Rewriting of Imperative Programs into Logical Equations

Olivier Ponsini, Carine Fédèle and Emmanuel Kounalis

Laboratoire I3S - UNSA - CNRS (UMR 6070)
Les Algorithmes - 2000, route des Lucioles - B.P. 121
06903 Sophia Antipolis Cedex - France

Abstract

This paper describes $SOS\mathrm{Sub}_{\mathcal{C}}$: a system for automatically translating programs written in $\mathrm{Sub}_{\mathcal{C}}$, a simple imperative language, into a set of first-order equations. This set of equations represents a $\mathrm{Sub}_{\mathcal{C}}$ 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 $\mathrm{Sub}_{\mathcal{C}}$ program, a unique set of equations. In our work, *simple* imperative programs are equational theories of a logical system within which proofs can be derived.

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.

Email addresses: ponsini@i3s.unice.fr, Carine.Fedele@unice.fr, kounalis@i3s.unice.fr (Olivier Ponsini, Carine Fédèle and Emmanuel Kounalis).

Preprint submitted to Elsevier Science *16 November 2004*

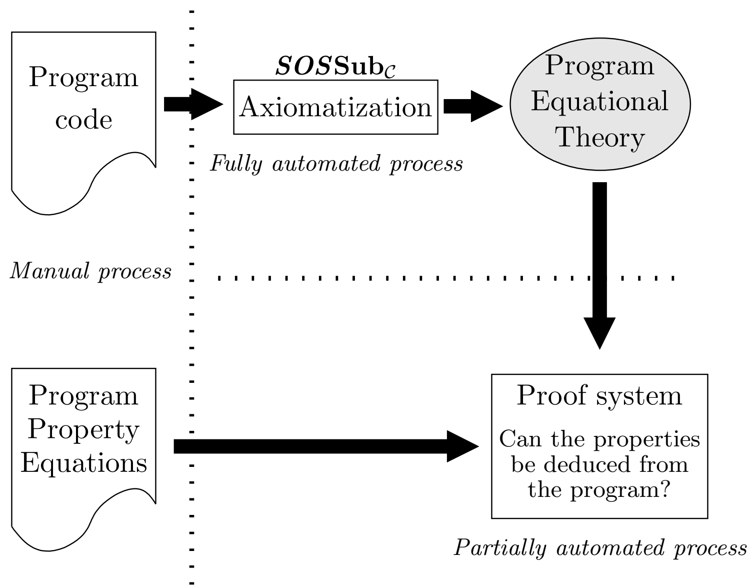

Fig. 1. Proof process overview.

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 semantics. This translation is part of a framework for automatically proving properties of programs. *Equational logic*, a subset of first-order logic, affords a simple and clear semantics (substitution of equals), as well as existing efficient algorithms and tools developed for it.

The general outline of our framework is shown in Fig. 1. Developers write down the Sub$_{\mathcal{C}}$ code of a program; they also write program specifications as a set of required properties expressed in equational logic: the *program property equations*. Then the *axiomatization* (*SOS*Sub$_{\mathcal{C}}$ system) automatically transforms the source code into a set of equations: the *program equational theory*. These equations constitute a theory within equational logic. They also can be seen as an algebraic specification. The properties to be proved are conjectured theorems. Therefore, proving these theorems from the equational theory is equivalent to proving the program meets its specifications. The program equational theory concerns the proof side of the process and is derived from

the program code. The *program specification* concerns the development side of the process and is derived from the developer's requirements. Since both the equational theory and the program specification are expressed in equational logic, the proof may be done within proof systems, automatically, using *theorem provers* able to do mathematical induction like *RRL* [3] or Spike [4], or interactively, using *proof checkers* [1] like PVS [5] or Coq [6].

In this paper, we address the following problem :

> `Axiomatization`
>
> `Input` : A Sub$_{\mathcal{C}}$ program P.
>
> `Output` : A program equational theory $< \sum_p, E_p >$.
>
> Where $\sum_p$ and E_p are, respectively, the signature and the semantics of P in equational logic.

Example 1 (list reverse) We illustrate here the proof process using a program to reverse the elements of a list. Fig. 2 is the Sub$_{\mathcal{C}}$ *program code*. A property of this program could be that the reverse of the reverse of a list is the list itself: reverse(reverse(L)) $= L$. This equation is a conjecture we must prove from the equations generated by our system (Fig. 3). Appendix C.1 discusses the proof of this conjecture in the theorem prover *RRL*.

```
list reverse (list L) {
  list W = NULL;
  while (L != NULL) {
    W = cons(car(L), W);
    L = cdr(L);
  }
  return w;
}
```

Fig. 2. Reverse of a list in Sub$_{\mathcal{C}}$.

$$L = NULL \Rightarrow \mathrm{LOOP}^1_W(L,\ W) = W$$

$$L \neq NULL \Rightarrow \mathrm{LOOP}^1_W(L,\ W) = \mathrm{LOOP}^1_W(\mathrm{cdr}(L),\ \mathrm{cons}(\mathrm{car}(L),\ W))$$

$$\mathrm{reverse}(L) = \mathrm{LOOP}^1_W(L,\ NULL)$$

Fig. 3. `reverse` equational theory as produced by *SOS*Sub$_{\mathcal{C}}$.

[1] This distinction between theorem provers and proof checkers can appear rather artificial since theorem provers often provide some kind of interactivity and proof checkers some kind of automation.

Imperative languages are widely used in the industrial world which requires simple and user-friendly tools for specification and verification. Several 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* derives programs from specifications. The specification language is a formal and high-level language 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 [7] and Specware [8]. However, this approach suffers from several drawbacks. The specification language can help in saying *what* a program must do, but often the language is not sufficient to express *how* it should be done. The generated code is not as efficient as one written by a programmer. In addition, these systems cannot be used for verifying or maintaining existing programs.
- Verification systems are the second category. They deal with source code in order to verify program properties. We find two sub-categories:
  - *Program annotation* requires that user inserts program specifications in the form of annotations directly into source code. These annotations will help the system to carry out the proof (see [9] and [10] for instance). In these frameworks, even simple properties can be difficult to prove. Moreover, these frameworks mix specifications 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 program specifications from source code and to verify them against user specifications. This kind of system needs no user interaction except, possibly, for the proof step. The common part of these methods relies on giving a meaning to programs. This can be done by formally defining the "standard" semantics of the programming language, which is the approach taken here. Another approach is to express the meaning of programs in a semantics different from the "standard" semantics. This is of particular interest in proofs of compiler correctness (see [11] for instance). The program is then transformed by applying a set of proven semantics-preserving transformations in order to do, for instance, partial evaluation or correctness proofs (this latter application is discussed in [12] or [13] for example). Another active research area introduces the notion of *categorical semantics* and *monads* to prove equivalence of programs (Moggi [14]). Monads seem well suited to express imperative properties of languages in declarative formalisms (see for instance [15], [9] or [16]).

In our approach, we work on source code written by programmers. In this way the code can be manually optimized. We do not use annotations and, thus, we distinguish the coding and specification activities. Also, a strong requirement is to automate the whole process. PESCA [17] 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 [18] theorem prover. The main differences with our work come from the restrictions the previous approaches impose on the programming language (*e.g.* no recursion or no conditional loops allowed), the method to give a semantics to programs, and the specification language.

Equational logic is an adequate formalism to achieve our goal of giving a meaning to a program by unifying two views of language semantics. Because we express programs as terms in the setting of equational logic:

(1) the model theory of this logic, algebra, gives a *denotational semantics* to the program directly without resorting to additional formalisms as traditionally done in Scott-Strachey semantics [19], LCF [20] or Hoare [21] logics;
(2) rewriting theory gives an *operational semantics* by defining how to compute values from the program.

In this respect, our approach shares foundational ideas with work done on algebraic denotational semantics in [22] or PIM's core algebraic component [23] (for a discussion of equational logic and program semantics see also [24]). However, we defined equationally the semantics of our programming language so that programs were themselves equational definitions. In this regard, we do not propose another intermediate representation of programs for further translation. As it is, equational logic is well-suited to theorem proving. Thus, we hope to alleviate substantial parts of correctness proofs.

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 for the *program towards equation* process. In particular:
  - We formulate the axiomatization mainly as a rewrite system.
  - We prove the rewrite system's *convergence* using *RRL*. Roughly speaking, this means that a $\mathrm{Sub}_{\mathcal{C}}$ program is translated into a *unique* set of equations.
  - We give a formal description of how equations are generated from environments.

(2) We developed a computer program, $SOS\mathrm{Sub}_{\mathcal{C}}$, which fully automatizes the axiomatization process. The main features of $SOS\mathrm{Sub}_{\mathcal{C}}$ are:
   - a parser and a scanner for $\mathrm{Sub}_{\mathcal{C}}$ generated by JavaCC[2] — it was used for the term generation step;
   - a Java version of a *generic* rewriting algorithm — the rewrite rules are loaded separately from a file;
   - an algorithm to generate equations from environments;
   - a set of $\mathrm{Sub}_{\mathcal{C}}$ programs.

1.3 Structure of this paper

In Section 2, we first introduce the $\mathrm{Sub}_{\mathcal{C}}$ language, and some basic definitions and notations for conditional equational logic and rewriting. Then, Section 3 gives a general outline of the $SOS\mathrm{Sub}_{\mathcal{C}}$ system. Section 4 describes in detail the steps involved in the axiomatization process. It also extensively discusses the rewrite system and its rules. It involves three steps:

(1) programs are written as terms (Section 4.1);
(2) terms rewrite into environments (Section 4.2);
(3) environments generate equations (Section 4.3).

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 proofs done with the generated equations in proof systems can be found in Appendix C.

2 Background

2.1 The $\mathrm{Sub}_{\mathcal{C}}$ language

For our experiments, we use a very simple imperative language. The $\mathrm{Sub}_{\mathcal{C}}$ syntax is similar to that of C. Fig. 4 shows the $\mathrm{Sub}_{\mathcal{C}}$ grammar given in EBNF.

The main features of the language are:

- assignments;
- functions: argument-passing by value, local variables;
- control flow statements: **`if`** . . . **`else`**, **`while`** and **`return`**;
- two predefined types: integers (**`int`**), and lists (**`list`**);

[2] Java Compiler Compiler, Metamata.

```
program ::= function*

function ::= type ident '(' ['void' | argumentList] ')' ( ';' | functionBody)

argumentList ::= argumentDeclaration (',' argumentDeclaration)*

argumentDeclaration ::= type ident

functionBody ::= '{' variableDeclaration* [statementList] returnStatement ';' '}'

variableDeclaration ::= type variableInitialisation (',' variableInitialisation)* ';'

variableInitialisation ::= ident ['=' expression]

statementList ::= statement+

statement ::= assignStatement ';'
   | whileStatement ';'
   | branchStatement ';'
   | '{' statementList '}'
   | '{' '}'

assignStatement ::= ident '=' expression

returnStatement ::= 'return' expression

whileStatement ::= 'while' '(' condition ')' statement

branchStatement ::= 'if' '(' condition ')' statement ['else' statement]

type ::= 'int' | 'list'
condition ::= a conditional expression
expression ::= an expression
ident ::= an identifier
```

Fig. 4. Sub$_{\mathcal{C}}$ EBNF grammar.

