

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES  
DE SOPHIA ANTIPOLIS  
UMR 6070

# REWRITING OF IMPERATIVE PROGRAMS INTO LOGICAL EQUATIONS

*Olivier Ponsini, Carine Fédèle, Emmanuel Kounalis*

*Projet LANGAGES & Projet COPRIN*

Rapport de recherche  
I3S/RR-2002-31-FR

Août 2002

---

RÉSUMÉ :

Ce rapport décrit un système permettant de traduire automatiquement des programmes écrits en subC, un langage impératif simple, en un ensemble d'équations du premier ordre. Cet ensemble d'équations représente un programme subC et a un sens mathématique précis ; de plus, les techniques standards de mécanisation du raisonnement équationnel peuvent être utilisées pour vérifier des propriétés des programmes. Une grande partie du système est elle-même formulée de façon abstraite comme un ensemble de règles de réécriture du premier ordre. Nous avons montré la terminaison et la confluence des règles du système de réécriture. Ceci signifie que notre système produit, pour un programme subC donné, un unique ensemble d'équations. Notre travail montre que les programmes impératifs simples peuvent être vus comme des systèmes logiques totalement formalisés, à l'intérieur desquels il est possible de prouver des théorèmes.

MOTS CLÉS :

Transformation de programmes impératifs, Sémantique Opérationnelle, Sémantique Équationnelle, Réécriture, Système de Réécriture, vérification de programmes

---

ABSTRACT:

This paper describes a system for automatically translating programs written in subC, a simple imperative language, into a set of first-order equations. This set of equations represents a subC program and has a precise mathematical meaning; moreover, the standard techniques for mechanizing equational reasoning can be used for verifying properties of programs. Part of the system itself is formulated abstractly as a set of first-order rewrite rules. Then, the rewrite rules are proven to be terminating and confluent. This means that our system produces, for a given subC program, a unique set of equations. Our work shows that simple imperative programs can be seen as fully formalized logical systems, within which theorems can be proved.

KEY WORDS :

Imperative Program Transformation, Operational Semantics, Equational Semantics, Rewriting, Rewrite System

# Rewriting of Imperative Programs into Logical Equations

Olivier PONSINI, Carine FÉDÈLE and Emmanuel KOUNALIS

*Laboratoire I3S - UNSA - CNRS*  
*Les Algorithmes*  
*2000, route des Lucioles*  
*B.P. 121*  
06903 SOPHIA ANTIPOLIS, FRANCE

---

## Abstract

This paper describes a system for automatically translating programs written in  $\text{sub}_C$ , a simple imperative language, into a set of first-order equations. This set of equations represents a  $\text{sub}_C$  program and has a precise mathematical meaning; moreover, the standard techniques for mechanizing equational reasoning can be used for verifying properties of programs. Part of the system itself is formulated abstractly as a set of first-order rewrite rules. Then, the rewrite rules are proven to be terminating and confluent. This means that our system produces, for a given  $\text{sub}_C$  program, a unique set of equations. Our work shows that *simple* imperative programs can be seen as fully formalized logical systems, within which theorems can be proved.

*Key words:* Imperative Program Transformation, Operational Semantics, Equational Semantics, Compiling, Rewriting, Rewrite System, Program Verification

---

## 1 Introduction

In most cases in which a program specification is done correctly, software deficiencies that come from the gap between the specification and its actual coding are by far more numerous than errors due, for instance, to hardware failure or to the compiler. To increase confidence in code production, effort should be focused on verifying that programs meet their requirements, that is, that they are sound with respect to their specifications.

When dealing with *program soundness*, developers usually use empirical methods like test sets. But this is not sufficient for applications that need a high

degree of reliability. For validation, applications would strongly benefit from *formal methods*, *i.e.*, mathematical tools and techniques aimed at specifying and verifying software or hardware systems. By verification, we mean the analysis that demonstrates a program has the desired properties.

### 1.1 Outline of our Approach and Related Work

In [1,2], we promote the idea of generating equations from imperative programs. The principle is to translate source code into a set of first-order equations expressing the program algebraic semantics. This translation is part of a framework for automatically proving properties of programs. The use of *equational logic* has advantages over other, more complex logics:

- it is very simple — the logic of substituting equals for equals;
- many problems associated with equations are decidable in equational logic;
- there are efficient algorithms for deciding many of these problems.

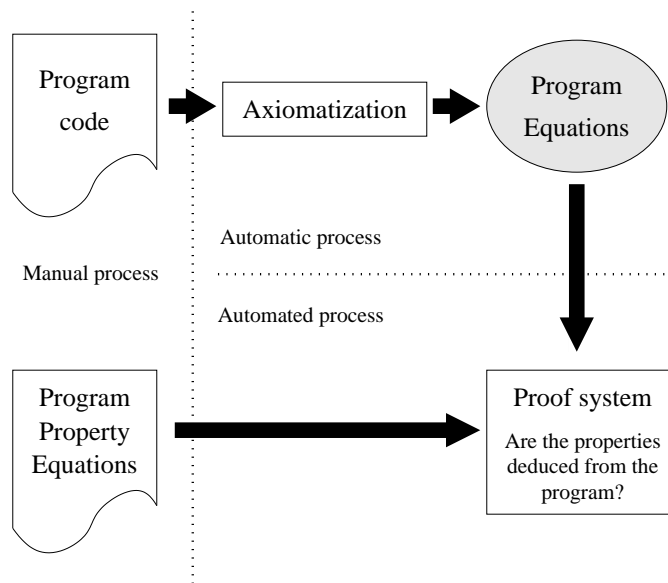


Fig. 1. Proof process overview.

The general outline of our framework is shown in figure 1. Users write down the sub<sub>c</sub> code of a program; they also write the program specification as a set of properties expressed in equational logic. The source code is then transformed automatically into a set of equations, the *axiomatization*. The equations of the program can be seen as the axioms, and the properties to be proved as the conjectured theorems of the axioms. Therefore, proving these theorems from the axioms is equivalent to proving the program meets its specification. Since both the axioms and the specification of the program are expressed in equational logic, the proof may be done within proof systems, automatically

using *theorem provers* able to do mathematical induction like NICE [3] or interactively using *proof checkers* like COQ [4].

Imperative languages are widely used in the industrial world, which requires simple and user-friendly specification and verification tools. Several other approaches address this challenge. Two large classes exist: approaches that generate code from specifications and approaches that work with source code as raw material.

- *Program synthesis* uses a formal and high-level language to describe program specifications. The specification language semantics is defined well enough to produce source code in various programming languages. Systems based on this approach differ mainly in the specification language, which is often tuned for a particular type of application. Examples of such systems are COGITO [5] and SPECWARE [6]. This approach suffers from several drawbacks. The specification language can help in saying *what* a program must do, but the language is often not sufficient to express *how* it should be done. The generated code is not as efficient as the one a programmer would write. In addition, these systems cannot be used for verifying existing programs or for maintenance.
- The second category of verification systems, *i.e.* the category of systems that deal with source code, can itself be divided into two sub-categories.
  - *Program annotation* requires that the user insert program specifications in the form of annotations directly into source code. These annotations will help the system to carry out the proof (see [7] and [8] for instance). In these frameworks, even simple properties can be difficult to prove. Moreover, these frameworks mix specification contents with code. Therefore, either the programmer must have a good understanding of the specification language or the specifier must have sufficient knowledge of the coding language. In both cases, the same person must master two disparate languages and adopt two different points of view.
  - *Specification generation* attempts to extract specifications from source code and verify them against user specifications. This kind of system needs no user interaction except, possibly, for the proof step. The method discussed in [9] uses a set of well-known semantics-preserving transformations to extract specifications. Specifications are then written in a language mixing high-level and low-level contents. PESCA [10] is close to our approach. This system uses algebraic semantics for the specification part and a basic imperative language for the programming part. The proofs are conducted in the LARCH PROVER [11] theorem prover. The main differences with our work come from the restrictions applied to the programming language (*e.g.* no recursion or no loops allowed), the method used to generate the specifications, and the specification language, which is not always easily amenable to automation.

