# Mutable Lists and Call-by-Reference with *SOS*Sub$_{\mathcal{C}}$

OLIVIER PONSINI
Laboratoire I3S - UNSA - CNRS
2000 route des lucioles, B.P. 121
06 903 Sophia Antipolis Cedex
FRANCE
ponsini@i3s.unice.fr

CARINE FÉDÈLE
Laboratoire I3S - UNSA - CNRS
2000 route des lucioles, B.P. 121
06 903 Sophia Antipolis Cedex
FRANCE
carine.fedele@unice.fr

*Abstract:* - Sub$_{\mathcal{C}}$ is a study imperative language around which we build the *SOS*Sub$_{\mathcal{C}}$ system as an attempt to help the development of certified programs. It allows formal reasoning about imperative programs by translating programs into equations, with which proofs can be carried out. In this paper, we add to the Sub$_{\mathcal{C}}$ language two important imperative features: mutable lists and call-by-reference passing mode. We present their implementation and semantics in Sub$_{\mathcal{C}}$, as well as their translation into conditional equations by the *SOS*Sub$_{\mathcal{C}}$ system.

*Key–Words:* - Program analysis, formal methods, correctness proofs, algebraic semantics, logics

## 1 Introduction

In computer science, formal methods provide a mathematical framework to logically reason about computer programs and systems.

Thanks to formal methods, designers gain an incomparable degree of confidence in their critical applications where human safety, security or financial costs are involved (*cf.* [3] for case studies). As computer aided tools appear and the range of applications widens, formal methods incite interest in industry. Still, in practice, they are often reproached a complicated and unusual implementation which prevents their actual and effective use.

We think that *equational logic* is well suited to serve as a mathematical foundation underlying formal methods. Indeed, equational logic is well understood, amenable to automation and existing tools are mature enough to conduct proofs within it, possibly without user interaction [4]. Moreover, it has the advantage of being well known by industry engineers. And since the imperative paradigm is the most widespread among industrial languages, we developed *SOS*Sub$_{\mathcal{C}}$ to fill the gap between imperative programs and equational logic.

*SOS*Sub$_{\mathcal{C}}$ translates programs written in Sub$_{\mathcal{C}}$, a simple yet powerful imperative language, into conditional equations. Specifications of programs are written, within equational logic, as properties on programs inputs and outputs. Thus, if it can be proved that the properties are deduced from the equations of the programs, it has been proved that the programs meet their specification. *SOS*Sub$_{\mathcal{C}}$ can be associated with a theorem prover or a proof checker to form a framework within which developers can program and prove properties of their programs.

In the preceding version of *SOS*Sub$_{\mathcal{C}}$ [12], the Sub$_{\mathcal{C}}$ language comprised the basics of imperative languages: sequence of statements, assignments, conditionals and `while` loops. It also provided procedure abstraction and call-by-value passing mode. The two data types were integers and functional-style lists (*i.e.*, the value of a list could not be changed).

The contributions of this paper are:

- the introduction of new imperative constructs in Sub$_{\mathcal{C}}$: mutable lists (*i.e.*, in-place modification of lists) and call-by-reference parameters;
- the semantic interpretation of these constructs and of their associated side effects, within equational logic;
- the automatic transformation into equations of Sub$_{\mathcal{C}}$ programs using these constructs.

In this paper, we assume some familiarity with equational logic and `C`. Section 2 presents an example which motivates the introduction of the new constructs in Sub$_{\mathcal{C}}$. We give an overview of *SOS*Sub$_{\mathcal{C}}$ in Sect. 3. In Sect. 4, we briefly discuss the background and main results which conduct to the method implemented in *SOS*Sub$_{\mathcal{C}}$. The method itself is explained in Sect. 5. Section 6 presents the equations generated by our system for the example of Sect. 2. Finally, Sect. 7 concludes and gives some perspectives of this work.

## 2 Motivation

We want to be able to reason about imperative programs. In practice and in our framework, this means

being able to prove properties of $\text{Sub}_{\mathcal{C}}$ programs with side effects on lists and function parameters.