- the usual arithmetic operators;
- operators on lists:
  - **car**(L) returns the first element of the list L;
  - **cdr**(L) returns a copy of the list L without its first element;
  - **NULL** represents an empty list;
  - **cons**(e, L) returns a new list by adding the element e to the head of a copy of L.

There is no restriction on the nesting level of control flow statements, but only one **return** statement per function is allowed (at the end of the function body). Moreover, several common features in imperative languages are unavailable in Sub$_{\mathcal{C}}$:

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

Fig. 5 defines the EBNF grammar of expressions in Sub$_{\mathcal{C}}$. Expressions are

```
condition ::= andExpression ('||' andExpression)*

andExpression ::= equalExpression ('&&' equalExpression)*

equalExpression ::= relExpression (('==' | '!=') relExpression)*

relExpression ::= expression (relOperator expression)*

expression ::= mulExpression (('+' | '−') mulExpression)*

mulExpression ::= unExpression (('*' | '/') unExpression)*

unExpression ::= [unOperator] factor

factor ::= '(' expression ')' | definedName | integer | 'NULL'

definedName ::= ident ['(' [effectiveArgumentList] ')']
   | 'cons' '(' expression ',' expression ')'
   | 'car' '(' expression ')'
   | 'cdr' '(' expression ')'

effectiveArgumentList ::= expression (',' expression)*

unOperator ::= '!' | '−' | '+'

relOperator ::= '<' | '<=' | '>' | '>='

integer ::= an integer
ident ::= an identifier
```

Fig. 5. Sub$_{\mathcal{C}}$ expressions EBNF grammar.

fully parsed but we are not interested in their semantics at the translation stage. Indeed, their meaning is usually largely predefined in proof systems (*cf.* Appendix C), consequently we only need to match the syntax of a specific proof system and we rely on it for the symbol's definitions.

Example 2 The conditional expression `!a&&b==c` will be disambiguated and translated, according to *RRL* syntax for instance, into *not(a) and b = c*.

♦

2.2 Equational logic

Let F be a set of symbols called a *signature*. Each symbol f in F is called a *function symbol* and has an arity. Elements of arity zero are also 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 $f(t_1, \ldots, t_n)$ is in $T(F, X)$ whenever the arity of f is n and $t_i \in T(F, X)$ for $i \in [1..n]$.

We call a pair of two terms l and r denoted by $l = r$ an *equation* and a pair denoted by $\neg(l = r)$, also $l \neq r$, a *negative equation*.

Let $Pred$ be the set of *predicate symbols*. Each symbol *pred* in $Pred$ has an arity. *Atoms* are constants from $Pred$, equations and formulas $pred(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ ($n \geq 1$) are terms and *pred* is an n-ary predicate symbol.

An *equational program*, also *Horn clause*, is written $c \Rightarrow e$ where e is an equation and $c \equiv \bigwedge_i (\bigvee_j c_{ij})$, with atoms c_{ij}.

2.3 Rewrite systems

Rewrite systems are sets of oriented equations. For an introduction to rewrite system theory see [25] for instance.

Let $T(F, X)$ denote the set of terms built from the signature F and a set X of *variables*. If t is a term and θ 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's 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 syntactic identity the $\mathcal{R}$-normal forms of t and s (*i.e.* $\mathrm{nf}(t) \stackrel{?}{\equiv} \mathrm{nf}(s)$, where $\mathrm{nf}(t)$ (resp. $\mathrm{nf}(s)$) denotes the normal form of t (resp. s)).

3 System overview

Axiomatization is the operation that takes as input *a* $Sub_{\mathcal{C}}$ *program* and gives as output *a set of equations* semantically equivalent to the program: the result of the execution of the $\mathrm{Sub}_{\mathcal{C}}$ program with input $\mathcal{I}$ is identical, up to translation of symbols, to the result (*i.e.* normal form) 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 Fig. 6.

The central part of $SOS\mathrm{Sub}_{\mathcal{C}}$ is the rewrite system $\mathrm{SS}_{\mathcal{R}}$ ($\mathrm{Sub}_{\mathcal{C}}$ Semantics Rewrite system) . $\mathrm{SS}_{\mathcal{R}}$ gives a semantics to $\mathrm{Sub}_{\mathcal{C}}$ programs (*cf.* Section 4.2). $\mathrm{SS}_{\mathcal{R}}$ is defined over the first-order language $\mathrm{SS}_{\mathcal{L}}$ ($\mathrm{Sub}_{\mathcal{C}}$ Semantics Language). In

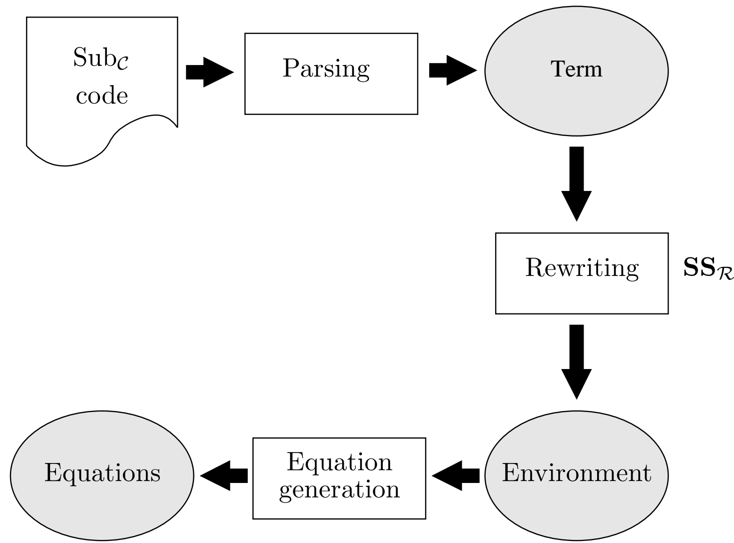

Fig. 6. Axiomatization process (*SOS*Sub$_{\mathcal{C}}$) overview.

particular, SS$_{\mathcal{L}}$ is built from a set of function symbols which are the translation of Sub$_{\mathcal{C}}$ constructs (*cf.* Section 4.1.1 for a description of SS$_{\mathcal{L}}$).

Example 3 For instance, the *assignment* statement, written `x = y` in Sub$_{\mathcal{C}}$, is translated into *C_Assign(x,y)*, where *C_Assign* belongs to the signature of SS$_{\mathcal{L}}$. Next, the semantics of an assignment is to update the current environment: this is the role of Rules (3) and (4) (*cf.* Appendix B). If the current environment were empty, Rule (3) would produce an environment with a single pair associating a variable to its value: *{C_Pair(x, y)}*.

♦

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 SS$_{\mathcal{L}}$. This term is then normalized using SS$_{\mathcal{R}}$ into a unique normal form, which is an *environment*. Roughly speaking, environments contain information about the state of the computation (*e.g.* the value of each variable and intermediate loop functions introduced by iterations) at the end of a "partial execution" of the program. We mean by "*partial execution*" an execution not done on a real input but carried out with the Sub$_{\mathcal{C}}$ function arguments as missing inputs. Finally, the *equations* are extracted from the environment.

Example 4 (identity) As an introductory example, let us see the different stages involved in the axiomatization of the very simple `identity` function as given by the following program:

```
int identity (int x) {
  return x;
}
```

At the first step, the `identity` function is transformed into a term over $SS_{\mathcal{L}}$. In the function `identity`, the only statement is a return statement. Therefore, 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 *C_Return* term: *{C_Return(identity, x)}*.

The *initial environment* of a function is a list of pairs associating $Sub_{\mathcal{C}}$ function arguments and their value when the function is called (*i.e.* the effective argument). Here, we find argument x associated with a *C_EA* term, which represents the missing value of x : *{C_Pair(x, C_EA(x))}*.

The statement list, which represents *the sequence of statements* of the source function, and the initial environment are gathered in a *GE* term:

GE({C_Return(identity, x)}, {C_Pair(x, C_EA(x))}).

At the second step, the *GE* term is rewritten by Rules (1) and (2) (Appendix B) in:

Comp(C_Return(identity, x), {C_Pair(x, C_EA(x))}).

And Rules (5), (16) and (17) generate the following environment:

{C_Pair(x, C_EA(x)) ·C_Pair(identity, C_Subst(x, C_EA(x), C_EA(x)))}.

C_EA terms have a special behavior regarding substitution: they prevent their argument from being substituted. This is consistent with these terms representing a value. The final environment is then:

{C_Pair(x, C_EA(x)) ·C_Pair(identity, C_EA(x))}.

We find in this environment two terms of type *C_Pair*. They express the value of argument x and the return value of the `identity` function at the end of the function. It shows that both values are equal to the value of x at the moment of the call.

Finally, from the $Sub_{\mathcal{C}}$ function prototype and the final environment, we extract the return value of the function and generate one equation:

$$identity(x) = x.$$

This equation expresses the semantics in equational logic of the `identity` function.

♦

4 Axiomatization description

In this section, we go over the three steps of the axiomatization in more detail. We also present formally $SS_{\mathcal{L}}$ and $SS_{\mathcal{R}}$.

4.1 Programs are terms

4.1.1 $SS_{\mathcal{L}}$

$SS_{\mathcal{L}}$ is the language over which the terms of $SS_{\mathcal{R}}$ are built. Its signature contains three types of function symbols:

- Function symbols which are used to build terms from $Sub_{\mathcal{C}}$ language constructs (*cf.* Fig. 7). They allow a term representation of $Sub_{\mathcal{C}}$ programs. As explained in Section 2.1, $Sub_{\mathcal{C}}$ expressions are not part of our rewrite theory and we give no definitions of the respective operators. We deal with *open* $Sub_{\mathcal{C}}$ programs in the sense that there can be missing inputs. In fact, $Sub_{\mathcal{C}}$ programs are seen as a collection of $Sub_{\mathcal{C}}$ functions and, at the level of a $Sub_{\mathcal{C}}$ function, every function argument is a missing input. The term *C_EA* characterizes a missing input and denotes the associated effective argument. More explanations and examples can be found in Section 4.1.

```
stmt      = statements
stmt_list = statement lists
var       = variables
exp       = expressions (exp ⊃ var)
cond      = conditions
fct_id    = function identifiers
nat       = natural numbers

statement lists constructors
C_L_Stmt            :  stmt_list × stmt → stmt_list
C_Empty_L_Stmt      :  → stmt_list

statements constructors
C_Return  :  fct_id × exp → stmt
C_If      :  cond × stmt_list × stmt_list → stmt
C_Assign  :  var × exp → stmt
C_While   :  nat × cond × stmt_list → stmt

expressions constructor
C_EA  :  var → exp

conditions constructors
C_Or   :  cond × cond → cond
C_And  :  cond × cond → cond
C_Not  :  cond → cond
```

Fig. 7. Function symbols for $Sub_{\mathcal{C}}$ language constructs.

env = *environments*
env_elt = *environment elements*
var_list = *variable lists*
id = *identifiers* $(id \supset var, id \supset fct_id)$

environment constructors
C_Env : $env_elt \times env \rightarrow env$
C_Empty_Env : $\rightarrow env$
C_Choice : $env \times env \rightarrow env$
C_Branch : $cond \times env \rightarrow env$

environment elements constructors
C_Pair : $id \times exp \rightarrow env_elt$
C_While_Closure : $nat \times cond \times env \times var_list \rightarrow env_elt$

variable lists constructors
C_L_Var : $var \times var_list \rightarrow var_list$
C_Empty_L_Var : $\rightarrow var_list$

Loop constructor
C_Loop : $nat \times var \times env \rightarrow exp$

Environment Generation from a list of statements
GE : $stmt_list \times env \rightarrow env$

Composition of a statement with an environment
Comp : $stmt \times env \rightarrow env$

Generation of calls to Loop functions
GL : $nat \times var_list \times env \rightarrow env$

Fig. 8. Function symbols for $\text{Sub}_\mathcal{C}$ language semantics.