In our approach, we endeavor to work on source code written by programmers. This way the code can be manually optimized. We do not use annotations and, thus, we distinguish the coding and specifying activities. Also, a strong requirement is to automate the whole process.

## 1.2 Highlights of our Approach

This paper is a full version of [1]. We focus on the program *axiomatization*: the operation that derives equations from source code.

- (1) We give an abstract framework to the *program towards equation* process. In particular :
  - We formulate the axiomatization mainly as a rewrite system.
  - We prove the rewrite system *completeness* using RRL<sup>1</sup>. Roughly speaking, this kind of completeness means that  $\text{sub}_C$  programs can be translated into a unique equational program.
  - We give a formal description of how equations are generated from environments.
- (2) We assert that the axiomatization process can be automatized since we made an implementation in Java.
  - JavaCC<sup>2</sup>, a parser and scanner generator, has been used for the term generation step.
  - We developed a Java version of a *generic* rewriting algorithm. The rewrite rules are loaded separately from a file.
  - We worked out an algorithm to generate equations from environments.
  - A set of computer experiments was done.

## 1.3 Structure of this paper

In section 2, we first introduce the  $\text{sub}_C$  language, and some basic definitions and notations about conditional equational logic and rewriting. Then, section 3 gives a general outline of the axiomatization. Section 4 describes in detail the steps involved in the axiomatization process. It also extensively discusses the rewrite system and its rules. It follows the scheme:

- (1) programs are terms (section 4.1);
- (2) terms rewrite into environments (section 4.2);
- (3) environments generate equations (section 4.3).

---

<sup>1</sup> Rewrite Rule Laboratory [12].

<sup>2</sup> Java Compiler Compiler, Metamata.

In section 5, we illustrate these steps through concrete examples. Some other examples are briefly presented in appendix A. Rewrite system rules are given in appendix B and the convergence proof is in appendix C.

## 2 Background

### 2.1 The $sub_C$ Language

For our experiments, we use a very simple imperative language. The  $sub_C$  syntax is similar to the C one and figure 2 shows the  $sub_C$  grammar given in EBNF. The main features of the language are:

- assignment;
- function notion: argument-passing by value, local variables;
- control flow statements: **if** ... **else**, **while** and **return**;
- two predefined types: integers (**int**), and lists (**list**);
- usual arithmetic operators;
- operators on lists,
  - **getHead** returns the first element of a list;
  - **getQueue** returns a list except for its first element;
  - **NULL** represents an empty list;
  - **cons** builds a new list by adding an element to the head of another list.

There is no restriction on the nesting level of control flow statements, but only one **return** statement per function is allowed. Moreover, several common features in imperative languages are unavailable in  $sub_C$ :

- no user's defined types;
- no global variables;
- no pointers directly accessible — of course some are used in the predefined abstract type **list**, but it is hidden.

### 2.2 Equational Logic

Let  $F$  be a set of symbols called *function symbols*, each symbol  $f$  in  $F$  has an arity. Elements of arity zero are called *constants*. Let  $X$  be a denumerable set of *variable symbols*. The set of (first-order) terms  $T(F, X)$  is the smallest set containing  $X$  and such that the string  $f(t_1, \dots, t_n)$  is in  $T(F, X)$  whenever the arity of  $f$  is  $n$  and  $t_i \in T(F, X)$  for  $i \in [1..n]$ . *Formulas* in the logic of equality are built from first-order terms and the equality predicate. We call a pair of two terms  $l$  and  $r$  denoted by  $l = r$  an *equation* and a pair denoted

```

program ::= function*

function ::= ( type | 'void' ) ident
           '(' [ ('void' | parameterlist ) ] ')'
           ( ';' | functionbody )

parameterlist ::= parameterdeclaration
               ( ',' parameterdeclaration ) *

parameterdeclaration ::= type ident

functionbody ::= '{' [ variabledeclarationlist ]
                [ statementlist ]
                [ returnstatement ';' ] '}'

variabledeclarationlist ::= variabledeclaration +

variabledeclaration ::= type variableinitialisation
                     ( ',' variableinitialisation ) * ';'

variableinitialisation ::= ident [ '=' exp ]

statementlist ::= statement +

statement ::= assignstatement ';'
           | whilestatement ';'
           | branchstatement ';'
           | '{' statementlist '}'
           | '{' '}'

assignstatement ::= ident '=' exp

returnstatement ::= 'return' exp

whilestatement ::= 'while' condition statement

branchstatement ::= 'if' condition statement
                 [ 'else' statement ]

condition ::= '(' exp ')

type ::= 'int'
       | 'list'

exp ::= an expression

```

Fig. 2.  $\text{sub}_C$  EBNF grammar.

by  $\neg(l = r)$  a *negative equation*. Equations and negative equations are called *atoms*. An *equational program*, also *Horn clause*, is written  $c \Rightarrow e$  where  $e$  is an equation,  $c \equiv \bigwedge_i (\bigvee_j c_{ij})$ ,  $c_{ij}$  are atoms, and  $\wedge$  and  $\vee$  denote respectively conjunction and disjunction.

### 2.3 Rewrite Systems

Rewrite systems follow from orientation of a set of equations. For an introduction to the rewrite system theory see [13] for instance.

Let  $T(F, X)$  denote the set of terms built out from the finite *vocabulary*  $F$  and a set  $X$  of *variables*. If  $t$  is a term and  $\theta$  is a *substitution* of terms for variables in  $t$ , then  $t\theta$  is an *instance* of  $t$ .

A *rewrite system*  $\mathcal{R}$  is a set of oriented equations  $l \rightarrow r$ , called *rewrite rules*. A rule is applied to a term  $t$  by finding a subterm  $s$  of  $t$  that is an instance of the left side  $l$  (*i.e.*  $s = l\theta$ ) and replacing  $s$  with the corresponding instance ( $r\theta$ ) of the rule right side. One computes with  $\mathcal{R}$  by repeatedly applying rules to rewrite (or reduce) an input term until a *normal form* (irreducible term) is obtained.

Let  $A$  be a set of equations, in the case where  $A$  can be compiled into a *convergent* (*i.e.* *terminating* and *confluent*) rewrite system  $\mathcal{R}$ , we can decide  $t =_A s$  by testing for identity the  $\mathcal{R}$ -normal forms of  $t$  and  $s$  (*i.e.*  $\text{nf}(t) \stackrel{?}{=} \text{nf}(s)$ ), where  $\text{nf}(t)$  (resp.  $\text{nf}(s)$ ) denotes the normal form of  $t$  (resp.  $s$ ).

## 2.4 The $\mathcal{L}$ Language

$\mathcal{L}$  is the language of our rewrite system (the complete set of rules is given in Appendix B). In this section we give the vocabulary of  $\mathcal{L}$  and a brief explanation of the semantics of some function symbols.

The vocabulary of  $\mathcal{L}$  contains function symbols denoting the  $\text{sub}_c$  syntactical constructs (see also section 4.1): `Assign`, `If`, `While`, `Return`.

We find also function symbols describing the  $\text{sub}_c$  operational semantics (see also section 4.2):

- `GE`, `Generate Environment`, translates the behavior of the *sequence* instruction. Statements are executed one after the other in the environment produced by the execution of the preceding statement (rules 1 and 2);
- `Comp`, `Compose`, is used to evaluate a new statement in the current environment;
- `Pair` terms are used to reflect the expression of a variable in an environment;
- `EP`, `Effective Parameter`, denotes the actual value of a function parameter.
- `LT`, `Loop Term`, will be used to generate the environment of a loop body and add new pairs to the current environment for the variables modified by the loop instructions as explained in section 4.2.1.
- `Choice` divides the environment into two parts. Each part is included in a `Branch` term and contains an `if` alternative depending on the condition value.

