SOS C--: a System for interpreting Operational Semantics of C-- programs

Olivier Ponsini, Carine Fédèle and Emmanuel Kounalis
Laboratoire I3S - UNSA - CNRS
2000, route des Lucioles, B.P. 121
06 903 Sophia Antipolis, FRANCE
email: {ponsini, carine, kounalis}@i3s.unice.fr

ABSTRACT
This paper describes a system for automatically transforming programs written in a simple imperative language (called C--), into a set of first-order equations. This means that a set of first-order equations used to represent a C-- program already has a precise mathematical meaning; moreover, the standard techniques for mechanizing equational reasoning can be used for verifying properties of programs. This work shows that simple imperative programs can be seen as fully formalized logical systems, within which theorems can be proved. The system itself is formulated abstractly as a set of first-order rewrite rules. Then, it is proven to be terminating and confluent using the RRL system.

KEY WORDS
Program Verification, Rewriting, Equational Semantics

1. Introduction

The need to be able to reason about computer programs in a rigorous formal way is self evident. In order to increase confidence in code production, efforts should be focused on verifying that programs meet their requirements, that is, that they are sound with respect to their specification. In a previous work [1], Fédèle and Kounalis introduced a theoretical framework for proving automatically properties of C-- programs, a simple imperative language. The idea was to translate source code into a set of first order equations expressing the program algebraic semantics. The use of equational logic, has some advantages over other, more complex logics:

1. it is very simple — the logic of substituting equals for equals;
2. many problems associated with equations, that are not decidable in more complex logics, are decidable in equational logic;
3. there are efficient algorithms for deciding many of these problems.

The general outline of our framework is shown in Fig. 1. Users write down the C-- 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, by the SOS C-- system, into a set of equations. 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, the proof of these theorems from the axioms is equivalent to the proof that the program meets its specification. This last part may be done automatically using theorem provers able to do mathematical induction like NICE [2] or interactively using proof checkers like CoQ [3].

This approach is conceptually different from other recent developments like COGITO [4] or SPECWARE [5], since these systems generate code from specifications. It also differs from [6] and [7] since they use annotations to prove the program behavior. Our approach is conceptually similar to the systems [8] and [9]. However, [9] can not treat programs with loops and [8] works in logics not easily amenable to automation.

In this paper, we extend [1] in various directions:

1. We give an abstract framework to the program towards equation process. In [1], there was only the basis of a rewrite system and no property had been proved on the system.
In particular we formulate the SOS C-- system as a rewrite system. We refine the rules to enrich and be closer to the C-- semantics.

We prove the rewrite system completeness (i.e. termination and confluence) using the RRL\(^1\). Roughly speaking, this kind of completeness means that every C-- program can be transformed into a unique equational program.

We give a formal description of how equations are generated from environments.

2. In [1], no implementation had been done. We now assert that the axiomatization process can be automatized since we made an implementation in Java.

- JavaCC\(^2\), 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 worked out an algorithm to generate equations from environments.

In what follows, we first introduce some basic definitions and notations. Then, section 3 gives a general outline of the SOS C-- system and illustrates it with an example; the section uses the following scheme:

1. programs are terms;
2. terms produce environments;
3. environments generate equations.

2. Definitions

2.1 Rewrite Systems

We assume familiarity with the basic notions of equational logic and rewrite systems (see [11] for instance). Let \( T(F, X) \) denote the set of terms built out of function symbols taken from the finite \textit{vocabulary} \( F \) and a denumerable set \( X \) of \textit{variables}. If \( t \) is a term and \( \theta \) is a substitution of terms for variables in \( t \), then \( t \theta \) is an \textit{instance} of \( t \). An \textit{equation} \( e \) is an element \( T(F, X) \times T(F, X) \) and is written as \( t = s \). A \textit{rewrite system} \( R \) is a set of oriented equations \( l \rightarrow r \), called \textit{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 \( R \) by repeatedly applying rules to rewrite (or reduce) an input term until a \textit{normal form} (irreducible term) is obtained. Let \( A \) be a set of equations, in the case where \( A \) can be compiled into a \textit{complete} (i.e. terminating and confluent) rewrite system \( R \), we can decide \( t \equiv_A s \) by testing for identity the \( R \)-normal forms of \( t \) and \( s \) (i.e. \( \text{nf}(t) \equiv \text{nf}(s) \)), where \( \text{nf}(t) \) (resp. \( \text{nf}(s) \)) denotes the normal form of \( t \) (resp. \( s \)).