- Function symbols which are used to give a semantics to $\text{Sub}_\mathcal{C}$ language constructs (*cf.* Fig. 8). They interpret $\text{Sub}_\mathcal{C}$ language constructs as functions from environments to environments. *Environments* capture the state of a computation. Iteration, through **while** statements, will be defined by recursion, through LOOP functions. More explanations and examples can be found in Section 4.2.
- Function symbols which are used to modify environments (*cf.* Fig. 9). In practice the differentiation between C_Subst_c et C_Subst_e is not necessary since the definitions of these operators are the same, and moreover we use an unsorted rewrite system. Therefore, in the sequel we will write in both cases *C_Subst*. Substitution appears as a constructor because it is not defined within the rewrite system. More explanations and examples can be found in Section 4.2.

$\text{SS}_\mathcal{R}$ is the definition of these function symbols (*cf.* Appendix B).

Note 1 The function symbols for lists are composed of a constant denoting the empty list and a constructor to add an element to an existing list. We find lists of variables (*C_L_Var* and *C_Empty_L_Var*), lists of statements (*C_L_Stmt* and *C_Empty_L_Stmt*) and environments (*C_Env* and *C_Empty_Env*). However, for convenience, lists will be represented enclosed in braces and elements in lists separated by dots, as in $\{e_1 \cdot e_2 \cdot e_3\}$.

substitution operators
C_Subst$_c$ *: var × cond × exp → cond*
C_Subst$_e$ *: var × exp × exp → exp*

Insertion and updating of a pair in an environment
Update_Env : env_elt × env → env

Updating of a condition according to an environment
Update_Cond : cond × env → cond

Generation of the Initial Environment of a loop function
GIE : env → env

Generation of a List Of Variables
GLOV : env → var_list

Generation of a List Of Modified Variables
GLOMV : env → var_list

Merge of two environments
Merge_Env : env × env → env

Insertion without updating of a pair in an environment
Insert_Pair : env_elt × env → env

Merge of two lists of variables
Merge_L_Var : var_list × var_list → var_list

Insertion of a variable in a list
Insert_Var : var × var_list → var_list

Fig. 9. Function symbols on environments.

4.1.2 Term generation

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

		Term generation
Input	:	A $\text{Sub}_{\mathcal{C}}$ program P made up of functions f_i.
Output	:	Terms $T_P^{f_i}$ over the rewrite system language $\text{SS}_{\mathcal{L}}$.

In order to build these terms, each $\text{Sub}_{\mathcal{C}}$ syntactical construct is mapped to a function symbol of $\text{SS}_{\mathcal{L}}$. Constructions and mapping are shown in Fig. 10 where "$\longmapsto$" means "maps to".

- Rule (Fct): a $\text{Sub}_{\mathcal{C}}$ `function` is seen as a list of statements and an initial environment gathered in a *GE* term. The initial environment is a list of *C_Pair* terms. Each *C_Pair* term is made up of a $\text{Sub}_{\mathcal{C}}$ function formal arguments combined with its *C_EA* term. This denotes missing values: the

$$\frac{\vdash \texttt{stmts} \longmapsto \textit{stmt_list} \quad \texttt{type}_t \in \{\texttt{int}, \texttt{list}\}}{\vdash \texttt{type fct_name (type}_1\ \texttt{p}_1\texttt{, ..., type}_n\ \texttt{p}_n\texttt{) \{ stmts \}} \longmapsto \textit{GE(stmt_list}, \{\textit{C_Pair}(p_1, \textit{C_EA}(p_1)) \cdot \ldots \cdot \textit{C_Pair}(p_n, \textit{C_EA}(p_n))\})} \text{(Fct)}$$

$$\frac{\vdash \texttt{stmt} \longmapsto \textit{stmt}' \quad \vdash \texttt{stmts} \longmapsto \textit{stmt_list}}{\vdash \texttt{stmt; stmts} \longmapsto \{\textit{stmt}' \cdot \textit{stmt_list}\}} \text{(Seq)}$$

$$\frac{\textit{fct_name} = \text{function name}}{\vdash \textbf{return}\ \texttt{exp} \longmapsto \textit{C_Return(fct_name, exp)}} \text{(Ret)}$$

$$\frac{}{\vdash \texttt{x = y} \longmapsto \textit{C_Assign(x, y)}} \text{(Assg)}$$

$$\frac{\vdash \texttt{stmts}_1 \longmapsto \textit{stmt_list}_1 \quad \vdash \texttt{stmts}_2 \longmapsto \textit{stmt_list}_2}{\vdash \textbf{if}\ \texttt{(c) stmts}_1\ \textbf{else}\ \texttt{stmts}_2 \longmapsto \textit{C_If(c, stmt_list}_1, \textit{stmt_list}_2)} \text{(Cond)}$$

$$\frac{\vdash \texttt{stmts} \longmapsto \textit{stmt_list} \quad \textit{loop_number} = \text{unused loop number}}{\vdash \textbf{while}\ \texttt{(c) stmts} \longmapsto \textit{C_While(loop_number, c, stmt_list)}} \text{(Iter)}$$

$$\frac{\texttt{type} \in \{\texttt{int}, \texttt{list}\}}{\vdash \texttt{type var = val} \longmapsto \textit{C_Assign(var, val)}} \text{(VarDecl1)}$$

$$\frac{}{\vdash \textbf{int}\ \texttt{var} \longmapsto \textit{C_Assign(var, 0)}} \text{(VarDecl2)}$$

$$\frac{}{\vdash \textbf{list}\ \texttt{var} \longmapsto \textit{C_Assign(var, NULL)}} \text{(VarDecl3)}$$

Fig. 10. Syntactic mapping rules.

effective arguments. Thus, the initial environment contains the value of the function variables as they would be after a call to this function, but before any statement of the function is evaluated.

- Rule (Seq): a `sequence` of statements is mapped to a list of statements.
- Rule (Ret): **`return`** statements produce a term *C_Return*. It links the name of the function containing the return statement and its return expression.
- Rule (Iter): **`while`** statements are mapped to *C_While* terms. Moreover, a unique number identifies each loop of the $\text{Sub}_{\mathcal{C}}$ program.
- Rules (VarDecl1), (VarDecl2) and (VarDecl3): `variable declarations` are part of the function statements. They are treated as assignments. If a variable is not initialized, a default value is assigned to it depending on its type.

Example 5 In the case of the `identity` function, the following rules apply to build the program term:

```
int identity (int x) {      ↦(Fct)   GE(□, {C_Pair(x, C_EA(x))})
 return x;                  ↦(Ret)   t = C_Return(identity, x)
}                           ↦(Seq)   □ = {t}
```

The symbol □ denotes a *hole* in a term which is filled while the remaining of the $\mathrm{Sub}_{\mathcal{C}}$ function is parsed. Finally, the *GE* term consists of a statement list whose single term is *C_Return(identity, x)*, and an initial environment *{C_Pair(x, C_EA(x))}*.

♦

4.2 Terms rewrite into environments

At the second step, each term $T_P^{f_i}$ is *rewritten*, according to $\mathrm{SS}_{\mathcal{R}}$ rules (*cf.* Appendix B), into an environment $Env_{T_P^{f_i}}$. Intuitively, this environment contains information about the variables of function f_i and their values: the evaluation of this environment yields the value of f_i in an execution of P. We have:

Term normalization

`Input` : A term $T_P^{f_i}$ over $< \mathrm{SS}_{\mathcal{L}}, \mathrm{SS}_{\mathcal{R}} >$.

`Output` : An $\mathrm{SS}_{\mathcal{R}}$-normalized term, $Env_{T_P^{f_i}}$, called *environment* of f_i.

4.2.1 Environments

We give in Fig. 11, in the form of a grammar, an overview of environment composition. The next sections explain how environments are used to express the $\mathrm{Sub}_{\mathcal{C}}$ semantics.

Control flow statements define different possible *execution paths* in a $\mathrm{Sub}_{\mathcal{C}}$ function. The environment produced by rewriting represents these distinct execution paths of a function, along with their associated conditions and the final variable state. An alternative between two execution paths appears in environments under a *C_Choice* term. When an execution path is associated to a condition it is enclosed in a *C_Branch* term. An iterated sequence of statements will be captured by a *C_While_Closure* term. A state of the variables is represented by a list of terms *C_Pair(var, exp)*, which corresponds to the value of each variable of a function at a step of the computation.

$\mathrm{SS}_{\mathcal{R}}$ defines several functions to handle environments:

env ::= choice | env_elt_list

env_elt_list ::= *'C_Env'* *'('* env_elt *','* env_elt_list *')'* | *'C_Empty_Env'*

env_elt ::= while_closure | pair

choice ::= *'C_Choice'* *'('* branch *','* branch *')'*

branch ::= choice | *'C_Branch'* *'('* cond *','* env *')'*

while_closure ::= *'C_While_Closure'* *'('* int *','* cond *','* env *','* var_list *')'*

pair ::= *'C_Pair'* *'('* var *','* exp *')'*

var_list ::= *'C_L_Var'* *'('* var *','* var_list *')'* | *'C_Empty_L_Var'*

loop ::= *'C_Loop'* *'('* int *','* var *','* env *')'*

exp ::= *a* $Sub_{\mathcal{C}}$ *expression where* loop *is added as an alternative* factor
cond ::= *a condition*
var ::= *a variable*
int ::= *an integer*

Fig. 11. Environments EBNF grammar.