There are function symbols for lists: they are composed of a constant denoting the empty list and a constructor to add an element to an existing list. We

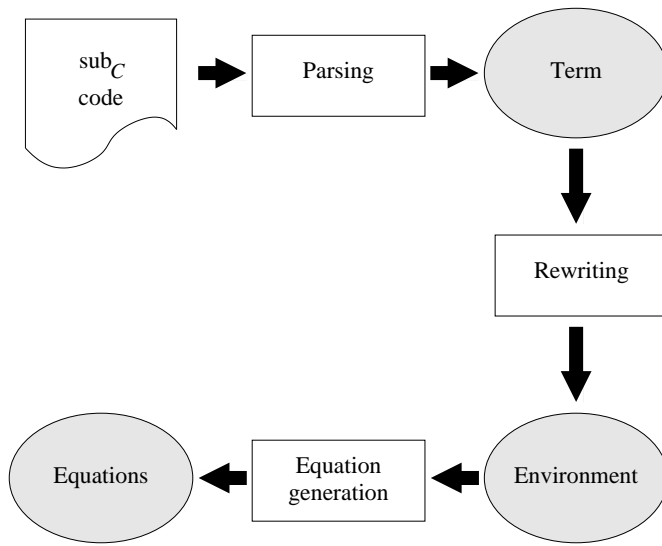


Fig. 3. Axiomatization process overview.

find lists of variables (`Cons_var` and `End_var`), lists of statements (`Cons_inst` and `End_inst`), environments (`Cons_env` and `End_env`) and lists of expressions (`Cons_exp` and `End_exp`). However, for convenience, lists will be represented enclosed in braces and elements in lists separated by dots, like in  $\{e_1 \cdot e_2 \cdot e_3\}$ .

Finally, we find function symbols handling the environment:

- `Update_env` to update the environment when a new pair is created (rules 14 to 17).
- `Merge_env` and `Merge_L_var` to merge two lists (rules 42, 43 and 32, 33);
- `Insert_pair` and `Insert_var` to add a pair or a variable to a list (rules 44 to 47 and 34 to 36);
- `GLOV`, `GLOMV` and `GLOE` to run through a list and build a new list by extracting, respectively, variables, modified variables or expressions from the initial list (rules 23 to 25, 26 to 31 and 37 to 39).

### 3 System Overview

The axiomatization is the operation that takes as input *a sub<sub>C</sub> program* and gives as output *a set of equations* semantically equivalent to the program: the result of the execution of the sub<sub>C</sub> program with input  $\mathcal{I}$  is identical, up to symbols translation, to the result (*i.e.* theorem) of the equational deduction, started with the same input. This section gives an informal description of the main stages underlying our method. The axiomatization is done in three steps, without any user interaction. These steps are shown in figure 3.

The central part of the system is a rewrite system  $\mathcal{R}$  over a first-order language

$\mathcal{L}$  built out from a set of function symbols. These symbols are translations of the constructs of  $\text{sub}_{\mathcal{L}}$ . For instance, the *assignment* statement, written  $x = y$  in  $\text{sub}_{\mathcal{L}}$ , is translated into  $\text{Assign}(x,y)$ , where  $\text{Assign}$  belongs to the vocabulary of  $\mathcal{L}$ .

The goal of the first step is to provide, from the source code, a correct input to the rewrite system, that is a *term* over  $\mathcal{L}$ . This term is then rewritten using  $\mathcal{R}$  into another term, which is an *environment*. Roughly speaking, environments contain information about the variable state, the value of each variable, at the end of an “abstract execution” of the program. We mean by “*abstract execution*” an execution not done on a real input but carried out with the textual variables given as parameters. Finally, the *equations* are extracted from the environment.

As an introductory example, let us see the different stages involved in the axiomatization of the very simple *identity* function of the following program:

---

```

int identity (int x) {
    return x;
}

```

---

At the first step, the *identity* function is transformed into a term over  $\mathcal{L}$ . In the function *identity*, the only statement is a return statement. We build a *statement list* containing only one element, which associates the expression of the return statement with the name of the function in a  $\text{Return}$  term:  $\{\text{Return}(\text{identity}, x)\}$ . The *initial environment* of the source function is a list of pairs associating function parameters and their value when the function is called. Here, we find parameter  $x$  associated with an EP term, which represents the value of  $x$  when the function is called:  $\{\text{Pair}(x, \text{EP}(x))\}$ . The statement list, which represents *the sequence of statements* of the source function, and the initial environment are gathered in a GE term:

$$\text{GE}(\{\text{Return}(\text{identity}, x)\}, \{\text{Pair}(x, \text{EP}(x))\}).$$

At the second step, the GE term is rewritten and generate the following environment:

$$\{\text{Pair}(x, \text{EP}(x)) \cdot \text{Pair}(\text{identity}, \text{EP}(x))\}.$$

We find in this environment the value of parameter  $x$  and the return value of the *identity* function at the end of the function. They are equal to the value of  $x$  at the moment of the call.

Finally, from this environment, one equation is generated:

$$\text{identity}(x) = x .$$

This equation expresses the algebraic semantics of the identity function.

## 4 Axiomatization Description

In this section, we go over the three steps of the axiomatization, in more detail.  $\mathcal{R}$  will denote the rewrite system we designed,  $\mathcal{L}$  will be the language of  $\mathcal{R}$ , and  $P$  will be a  $\text{sub}_C$  program.

### 4.1 Programs are Terms

The first step consists in parsing the source program  $P$ . The result of this syntactical analysis of  $P$  is a list of term  $T_P^f$  over a language  $\mathcal{L}$ ; one term for each function  $f$  of  $P$ . Intuitively, a term  $T_P^f$  is equivalent to a source function  $f$  of  $P$  and suitable for the rewriting at the second step. In terms of input/output, this step is described by:

**Input** : A  $\text{sub}_C$  program.  
**Output** : Terms over the rewrite system language  $\mathcal{L}$ .

Each  $\text{sub}_C$  syntactical construct is mapped to a function symbol of  $\mathcal{L}$ . Constructions and mapping are the following, where “ $\mapsto$ ” means “maps to”:

- A `function` is seen as a list of statements and an initial environment gathered in a `GE` term. The initial environment is a list of `Pair` terms. It is made up of the function formal parameters combined with `EP` terms in a `Pair`. An `EP` term denotes an effective parameter: `EP(x)` is the value given to parameter `x` when the function is called. Thus, the initial environment contains the value of the function variables after the call, but before any instruction of the function is evaluated.

$$\text{type func\_name (type}_1 \text{ p}_1, \dots, \text{type}_n \text{ p}_n) \{ \text{stmts} \} \mapsto \\ \text{GE}(\text{stmt\_list}, \{ \text{Pair}(\text{p}_1, \text{EP}(\text{p}_1)) \cdot \dots \cdot \text{Pair}(\text{p}_n, \text{EP}(\text{p}_n)) \})$$

where  $\text{stmts} \mapsto \text{stmt\_list}$

- A sequence of statements is a list of terms.

$$\text{stmt}_1 ; \text{stmt}_2 \mapsto \{ \text{stmt}'_1 \cdot \text{stmt}'_2 \}$$

where  $\text{stmt}_1 \mapsto \text{stmt}'_1$  and  $\text{stmt}_2 \mapsto \text{stmt}'_2$

