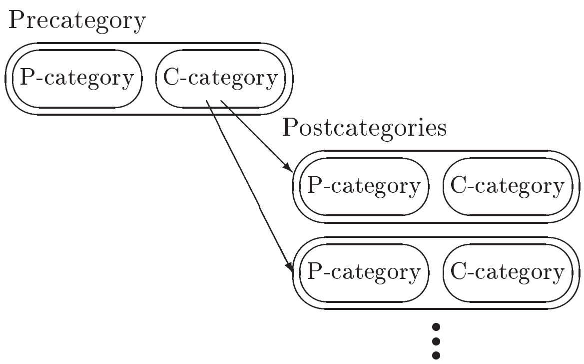

Figure 3: The categorical representation of one method specification

A component's categorical representation is a list of methods' categorical representations. For each method its categorical representation includes a precategory and a list a postcategories. The precategory represents the method precondition, and for each consequent part of a method postcondition there is a postcategory that represent it.
Figure 3 depicts the categorical representation of one method specification, which is a function from the precategory C-objects to the postcategories. Therefore, for each component postcondition there is a C-object that represents the antecedent part, which is related with a postcategory that represents the consequent part.

There are two different types of categories; The C-category represents the logic implications between sets of assertions, and the P-category represents the logic implications between predicates.
The C-objects represents one part of a logic implication, so it can represents an assertion, a minimum (conjunction) of assertions or a maximum (disjunction) of assertions.
An assertion represents a proposition, where the constants *true* and *false* are replaced, respectively, by the terminal (1) and initial (0) objects. A predicate can be a normal predicate or the special equality predicate. Variable names with question and exclamation marks represent, respectively, input and output parameters.

The result of translating an algebraic specification is an inside categorical representation, which is represented by lists and structures. Afterwards is created the equivalent outside categorical representation, which is written in a file.

5.3 Specifications translation

In this section is presented a generic description of the algorithms identified, which were used to implement the computer application that translates a component's algebraic specification into its categorical representation.

Component translation
Input: A component's algebraic specification.
Output: A component's categorical representation.

1. For each component's method run the *Method translation* algorithm.
2. Create a list with all the categorical representations obtained in the previous step.

Method translation
Input: A method's algebraic specification.
Output: A method's categorical representation.

1. Run the *Clauses translation* algorithm with the following arguments:
   (a) $\mathcal{A}$ equal to a set of pseudo-clauses equivalent to $\{\mathcal{E}_{ai} \Rightarrow \mathcal{R}$, $(i = 0,\ldots,$*number of postconditions*$)\}$, where $\mathcal{E}_{ai}$ is an antecedent part of a postcondition and $\mathcal{R}$ the precondition.
   (b) $\mathcal{V}$ is the set of state and method input variables.

   The result of this step is the precategory.
2. For each consequent part of a postcondition, $\mathcal{P}$, run the *Clauses translation* algorithm with the following arguments:
   (a) $\mathcal{A}$ equal to $\mathcal{P}$.
   (b) $\mathcal{V}$ is the set of state and method input/output variables.
3. For each postcondition associate its postcategory to the precategory C-object that represents the antecedent part.

Clauses translation
Input: $\mathcal{A}$ as a set of pseudo-clauses.
$\mathcal{V}$ as a set variables.
Output: A C-category and a P-category.

1. Join to $\mathcal{A}$ the set of pseudo-clauses that represents the Invariant.
2. For each variable $v_i \in \mathcal{V}$, join the pseudo-clause *true* $\Rightarrow \exists x\,(v_i = x)$ to $\mathcal{A}$.
3. Run the *Clause translation* algorithm for each pseudo-clause in $\mathcal{A}$.
4. Join all the objects and arrows, obtained in the previous step, into the C-category and P-category.

Clause translation
Input: $\mathcal{C}$ as a pseudo-clause.
Output: A set of C-objects.
A set of C-arrows.
A set of P-objects.
A set of P-arrows.

1. If $\mathcal{C}$ represents a predicate implication, then $\mathcal{C}$ as the following form:
   $P_1 \rightarrow P_2$ (P_1 and P_2 are any predicates)
   (a) Create the P-objects P_1 and P_2.
   (b) Create the P-arrow whose domain is P_1 and codomain is P_2.

   In this case the algorithm exits here.
2. Run the *Replace quantifiers* algorithm for $\mathcal{C}$.
3. For each clause $\mathcal{C}_i$, that resulted from the previous step, do:
   (a) Knowing that $\mathcal{C}_i$ is represented in the following form:
   $\{p_1 \wedge \ldots \wedge p_n \Rightarrow q_1 \vee \ldots \vee q_m\}$, where:
      - m and n are any two integers numbers greater than zero.
      - Each one of the symbols $p_1, \ldots, p_n$, $q_1, \ldots, q_m$ represent a proposition.