- *Update_Env* inserts a new pair in an environment (Rules (15) to (18)). The value of the new pair is also updated according to the value of the other variables in the environment.
- *Merge_Env* and *Merge_L_Var* merge, respectively, two environments and two lists of variables (Rules (36)–(37) and (42)–(43));
- *GLOV* and *GLOMV* run through an environment and build a list by extracting, respectively, variables and modified variables (Rules (27) to (29) and (30) to (35)).

4.2.2 $Sub_{\mathcal{C}}$ semantics

This second step is a semantic evaluation of the program P. The rules of $SS_{\mathcal{R}}$ express the *equational semantics* of the $Sub_{\mathcal{C}}$ language as operations from environments to environments. The term $T_P^{f_i}$ produced by the parsing of the program is normalized according to $SS_{\mathcal{R}}$ rules, that is, the term is rewritten until no more rules can be applied. The resulting term is a $Sub_{\mathcal{C}}$ function environment.

For the most part, the $Sub_{\mathcal{C}}$ semantics does not require much explanation since it behaves as one could expect from a "standard" imperative language. *Arguments* of a function behave like function local variables to which are assigned effective arguments (because we only deal with arguments passed by value). The actual value of an effective argument is not known of course, but we denote it by a *C_EA* term. The *function return value* is handled through a local variable whose name is the function name. The return statement is an assignment to this particular variable. *Local variable declarations* are treated as assignments.

Example 6 Let us consider the following $\mathrm{Sub}_{\mathcal{C}}$ function:

```
int g (int y, int z) {
  int x = 3;
  y = f(x, 5);
  z = y + z;
}
```

According to rules of Fig. 10, the corresponding program term is:

GE({C_Assign(x, 3) · C_Assign(y, f(x, 5)) · C_Assign(z, y+z)},
{C_Pair(y, C_EA(y)) · C_Pair(z, C_EA(z))}).

GE (Generate Environment) translates the behavior of a *sequence* of statements. By application of Rules (1) and (2), we obtain:

Comp(C_Assign(z, y+z),
Comp(C_Assign(y, f(x, 5)),
Comp(C_Assign(x, 3),
{C_Pair(y, C_EA(y)) · C_Pair(z, C_EA(z))}))).

Comp (Compose) is used to evaluate a new statement in the current environment. By Rule (4), we have:

Comp(C_Assign(z, y+z),
Comp(C_Assign(y, f(x, 5)),
Update_Env(C_Pair(x, 3),
{C_Pair(y, C_EA(y)) · C_Pair(z, C_EA(z))}))).

Then, Rules (16) and (17) defining *Update_Env* apply. They insert the new pair in the current environment. After substitutions, we obtain:

Comp(C_Assign(z, y+z),
Comp(C_Assign(y, f(x, 5)),
{C_Pair(y, C_EA(y)) · C_Pair(z, C_EA(z)) · C_Pair(x, 3)})).

The same pattern repeats for the other two assignments, with the addition of Rule (3), and leads to the final environment:

{C_Pair(x, 3) · C_Pair(y, f(3, 5)) · C_Pair(z, f(3, 5)+C_EA(z))}.

This means that at the end of `g`, the value of `x` is 3, the value of `y` is the result of a call to `f` (with effective arguments 3 and 5), and the value of `z` is the sum of the result of a call to `f` and the value of the effective argument associated to `z` when `g` is called.

♦

Still, some constructs have a specific meaning as we are going to see particularly in the two paragraphs dedicated to the *conditional* and *iterative* statements.

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. This defines two paths which are enclosed in a *C_Choice* term. Each **if** alternative is included in a *C_Branch* term with the associated condition (Rules (7) and (8)). Other statements following an **if** are executed in both paths (Rules (11) and (12)).

Example 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
  ...
}
```

When the **if** statement is composed with the initially empty environment (Rule (7)) it will produce:

C_Choice(C_Branch(cond, GE(then_part, C_Empty_Env)),
C_Branch(C_Not(cond), GE(else_part, C_Empty_Env))).

♦

Iterative statements. We now turn our attention to the iterative construct *while*. The semantics of **while** statements is more complicated. Indeed, each **while** statement is considered as a family of separate recursive functions, each function having the same arguments and body. The idea is to replace iteration by recursion. A loop is a *loop function* that calls itself recursively with the value of the variables modified accordingly to the loop body. But a function can only return a single value, and yet several variables can be modified by a loop. To address this, a new loop function is defined for each variable modified by the loop body; its return value is the value of the modified variable. As a consequence we get a family of loop functions. In addition, since any variable of the $\mathrm{Sub}_{\mathcal{C}}$ function may be used inside the loop body, the loop function takes all the variables as arguments.

Now, we have to establish a connection between the loop functions and the $\mathrm{Sub}_{\mathcal{C}}$ function where the loop occurs. To this end, we replace, in the environ-

ment of the $\text{Sub}_\mathcal{C}$ function, the value of a variable x modified by the loop body, by a call to the loop function defined for x.

Consequently, when a loop is encountered, the environment is modified as follows:

- A new *C_While_Closure* 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 loop condition, the environment generated by the statements of the loop body and the list of all the $\text{Sub}_\mathcal{C}$ function local variables. This term will be used at the third step to generate a family of equations (see Section 4.3).
- A call to the corresponding loop function, with as argument the current value of the variables in the $\text{Sub}_\mathcal{C}$ function, is assigned to each variable modified by the **`while`** statement. This modifies the environment just as if equivalent assignments had been added to the $\text{Sub}_\mathcal{C}$ function.

Example 8 (iteration) This example shows how iterative statements are handled. Let us suppose a $\text{Sub}_\mathcal{C}$ function declares three variables (`x, y, z`) and that only two of them (`x, y`) are modified in the 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 (Rule (10)):

$$C_While_Closure(1, y > 0, GE(loop_body, initial_env), \{x \cdot y \cdot z\}).$$

We find in *C_While_Closure* a loop number, the loop condition, a *GE* term, which means that a new environment will be evaluated for the loop body, and the list of local variables of function `f`. *loop_body* is composed of the statements of the **`while`** body: these are the two assignments modifying `x` and `y`. *initial_env* is the initial environment for the loop functions in which the statements of the **`while`** body will be evaluated: it is the list of pairs $(x, C_EA(x))$, $(y, C_EA(y))$ and $(z, C_EA(z))$, one for each variable or arguments of `f`. Once the *GE* term is rewritten, we have:

$$\begin{array}{l} C_While_Closure(1, y > 0, \\ \quad \{C_Pair(z, C_EA(z)) \cdot C_Pair(x, C_EA(x)+C_EA(z)) \cdot \\ \quad\; C_Pair(y, C_EA(y)-1)\}, \\ \quad \{x \cdot y \cdot z\}). \end{array}$$

At the third step of the process (*cf.* Example 9, Page 24), this term will lead to the definition of two functions, LOOP_x^1 and LOOP_y^1, one for each variable modified inside the loop.

In addition, the pairs

C_Pair(x, C_Loop(1, x, {C_Pair(z, 3) · C_Pair(y, 2) · C_Pair(x, 1)}))

and

C_Pair(y, C_Loop(1, y, {C_Pair(z, 3) · C_Pair(y, 2) · C_Pair(x, 1)}))

update the environment of function `f` to reflect the new state of variables x and y which are modified by the loop body. This is just like replacing the loop in function `f` by the assignments:

`x = LOOP`$_x^1$`(1, 2, 3)` and `y = LOOP`$_y^1$`(1, 2, 3)`.

♦

4.2.3 $SS_{\mathcal{R}}$ convergence

In order to be sure that *every* $\text{Sub}_{\mathcal{C}}$ program always rewrites into a *unique* normal form, we must prove that $\text{SS}_{\mathcal{R}}$ is *convergent*. Convergence is equivalent to termination and confluence. Convergence means that our system is able to provide a unique equational formulation for $\text{Sub}_{\mathcal{C}}$ programs.

A rewrite system *terminates* if the rewriting process eventually ends for any input term. It means that every term of $\text{SS}_{\mathcal{L}}$ has at least one $\text{SS}_{\mathcal{R}}$-normal form. We show that the rewrite system terminates by exhibiting a mapping ϕ, from elements of the rewrite system $(\text{SS}_{\mathcal{L}}, \rightarrow)$ 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 $\phi(x_0) > \phi(x_1) > \cdots$ and therefore no infinite chain $x_0 \rightarrow x_1 \rightarrow \cdots$. Consequently the rewrite process terminates. For the proof, we define a well-founded order relation $>$ on function symbols of $\text{SS}_{\mathcal{L}}$ 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 θ, then both rules can be applied

C_And is associative and commutative.
GE and Comp are equivalent.
GE lexicographic path is left to right.
Update_Env lexicographic path is right to left.
Update_Cond lexicographic path is right to left.
C_Branch lexicographic path is right to left.

Comp > C_Env Comp > C_Pair Comp > Update_Env
Comp > C_Choice Comp > C_Branch Comp > C_Not
Comp > Update_Cond Comp > C_While_Closure
Comp > GLOV Comp > Merge_Env Comp > GL Comp > GLOMV
Comp > GIE

Update_Cond > C_Subst

Update_Env > C_Pair Update_Env > C_Subst
Update_Env > C_Env

C_Branch > C_Choice C_Branch > C_And

GIE > C_Env GIE > C_Pair GIE > C_EA

GLOV > C_L_Var

GLOMV > C_L_Var GLOMV > Merge_L_Var

Merge_L_Var > Insert_Var

Insert_Var > C_L_Var

GL > C_Env GL > C_Pair GL > C_Loop

Merge_Env > Insert_Pair

Insert_Pair > C_Env Insert_Pair > C_While_Closure
Insert_Pair > C_Pair

Fig. 12. Partial ordering on function symbols of $\mathrm{SS}_{\mathcal{L}}$.

to an instance of term $l_1\theta$. In this case, $r_1\theta$ and $(l_1\theta)[r_2\theta]_p$ [3] 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 [26]. 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* to prove the convergence of $\mathrm{SS}_{\mathcal{R}}$. 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 $\mathrm{SS}_{\mathcal{L}}$ to prove the termination. Then we successfully applied the completion algorithm on $\mathrm{SS}_{\mathcal{R}}$ (no new rules were generated), thus proving the confluence. Fig. 12 shows the partial ordering used in the proof.

[3] This is $l_1\theta$ where subterm at position p is replaced by $r_2\theta$.

This is the third and last step of the axiomatization process. A set of equations is generated from each environment $Env_{T_P^{f_i}}$. These sets of equations define the *equational theory* of program P. The problem definition for this step is:

Equations generation		
`Input`	:	An environment *Env*.
`Output`	:	A representation of *Env* within equational logic.

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:

- *C_Subst* terms. *C_Subst(x, exp*$_1$*, exp*$_2$*)* denotes the substitution of expression *exp*$_2$ for variable *x* in expression *exp*$_1$. The substitution is simply applied. Note that no substitution is done if *x* occurs in a term *C_EA(x)* – since this special term is precisely introduced to distinguish a formal argument *x*, which can be modified, and its effective value, which cannot.
- *C_EA* terms. They are not necessary anymore since the distinction between effective and formal arguments is only required for substitutions. *C_EA(x)* is replaced by x.
- *C_Loop* terms. They undergo a syntactical transformation.

  $$\textit{C_Loop(num, variable, \{exp}_1\cdot\ldots\cdot\textit{exp}_n\textit{\})}$$

  is replaced by

  $$\text{LOOP}^{\text{num}}_{\text{variable}}(exp_1, \ldots, exp_n).$$

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

- *lists of pairs*. 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` : *{C_Pair(...)* $\cdot\ldots\cdot$ *C_Pair(fct_name, exp)}*

  `Equation` : $fct_name = exp$

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

conditional equation.

`Generator` : *C_Branch(cond, {C_Pair(...) ·...·C_Pair(fct_name, exp)})*

`Equation` : $cond \Rightarrow fct_name = exp$

- *C_While_Closure* terms. They generate a family of conditional equations that defines recursively the loop functions – one loop function for each variable modified by the **`while`** statement. Two equations are needed:

(1) One equation for the recursive call. The variable state is modified according to the statements of the loop body. If e_i is the value of variable v_i after an abstract computation of the **`while`** statements, then, if the condition is true, the loop function is called again with the value e_i for the argument 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 argument to the loop function.

`Generator` :

C_While_Closure(n, cond, {C_Pair(v₁, e₁) · ... · C_Pair(vₙ, eₙ)}, {v₁ · ... · vₙ})

`Equation` :

$$\bigcup_{j \in M} \begin{cases} cond \Rightarrow \mathrm{LOOP}^n_{v_j}(v_1,\ \ldots,\ v_n) = \mathrm{LOOP}^n_{v_j}(e_1,\ \ldots,\ e_n) \\ \neg(cond) \Rightarrow \mathrm{LOOP}^n_{v_j}(v_1,\ \ldots,\ v_n) = v_j \end{cases}$$

Here, M denotes the set of modified variables in the loop body, v_j denotes one of these variables and $v_1, \ldots, v_n$ are the local variables appearing in the $\mathrm{Sub}_{\mathcal{C}}$ function. A variable *v* is known to have been modified when its value differs from *C_EA(v)*, which is the value assigned to it before getting into the loop.

Example 9 (iteration continued) With the *C_While_Closure* term of Example 8, the following function definitions will be generated:

$$\begin{cases} y > 0 \Rightarrow \mathrm{LOOP}^1_x(x,\ y,\ z) = \mathrm{LOOP}^1_x(x+z,\ y-1,\ z) \\ \neg(y > 0) \Rightarrow \mathrm{LOOP}^1_x(x,\ y,\ z) = x \end{cases}$$

$$\begin{cases} y > 0 \Rightarrow \mathrm{LOOP}^1_y(x,\ y,\ z) = \mathrm{LOOP}^1_y(x+z,\ y-1,\ z) \\ \neg(y > 0) \Rightarrow \mathrm{LOOP}^1_y(x,\ y,\ z) = y \end{cases}$$

♦

5 Extended examples

This section goes over the axiomatization process again, illustrating how its three steps apply to two case studies, namely *sum of the first n integers* and *insertion sort*. Appendix A contains more examples.

5.1 Sum of the first n integers

Let us suppose that someone wants to compute the sum of the first n integers and writes the erroneous program showed in Fig. 13.