- Expressions, appearing as right-values or as effective parameters in function calls, are not parsed.
- Assignments, **while** and **if** statements simply map to function symbols of  $\mathcal{L}$ .

$$x = y \mapsto \text{Assign}(x, y)$$

$$\mathbf{if} (c) \text{ stmts}_1 \mathbf{else} \text{ stmts}_2 \mapsto \text{If}(c, \text{stmt\_list}_1, \text{stmt\_list}_2)$$

$$\mathbf{while} (c) \text{ stmts} \mapsto \text{While}(\text{loop\_number}, c, \text{stmt\_list})$$

where  $\text{stmts}_n \mapsto \text{stmt\_list}_n$

- Variable declarations are part of the function statements. They are treated as assignments and added to the list of statements of the function. If a variable is not initialized, a default value is assigned to it depending on its type.

$$\mathbf{type} \text{ var} = \text{val} \mapsto \text{Assign}(\text{var}, \text{val})$$

$$\mathbf{int} \text{ var} \mapsto \text{Assign}(\text{var}, 0)$$

$$\mathbf{list} \text{ var} \mapsto \text{Assign}(\text{var}, \text{null})$$

- Return statements produce a pair, linking the name of the function containing the return statement and its return expression.

$$\mathbf{return} \text{ exp} \mapsto \text{Pair}(\text{func\_name}, \text{exp})$$

In the case of the identity function, the GE term consists of an instruction list whose single instruction term is  $\text{Return}(\text{identity}, x)$ , and an initial environment  $\{ \text{Pair}(x, \text{EP}(x)) \}$  since  $x$  is the unique function parameter.

#### 4.2 Terms rewrite into Environments

At the second step, each term  $T_P^f$  is *rewritten*, according to  $\mathcal{R}$ , into an environment  $E_{T_P^f}$ . Intuitively, this environment contains all the information about the variables of function  $f$  and their corresponding expressions — the evaluation of which yields the value of  $f$  in an execution of  $P$ . In terms of input/output, we have:

**Input** : A term over  $\mathcal{L}$ .

**Output** : An  $\mathcal{R}$ -normalized term, called environment.

This second step is a semantic evaluation of  $P$ . The rules of  $\mathcal{R}$  express the *operational semantics* of the  $\text{sub}_C$  language. The term  $T_P^f$  produced by the program parsing is normalized according to  $\mathcal{R}$  rules, that is, the term is rewritten until no more rule can be applied. The resulting term is a  $\text{sub}_C$  function environment.

Function statements are executed in an order that depends on the control flow statements and their associated conditions. These different possible orderings constitute the *execution paths* of a function. The environment produced by

rewriting represents the distinct execution paths of a function, along with their associated conditions and the final variable state. Variable states are represented by lists of terms `Pair(variable, expression)`.

#### 4.2.1 *sub<sub>c</sub> semantics*

For most part, the `subc` semantics does not require much explanation since it behaves as one could expect from a “standard” imperative language. For example, in a sequence of statements, each statement is executed one after the other (see rules 1 and 2 in appendix B); or an assignment modifies the value of a variable (see rules 3, 4 and 14 to 17 in appendix B). Still, some constructs have a specific meaning as we are going to see below and particularly in the two paragraphs dedicated to the *conditional* and *iterative* statements.

*Parameters* of a function behave like function local variables to which are assigned effective parameters. The actual value of an effective parameter is not known of course, but we denote it by an EP term. The *function result* is a local variable whose name is the function name. Assignments to this particular variable are done through the return instruction. *Local variable declarations* are assignments to a default or an explicit value.

*Conditional Statements.* An *if* statement splits an execution path into two parts: function statements are divided into those executed when the condition is true and those executed when the condition is false. Statements following an *if* statement are executed in both cases. This defines two paths, which are enclosed in a Choice term. Each *if* alternative is included in a Branch term, which associates a condition to a variable state. This is expressed by rule 7. For example, the following function defines two execution paths, one for the *then\_part* and one for the *else\_part*:

---

```
int alternative () {
  if (cond)
    then_part
  else
    else_part
  ...
}
```

---

Therefore, the environment will contain the term:

`Choice(Branch(cond, then_part), Branch(Not(cond), else_part))`.

*Iterative Statements.* We now turn our attention to the iterative construct *while*. The semantics of *while* statements is more complicated. Indeed, each loop is considered as a family of separate recursive functions, each function

having its own parameters and own body. The idea is that a loop is a function that calls itself recursively with the value of the variables modified accordingly to the loop instructions. But a function can only return a single value, and yet several variables can be modified by a loop, so, for each variable modified by the loop instructions, such a loop function is defined. Its return value is the modified variable one: as a consequence we get a family of functions. In addition, since any function variable may be used inside the loop body, the loop function takes all the variables as parameters. Then, in the function where the loop occurs, the value of a variable  $x$ , if  $x$  is modified by the loop instructions, is the result of a call to the loop function defined for  $x$ . Consequently, when a loop is encountered, the environment is modified as follows:

- A new LT term containing all the information needed to generate the loop functions is created and added to the current environment without altering it. The information is the loop number, the exit condition, the statements of the loop body and the list of all the function variables. This loop term will be used at the third step to generate a family of equations (see section 4.3).
- Each variable modified by the loop instructions is assigned a call to the corresponding loop function. This loop function takes as argument the current state of the variables in the calling function.

The following example shows how iterative statements are handled. Let us suppose a  $\text{sub}_c$  function declares three variables —  $x, y, z$  — two of which are modified in a loop body :

---

```

int f ( ) {
    int x, y, z;

    x = 1; y = 2; z = 3;

    while (y > 0) {
        x = x + z;
        y = y - 1;
    }
    ...
}

```

---

During rewriting of the term corresponding to function  $f$ , when the *while* statement is encountered, the following loop term is created:

$$\text{LT}(1, y > 0, \text{GE}(\text{loop\_body}, \text{initial\_environment}), (x, y, z)).$$

We find in LT the loop number, the loop condition, a GE term, which means that a new environment will be evaluated for the loop body, and a list of all the variables of function  $f$ . `loop_body` is composed of the loop instructions:

they are the two assignments modifying  $x$  and  $y$ . `initial_environment` is the environment in which loop instructions will be evaluated: it is the list of pairs  $(x, EP(x))$ ,  $(y, EP(y))$  and  $(z, EP(z))$ , one for each variable of  $f$ . Once the GE term is rewritten, we have:

$$LT(1, y > 0, \{Pair(z, EP(z)) \cdot Pair(x, EP(x)+EP(z)) \cdot Pair(y, EP(y)-1)\}, (x,y,z)).$$

At the third step of the process, this will lead to the definition of two functions,  $LOOP_x^1$  and  $LOOP_y^1$ , one for each variable modified inside the loop.

$$\begin{cases} y > 0 \Rightarrow LOOP_x^1(x, y, z) = LOOP_x^1(x + z, y - 1, z) \\ \text{not}(y > 0) \Rightarrow LOOP_x^1(x, y, z) = x \end{cases}$$

$$\begin{cases} y > 0 \Rightarrow LOOP_y^1(x, y, z) = LOOP_y^1(x + z, y - 1, z) \\ \text{not}(y > 0) \Rightarrow LOOP_y^1(x, y, z) = y \end{cases}$$

In addition, the pairs

$$Pair(x, LOOP_x^1(\text{variable\_current\_state}))$$

and

$$Pair(y, LOOP_y^1(\text{variable\_current\_state})),$$

update the environment of function  $f$  to reflect variables  $x$  and  $y$  new state. `Variable_current_state` refers to the state of the variables of  $f$  just before the *while* statement. This is just like replacing the loop in function  $f$  by the assignments and function calls:  $x = LOOP_x^1(1, 2, 3)$  and  $y = LOOP_y^1(1, 2, 3)$ .