   (b) Create the following C-objects:
      - $min = \begin{cases} \&\{p_1, \ldots, p_n\} & \text{if } n > 1 \\ \#p_1 & \text{otherwise} \end{cases}$
      - $max = \begin{cases} |\{q_1, \ldots, q_m\} & \text{if } n > 1 \\ \#q_1 & \text{otherwise} \end{cases}$

   (c) Create the C-arrow whose domain is min and codomain is max.

(d) For each one of the symbols $p_1, \ldots, p_n$, $q_1, \ldots, q_m$ who represents a proposition of the form $P(t)$, create the P-object that represents the predicate P.

Replace quantifiers
Input: $\mathcal{C}$ as a pseudo-clause.
The set Sort.
Output: A set of clauses.

1. Knowing that $\mathcal{C}$ is represented in the following form:
   $\{p_1 \wedge \ldots \wedge p_n \Rightarrow q_1 \vee \ldots \vee q_m\}$, where:
   - m and n are any two integers numbers greater than zero.
   - The $p_1, \ldots, p_n, q_1, \ldots, q_m$ symbols represent pseudo-propositions.
2. Knowing that the set Sort is represented by $\{x_1, \ldots, x_r\}$, where:
   - r is any integer number greater than zero.
   - Each one of the symbols $x_1, \ldots, x_r$ represents constants.
3. Apply the result 4.1 to the pseudo-clause $\mathcal{C}$, i.e., for each one of the symbols $p_1, \ldots, p_n$, $q_1, \ldots, q_m$ that represents:
   - $\forall x\, P(x)$, replace by $P(x_1) \wedge \ldots \wedge P(x_r)$.
   - $\exists x\, P(x)$, replace by $P(x_1) \vee \ldots \vee P(x_r)$.
4. Apply the result 4.2 on the formulas obtained in the previous step, and the result will be a set clauses equivalent to $\mathcal{C}$.

Example 5.2 *Here is described the steps to obtain the categorical representation of the following pseudo-clause $\mathcal{C}$:*
$\forall x\, P_1(x) \wedge \exists x\, P_2(x) \wedge P_3(c_1) \Rightarrow$
$\forall x\, P_4(x) \vee \exists x\, P_5(x) \vee P_6(c_2)$
Assuming that the set Sort is $\{x_1, x_2\}$, we have the following steps:

1. *The outcome of applying the result 4.1 to $\mathcal{C}$ is the following formula:*
   $P_1(x_1) \wedge P_1(x_2) \wedge (P_2(x_1) \vee P_2(x_2)) \wedge P_3(c_1) \Rightarrow$
   $(P_4(x_1) \wedge P_4(x_2)) \vee P_5(x_1) \vee P_5(x_2) \vee P_6(c_2)$
2. *The outcome of applying the result 4.2 to the previous formula is the following set of clauses:*
   $\{P_1(x_1) \wedge P_1(x_2) \wedge P_2(x_1) \wedge P_3(c_1) \Rightarrow$
   $P_4(x_1) \vee P_5(x_1) \vee P_5(x_2) \vee P_6(c_2),$
   $P_1(x_1) \wedge P_1(x_2) \wedge P_2(x_2) \wedge P_3(c_1) \Rightarrow$
   $P_4(x_1) \vee P_5(x_1) \vee P_5(x_2) \vee P_6(c_2),$
   $P_1(x_1) \wedge P_1(x_2) \wedge P_2(x_1) \wedge P_3(c_1) \Rightarrow$
   $P_4(x_2) \vee P_5(x_1) \vee P_5(x_2) \vee P_6(c_2),$
   $P_1(x_1) \wedge P_1(x_2) \wedge P_2(x_2) \wedge P_3(c_1) \Rightarrow$
   $P_4(x_2) \vee P_5(x_1) \vee P_5(x_2) \vee P_6(c_2)\}$
3. *Create the following C-objects:*
   - $\&\{P_1(x_1), P_1(x_2), P_2(x_1), P_3(c_1)\}$
   - $\&\{P_1(x_1), P_1(x_2), P_2(x_2), P_3(c_1)\}$
   - $|\{P_4(x_1), P_5(x_1), P_5(x_2), P_3(c_1)\}$
   - $|\{P_4(x_2), P_5(x_1), P_5(x_2), P_3(c_1)\}$
4. *Create the following C-arrows:*
   - *domain:*
     $\&\{P_1(x_1), P_1(x_2), P_2(x_1), P_3(c_1)\}$
     codomain:
     $|\{P_4(x_1), P_5(x_1), P_5(x_2), P_3(c_1)\}$
   - *domain:*
     $\&\{P_1(x_1), P_1(x_2), P_2(x_2), P_3(c_1)\}$
     codomain:
     $|\{P_4(x_1), P_5(x_1), P_5(x_2), P_3(c_1)\}$
   - *domain:*
     $\&\{P_1(x_1), P_1(x_2), P_2(x_1), P_3(c_1)\}$
     codomain:
     $|\{P_4(x_2), P_5(x_1), P_5(x_2), P_3(c_1)\}$
   - *domain:*
     $\&\{P_1(x_1), P_1(x_2), P_2(x_2), P_3(c_1)\}$
     codomain:
     $|\{P_4(x_2), P_5(x_1), P_5(x_2), P_3(c_1)\}$
5. *Create the P-objects P_1, P_2, P_3, P_4, P_5 and P_6.*