```
int sum (int n) {
  int ret = 0;
  while (n != 0) {
    n = n - 1;
    ret = ret + n;
  }
  return ret;
}
```

Fig. 13. (Erroneous) “sum of the first n integers” in $\text{Sub}_{\mathcal{C}}$.

This program consists of 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. We show here how the program term is built from the $\text{Sub}_{\mathcal{C}}$ code according to rules in Fig. 10:

`int sum (int n) {`	$\overset{(Fct)}{\longmapsto}$	$GE(\square_0, \{C_Pair(n, C_EA(n))\})$
`int ret = 0;`	$\overset{(VarDecl1)}{\longmapsto}$	$t_1 = C_Assign(ret, 0)$
`while (n != 0) {`	$\overset{(Iter)}{\longmapsto}$	$t_2 = C_While_Closure(1, n \neq 0, \square_1)$
`n = n - 1;`	$\overset{(Assg)}{\longmapsto}$	$t_3 = C_Assign(n, n - 1)$
`ret = ret + n;`	$\overset{(Assg)}{\longmapsto}$	$t_4 = C_Assign(ret, ret + n)$
`}`	$\overset{(Seq)}{\longmapsto}$	$\square_1 = \{t_3 \cdot t_4\}$
`return ret;`	$\overset{(Ret)}{\longmapsto}$	$t_5 = C_Return(sum, ret)$
`}`	$\overset{(Seq)}{\longmapsto}$	$\square_0 = \{t_1 \cdot t_2 \cdot t_5\}$

Thus, at the end of the first step, the program is translated into the following $\text{SS}_{\mathcal{L}}$ term:

$$
\begin{array}{l}
GE(\{C_Assign(ret, 0) \cdot \\
\quad C_While(1, n \neq 0, \{C_Assign(n, (n - 1)) \cdot C_Assign(ret, (ret + n))\}) \cdot \\
\qquad\qquad C_Return(sum(n), ret)\}, \\
\quad \{C_Pair(n, C_EA(n))\}).
\end{array}
$$

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

The environment obtained after rewriting of the previous term with the rules of $SS_{\mathcal{R}}$ is:

{C_ While_ Closure(1, $n \neq 0$, {C_ Pair(n, (n−1)) · C_ Pair(ret, (ret+(n−1)))},
{n · ret}) ·
C_ Pair(n, C_ Loop(1, n, {C_ Pair(n, C_ EA(n)) · C_ Pair(ret, 0)})) ·
C_ Pair(ret, C_ Loop(1, ret, {C_ Pair(n, C_ EA(n)) · C_ Pair(ret, 0)})) ·
C_ Pair(sum(n), C_ Loop(1, ret, {C_ Pair(n, C_ EA(n)) · C_ Pair(ret, 0)}))}.

The *C_ While* term was rewritten into a *C_ While_ Closure* term. It 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. A *C_ Loop* term denotes a call to such a LOOP function. For instance, $\mathrm{LOOP}^1_{ret}(n, 0)$ is denoted by *C_ Loop(1, ret, {C_ Pair(n, C_ EA(n)) ·C_ Pair(ret, 0)})*.

Finally, the third step gives the equations of Fig. 14.

$$n \neq 0 \Rightarrow \mathrm{LOOP}^1_{ret}(n,\ ret) = \mathrm{LOOP}^1_{ret}((n-1),\ (ret + (n-1)))$$
$$n = 0 \Rightarrow \mathrm{LOOP}^1_{ret}(n,\ ret) = ret$$
$$\mathrm{sum}(n) = \mathrm{LOOP}^1_{ret}(n,\ 0)$$

Fig. 14. “Sum of the first n integers” equations.

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

5.2 Insertion sort

A $\mathrm{Sub}_{\mathcal{C}}$ version of *insertion sort* (see listing Fig. 15) will serve for this second case study. 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`.

5.2.1 Programs are terms

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

```
list ISort (list L) {
  list ret = NULL;
  if (L == NULL)
    ret = NULL;
  else
    ret = ins(car(L), ISort(cdr(L)));
  return ret;
}

list ins (int e, list L) {
  list ret = NULL;
  if (L == NULL)
    ret = cons(e, NULL);
  else if (e <= car(L))
    ret = cons(e, L); /* cons duplicates L */
  else {
    ret = ins(e, cdr(L));
    ret = cons(car(L), ret);
  }
  return ret;
}
```

Fig. 15. Insertion sort program in $\mathrm{Sub}_{\mathcal{C}}$.

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

GE({C_Assign(ret, NULL) ·
C_If(L = NULL,
{C_Assign(ret, NULL)},
{C_Assign(ret, ins(car(L), ISort(cdr(L))))}) ·
C_Return(ISort(L), ret)},
{C_Pair(L, C_EA(L))}).

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

- an *assignment* coming from the initialization of the variable `ret` during its declaration;
- a *C_If* term including the condition and two statements 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 *C_Return* term where the function name and arguments appear.

The initial environment is one pair, *C_Pair(L, C_EA(L))*, associating `L`, the only argument of the function, and the value `L` will take at the function call: its effective value denoted by *C_EA(L)*.

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

GE({C_Assign(ret, NULL) ·
C_If(L = NULL,

```
    {C_Assign(ret, cons(e, NULL))},
    {C_If(e ≤ car(L),
      {C_Assign(ret, cons(e, L))},
      {C_Assign(ret, ins(e, cdr(L))) · C_Assign(ret, cons(car(L), ret))})}) ·
   C_Return(ins(e, L), ret)},
  {C_Pair(L, C_EA(L)) · C_Pair(e, C_EA(e))}).
```

5.2.2 Terms rewrite into environments

At the second step, each term previously produced is rewritten, using $SS_{\mathcal{R}}$, into a final environment. Here is the environment of `ISort`:

```
C_Choice(
 C_Branch(L=NULL,
     {C_Pair(L, L) · C_Pair(ret, NULL) · C_Pair(ISort(L), NULL)}),
 C_Branch(L ≠ NULL,
     {C_Pair(L, L) · C_Pair(ret, ins(car(L), ISort(cdr(L)))) ·
      C_Pair(ISort(L), ins(car(L), ISort(cdr(L))))})).
```

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