#### 4.2.2 Rewrite system convergence

In order to be sure that *every*  $sub_C$  program always rewrites in a *unique* normal form, we must prove that the rewrite system is *convergent*. Convergence is equivalent to termination and confluence. Convergence means that our system is able to provide a unique equational formulation for  $sub_C$  programs.

A rewrite system *terminates* if the rewriting process eventually ends for any input term. It means that every term of  $\mathcal{L}$  has at least one  $\mathcal{R}$ -normal form. We show that the rewrite system terminates by exhibiting a mapping  $\phi$ , from elements of the rewrite system  $(\mathcal{L}, \rightarrow)$  to elements of a system  $(\mathcal{E}, >)$ , where  $>$  is a well-founded order, and such that if  $x \rightarrow x'$  then  $\phi(x) > \phi(x')$ . Since  $>$  is well-founded there cannot be an infinite chain  $\phi(x_0) > \phi(x_1) > \dots$  and so it is for  $x_0 \rightarrow x_1 \rightarrow \dots$ . Therefore the rewrite process terminates. For the

proof, we define a well-founded order relation  $>$  on function symbols of  $\mathcal{L}$  and we extend it to terms through a *lexicographic path order*  $>_{\text{lex}}$ . We then verify that  $s \rightarrow t \Rightarrow s >_{\text{lex}} t$ .

A rewrite system is *confluent* if whenever two rules can be applied to the same term, the result, after some rewritings, is identical whichever rule was applied initially. It means that if there exists a normal form, then it is unique. Suppose two rules  $l_1 \rightarrow r_1$  and  $l_2 \rightarrow r_2$ . If the subterm of  $l_1$  at position  $p$  can be unified with  $l_2$  by a substitution  $\theta$ , then both rules can be applied to an instance of term  $l_1\theta$ . In this case,  $r_1\theta$  and  $(l_1\theta)[r_2\theta]_p$ <sup>3</sup> constitute a *critical pair*. A terminating rewrite system is confluent if all its critical pairs are joinable, that is, if the two terms of a critical pair rewrite to a same term. This property can be shown with the help of the Knuth–Bendix completion algorithm [14]. This algorithm computes all critical pairs of the system and verifies they are joinable. If the completion algorithm terminates successfully, then a terminating rewrite system is convergent.

We used RRL, the *Rewrite Rule Laboratory* [12], to prove the convergence of  $\mathcal{R}$ . Indeed, RRL implements a completion algorithm and allows the user to define order relations. So, we defined a lexicographic path order on the terms of  $\mathcal{L}$ , to prove the termination, and successfully applied the completion algorithm on  $\mathcal{R}$ , thus proving the confluence. The trace of the RRL session proving our rewrite system is convergent can be found in appendix C.

### 4.3 Environments generate Equations

This is the third and last step of the axiomatization process. A set of equation is generated from each environment  $E_{T_P}$ . These equation sets define the *algebraic semantics* of program  $P$ . The problem definition for this step is:

**Input** : An environment.  
**Output** : A set of equation.

Only a few elements in environments will generate equations: the *equation generators*. This final step refines environments, extracts equation generators from environments and generates the corresponding equations.

First, environments are made clearer through evaluation of the following terms:

- **Subst terms.**  $\text{Subst}(x, \text{exp}_1, \text{exp}_2)$  denotes the substitution of expression  $\text{exp}_2$  for variable  $x$  in expression  $\text{exp}_1$ . The substitution is simply applied. Note that no substitution is done if  $x$  occurs in term  $\text{EP}(x)$  — since this spe-

<sup>3</sup> This is  $l_1\theta$  where subterm at position  $p$  is replaced by  $r_2\theta$ .

cial term is precisely introduced to distinguish formal parameter  $x$ , which can be modified, and its effective value, which cannot.

- EP terms. They are not necessary anymore since the distinction between effective and formal parameters is only required for substitutions.  $EP(x)$  is replaced by  $x$ .
- Loop terms. They undergo a syntactical transformation:  $Loop(num, variable, \{ exp_1 \cdot \dots \cdot exp_n \})$  is replaced by:  $LOOP_{variable}^{num}(exp_1, \dots, exp_n)$ .

Then equation generators are transformed into equations. The equation generators are:

- *lists of pair*. They represent the variable state at the end of an abstract computation. But, only the function return value is interesting, therefore, only the pair containing the function name will generate an equation.

Generator :  $\{ Pair(\dots) \cdot \dots \cdot Pair(function\_name, expression) \}$

Equation :  $function\_name = expression$

- *Branch* terms. They appear because of an *if* statement and represent an alternative. They link a condition and a list of pair. Again, only the pair with the function name is of interest. Each *Branch* term generates one conditional equation.

Generator :

Branch(condition,  $\{ Pair(\dots) \cdot \dots \cdot Pair(function\_name, expression) \}$ )

Equation :

$condition \Rightarrow function\_name = expression$

- *LT* terms. They generate a family of conditional equations that defines recursively the loop functions — one loop function for each variable modified by the loop instructions. Two equations are needed:
  - (1) One equation for the recursive call. The variable state is modified accordingly to the statements of the loop body. If  $exp_i$  is the value of variable  $v_i$  after an abstract computation of the loop instructions, then, if the condition is true, the loop function is called again with the value  $exp_i$  for the parameter  $v_i$ .
  - (2) One equation for the exit case, when the *while* condition is false. This equation gives the result of the loop function, that is the current value of the considered modified variable. This current value was passed as parameter to the loop function.

Generator :

$LI(n, cond, \{ Pair(v_1, exp_1) \cdot \dots \cdot Pair(v_n, exp_n) \}, \{ v_1 \cdot \dots \cdot v_n \})$

Equation :

$$j \in \left\{ \begin{array}{l} \cup \\ \text{modified} \\ \text{variables} \end{array} \right\} \left\{ \begin{array}{l} cond \Rightarrow LOOP_{v_j}^n(v_1, \dots, v_n) = \\ \quad LOOP_{v_j}^n(exp_1, \dots, exp_n) \\ \text{not}(cond) \Rightarrow LOOP_{v_j}^n(v_1, \dots, v_n) = v_j \end{array} \right\}$$

Here,  $v_j$  denotes a variable modified in the loop body and  $v_1, \dots, v_n$  are all the variables appearing in the  $sub_c$  function. A variable  $v$  is known to have been modified when its value differs from  $EP(v)$ , which is the value assigned to it before getting into the loop.

## 5 Extended Examples

This section goes over the axiomatization process again, illustrating how its three steps apply to two case studies.

### 5.1 Insertion Sort

A  $sub_c$  version of *insertion sort* will serve as support (see appendix A for some other examples) for this first case study:

---

```

list ISort (list L) {
  list ret = NULL;
  if (L == NULL)
    ret = NULL;
  else
    ret = ins(getHead(L), ISort(getQueue(L)));
  return ret;
}

list ins (int e, list L) {
  list ret = NULL;
  if (L == NULL)
    ret = cons(e, NULL);
  else if (e <= getHead(L))
    ret = cons(e, L);
  else {
    ret = ins(e, getQueue(L));
  }
}

```

```

    ret = cons(getHead(L), ret);
  }
  return ret;
}

```

---

This program contains two functions. Function `ins` takes an integer `e` and a sorted list `L` and returns a new sorted list containing `e`. `ISort` takes a list `L` as argument and returns a sorted version of `L` by inserting at the proper position (call to function `ins`) the first element of `L` in the already sorted queue of `L`.

### 5.1.1 Programs are Terms

In the first step, each function of the *sorting program* gives a term over  $\mathcal{L}$ . The two functions, `ISort` and `ins`, are treated separately.