2.2 The C-- Language

For our experiments we use a very simple imperative language. The C-- syntax is similar to the C one. The main features of the language are:

- assignment;
- control flow statements: \texttt{if} . . . \texttt{else, while} and \texttt{return};
- two predefined types: integers (\texttt{int}), and lists of integer (\texttt{list});
- usual arithmetic operators;
- operators on lists: \texttt{getHead} which returns the first element of a list, \texttt{getQueue} which returns a copy of a list except for the first element, \texttt{NULL} which represents an empty list, and \texttt{cons} which inserts an element at the beginning of a list.

However, several common features in imperative languages are unavailable in C--: no user’s defined types; no global variables; no pointers directly accessible — of course some are used in the predefined type \texttt{list}.

3. A General Outline of the SOS C-- System

The SOS C-- system can be represented as a rewrite system \( R \) over a first-order language \( L \) built out from a set of function symbols. These symbols are translations of the constructs of the source language (C--). For instance, the assignment statement, written \( x = y \) in C--, is translated into \texttt{Assign}(\( x, y \)), where \texttt{Assign} belongs to the vocabulary of \( L \).

The system takes as input a C-- program and returns as output a set of equations semantically equivalent to the program: the result of the execution of the C-- program with input I is identical to the result (i.e. theorem) of the equational deduction, started with the very same input. The transformation process, called C-- axiomaticiation, is done in three steps and is carried out without any user interaction. Figure 2 shows the steps involved in the C-- axiomatization.

To illustrate these steps, we will use the sorting program of listing 1. This program is a C-- version of the insertion sort. Function \texttt{ins} takes an integer \( e \) and a sorted list \( L \) and returns a new sorted list which is a copy of \( L \) containing \( e \). \texttt{Isort} takes a list \( L \) as argument and returns a sorted copy of \( L \) by inserting at the right position (call to function \texttt{ins}) the first element of \( L \) in the already sorted queue of \( L \).

\(^1\)Rewrite Rule Laboratory [10]. The proof is part of the full version of this paper.
\(^2\)Java Compiler Compiler, Metamata.
3.1 Programs are Terms

The first step consists in analyzing the functions of the source program $P$. The result of this syntactic analysis is a list of term $T_f^P$ over $L$: one term for each function $f$ of $P$. Intuitively, a term is equivalent to a source function and suitable for rewriting.

Each C-- function is seen as a list of statements and an initial environment gathered in a $GE$ term. The initial environment is made up of the function formal parameters combined in a $Pair$ term with $EP$ terms. An $EP$ term denotes an effective parameter. Thus, formal parameters behave like local variables to which are assigned the effective parameters.

Variable declarations are considered as assignments and are therefore added to the list of statements of the function.

A sequence of statements is simulated by a list of terms.

Return statements engender a pair linking the name of the function and its return expression.

Expressions appearing as right value in an assignment or as value in a function call are left as is.

The other constructs of the C-- language are simply matched with equivalent terms of the rewrite system: the if statement, if $(c) s_1$ else $s_2$, is translated into If$(c, s_1, s_2)$; the while statement, while$(c) s$, is translated into While(while_number, $c, s$).

For instance, the term which corresponds to the $ISort$ function of the sorting program is:

\[
GE( \{ Assign(ret, NULL), \nonumber \\
If(L = NULL, \nonumber \\
\quad \{ Assign(ret, NULL) \}, \nonumber \\
\quad \{ Assign(ret, inst(getHead(L), \nonumber \\
\quad \quad ISort(getQueue(L)))) \}), \nonumber \\
\quad Return(ISort(L), ret) \} \).
\]

A $GE$ term contains two elements. The first one represents the sequence of instructions of the source function:

- the declaration of $ret$ gives the first $Assign$ term;
- then comes the $If$ term with the condition and its two lists of statements, one for each alternative;
- and finally the $Return$ term.