```
C_Choice(
 C_Branch(L = NULL,
     {C_Pair(L, L) · C_Pair(e, e) · C_Pair(ret, cons(e, NULL)) ·
      C_Pair(ins(e, L), cons(e, NULL))}),
 C_Choice(
  C_Branch(C_And(L ≠ NULL, e ≤ car(L)),
      {C_Pair(L, L) · C_Pair(e, e) · C_Pair(ret, cons(e, L)) ·
       C_Pair(ins(e, L), cons(e, L))}),
  C_Branch(C_And(L ≠ NULL, e > car(L)),
      {C_Pair(L, L) · C_Pair(e, e) · C_Pair(ret, cons(car(L), ins(e, cdr(L)))) ·
       C_Pair(ins(e,L), cons(car(L), ins(e, cdr(L))))}))).
```

5.2.3 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*:

- *C_Branch* term with condition $L = NULL$;
- *C_Branch* term with condition $L \neq NULL$.

Finally, these two *equation generators* give the equations of Fig. 16 that define the semantics of the function `ISort`.

$$L = NULL \Rightarrow \mathrm{ISort}(L) = NULL$$
$$L \neq NULL \Rightarrow \mathrm{ISort}(L) = \mathrm{ins}(\mathrm{car}(L),\ \mathrm{ISort}(\mathrm{cdr}(L)))$$

Fig. 16. `ISort` equations.

Similarly, the *equation generators* for the `ins` function are the three *C_Branch* terms and give the conditional equations of Fig. 17.

$$L = NULL \Rightarrow \mathrm{ins}(e,\ L) = \mathrm{cons}(e,\ NULL)$$
$$L \neq NULL \wedge e \leq \mathrm{car}(L) \Rightarrow \mathrm{ins}(e,\ L) = \mathrm{cons}(e,\ L)$$
$$L \neq NULL \wedge e > \mathrm{car}(L) \Rightarrow \mathrm{ins}(e,\ L) = \mathrm{cons}(\mathrm{car}(L),\ \mathrm{ins}(e,\ \mathrm{cdr}(L)))$$

Fig. 17. `ins` equations.

Union of the two sets of equation constitutes the equational theory of the entire $\mathrm{Sub}_{\mathcal{C}}$ *sorting program*.

Appendix C.2 discusses the proof of two properties of `ISort` with the proof system PVS.

6 Conclusion

In this paper, we 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 equational 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 $\text{Sub}_{\mathcal{C}}$ programs into equations. An implementation of the axiomatization has been carried out in Java, thus proving that the process is fully automatic. A parser and scanner generator, was 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.

Our method focuses on functions as base elements for verification. This level of granularity should allow the system to perform as well with small programs as with larger ones, provided that they are sufficiently functionally decomposed. Proof systems offer user-friendly environments and help the user in keeping the proof modular. Still proofs can be hard and long to conduct not because of the size of the program, but because of its algorithm where lies the complexity. Before being proved, a property can require several intermediate lemmas which may not be found automatically by the proof system.

We present in Appendix A some interesting experiments with our system (algorithms about trees and graphs), which encourage us to continue our work in the following directions:

- adding functionalities to $\text{Sub}_{\mathcal{C}}$ in order to come closer to real imperative languages (*e.g.* side effects, call by reference);
- 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 two more examples of $\text{Sub}_{\mathcal{C}}$ programs and the equations produced by our $SOS\text{Sub}_{\mathcal{C}}$ system. These examples are well known algorithms on data types more complicated than the previous ones: trees and graphs. Lists are used to model these data types. For the sake of readability, conditions are partially evaluated in the equations.

A.1 Binary search tree

Fig. A.1 shows the code of a $\text{Sub}_{\mathcal{C}}$ program that searches an 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 triplet made up of the value of the node (an integer), and two other tree nodes for the left and right children (*i.e.* $\{root \cdot left\ child \cdot right\ child\}$). We represent a triplet by a list of three heterogeneous components. The first element of the list is the node value (an integer), the second and third ones are triplets (again lists of three elements).

```
int root (list tree) { return car(tree); }
list lc (list tree) { return car(cdr(tree)); }
list rc (list tree) {
  return car(cdr(cdr(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;
}
```

Fig. A.1. Binary search in a tree in $\text{Sub}_{\mathcal{C}}$.

The produced equations are in Fig. A.2.

$$\text{bs}(e,\ \emptyset) = 0$$
$$\text{bs}(e,\ \{e \cdot L\}) = 1$$
$$e < r \Rightarrow \text{bs}(e,\ \{r \cdot L\}) = \text{bs}(e,\ \text{lc}(\{r \cdot L\}))$$
$$e > r \Rightarrow \text{bs}(e,\ \{r \cdot L\}) = \text{bs}(e,\ \text{rc}(\{r \cdot L\}))$$

Fig. A.2. Binary search equations.

Figs. A.3 and A.4 show the code of a $\mathrm{Sub}_{\mathcal{C}}$ 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 `adjlist`. The produced equations are shown in Figs. A.5 and A.6.

```
list adj (int v, list adjlist) {
  list ret;
  if (v == 1) ret = car(adjlist);
  else ret = adj(v−1, cdr(adjlist));

  return ret;
}

int dfs (list vertices, list marked, list adjlist) {
  int ret, v;
  if (vertices == NULL) ret = 1;
  else {
    v = car(vertices);
    ret = depth(vertices, adj(v, adjlist),
            cons(v, marked), adjlist);
  }
  return ret;
}
```

Fig. A.3. Depth-first search in $\mathrm{Sub}_{\mathcal{C}}$ (part 1).

```
int member (int e, list L) {
  int ret;
  if (L == NULL) ret = 0;
  else if (e == car(L)) ret = 1;
  else ret = member(e, cdr(L));

  return ret;
}

int depth (list vertices, list adjacents, list marked,
           list adjlist) {
  int ret, v;
  if (adjacents == NULL)
    ret = dfs(cdr(vertices), marked, adjlist);
  else {
    v = car(adjacents);
    if (member(v, marked) == 1)
      ret = depth(vertices, cdr(adjacents),
                  marked, adjlist);
    else
      ret = dfs(cons(v, vertices), marked, adjlist);
  }
  return ret;
}
```

Fig. A.4. Depth-first search in $\mathrm{Sub}_{\mathcal{C}}$ (part 2).

$$\begin{aligned}
\mathrm{adj}(1,\ \{L_1 \cdot L\}) &= L_1\\
\mathrm{adj}(n+1,\ \{L_1 \cdot L\}) &= \mathrm{adj}(n,\ L)\\
\mathrm{dfs}(\emptyset,\ M,\ A) &= 1\\
\mathrm{dfs}(\{v \cdot L\},\ M,\ A) &= \mathrm{depth}(\{v \cdot L\},\ \mathrm{adj}(v,\ A),\ \{v \cdot M\},\ A)
\end{aligned}$$

Fig. A.5. Depth-first search (part 1): `adj` and `dfs` equations.

$$\begin{aligned}
&& \mathrm{member}(e,\ \emptyset) &= 0\\
&& \mathrm{member}(e,\ \{e \cdot L\}) &= 1\\
e \neq c \Rightarrow && \mathrm{member}(e,\ \{c \cdot L\}) &= \mathrm{member}(e,\ L)\\
&& \mathrm{depth}(\{v \cdot L\},\ \emptyset,\ M,\ A) &= \mathrm{dfs}(L,\ M,\ A)\\
\mathrm{member}(v,\ M) = 1 \Rightarrow && \mathrm{depth}(L,\ \{v \cdot L_1\},\ M,\ A) &= \mathrm{depth}(L,\ L_1,\ M,\ A)\\
\mathrm{member}(v,\ M) = 0 \Rightarrow && \mathrm{depth}(L,\ \{v \cdot L_1\},\ M,\ A) &= \mathrm{dfs}(\{v \cdot L_1\},\ M,\ A)
\end{aligned}$$

Fig. A.6. Depth-first search (part 2): `member` and `depth` equations.

B $\mathrm{SS}_{\mathcal{R}}$

Note 2 Most of the operators of $\mathrm{Sub}_{\mathcal{C}}$ expressions (*e.g.* **cons**, +) do not appear in these rules because they do not affect the translation process. Their definition is left to the proof system.

Note 3 The syntax is that of *RRL*. In order to fulfill it and have an homogeneous notation, we prefix names of rule variables with v_ and names of constructors with C_. All other names are function names if they begin with a capital letter, or sorts otherwise. Moreover sorts are given for information because they are not taken into account during the rewriting process. Indeed, $\mathrm{SS}_{\mathcal{R}}$ is unsorted.

```
;;;; sorts

;; stmt: statements                ;; stmt_list: statement lists
;; var: variables                  ;; var_list: variable lists
;; fct_id: function identifiers
;; id: identifiers
;; exp: expressions
;; cond: conditions
;; subst_exp: substitutable expressions
;; nat: natural numbers
;; env: environments
```

;; env_elt: environment elements

;; statement lists constructors
[C_L_Stmt : stmt_list, stmt →stmt_list]
[C_Empty_L_Stmt : stmt_list]

;; statements constructors
[C_Return : fct_id, exp →stmt]
[C_If : cond, stmt_list, stmt_list →stmt]
[C_Assign : var, exp →stmt]
[C_While : nat, cond, stmt_list →stmt]

;; environment constructors
[C_Env : env_elt, env →env]
[C_Empty_Env : env]
[C_Choice : env, env →env]
[C_Branch : cond, env →env]

;; environment elements constructors
[C_Pair : id, exp →env_elt]
[C_While_Closure : nat, cond, env, var_list →env_elt]

;; variable lists constructors
[C_L_Var : var, var_list →var_list]
[C_Empty_L_Var : var_list]

;; conditions constructors
[C_Or : cond, cond →cond]
[C_And : cond, cond →cond]
[C_Not : cond →cond]

;; expressions constructors
[C_EA : var →exp]
[C_Loop : nat, var, env →exp]
[C_Subst : var, subst_exp, exp →subst_exp]

;; var is a subsort of expressions
[var < exp]

;; a variable is an identifier
[var < id]

;; a function identifier is an identifier
[fct_id < id]

```
;; Environment Generation from a list of statements
[ GE : stmt_list, env →env ]

(1)  GE(C_L_Stmt(v_l_stmt, v_stmt), v_env) →Comp(v_stmt, GE(v_l_stmt, v_env))

(2)  GE(C_Empty_L_Stmt, v_env) →v_env

;; Composition of a statement with an environment
[ Comp : stmt, env →env ]

(3)  Comp(C_Assign(v_var, v_exp), C_Empty_Env) →
       C_Env(C_Pair(v_var, v_exp), C_Empty_Env)

(4)  Comp(C_Assign(v_var, v_exp), C_Env(v_pair, v_env)) →
       Update_Env(C_Pair(v_var, v_exp), C_Env(v_pair, v_env))

(5)  Comp(C_Return(fct, v_exp), C_Env(v_pair, v_env)) →
       Update_Env(C_Pair(fct, v_exp), C_Env(v_pair, v_env))

(6)  Comp(C_Return(fct, v_exp), C_Empty_Env) →
       C_Env(C_Pair(fct, v_exp), C_Empty_Env)

(7)  Comp(C_If(v_cond, v_l_stmt1, v_l_stmt2), C_Empty_Env) →
       C_Choice(C_Branch(v_cond, GE(v_l_stmt1, C_Empty_Env)),
                C_Branch(C_Not(v_cond), GE(v_l_stmt2, C_Empty_Env)))

(8)  Comp(C_If(v_cond, v_l_stmt1, v_l_stmt2), C_Env(v_pair, v_env)) →
       C_Choice(C_Branch(Update_Cond(v_cond, C_Env(v_pair, v_env)),
                         GE(v_l_stmt1, C_Env(v_pair, v_env))),
                C_Branch(C_Not(Update_Cond(v_cond, C_Env(v_pair, v_env))),
                         GE(v_l_stmt2, C_Env(v_pair, v_env))))

(9)  Comp(C_While(v_num, v_cond, v_l_stmt), C_Empty_Env) →
       C_Env(C_While_Closure(v_num, v_cond, GE(v_l_stmt, C_Empty_Env),
                         C_Empty_L_Var),
             GL(v_num, GLOMV(GE(v_l_stmt, C_Empty_Env)), C_Empty_Env))