The term obtained at the end of this first step for function `ISort` is :

$$\begin{aligned}
 GE(\{ & \text{Assign}(\text{ret}, \text{NULL}) \cdot \\
 & \text{If}(\text{L} = \text{NULL}, \\
 & \quad \{ \text{Assign}(\text{ret}, \text{NULL}) \}, \\
 & \quad \{ \text{Assign}(\text{ret}, \text{ins}(\text{getHead}(\text{L}), \text{ISort}(\text{getQueue}(\text{L}))) \} ) \cdot \\
 & \text{Return}(\text{ISort}(\text{L}), \text{ret}) \} , \\
 & \{ \text{Pair}(\text{L}, \text{EP}(\text{L})) \} )
 \end{aligned}$$

The  $GE$  term can be identified, with its list of instruction and its initial environment. The instruction list is made up of the statements of the source function:

- an *assignment* coming from the initialization of the variable `ret` during its declaration;
- an *If* term including the *if condition* and two instruction lists:
  - one for the *then part*, made up of the assignment of `NULL` to `ret`;
  - one for the *else part*, made up of the assignment of the result of function `ins` to variable `ret`;
- a *Return* term where the function name and parameters appear.

The initial environment is one pair, `Pair(L, EP(L))`, associating the only parameter of the function, `L`, and the value `L` will take at the function call — its effective value denoted by `EP(L)`.

Likewise, the term that corresponds to the `ins` function is :

$$\begin{aligned}
 GE(\{ & \text{Assign}(\text{ret}, \text{NULL}) \cdot \\
 & \text{If}(\text{L} = \text{NULL}, \\
 & \quad \{ \text{Assign}(\text{ret}, \text{cons}(\text{e}, \text{NULL})) \} , \\
 & \quad \{ \text{If}(\text{e} \leq \text{getHead}(\text{L}),
 \end{aligned}$$

```

    { Assign(ret, cons(e, L)) },
    { Assign(ret, ins(e, getQueue(L))) .
      Assign(ret, cons(getHead(L), ret)) } )
  } ) .
  Return(ins(e, L), ret) },
{ Pair(L, EP(L)) . Pair(e, EP(e)) } )

```

### 5.1.2 Terms rewrite into Environments

At the second step, each term previously produced is rewritten, using the rewrite system, into a final environment. Here is the environment of `ISort`, once refined for readability — as explained in section 4.3.

```

Choice(
  ( Branch(L=NULL,
    { Pair(L, L) . Pair(ret, NULL) . Pair(ISort(L), NULL)})),
  ( Branch(L ≠ NULL,
    { Pair(L, L) .
      Pair(ret, ins(getHead(L), ISort(getQueue(L)))) .
      Pair(ISort(L), ins(getHead(L), ISort(getQueue(L))))})))

```

Function `ISort` includes one **if** statement, so we find in the environment a `Choice` term composed of the two alternatives: the two `Branch` terms. These latter terms partition the instructions of the function between those executed when `L` is equal to `NULL` and those executed when `L` is different from `NULL`. In each `Branch` term there is a list of `pair`, which represents the state of the variables at the end of an abstract computation of function `ISort`. For instance, when `L = NULL`, `Pair(L, L)` means that `L` is not modified by the function; `Pair(ret, NULL)` means that the value of `ret` is `NULL`; the `Pair` term containing the function name, `Pair(ISort(L), NULL)` means that the function return value is `NULL`.

Likewise, here is the environment of the `ins` function. In this function, one **if** statement is embedded in the `else` part of another **if** statement, consequently we find two `Choice` terms. Conditions of the nested alternatives were gathered in a conjunction.

```

Choice(
  ( Branch(L = NULL,
    { Pair(L, L) . Pair(e, e) . Pair(ret, cons(e, NULL)) .
      Pair(ins(e, L), cons(e, NULL)) } ) ),
  ( Choice(
    ( Branch(L ≠ NULL and e ≤ getHead(L),
      { Pair(L, L) . Pair(e, e) .
        Pair(ret, cons(e, L)) } )

```

```

    Pair(ins(e, L), cons(e, L)) } ) ),
( Branch(L ≠ NULL and e > getHead(L),
{ Pair(L, L) · Pair(e, e) ·
  Pair(ret, cons(getHead(L), ins(e, getQueue(L)))) ·
  Pair(ins(e, L), cons(getHead(L), ins(e, getQueue(L)))) }
) ) ) ) )

```

### 5.1.3 Environments generate Equations

At the equation generation step, environments are parsed for *equation generators*. The `ISort` environment is made up of the following *equation generators*:

- Branch term with condition  $L = \text{NULL}$ ;
- Branch term with condition  $L \neq \text{NULL}$ .

Finally, these two *equation generators* give the following equations that define the algebraic semantics of the function `ISort`.

$$L = \text{NULL} \Rightarrow \text{ISort}(L) = \text{NULL}$$

$$L \neq \text{NULL} \Rightarrow \text{ISort}(L) = \text{ins}(\text{getHead}(L), \text{ISort}(\text{getQueue}(L)))$$

Similarly, the *equation generators* for the `ins` function are the three Branch terms and give the following conditional equations.

$$L = \text{NULL} \Rightarrow \text{ins}(e, L) = \text{cons}(e, \text{NULL})$$

$$L \neq \text{NULL} \text{ and } e \leq \text{getHead}(L) \Rightarrow \text{ins}(e, L) = \text{cons}(e, L)$$

$$L \neq \text{NULL} \text{ and } e > \text{getHead}(L) \Rightarrow$$

$$\text{ins}(e, L) = \text{cons}(\text{getHead}(L), \text{ins}(e, \text{getQueue}(L)))$$

Union of the two sets of equation constitutes the algebraic semantics of the entire `subc sorting program`.

## 5.2 Sum of the First Integers

Let us suppose that someone wants to compute the sum of the first  $n$  integers and writes the following program:

---

```
int sum (int n) {
  int ret = 0;

  while (n != 0) {
    n = n - 1;
    ret = ret + n;
  }

  return ret;
}
```

---

This program consists in a main **while** loop where variable  $n$  decreases up to 0 and variable  $ret$  is used as an accumulator of the successive values of  $n$ .

At the end of the first step, the program is translated into the following term:

$$\text{GE}(\{ \text{Assign}(ret, 0) \cdot \\ \text{While}(1, n \neq 0, \{ \text{Assign}(n, (n - 1)) \cdot \\ \text{Assign}(ret, (ret + n)) \}) \cdot \\ \text{Return}(\text{sum}(n), ret) \}, \\ \{ \text{Pair}(n, \text{EP}(n)) \} )$$

This term contains a **While** term. Since it is the first **while** of the program, it is given the number 1. Then, follow the condition and the statements of the loop body.

The environment obtained after rewriting of the previous term is :

$$\{ \text{LT}(1, n \neq 0, \{ \text{Pair}(n, (n-1)) \cdot \text{Pair}(ret, (ret+(n-1))) \}, \\ \{ n \cdot ret \}) \cdot \\ \text{Pair}(n, \text{LOOP}(1, n, \{ n \cdot 0 \})) \cdot \\ \text{Pair}(ret, \text{LOOP}(1, ret, \{ n \cdot 0 \})) \cdot \\ \text{Pair}(\text{sum}(n), \text{LOOP}(1, ret, \{ n \cdot 0 \})) \}$$

The **While** term was rewritten into an **LT** term. **LT** contains the result of the evaluation of the loop body statements in a new environment. This will lead to the definition of a new function **LOOP**. The expressions of the variables modified in the loop are now expressed as calls to this new **LOOP** function.