Program MergeSort will serve as a running example to illustrate how *mutable lists* and *reference parameters* are handled in *SOS*$\text{Sub}_{\mathcal{C}}$. Indeed, this program makes an extensive use of side effects on lists. This program is written in the $\text{Sub}_{\mathcal{C}}$ language. The syntax is similar to the C language with some slight differences (*e.g.*, we use element(L) instead of L->element to read the element at the head of list L).

```
list mergeSort(list & L) {
  list secondList, l1, l2;
  if(L == NULL) return NULL;
  else if(next(L) == NULL) return L;
  else {
    secondList = split(L);
    l1 = mergeSort(L);
    l2 = mergeSort(secondList);
    return merge(l1, l2); }
}
list merge(list & L1, list & L2) {
  if(L1 == NULL) return L2;
  else if(L2 == NULL) return L1;
  else if(element(L1) <= element(L2)) {
    L1->next = merge(next(L1), L2);
    return L1;
  } else {
    L2->next = merge(L1, next(L2));
    return L2; }
}
list split(list & L) {
  list pSecondCell;
  if(L == NULL)
    return NULL;
  else if(next(L) == NULL)
    return NULL;
  else {
    pSecondCell = next(L);
    L->next = next(pSecondCell);
    pSecondCell->next =
      split(next(pSecondCell));
    return pSecondCell; }
}
```

Fig. 1: The MergeSort program in $\text{Sub}_{\mathcal{C}}$.

Let us suppose we would like to make some assertions on this program, such that merge returns an ordered list, or more generally that mergeSort actually sorts the list we give to it. We would then need to have a comprehensive understanding of the programming language used and also to do an in-depth analysis of each statement appearing in the program.

If we do this informally, we will say that this recursive sorting program consists of three functions:

- Function mergeSort takes a list L as parameter and returns a sorted version of the list referenced by L. The ampersand (&) preceding the parameter's name denotes that a side effect can occur on this parameter (not to be confused with the C address operator). If the list contains less than two elements, it is already sorted. Otherwise, half of the elements is removed from L and put in secondList, this is done by the function split. Both lists are then recursively sorted before joining them again through the call to function merge.
- Function split takes a list as parameter and removes one out of two elements from it and put them in another list which is returned.
- Function merge takes two sorted lists and merges them in one sorted list which is returned.

However, because this is informal, this is of little help in *proving* the assertions which motivated the analysis. And the formal counterpart of this kind of analysis, *i.e.* deductive reasoning about imperative programs as introduced by [7], is often found tedious by programmers and rarely used in practice. With *SOS*$\text{Sub}_{\mathcal{C}}$, we propose to translate the $\text{Sub}_{\mathcal{C}}$ program into conditional equations. The equations resulting from this translation are showed in Fig. 5, 6 and 7. The assertions on programs are expressed as formulae within equational logic. From the conditional equations, we are then able to reason and prove the assertions in a more natural way. For instance, we proved the two formulae (*cf.* Sect. 6): $\mathit{permutation}(l, \mathit{mergeSort}(l)) = \mathit{true}$ and $\mathit{sorted}(\mathit{mergeSort}(l)) = \mathit{true}$.

In $\text{Sub}_{\mathcal{C}}$, however, up to now, programs were only allowed to manipulate single values—integers and functional lists—and modify their state through assignment. This means that, though a list is a sequence of elements, it was seen as an atomic type; for instance, there was no way to change the value of a particular element without building another list. But this is not sufficient to support some of the most useful features commonly found in imperative languages and at work in the MergeSort program.

Indeed, the three functions of the MergeSort program never duplicate nor create a list element (although the algorithm is not constant in space), they only rearrange the links between elements of the initial list passed to mergeSort. This way of proceeding is very efficient and typical of imperative programming. This relies on an important feature of imperative languages: *side effects*. The objects considered in imperative programs are mutable, *i.e.*, their state can change over the execution of the program. In order to allow this in $\text{Sub}_{\mathcal{C}}$, we add the following two features: mutable lists and reference parameters.

We introduce *mutable lists* in $\text{Sub}_{\mathcal{C}}$, a data structure for lists in which a list is a *pointer* to a *cell*. A cell is made up of two fields: element which can be any $\text{Sub}_{\mathcal{C}}$ data type, and next which is a list. A cell is not a type by itself and is only accessible through its

`element` and `next` fields. We offer the following operators on lists, assuming `L` is a list:

- as expected, operator `element(L)` returns the head of `L`, while `next(L)` returns the tail of `L`;
- operator `add(elt, L)` allows to dynamically create a new cell by specifying its `elt` and `next` fields;
- constructs `L->element` and `L->next` can be used as left values in an assignment to modify the `element` and `next` fields, respectively, of the cell referenced by `L`.