(10)Comp(C_While(v_num, v_cond, v_l_stmt), C_Env(v_pair, v_l_pair)) →
       C_Env(C_While_Closure(v_num, v_cond,
                         GE(v_l_stmt, GIE(C_Env(v_pair, v_l_pair))),
                         GLOV(C_Env(v_pair, v_l_pair))),
             Merge_Env(GL(v_num,
                        GLOMV(GE(v_l_stmt, GIE(C_Env(v_pair, v_l_stmt)))),
                      C_Env(v_pair, v_l_pair)), C_Env(v_pair, v_l_pair)))
```

```
(11)Comp(stmt, C_Choice(v_exp1, v_exp2)) →
        C_Choice(Comp(stmt, v_exp1), Comp(stmt, v_exp2))

(12)Comp(stmt, C_Branch(v_cond, v_env)) →
        C_Branch(v_cond, Comp(stmt, v_env))

     ;; Generation of calls to Loop functions
     [ GL : nat, var_list, env →env ]

(13)GL(v_num, C_L_Var(v_var, v_l_var), v_env) →
        C_Env(C_Pair(v_var, C_Loop(v_num, v_var, v_env)),
                GL(v_num, v_l_var, v_env))

(14)GL(v_num, C_Empty_L_Var, v_env) →C_Empty_Env

     ;; Insertion and updating of a pair in an environment
     [ Update_Env : env_elt, env →env ]

(15)Update_Env(C_Pair(v_var, v_exp1), C_Env(C_Pair(v_var, v_exp2), v_env)) →
        Update_Env(C_Pair(v_var, C_Subst(v_var, v_exp1, v_exp2)), v_env)

(16)Update_Env(C_Pair(v_var1, v_exp1), C_Env(C_Pair(v_var2, v_exp2), v_env)) →
        C_Env(C_Pair(v_var2, v_exp2),
        Update_Env(C_Pair(v_var1, C_Subst(v_var2, v_exp1, v_exp2)), v_env))
     if not equal(v_var1, v_var2)

(17)Update_Env(C_Pair(v_var, v_exp), C_Empty_Env) →
        C_Env(C_Pair(v_var, v_exp), C_Empty_Env)

(18)Update_Env(C_Pair(v_var, v_exp1),
          C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var), v_env)) →
        C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var),
                  Update_Env(C_Pair(v_var, v_exp1), v_env))


     ;; Updating of a condition according to an environment
     [ Update_Cond : cond, env →cond ]

(19)Update_Cond(v_cond, C_Env(C_Pair(v_var, v_exp), v_env)) →
        Update_Cond(C_Subst(v_var, v_cond, v_exp), v_env)

(20)Update_Cond(v_cond, C_Empty_Env) →v_cond

(21)Update_Cond(v_cond,
          C_Env(C_While_Closure(v_num, v_condc, v_envc, v_l_var), v_env)) →
        Update_Cond(v_cond, v_env)
```

(22)C_Branch(v_cond, C_Choice(v_env1, v_env2)) →
    C_Choice(C_Branch(v_cond, v_env1), C_Branch(v_cond, v_env2))

(23)C_Branch(v_cond1, C_Branch(v_cond2, v_env)) →
    C_Branch(C_And(v_cond1, v_cond2), v_env)

;; Generation of the Initial Environment of a loop function
[GIE : env →env]

(24)GIE(C_Empty_Env) →C_Empty_Env

(25)GIE(C_Env(C_Pair(v_var, v_exp), v_env)) →
    C_Env(C_Pair(v_var, C_EA(v_var)), GIE(v_env))

(26)GIE(C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var), v_env)) →
    GIE(v_env)

;; Generation of a List Of Variables
[GLOV : env →var_list]

(27)GLOV(C_Empty_Env) →C_Empty_L_Var

(28)GLOV(C_Env(C_Pair(v_var, v_exp), v_env)) →C_L_Var(v_var, GLOV(v_env))

(29)GLOV(C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var), v_env)) →
    GLOV(v_env)

;; Generation of a List Of Modified Variables
[GLOMV : env →var_list]

(30)GLOMV(C_Empty_Env) →C_Empty_L_Var

(31)GLOMV(C_Env(C_Pair(v_var, C_EA(v_var)), v_env)) →GLOMV(v_env)

(32)GLOMV(C_Env(C_Pair(v_var, v_exp), v_env)) →
    C_L_Var(v_var, GLOMV(v_env))
  if not equal(C_EA(v_var), v_exp)

(33)GLOMV(C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var), v_env)) →
    GLOMV(v_env)

(34)GLOMV(C_Branch(v_cond, v_env)) →GLOMV(v_env)

(35)GLOMV(C_Choice(v_env1, v_env2)) →
    Merge_L_Var(GLOMV(v_env1), GLOMV(v_env2))

```
;; Merge of two environments
[ Merge_Env : env, env →env ]

(36)Merge_Env(C_Env(v_pair, v_l_pair), v_env) →
      Insert_Pair(v_pair, Merge_Env(v_l_pair, v_env)

(37)Merge_Env(C_Empty_Env, v_env) →v_env

;; Insertion without updating of a pair in an environment
[ Insert_Pair : env_elt, env →env ]

(38)Insert_Pair(C_Pair(v_var, v_exp),
        C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var), v_env)) →
      C_Env(C_While_Closure(v_num, v_cond, v_envc, v_l_var),
            Insert_Pair(C_Pair(v_var, v_exp), v_env))

(39)Insert_Pair(C_Pair(v_var, v_exp1), C_Env(C_Pair(v_var, v_exp2), v_env)) →
      C_Env(C_Pair(v_var, v_exp1), v_env)

(40)Insert_Pair(C_Pair(v_var1, v_exp1), C_Env(C_Pair(v_var2, v_exp2), v_env)) →
      C_Env(C_Pair(v_var2, v_exp2), Insert_Pair(C_Pair(v_var1, v_exp1), v_env))
    if not equal(v_var1, v_var2)

(41)Insert_Pair(C_Pair(v_var, v_exp), C_Empty_Env) →
      C_Env(C_Pair(v_var, v_exp), C_Empty_Env)

;; Merge of two lists of variables
[ Merge_L_Var : var_list, var_list →var_list ]

(42)Merge_L_Var(C_L_Var(v_var, v_l_var1), v_l_var2) →
      Insert_Var(v_var, Merge_L_Var(v_l_var1, v_l_var2))

(43)Merge_L_Var(C_Empty_L_Var, v_l_var) →v_l_var

;; Insertion of a variable in a list
[ Insert_Var : var, var_list →var_list ]

(44)Insert_Var(v_var, C_L_Var(v_var, v_l_var)) →C_L_Var(v_var, v_l_var)

(45)Insert_Var(v_var1, C_L_Var(v_var2, v_l_var)) →
      C_L_Var(v_var2, Insert_Var(v_var1, v_l_var))
    if not equal(v_var1, v_var2)

(46)Insert_Var(v_var, C_Empty_L_Var) →C_L_Var(v_var, C_Empty_L_Var)
```

C Proofs

We illustrate in this appendix the proof of program properties from equations produced by our $SOS\text{Sub}_C$ system. We show three different proofs conducted in two proof systems (*RRL* and PVS). We take the opportunity of these examples to cover a part of the theory which is assumed and not expressed in our system since it is specific to the proof system we use.

C.1 Reverse program and RRL

The *Rewrite Rule Laboratory* (*RRL*) is a theorem prover based on rewriting techniques and automated reasoning algorithms. Its ability to automate proofs by induction is of particular interest. We prove here the property presented in Example 1 Page 3 with *RRL*.

The input to *RRL* is a first-order theory. In our case, we provide the program equations of Fig. 3 Page 3. *RRL* is rather poor in predefined theories, so in addition, we define constructors (***C_ Null*** and ***C_ Cons***) and functions (***Cdr***, ***Car*** and ***Append***) on lists:

```
[C_Null : list]
[C_Cons : univ, list -> list]

[Cdr : list -> list]
Cdr(C_Cons(v_e, v_l)) := v_l

[Car : list -> univ]
Car(C_Cons(v_e, v_l)) := v_e

[Append : list, list -> list]
Append(C_Null, v_y) := v_y
Append(v_x, C_Null) := v_x
Append(C_Cons(v_x, v_y), v_z) := C_Cons(v_x, Append(v_y, v_z))
Append(Append(v_l1, v_l2), v_l3) ==
  Append(v_l1, Append(v_l2, v_l3))
```

Next we try to generate from the equations a set of rules which will serve as a decision procedure for the theory. We give an ordering and obtain:

```
  [1] CDR(C_CONS(V_E, V_L)) ---> V_L  [DEF, 1]
  [2] CAR(C_CONS(V_E, V_L)) ---> V_E  [DEF, 2]
  [3] APPEND(C_NULL, V_Y) ---> V_Y  [DEF, 3]
  [4] APPEND(V_X, C_NULL) ---> V_X  [DEF, 4]
```

```
  [5] APPEND(C_CONS(V_X, V_Y), V_Z) --->
        C_CONS(V_X, APPEND(V_Y, V_Z))  [DEF, 5]
  [6] LOOP1W(C_NULL, V_W) ---> V_W  [DEF, 6]
  [7] LOOP1W(C_CONS(V_E, V_L), V_W) --->
        LOOP1W(V_L, C_CONS(V_E, V_W))  [DEF, 7]
  [8] REVERSE(V_L) ---> LOOP1W(V_L, C_NULL)  [DEF, 8]
  [9] APPEND(APPEND(V_L1, V_L2), V_L3) --->
        APPEND(V_L1, APPEND(V_L2, V_L3))  [USER, 9]
```

Car and *Cdr* are partially defined but we can safely assume their definition is complete since the missing case corresponds to an error.

We indicate we want the proofs to be done by explicit induction (*cover set* method), which does not require the set of rules to be convergent or the definitions to be complete. First, we introduce two lemmas. Both of them relate *Loop1w* to *Append*. The first one makes apparent the role of accumulator of *Loop1w* second argument:

```
prove
Loop1w(v_l1, C_Cons(v_e, v_l2)) ==
  Append(Loop1w(v_l1, C_Null), C_Cons(v_e, v_l2))
...
hypothesis
hypothesis
...
  [USER, 10] is an inductive theorem in the current system.
```

Proofs with *RRL* are largely automated but for this particular lemma, we need to explicitly tell the system to rewrite with the induction hypothesis on two occasions. This may come from *RRL* which does not sufficiently generalize induction hypotheses.

The second lemma expresses the distributivity of *Loop1w* over *Append*:

```
prove
Loop1w(Append(v_l1, v_l2), C_Null) ==
  Append(Loop1w(v_l2, C_Null), Loop1w(v_l1, C_Null))
...
  [USER, 11] is an inductive theorem in the current system.
```

Finally, we prove the property of *Reverse*:

```
prove
Reverse(Reverse(v_l)) == v_l)
...
  [USER, 12] is an inductive theorem in the current system.
```

These two last proofs require no user intervention. We can now assert that the $\mathrm{Sub}_{\mathcal{C}}$ function `reverse` observes the property:

$$\forall(l : list)\ reverse(reverse(l)) = l.$$

C.2 ISort program and PVS

We now want to prove that the program of Fig. 15 Page 27 actually sorts a list of natural numbers. To this end, we need to prove that the initial list is a permutation of the list returned by the program (that is, they contain exactly the same elements) and that this latter list is ordered. The proof was done within PVS (Prototype Verification System) which is designed to act interactively under user guidance. Thus it is less automated than *RRL* but it gives a full and easier control over the proof.

C.2.1 Predefined theory

A *prelude file* built in to PVS contains several predefined theories among which we find:

- Predicate logic.