Finally, the third step gives the following equations :

$$\begin{aligned}
n \neq 0 &\Rightarrow \text{LOOP}_{ret}^1(n, ret) = \text{LOOP}_{ret}^1((n - 1), (ret + (n - 1))) \\
n = 0 &\Rightarrow \text{LOOP}_{ret}^1(n, ret) = ret \\
\text{sum}(n) &= \text{LOOP}_{ret}^1(n, 0)
\end{aligned}$$

Thanks to these equations we can show that the source program computes the sum of the first  $n - 1$  integers and not the sum of the first  $n$  integers.

## 6 Conclusion

In this paper we have discussed a method to obtain an equivalent equational formulation of a program from source code. Thereby programs can be understood as formalized logical systems. This allows to reason about programs from equations rather than from source code, which is more natural and more efficient. Indeed, there exist powerful tools dealing with equations.

The program to be translated is written in a language with imperative features and there is no need for program annotations. The axiomatization, that is, the process leading to the equations, is automatic and requires no user interactions. It is done in three steps: a syntactic analysis, then a semantic analysis and finally a translation into an equational language. The main point of the discussed method is the generation of an environment using a rewrite system. The rewrite system implements the operational semantics of the source language. The first stage consists in building a term suitable for rewriting through the syntactic analysis of the program code. The last stage consists in translating environments information into equations. We showed in this paper that our method can translate  $\text{sub}_c$  programs into equations. An implementation of the axiomatization has been carried out in Java, thus proving that the process is fully automatic. JavaCC<sup>4</sup>, a parser and scanner generator, has been used for the term generation step. We developed a Java version of a generic rewriting algorithm. The rewrite rules are loaded separately from a file so as to elaborate the rules with ease.

We present in appendix some interesting experiments on our system (simple algorithms about trees and graphs), which encourages us to continue our work in the following directions:

---

<sup>4</sup> Java Compiler Compiler, Metamata.

- adding functionalities to  $\text{sub}_C$  in order to come closer to real imperative languages;
- our approach for program analysis strongly relies on equational tools, which are still actively developed, so we need to investigate how our equations are handled by proof systems. This implies:
  - implementing interfaces towards proof systems, that is, providing the equations in the specific system syntax;
  - experimenting on a larger scale proving properties from the equations in proof systems — this in order to identify a class of properties and programs that can be proven sound using our method;
  - enhancing existing proof systems, especially ours, to increase the class of provable properties.

## A Examples

This appendix presents some examples of  $\text{sub}_C$  programs and the equations produced by our axiomatization process.

### A.1 Binary Search Tree

Here are the code and equations of a  $\text{sub}_C$  program that searches element  $e$  in a binary search tree. Function `BS` returns 1 if element  $e$  is found in `tree` and 0 otherwise. A tree node is a list made up of an integer and two other nodes for the left and right children (*i.e.*  $\{ \text{root} \cdot \text{left child} \cdot \text{right child} \}$ ).

---

```

int root (list tree) { return getHead(tree); }
list lc (list tree) { return getHead(getQueue(tree)); }
list rc (list tree) {
    return getHead(getQueue(getQueue(tree)));
}
int bs (int e, list tree) {
    int ret;
    if (tree == NULL)
        ret=0;
    else if (e == root(tree))
        ret = 1;
    else if (e < root(tree))
        ret = bs(e,lc(tree));
    else
        ret = bs(e,rc(tree));
    return ret;
}

```

---

Equations produced by our system:

$$\begin{aligned}bs(e, \emptyset) &= 0 \\bs(e, [ e \cdot lc \cdot rc ]) &= 1 \\e < r \Rightarrow bs(e, [ r \cdot lc \cdot rc ]) &= bs(e, lc) \\e > r \Rightarrow bs(e, [ r \cdot lc \cdot rc ]) &= bs(e, rc)\end{aligned}$$

## A.2 Depth-First Search

Here are the code and equations of a  $\text{sub}_c$  program that goes through all the vertices of a graph using a depth-first search. Graph vertices are integers. We use a list of adjacency lists to represent the graph. Element at position  $p$  in this list is the list of all the vertices connected to vertex  $p$ . Function `member` returns 1 if element  $e$  is in list  $L$  and 0 otherwise. Function `adj` returns the list of vertices adjacent to vertex  $v$  in the list of adjacency lists `listadj`.

---

```
int member (int e, list L) {
  int ret;
  if (L == NULL)
    ret = 0;
  else if (e == getHead(L))
    ret = 1;
  else
    ret = member(e, getQueue(L));
  return ret;
}
list adj (int v, list listadj) {
  list ret;
  if (v == 1)
    ret = getHead(listadj);
  else
    ret = adj(v-1, getQueue(listadj));
  return ret;
}
int dfs (list vertices, list marked,
         list listadj) {
  int ret, v;
  if(vertices == NULL)
    ret = 1;
  else {
    v = getHead(vertices);
    ret = depth(vertices, adj(v, listadj),
```

```

        cons(v, marked), listadj);
    }
    return ret;
}
int depth (list vertices, list adjacents, list marked,
           list listadj) {
    int ret, v;
    if(adjacents == NULL)
        ret = dfs(getQueue(vertices), marked, listadj);
    else {
        v = getHead(adjacents);
        if(member(v, marked) == 1)
            ret = depth(vertices, getQueue(adjacents),
                        marked, listadj);
        else
            ret = dfs(cons(v, vertices), marked, listadj);
    }
    return ret;
}
}

```

---

Equations produced by our system:

$$\text{member}(e, \emptyset) = 0$$

$$\text{member}(e, [ e \cdot L ]) = 1$$

$$e \neq c \Rightarrow \text{member}(e, [ c \cdot L ]) = \text{member}(e, L)$$

$$\text{adj}(1, [ L_1 \cdot L ]) = L_1$$

$$\text{adj}(n + 1, [ L_1 \cdot L_2 \cdot L ]) = \text{adj}(n, [ L_2 \cdot L ])$$

$$\text{depth}([ v \cdot L ], \emptyset, M, LA) = \text{dfs}(L, M, LA)$$

$$\text{member}(v, M) = 1 \Rightarrow \text{depth}(L, [v \cdot L_1], M, LA) = \text{depth}(L, L_1, M, LA)$$

$$\text{member}(v, M) = 0 \Rightarrow \text{depth}(L, [v \cdot L_1], M, LA) = \text{dfs}([v \cdot L_1], M, LA)$$

$$\text{dfs}(\emptyset, M, LA) = 1$$

$$\text{dfs}([v \cdot L], M, LA) = \text{depth}([v \cdot L], \text{adj}(v, LA), [v \cdot M], LA)$$