This models the linked list data structure of imperative languages. We find use of it in the three functions of program `MergeSort`.

We also enrich the language $\mathrm{Sub}_{\mathcal{C}}$ with a *call-by-reference* parameter passing mode denoted by an ampersand before the name of a parameter.

Thanks to these additions, the $\mathrm{Sub}_{\mathcal{C}}$ language embraces a wider class of algorithms. Above all, algorithms can be expressed as naturally as it would be done with a standard imperative language, while we are still able to restrict the use of pointers in $\mathrm{Sub}_{\mathcal{C}}$. For instance, there is no generic pointer type, nor pointer arithmetic. Nevertheless, even this restrictive usage of pointers in $\mathrm{Sub}_{\mathcal{C}}$ leads to difficult problems arising from aliasing. An alias occurs at some point during execution of a program when two or more names exist for the same storage location. There are two ways to create aliases in $\mathrm{Sub}_{\mathcal{C}}$:

- Explicitly, since variables of type list are pointers, the assignment of two expressions of type list creates an alias, *e.g.*, `L1 = L2` creates an alias between `L1` and `L2`.
- Through call-by-reference parameters, if the same variable is passed several times in a function call, then all the formal parameters are aliases in the body of the function. For instance, as a consequence to the call `merge(L, L)`, `L1` and `L2` would be aliases in `merge`.

The reason why we are interested in aliases lies on the propagation of side effects. If a side effect occurs on an object aliased, then the side effect must be propagated to all the aliases holding the value of the object whose state has changed.

# 3 Our Framework

We designed the programming language $\mathrm{Sub}_{\mathcal{C}}$ as a subset of the imperative language `C`. The syntax and constructs are very similar though limited in $\mathrm{Sub}_{\mathcal{C}}$. The principle of our approach for proving properties of imperative programs is to translate the $\mathrm{Sub}_{\mathcal{C}}$ source code into conditional equations. *SOS*$\mathrm{Sub}_{\mathcal{C}}$ is the system which performs the translation automatically (under the assumptions discussed in Sect. 5.4). This process is called the axiomatization. This expresses the semantics of $\mathrm{Sub}_{\mathcal{C}}$ programs in equational logic. Next, the equations are used in a proof system to derive properties of the source program. The whole process is illustrated by Fig. 2.

# 4 Related Work

Reasoning about pointers in programs has been challenging static analysis for decades and is still an active research area. One way to tackle the problem, while being of practical interest, is to focus on (recursive) pointer data structures. The various models proposed for reasoning about pointer data structures differentiate on the specific balance between expressiveness and efficient decision procedures.

Separation logic [10] is an extension to Hoare's logic for reasoning about programs with pointers. It postulates that concentrating on the memory cells a program actually accesses is sufficient for reasoning about mutable data structures and simpler than the previous attempts to axiomatize pointer operations. Separation logic introduces an ad hoc logic supposed to cope with the complexity arising from aliasing.

PAL [9] is a decidable logic to reason about structural aspects of a class of data structures. The trade-off for decidability is less expressiveness in the assertions which can be checked and the need to supply detailed *valid* invariants whose validity cannot always be automatically guaranteed.

Separation logic or PAL are specifically designed for reasoning about pointer data structures. With *SOS*$\mathrm{Sub}_{\mathcal{C}}$, our ambition is to deal with more general program properties. As a consequence, we are interested in a more general logic. Indeed, we want to describe mutable lists and reference parameters within equational logics. But these mechanisms are not part, as such, of equational logic: in equational logic, the state of a variable cannot change and the only parameter passing mode available is call-by-value. Moreover, with mutable lists and the `add` operator, we introduce dynamic memory and, as stated in [8], a static analysis of programs with dynamic memory needs the notion of memory cell because the program variables are not sufficient to name all the accessible memory locations. These considerations lead us to a representation of *memory cells* in equational logic.