The second one represents the function’s initial environment. It is a list of $Pair$ terms. A $Pair$ associates a variable and a value. Thus, the initial environment is the value of the function’s variables after the call but before any instruction is evaluated.

Likewise, the term which corresponds to the $ins$ function is:

\[
GE( \{ Assign(ret, NULL) , \nonumber \\
If(L = NULL, \nonumber \\
\quad \{ Assign(ret, cons(e, NULL)) \}, \nonumber \\
\quad \{ If(e \leq getHead(L), \nonumber \\
\quad \quad Assign(ret, cons(e, L)) \}, \nonumber \\
\quad \quad Assign(ret, inst(e, getQueue(L))), \nonumber \\
\quad \quad Assign(ret, cons(getHead(L), ret))) \} \} 
\]

Return(ins(e, L), ret) } , 
\quad \{ Pair(L, EP(L)) , Pair(e, EP(e)) \} 
\) .

3.2 Terms produce Environments

At the second step, the system transforms a term $T_f^P$ into an environment $E_{T_f^P}$. Intuitively, this environment contains all information about the variables of function $f$ and their corresponding expressions (the evaluation of which yields the value of $f$ in an execution of $P$).

3.2.1 Description

To obtain this environment, each term $T_f^P$ is normalized according to the rewrite rules $R$. The rules are divided

---

$GE$ stands for Generate Environment.

$EP(x)$ stands for Effective Parameter $x$. It is the value given to the function parameter $x$ when the function is called.

The complete set of rules can be found in the full version of this paper.
Function statements are executed in an order which 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 expression of variables and function. The state of the variables is represented by a list of Pair terms. A Branch term associates a condition to a variables state. The paths are enclosed in Choice terms.

For instance, the environment of the ISort function of listing 1 is:

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 comprises one if statement, so we find in the environment a Choice term composed of two Branch terms. These latter terms partition the statements of the function between those which are executed when the condition $L = NULL$ is true and those which are executed when this same condition is false. In each Branch term there is a list of Pair terms which represents the state of the variables at the end of an “abstract” execution of the ISort function. For instance, in the case where $L = NULL$, Pair(L, L) means that $L$ is not modified by the function; Pair(ret, NULL) means that the value of variable ret is NULL; the Pair term containing the function name, Pair(ISort(L), NULL) means that the function return value is NULL.

Likewise, the environment of the ins function is:

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))) ) ] ] ]
)

3.2.2 While Statements

We now turn our attention to the iterative construct while. This section gives an insight into how we handle while statements, and we then formalize it in sections 3.2.3 and 3.3.

The semantics of while statements is quite specific. Indeed, each loop is considered as a family of separate recursive functions with their own parameters and body. The idea is that a loop is a function which calls itself recursively with the value of the variables modified accordingly to the statements of the loop body. Such a loop function is defined for each variable modified in a loop body. Then, in the function containing the loop, the value of a variable modified in the loop body is the result of a call to the specific loop function. Consequently, when a loop is encountered, the environment is modified as follows:

- A new $LT^6$ term containing all the information needed to generate the loop functions is created. The information is the loop number, the exit condition, the statements of the loop body and the list of all the variables. This Loop Term will be used at the third step to generate a family of equations (see section 3.3).
- Each variable modified in the loop body is assigned a call to the corresponding loop function. This function takes as argument the current state of the variables.

The following example shows how while statements are handled. Let us suppose a C- function declares three variables — $x$, $y$, $z$ — two of which are modified in a loop body, like in listing 2. During rewriting of the term corres-

Listing 2. A loop.

```c
int f() {
    int x,y,z;
    x=1; y=2; z=3;
    while(y>0) {
        x=x+z;
        y=y−1;
    }
    . . .
}
```

Corresponding to function $f$, the following Loop Term is created:

$LT(1, y > 0, GE(statements, initial environment), (x, y, z))$.