## B Rewrite System

- (1)  $GE(Cons\_inst(L\_in, inst), env) \rightarrow Comp(inst, GE(L\_inst, env))$
- (2)  $GE(End\_inst, env) \rightarrow env$
- (3)  $Comp(Assign(var, exp), Empty\_env) \rightarrow Cons\_env(Pair(var, exp), Empty\_env)$
- (4)  $Comp(Assign(var, exp), Cons\_env(pair, env)) \rightarrow$   
 $Update\_env(Pair(var, exp), Cons\_env(pair, env))$
- (5)  $Comp(Return(fct, exp), Cons\_env(pair, env)) \rightarrow$   
 $Update\_env(Pair(fct, exp), Cons\_env(pair, env))$
- (6)  $Comp(Return(fct, exp), Empty\_env) \rightarrow Cons\_env(Pair(fct, exp), Empty\_env)$
- (7)  $Comp(If(cond, L\_inst1, L\_inst2), Cons\_env(pair, env)) \rightarrow$   
 $Choice( Branch( Update\_cond(cond, Cons\_env(pair, env)),$   
 $GE(L\_inst1, Cons\_env(pair, env))),$   
 $Branch( not(Update\_cond(cond, Cons\_env(pair, env))),$   
 $GE(L\_inst2, Cons\_env(pair, env))))$
- (8)  $Comp(While(num, cond, L\_inst), Cons\_env(pair, L\_pair)) \rightarrow$   
 $Cons\_env($   
 $LT(num, cond, GE(L\_inst, GIE(Cons\_env(pair, L\_pair))),$   
 $GLOV(Cons\_env(pair, L\_pair))),$   
 $Merge\_env(GL (num,$   
 $GLOMV(GE(L\_inst, GIE(Cons\_env(pair, L\_inst))))),$   
 $GLOE(Cons\_env(pair, L\_pair))), Cons\_env( pair, L\_pair))$
- (9)  $Comp(inst, Choice(exp1, exp2)) \rightarrow Choice(Comp(inst, exp1), Comp(inst, exp2))$
- (10)  $Comp(inst, Branch(cond, env)) \rightarrow Branch(cond, Comp(inst, env))$
- (11)  $Update\_cond(cond, Cons\_env(Pair(var, exp), env)) \rightarrow$   
 $Update\_cond(Subst(var, cond, exp), env)$
- (12)  $Update\_cond(cond, Empty\_env) \rightarrow cond$
- (13)  $Update\_cond(cond, Cons\_env(LT(...), env)) \rightarrow Update\_cond(cond, env)$
- (14)  $Update\_env(Pair(x, exp1), Cons\_env(Pair(x, exp2), env)) \rightarrow$   
 $Update\_env(Pair(x, Subst(x, exp1, exp2)), env)$
- (15)  $Update\_env(Pair(x, exp1), Cons\_env(Pair(y, exp2), env)) \rightarrow$   
 $Cons\_env(Pair(y, exp2), Update\_env(Pair(x, Subst(y, exp1, exp2)), env))$   
 $if\ not\ equal(x, y)$
- (16)  $Update\_env(Pair(var, exp), Empty\_env) \rightarrow Cons\_env(Pair(var, exp), Empty\_env)$
- (17)  $Update\_env(Pair(var, exp1), Cons\_env(LT(...), env)) \rightarrow$



```

Cons_env(Pair(y, exp2), Insert_pair(Pair(x, exp1), env))
                                if not equal(x, y)
(47) Insert_pair(Pair(x, exp), Empty_env) → Cons_env(Pair(x, exp), Empty_env)

```

## C Convergence proof

This appendix reproduces, in part, the interaction with RRL that leads to the proof of the convergence of our rewrite system. First, we define a partial order on the functions of the rewrite system language. Then, with the *MAKERULE* command, we let RRL orient the equations with respect to the *lexicographic path order* based on the order we defined. Next, we use the *KB* command to begin the completion algorithm. Once the algorithm completed, we have confirmation that all the *critical pairs* are joinable. Hence, our rewrite system is convergent. We renamed all function symbols and variables, by beginning them by a “F” and a “V” respectively, to satisfy the RRL syntax.

```

RRL-> read
Type filename: system
Using Conditional Rewriting Method ...

'FAND' is associative & commutative now.
FGE and FCOMP made equivalent.
Operator, FGE, given status: LR
Operator, FUPDATE_ENV, given status: RL
Operator, FUPDATE_COND, given status: RL
Operator, FBRANCH, given status: RL

FCOMP > FCONS_ENV  FCOMP > FPAIR  FCOMP > FUPDATE_ENV
FCOMP > FCHOICE  FCOMP > FBRANCH  FCOMP > FNOT
FCOMP > FUPDATE_COND  FCOMP > FWC  FCOMP > FGLOV
FCOMP > FMERGE_ENV  FCOMP > FGL  FCOMP > FGLQMV
FCOMP > FGIE  FCOMP > FGLOE
FUPDATE_COND > FSUBST  FUPDATE_ENV > FPAIR  FUPDATE_ENV > FSUBST
FUPDATE_ENV > FCONS_ENV
FBRANCH > FCHOICE  FBRANCH > FAND
FGIE > FCONS_ENV  FGIE > FPAIR  FGIE > FEA
FGLOV > FCONS_VAR
FGLQMV > FCONS_VAR  FGLQMV > FMERGE_L_VAR
FMERGE_L_VAR > FINSERT_VAR
FINSERT_VAR > FCONS_VAR
FGLOE > FCONS_EXP
FGL > FCONS_ENV  FGL > FPAIR  FGL > FLOOP  FGL > FCONS_EXP

```

```
FMERGE_ENV > FINSERT_PAIR
FINSERT_PAIR > FCONS_ENV  FINSERT_PAIR > FWC
FINSERT_PAIR > FPAIR
```

```
RRL-> MAKERULE
```

```
...
```

```
RRL-> KB
```

```
...
```

```
Your system is possibly canonical.
```

```
...
```

```
Number of rules generated      = 47
Number of rules retained       = 47
Number of critical pairs       = 57
Number of unblocked critical pairs = 0
```

## References

- [1] Ponsini O., Fédèle C., Kounalis E., Sos *C*—: A system for interpreting operational semantics of *C*— programs, in: Proceedings of the IASTED International Conference on Applied Informatics, 2002.
- [2] Fédèle C., Kounalis E., Automatic proofs of properties of simple *C*— modules, in: 14th IEEE International Conference on Automated Software Engineering, 1999.
- [3] Kounalis E., Urso P., Generalization discovery for proofs by induction in conditional theories, in: Proceedings of 12th International FLAIRS, AAAI Press, 1999.
- [4] Barras B., Boutin S., Cornes C., Courant J., Filliâtre J.C., Giménez E., Herbelin H., Huet G., Munoz C., Murthy C., Parent C., Paulin C., Saïbi A., Werner B., The Coq Proof Assistant Reference Manual – Version V6.1, Tech. Rep. 0203, INRIA, <http://pauillac.inria.fr/coq/coq-fra.html> (August 1997).
- [5] Traynor O., Hazel D., Kearney P., Martin A., Nickson R., Wildman L., The Cogito development system, in: Johnson M. (Ed.), Algebraic Methodology and Software Technology (AMAST), Berlin, Vol. 1349 of LNCS, Springer-Verlag, 1997, pp. 586–591.
- [6] Juellig R., Srinivas Y., Liu J., SPECWARE: An advanced environment for the formal development of complex software systems, Lecture Notes in Computer Science 1101 (1996) 551.
- [7] Filliâtre J.C., Proof of imperative programs in type theory, in: Altenkirch T., Naraschewski W., Reus B. (Eds.), Types for Proofs and Programs, Vol. 1657 of Lecture Notes in Computer Science, Springer-Verlag, 1998, p. 78.

- [8] Antoy S., Gannon J., Using Term Rewriting to Verify Software, *IEEE Transactions on Software Engineering* 20 (4) (1994) 259–274.
- [9] Ward M., Abstracting a specification from code, *Journal of Software Maintenance: Research and Practice* 5 (2) (1993) 101–122.  
URL <http://ws-mj4.dur.ac.uk/martin/papers/prog-spec.ps.gz>
- [10] Schweizer D., Denzler C., Verifying the specification-to-code correspondence for abstract data types, in: Dal Cin M., Meadows C., Sanders W. (Eds.), *Dependable Computing for Critical Applications 6*, Vol. 11 of *Dependable Computing and Fault-Tolerant Systems*, IEEE Computer Society Press, 1997.
- [11] Gar S. J., Guttag J. V., A guide to LP, the larch prover, Tech. Rep. 82, Digital Equipment Corporation, Systems Research Centre (31 Dec. 1991).
- [12] Kapur D., Zhang H., RRL : Rewrite Rule Laboratory (May 1989).
- [13] Baader F., Nipkow T., *Term Rewriting and All That*, Cambridge University Press, 1998.
- [14] Knuth D. E., Bendix P. B., *Simple word problems in universal algebras*, Computational Problems in Abstract Algebra Pergamon Press.