One way to handle memory cells in equational logic is to model the memory by a *store*, as done by Goguen and Malcolm in [5] for instance. A store is an abstraction which associates values to indices. How-

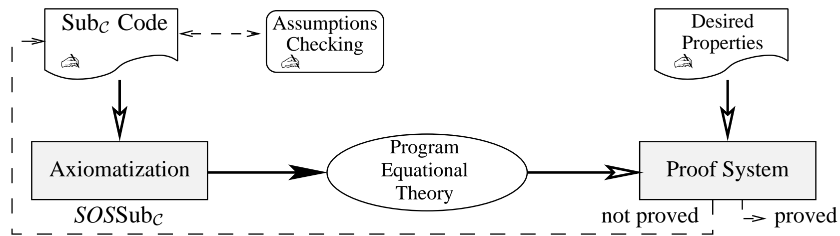


Fig. 2: The proof process.

ever, adopting this view of the memory would complicate the theory within which program proofs are carried out. For instance, within a theory provided with a store, the induction scheme on lists is not as natural as structural induction on the simple list data type.

We address this problem by introducing a *naming scheme* for memory cells dynamically allocated (*cf.* Sect. 5). This allows to name every object accessible to the program and associate to it a variable of the equational logic. Several schemes for naming anonymous objects have already been proposed as in [2]. Their goal is to model any kind of heap allocated structures so as to detect dependences or perform shape analysis (*e.g.*, is it a list? a tree?). Moreover, their naming scheme is designed to suit the static representation of all the memory layouts yielded by a given program. Thus, they are confronted to unbounded structures. As such, they can only approximate the actual layouts.

In our case, any approximation would just not be precise enough since we intend to be semantically equivalent to the source program. Conveniently, two particularities of our method contribute to the design of a simpler naming scheme for memory cells:

- In Sub$_{\mathcal{C}}$, the only dynamically allocated structures are lists, possibly lists of lists. This is expressive enough to represent all kind of data structures while keeping the naming scheme simple.
- Equational logic semantics is dynamic through recursion. Thus, unbounded recursive structures find a very natural expression in equational logic and do not have to be represented in extension.

Nevertheless, this would not be sufficient to ensure the correctness of the translation: we need some further assumptions on the aliases in programs. Actually, exactly determining aliases is a prerequisite for propagating side effects to program values. Unfortunately, this problem is known to be undecidable for languages such as Sub$_{\mathcal{C}}$ [1]. All the existing alias analysis methods (*cf.* [6] for a survey) are approximations of the actual aliases. However, approximations would lead to erroneous translation of programs.

Our paper presents a method to address the translation into equational logic of an imperative language with mutable lists and reference parameters:

- Mutable lists are handled by naming the memory locations accessed in programs. The naming scheme also integrates a level of indirection which allows to take into account alias relations (with the restrictions which are discussed in Sect. 5.4).
- Call-by-reference is dealt with by generating specific functions describing the value of each parameter after the call.

# 5 Translation into Equations

The translation process, called *axiomatization*, is a static analysis of the source code. The goal of the axiomatization is to produce, from each Sub$_{\mathcal{C}}$ function `f` of the source program, an equational definition of a function transfer $f^t$ from input terms to output terms. Let $\phi$ be an isomorphism between values in Sub$_{\mathcal{C}}$ and terms in equational logic, the translation must ensure that `f` with input $I$ gives output $O$ if and only if $f^t(\phi(I)) = \phi(O)$.

Therefore, we are interested in how the statements affect the values manipulated by a program through its variables. All along the axiomatization, we keep a state of the program variables in what we call an *environment*. Environments are sets of equations which synthesize the current state of the computation in all the execution paths of the program (*cf.* [12] for a full description). The Sub$_{\mathcal{C}}$ semantics is expressed as transformations of environments.

## 5.1 Mutable Lists Representation

A list value is a sequence of elements denoting the chaining of the cells in the list. These elements can be integer values, lists, or named list elements. A named list element is introduced each time we need to refer to a list cell with an unknown value, whether this is an integer or a list. This occurs in a function with list parameters because, as we do not always know the context of the call, we do not have access to the value of these parameters in the function and we deal with them as unknown inputs. So, when we need to access

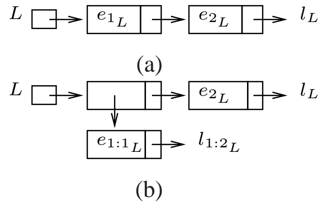


Fig. 3: List decompositions.

```
1  void f(list & L) {
2    list La, Lb;
3    La = next(L);
4    Lb = add(1, add(2, La));
5    La->element = 3;
6    L->next = next(La);
7  }
```

Fig. 4: Mutable lists.

a particular cell in a list parameter, we *decompose* this list in as many named elements as needed to reach the desired cell. The decomposition makes the shape of the list apparent; moreover, the named elements keep track of the memory cells accessible to the program.