```
booleans: THEORY
 BEGIN
  boolean: NONEMPTY_TYPE
  bool: NONEMPTY_TYPE = boolean
  FALSE, TRUE: bool
  NOT: [bool -> bool]
  AND, OR, IMPLIES, WHEN, IFF: [bool, bool -> bool]
 END booleans
```

- Natural numbers (`nat`), arithmetic and order relations on them.
- Parameterized lists.

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr: list):cons?
 END list
...
  member(x, l): RECURSIVE bool =
    CASES l OF
      null: FALSE,
      cons(hd, tl): x = hd OR member(x, tl)
    ENDCASES
   MEASURE length(l)
```

In addition to the equations obtained from our system (*cf.* Figs. 16 Page 29 and 17 Page 29) that we add as axioms of a new theory, and since we have at our disposal the prelude theories, we only need to specify that we are interested in lists of natural numbers:

```
insertion_sort: THEORY
BEGIN
  lists : TYPE = list[nat]
  e, x, y : VAR nat
  l, l1, l2 : VAR lists

  ins : [ nat, lists -> lists ]
  ins_empty: AXIOM l=null IMPLIES ins(e, l) = cons(e, null)
  ins_le: AXIOM l/=null and e <= car(l) IMPLIES
              ins(e, l) = cons(e, l)
  ins_gt: AXIOM l/=null and e > car(l) IMPLIES
              ins(e, l) = cons(car(l), ins(e, cdr(l)))

  ISort : [ lists -> lists ]
  ISort_empty: AXIOM l=null IMPLIES isort(l) = null
  ISort_rec: AXIOM l/=null IMPLIES
                 isort(l) = ins(car(l), isort(cdr(l)))
END insertion_sort
```

C.2.2 Permutation property

We prove here that $ISort(l)$ is a permutation of l. We first define what a permutation is through predicate *perm*. l' is a permutation of l, if l' is empty once we removed from it (function *del*) each element of l. The following is added to `insertion_sort` theory:

```
  del : [ nat, lists -> lists ]
  del_empty: AXIOM del(x, null) = null
  del_eq: AXIOM x=y IMPLIES del(x, cons(y, l)) = l
  del_diff: AXIOM x/=y IMPLIES
                del(x, cons(y, l)) = cons(y, del(x, l))

  perm : [ lists, lists -> bool ]
  perm_empty1: AXIOM perm(null, cons(x, l)) = false
  perm_empty2: AXIOM perm(null, null) = true
  perm_not: AXIOM not(member(x, l2)) IMPLIES
                perm(cons(x, l1), l2) = false
  perm_rec: AXIOM member(x, l2) IMPLIES
                perm(cons(x, l1), l2) = perm(l1, del(x, l2))
```

We introduce two lemmas:

```
ins_member: LEMMA ins(x, l) = l2 IMPLIES member(x, l2)

del_ins: LEMMA del(x, ins(x, l)) = l
```

The first lemma states that the result of inserting x in l contains x; the second one says that removing the element just inserted in a list does not change the list. The important steps in the proofs of these lemmas are a structural induction on list l and a case reasoning on whether $x \leq car(l)$ so that we can apply axiom `ins_le` or `ins_gt`.

Then the theorem can be proved by induction on l:

```
th_perm: THEOREM perm(l, ISort(l))
```

C.2.3 Sort property

We now prove that $ISort(l)$ is ordered. We define a new predicate, `ordered`, in `insertion_sort` theory:

```
ordered : [ lists -> bool ]
ordered_empty: AXIOM ordered(null) = true
ordered_one: AXIOM ordered(cons(x, null)) = true
ordered_le: AXIOM x <= y IMPLIES
            ordered(cons(x, cons(y, l))) = ordered(cons(y, l))
ordered_gt: AXIOM not(x <= y) IMPLIES
                  ordered(cons(x, cons(y, l))) = false
```

The following lemmas are properties of *ordered* and could be part of its definition even though they can be derived from its axioms:

```
lemma_ordered1: LEMMA ordered(cons(x, l)) IMPLIES ordered(l)

lemma_ordered2: LEMMA ordered(cons(x, cons(y, l)))
                      IMPLIES ordered(cons(x, l))

lemma_ordered3: LEMMA ordered(cons(x, l)) and member(y, l)
                      IMPLIES x <= y
```

The last lemma says that the first element of an ordered list is also the least one.

Here are three more lemmas about *ins*:

```
ins_not_empty: LEMMA exists (y: nat, l2: lists):
                     ins(x, l1) = cons(y, l2)

ins_member2: LEMMA x /= y and ins(x, l1) = cons(y, l2)
                   IMPLIES member(y, l1)

lemma_ins: LEMMA ordered(l) IMPLIES ordered(ins(x, l))
```

The first one states that inserting an element leads to a non empty list; the second one is easily derived from the previous lemma `ins_member`. At last, the third one is the most important part of the proof. It states that inserting an element in an ordered list leads to an ordered list. Again the proof is done by induction on l and case reasoning on the ordering of lists elements.

Finally, we prove the theorem by induction on l:

```
th_ordered: THEOREM ordered(isort(l))
```

Acknowledgements

The authors thank the anonymous referees for their valuable comments which significantly improved the quality of the present paper.

References

[1] O. Ponsini, C. Fédèle, E. Kounalis, Sos $C--$: A system for interpreting operational semantics of $C--$ programs, in: M. H. Hamza (Ed.), Proc. IASTED Internat. Conf. on Applied Informatics, Innsbruck, Austria, 2002, pp. 164–169.

[2] C. Fédèle, E. Kounalis, Automatic proofs of properties of simple $C--$ modules, in: Proc. 14th IEEE Internat. Conf. on Automated Software Engineering, Cocoa Beach, Floride, USA, 1999, pp. 283–286.

[3] D. Kapur, H. Zhang, RRL : A Rewrite Rule Laboratory, in: Proc. 9th Internat. Conf. on Automated Deduction (CADE), Vol. 310 of Lecture Notes in Computer Science, Springer, Argonne, Illinois, USA, 1988, pp. 768–769.

[4] A. Bouhoula, E. Kounalis, M. Rusinowitch, Spike: An automatic theorem prover, in: A. Voronkov (Ed.), Proc. Internat. Conf. on Logic Programming and Automated Reasoning, Vol. 624 of Lecture Notes in Artificial Intelligence, Springer-Verlag, St. Petersburgh, Russia, 1992, pp. 460–462.

[5] S. Owre, J. M. Rushby, N. Shankar, PVS: A prototype verification system, in: D. Kapur (Ed.), Proc. 11th Internat. Conf. on Automated Deduction (CADE),

Vol. 607 of Lecture Notes in Artificial Intelligence, Springer-Verlag, Saratoga, NY, 1992, pp. 748–752.

[6] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, B. Werner, The Coq Proof Assistant Reference Manual – Version V6.1, Tech. Rep. 0203, INRIA (August 1997).

[7] O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, L. Wildman, The Cogito development system, in: M. Johnson (Ed.), Proc. Algebraic Methodology and Software Technology (AMAST), Vol. 1349 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, pp. 586–591.

[8] R. Juellig, Y. Srinivas, J. Liu, SPECWARE: An advanced environment for the formal development of complex software systems, Lecture Notes in Computer Science 1101 (1996) 551.

[9] J. C. Filliâtre, Proof of imperative programs in type theory, in: T. Altenkirch, W. Naraschewski, B. Reus (Eds.), Types for Proofs and Programs, Vol. 1657 of Lecture Notes in Computer Science, Springer-Verlag, 1998, p. 78.

[10] S. Antoy, J. Gannon, Using Term Rewriting to Verify Software, IEEE Transactions on Software Engineering 20 (4) (1994) 259–274.

[11] F. Q. da Silva, On observational equivalence and compiler correctness, in: Proc. Internat. Conf. on Computing and Information (ICCI), 1994, pp. 17–34.

[12] M. Ward, Abstracting a specification from code, Journal of Software Maintenance: Research and Practice 5 (2) (1993) 101–122.

[13] A. Butterfield, S. Flynn (Eds.), Declarative View of Imperative Programs, Workshops in Computing, The British Computer Society, Cork, Ireland, 1998.

[14] E. Moggi, Notions of computation and monads, Information and Computation 93 (1) (1991) 55–92.

[15] P. Wadler, The marriage of effects and monads, in: Proc. 3rd ACM SIGPLAN Internat. Conf. on Functional Programming, ACM Press, Baltimore, Maryland, USA, 1998, pp. 63–74.

[16] B. Jacobs, E. Poll, Coalgebras and monads in the semantics of java, Theoretical Computer Science 291 (3) (2003) 329–349.

[17] M. D. Cin, C. Meadows, W. Sanders (Eds.), Verifying the Specification-to-Code Correspondence for Abstract Data Types, Vol. 11 of Dependable Computing and Fault-Tolerant Systems, IEEE Computer Society Press, 1997.

[18] S. Garland, J. Guttag, A guide to LP, the larch prover, Tech. Rep. 82, Digital Equipment Corporation, Systems Research Centre (Dec. 1991).

[19] D. Scott, C. Strachey, Towards a mathematical semantics for computer languages, in: 21st Symp. on Computers and Automata, Polytechnic institute of Brooklyn, 1971, pp. 19–46.

[20] R. Milner, LCF: A way of doing proofs with a machine, in: J. Becvár (Ed.), 8th Symp. on Mathematical Foundations of Computer Science, Vol. 74 of Lecture Notes in Computer Science, Springer, Olomouc, Czechoslovakia, 1979, pp. 146–159.

[21] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.

[22] J. A. Goguen, G. Malcolm, Algebraic Semantics of Imperative Programs, Foundations of Computing, The MIT Press, 1996.

[23] J. A. Bergstra, T. B. Dinesh, J. Field, J. Heering, Toward a complete transformational toolkit for compilers, ACM Transactions on Programming Languages and Systems 19 (5) (1997) 639–684.

[24] J. Field, J. Heering, T. B. Dinesh, Equations as a uniform framework for partial evaluation and abstract interpretation, ACM Computing Surveys 30 (3es) (1998) 2.

[25] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[26] D. E. Knuth, P. B. Bendix, Simple word problems in universal algebras, Computational Problems in Abstract Algebra (1970) 263–297.