The statements are the two assignments modifying $x$ and $y$. The initial environment is the list of pairs $(x, EP(x))$, $(y, EP(y))$ and $(z, EP(z))$. The term $GE$ means that a new

6Loop Term.
Among the rules updating an environment, we find:

- Merge.env and Merge.L var to merge two lists;
- Insert.pair and Insert.var to add a pair or a variable to a list;
- 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.

### 3.3 Environments generate Equations

At the last step, a set of equations is generated from each $E_{T.f}$. This set of equations defines the algebraic semantics of $P$. In the example of the sorting program, we obtain:

\[
\begin{align*}
L &= NULL \Rightarrow \text{ins}(e, L) = \text{cons}(e, NULL) \\
L \neq NULL \land e \leq \text{getHead}(L) &\Rightarrow \\
\text{ins}(e, L) &= \text{cons}(e, L) \\
L \neq NULL \land e > \text{getHead}(L) &\Rightarrow \\
\text{ins}(e, L) &= \text{cons}(\text{getHead}(L), \text{ins}(e, \text{getQueue}(L))) \\
L &= NULL \Rightarrow \text{ISort}(L) = NULL \\
L \neq NULL &\Rightarrow \\
\text{ISort}(L) &= \text{ins}(\text{getHead}(L), \text{ISort}(\text{getQueue}(L)))
\end{align*}
\]

Only a few elements in an environment will generate equations: these are the equation generators. The third and final step of the axiomatization process refines environments, extracts equation generators from environments and generates the corresponding equations. The equation generators are:

- lists of pairs. They represent the state of the variables at the end of the computation. But, only the function return value is of interest, therefore, only the pair containing the function name will generate an equation.

  \[
  \begin{align*}
  \text{Generator} : \\
  \text{Pair}(\ldots \ldots) \cdot \cdot \cdot \text{Pair}(\text{func}_\text{name}, \text{expression})
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Equation} : & \quad \text{func}_\text{name} = \text{expression}
  \end{align*}
  \]

- 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 Branch term generates one conditional equations.

  \[
  \begin{align*}
  \text{Generator} : \quad & \text{Branch}(\text{condition}, \\
  \text{Pair}(\ldots \ldots) \cdot \cdot \cdot \text{Pair}(\text{func}_\text{name}, \text{exp}))
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Equation} : & \quad \text{condition} = \text{True} \Rightarrow \text{func}_\text{name} = \text{exp}
  \end{align*}
  \]

- LT terms. They generate a family of conditional equations that defines recursively the loop functions — one loop function for each modified variable in the loop body. Two equations are needed, one for the recursive call — with the variables state modified according to
the loop body — and one for the exit case which gives the result of the loop function, that is the current value of the considered modified variable.

\[
\begin{align*}
\text{Generator} & : \text{LT}(\text{num}, \text{cond}, \\
& \quad \text{Pair}(v_1, e_1), \ldots, \text{Pair}(v_n, e_n), \{v_1, \ldots, v_n\}) \\
\text{Equation} & : \\
\bigcup_{1 \leq i \leq m} \begin{cases} \\
\text{cond} = \text{True} \Rightarrow \\
\text{LOOP}^{\text{num}}_{\text{mod}}(v_1, \ldots, v_n) = \\
\text{LOOP}^{\text{num}}_{\text{mod}}(e_1, \ldots, e_n) \\
\text{cond} = \text{False} \Rightarrow \\
\text{LOOP}^{\text{num}}_{\text{mod}}(v_1, \ldots, v_n) = v_{\text{mod}}. \\
\end{cases}
\end{align*}
\]

Here, \(v_{\text{mod}}, \ldots, v_{\text{mod}, n}\) are the variables modified in the loop body and \(v_1, \ldots, v_n\) are all the variables appearing in the C-- function. A variable \(v\) is known to have been modified when its value differs from \(\text{EP}(v)\) which is the value assigned to it before getting in the loop.

4. Conclusion

In this paper we have discussed a system to automatically obtain an equivalent equational formulation of a C-- program from source code. The process leading to the equations requires three steps. The central point of the discussed method is the generation of an environment by means of a rewrite system which implements the operational semantics of the C-- language. The first stage consists in building an environment — and one for the exit case which gives the result of the loop function, that is the current value of the considered modified variable.

References