For instance, if we need to access `next(next(L))`, supposing `L` is a list parameter, we will decompose[1] `L` in $L = e_{1_L} \cdot e_{2_L} \cdot l_L$ (see Fig. 3(a)). This means that `L` contains at least two elements, $e_{1_L}$ and $e_{2_L}$, which can be integer values or lists. Then, `next(next(L))` denotes $l_L$. As a list, $l_L$ can match the empty list (`NULL`) or a non empty list which could be further decomposed to show more elements if need be. We call $e_{1_L}$, $e_{2_L}$ and $l_L$ named list elements.

As a second example, let us consider the case where `L` is a list of lists and its first element, $e_{1_L}$, is a list that we need to decompose. We simply append another number to denote the decomposition: $e_{1_L}$ becomes $e_{1:1_L} \cdot l_{1:2_L}$. We read $l_{1:2_L}$ as the second element of the first element of `L`. We would then have $L = e_{1:1_L} \cdot l_{1:2_L} \cdot e_{2_L} \cdot l_L$ (see Fig. 3(b)). This naming scheme allows to represent any combination of list of lists.

A $\mathrm{Sub}_{\mathcal{C}}$ variable of type list is a reference to a list cell. We use the special reference `NULL` to indicate that a reference is not associated to any actual cell. We manage a set where each cell is uniquely assigned a reference. New references are introduced when lists are decomposed or new cells are dynamically created.

In order to reconstitute the list associated to a list variable, one just has to follow the links from one reference to the other – starting with the reference which is the value of the list variable – and concatenate elements found in the cells on the path. If a cycle is encountered, we isolate the corresponding part from the main list as a new list with a recursive definition (such as $L = e_1 \cdot \cdots \cdot e_n \cdot L$). This new list can then be used in the value of the main list.

We will use the listing of Fig. 4 to illustrate how we handle mutable lists in *SOS*$\mathrm{Sub}_{\mathcal{C}}$. In the translation environment, we are interested in three sets of values:

- the set of function variables and parameters;
- the set of cells;
- the set of the decompositions of function parameters.

At line 3, after the declarations of the function and local variables, we have:

variables: $L = ref_L \quad La = null \quad Lb = null$
cells: $ref_L = l_L$

Thus, after the initializations, we only have one named element, $l_L$, for the list parameter $L$. Next, before line 4 and after accessing `next(L)`, $L$ is decomposed as $L = e_{1_L} \cdot l_L$ and we have:

variables: $L = ref_L \quad La = ref_1 \quad Lb = null$
cells: $ref_1 = l_L \quad ref_L = e_{1_L} \cdot ref_1$

At line 4, two cells are dynamically created and we obtain:

variables: $L = ref_L \quad La = ref_1 \quad Lb = ref_2$
cells: $ref_2 = 1 \cdot ref_3 \quad ref_3 = 2 \cdot ref_1$
$ref_L = e_{1_L} \cdot ref_1 \quad ref_1 = l_L$

## 5.2 Side Effects on Mutable Lists

Side effects on lists find a natural expression within the chosen representation of mutable lists. A modification of the `element` field, `L->element = x`, will replace by `x` the `element` in the cell referenced by `L`, *i.e.*, the first element of `L`. If necessary, `L` will be decomposed so as to make its first element apparent. A modification of the `next` field, `L->next = x`, will replace by `x` the reference to the next cell in the cell referenced by `L`, *i.e.*, the tail of `L`. If necessary, `L` will be decomposed so as to make its reference to the next cell apparent.

With the example of the listing in Fig. 4, we would obtain at the end of the function:

variables: $L = ref_L \quad La = ref_1 \quad Lb = ref_2$
cells: $ref_1 = 3 \cdot ref_4 \quad ref_2 = 1 \cdot ref_3$
$ref_3 = 2 \cdot ref_1 \quad ref_4 = l_L$
$ref_L = e_{L:1} \cdot ref_4$

And $L$ is decomposed as $L = e_{1_L} \cdot e_{2_L} \cdot l_L$.