6 Conclusions

The first phase of this project has already been concluded. Its first stage consisted in the revision and the feasibility study of the methods proposed in the literature [Cre98, Lai76]. In this stage was necessary to make some extensions to the originals methods, so they can be efficiently implemented

in a computer application. These extensions consisted in: developing new methods for manipulate quantifiers; representing the formulas in a pseudo-clausal form; extending the algebraic specification language; giving a special meaning to the equality predicate; and transforming the equality predicate of arity two into a predicate of arity one.
Therefore, we revised and developed new algorithms to translate algebraic specifications into its categorical representation.

The second stage implemented the algorithms identified in the first stage. For this, it was necessary to develop efficient data structures to represent algebraic specifications and categorical representations. It was developed two ways of representing them:

- Outside Representation, which is used to keep a specification in a file.

- Inside Representation, which is used to manipulate a specification in the memory.

In this project, it was used two algebraic specifications. One is a simple specification often used in the literature, which specify the functionality of a simple switch. The other specification is depicted in the example 5.1, and it was created in this work with the propose of testing the computer application presented here. Therefore, it includes almost all types of formulas combinations that are permitted in ours algebraic specifications.

6.1 The computer application

The upshot of the work done so far is a computer application that translates a component's algebraic specification into its categorical representation.

The result of using the tools selected was positive. It was necessary to take a long time to learn how to use these tools, because they have little and a not very good documentation. But, even so, this time was recuperated in the implementation of the application because the tools provided an abstraction level very close to the representation of the algorithms developed.

After the application was conclude, it was used with the two algebraic specifications created. The categorical representations produced by the application are much more longer and less perceptive than their algebraic specifications, but that was already expected.

This application and all its documentation can be found on the Internet at
http://www.math.ist.utl.pt/fcouto/tfc/index.html

6.2 Future work

In this project, it's used a single-sort algebraic specification language with predicates of arity one. This language is very restricted and, therefore, the work must be extended to a more complete specification language, many-sorted with multiple arity predicates. The representation of such logics is depicted in [Lai76].

The application of quantifiers in this project can only be applied to a single predicate. During this work it was been studied new methods which can solve this limitation, but it wasn't possible to complete them until now. Therefore, in the future it'll be necessary to complete these methods so it can be possible to use the quantifiers without restrictions.

Like it was said, this work was split in two phases. The first one is already complete, now it'll start the second phase. This second phase consists in developing and implementing efficient methods that identify reusable components through their categorical representation. This phase will be concluded until September of 2001.

Acknowledgments

I'm grateful to Prof. Crespo for his support and advices throughout the developing of this work. I also want to express my gratitude to Prof. Paula Gouveia for her precious suggestions.

References

[Cre98] Rui Gustavo Crespo. Matching single-sort algebraic specifications for software reuse. *International Journal of Software an Knowledge Engineering*, 8(3):401–425, 1998.

[Gin86] S. G. Gindikin. *Algebraic Logic*. Problem Books in Mathematics. Springer-Verlag, 1986.

[Hog90] C. J. Hogger. *Essentials of Logic Programming*. Oxford Press, 1990.

[Lai76] Luis M. Laita. Un estudio de la lógica algebraica desde el punto de vista de la teoría de categorías. *Notre Dame Journal of Formal Logic*, 17(1):89–118, January 1976.

[Pie93] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1993.

[RB98] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory*. Prentice Hall, 1998.

[TM87] W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs*. Addison-Wesley, 1987.

[Wal91] R. F. C. Walters. *Categories and Computer Science*. Cambridge University Press, 1991.