We can now reconstitute the value of the lists by following the linking. If at the moment of the call we had $L = e_{1_L} \cdot e_{2_L} \cdot l_L$, then at the end of the function, we have: $L = ref_L = e_{1_L} \cdot ref_4 = e_{1_L} \cdot l_L$; $La = 3 \cdot l_L$; $Lb = 1 \cdot 2 \cdot 3 \cdot l_L$.

[1] List elements are separated by a dot in our notation.

### 5.3 Call-by-Reference Parameters

In the case of a $\text{Sub}_\mathcal{C}$ function without reference parameters, the semantics of the $\text{Sub}_\mathcal{C}$ function is expressed by the definition of an equational function. Each execution path in the $\text{Sub}_\mathcal{C}$ function will generate a conditional equation (see [12]). In order to express the semantics of the call-by-reference passing mode in equational logic, which only has call-by-value functions, we generate in addition a different equational definition for each reference parameter of the $\text{Sub}_\mathcal{C}$ function.

In details, when a procedure is called with a reference parameter $p$ referencing a program object $o$, we generate a call to an equational function whose result is the value of $p$ at the end of the procedure. This result is then assigned to $o$. This means that if $p$ is of type integer, then $o$ is an integer variable of the calling function and the variable $o$ will be modified. If $p$ is of type list, then $o$ is a cell referenced by another cell or by a list variable in the calling function. The cell $o$ and/or the list variable referencing it may be modified depending on the function statements. For instance, *SOS*$\text{Sub}_\mathcal{C}$ generates two functions for the $\text{Sub}_\mathcal{C}$ function `split` of Fig. 1: $split$ for the return value, and $split_L$ for the reference parameter as shown in Fig. 5.

On the side of a calling $\text{Sub}_\mathcal{C}$ function, when a function with reference parameters is called, we add to the translation environment fresh variables whose values are those of the modified objects after the call. We then substitute these fresh variables to the preceding value of the modified objects. The return value is also mapped to a fresh variable. For instance, in function `mergeSort` of Fig. 1, after the call to `split` in `secondList = split(L)`, we will add in the environment two fresh variables: $l_{v_1} = split_L(L)$ and $l_{v_2} = split(L)$. $split_L(L)$ is the value of the object referenced by $L$ after the call to `split`. $split(L)$ denotes the return value of `split`. Next, the values of $L$ and $secondList$ are updated differently since one update is done through an assignment to a variable, and the other one is done through a side effect on a cell. In the case of the assignment, we add to the environment $ref_{v_1} = l_{v_1}$ and we set $L = ref_{v_1}$. For the reference parameter, we update $ref_2 = l_{v_2}$ provided that $secondList = ref_2$.

### 5.4 Assumptions

In our method, undecidability of alias analysis (*cf.* [1]) and unknown function call contexts involve losing alias relation through function calls and `while` loops (which are translated into recursive functions). Thus, in order to do an exact alias analysis, we must consider the following assumptions on how aliases should be used in $\text{Sub}_\mathcal{C}$:

1. Different call-by-reference parameters should not be aliases and if they are lists, their elements should not be aliases; or there should be no ambiguous use of the concerned formal parameters in the callee and in the following code.

2. A list should not be a call-by-reference parameter if there exists aliases for sublists of this list at the moment of the call; or there should be no ambiguous use of the concerned aliases in the callee and in the following code.

3. The variables of type list modified in `while` loops should not have aliases on them or on any of their sublists; or there should be no ambiguous use of the concerned variables in the `while` loop and in the following code.

4. The statements in a `while` loop body should not create aliases; or there should be no ambiguous use of the concerned variables after the `while` loop.

By ambiguous use of two aliases, we mean applying a side effect on one of them and then reading the value of the other one.

Assumptions 1 and 2 ensure that function calls do not interfere with aliasing. This comes from the undecidability of aliasing, but also from the fact that our method is designed to work with “incomplete” programs, *i.e.*, programs whose all input values are not known before execution. Since we possibly do not know the context of a function call, we enforce the safest assumptions regarding aliases. The `while` loops are treated as recursive functions with one reference parameter for each function variable. This leads to Assumptions 3 and 4.

Efficiency put aside, these assumptions can be seen as good practice rules for programming, even though expert programmers would occasionally find them restricting the way they can express the algorithms.

## 6 Example

We report here our experience on using the *SOS*$\text{Sub}_\mathcal{C}$ system to prove the correctness of the `MergeSort` program of Fig. 1. First, we have to check that the four assumptions presented in Sect. 5.4 are not violated by the program. All the alias analysis tools we tried were over-approximating the alias set and failed. Consequently, the checking of the assumptions had to be done manually. However, the translation into conditional equations is automatic since we developed in *SOS*$\text{Sub}_\mathcal{C}$ the algorithms designed from the principles described in this paper.

Since there is no `while` loops in the `MergeSort` program, we are only concerned with the first two assumptions. The proof is straightforward for procedure `merge`. For `split` and `mergeSort` we first need to prove that the list returned by `split` is not aliased to the parameter of `split`.

**Lemma 1** *Let $L$ be an acyclic list. Let $L_{ret}$ be the list returned by `split(L)` and let $L_{mod}$ be the list $L$ after the call `split(L)`. $L_{ret}$ is not aliased to $L_{mod}$.*

**Proof:** We proceed by well-founded induction on the length of the list $L$. The induction hypothesis is that for all list $L'$ whose length is less than the one of $L$, $L'_{\mathrm{ret}}$ is not aliased to $L'_{\mathrm{mod}}$. We continue by case analysis on $L$:

- $L = \mathrm{NULL}$. The lemma is trivially true since the returned value is `NULL`.
- $L = e_1 \cdot L'$. Again, by case analysis on $L'$:
  - $L = e_1 \cdot \mathrm{NULL}$. The lemma is trivially true.
  - $L = e_1 \cdot e_2 \cdot L''$. This corresponds to the last `else` block in procedure `split`. Before the recursive call to `split`, we have: `p2ndCell`$= e_2 \cdot L''$ and `L`$= e_1 \cdot L''$. Then, `split` is called on $L''$ and we have `p2ndCell`$= e_2 \cdot L''_{\mathrm{ret}}$ and `L`$= e_1 \cdot L''_{\mathrm{mod}}$. By induction hypothesis, $L''_{\mathrm{ret}}$ is not aliased to $L''_{\mathrm{mod}}$, so `p2ndCell` is not aliased to `L`, hence the lemma. □

**Theorem 2** *The `MergeSort` program verifies the assumptions.*

**Proof:** We assume acyclic lists as the program does. Lemma 1 ensures that the assumptions hold for `split`. It also ensures that `secondList` is not aliased to `L` in procedure `mergeSort` after the call to `split`. Therefore, `mergeSort` verifies the assumptions. The case of `merge` is simpler since there is no alias in it and the call to it in `mergeSort` is not ambiguous. □

Theorem 2 allows us to safely apply *SOS*Sub$_C$ on program `MergeSort`, which yields the equations of Fig. 5, 6 and 7. These natural equations can be obtained thanks to a simple evaluation, based on boolean simplification and on the structure of lists, of the conditions of the equations.

We used the verification system PVS [11] to prove with these equations several properties of the `MergeSort` program. The proof of correctness within PVS required a lot of interactions with the system as can be expected with a program as optimized as `MergeSort`. On simpler examples, an automatic theorem prover may be used. In our case, we had to introduce several difficult lemmas to guide the proof. For instance, we proved that `split` divides into two partitions the list which is passed to it. We proved that the result of `merge` is a permutation of the concatenation of the two lists passed to it. Moreover, we proved that if these latter are sorted, then the result is sorted. Finally, we could prove that the `MergeSort` program actually sorts any list given to it. To this end, we proved the two formulae: $\mathit{permutation}(l, \mathit{mergeSort}(l)) = \mathit{true}$ and $\mathit{sorted}(\mathit{mergeSort}(l)) = \mathit{true}$ which state that the list returned by *mergeSort(l)* is an ordered permutation of $l$. Functions *permutation* and *sorted* are also defined by equations.

$$
\begin{aligned}
\mathrm{mergeSort}(\mathrm{NULL}) &= \mathrm{NULL}\\
\mathrm{mergeSort}(e_{L:1}\cdot \mathrm{NULL}) &= e_{L:1}\cdot \mathrm{NULL}\\
\mathrm{mergeSort}(e_{L:1}\cdot e_{L:2}\cdot l_{L:3}) &= \\
&\mathrm{merge}(\mathrm{mergeSort}(\mathit{split}_L(e_{L:1}\cdot e_{L:2}\cdot l_{L:3})),\\
&\quad \mathrm{mergeSort}(\mathit{split}(e_{L:1}\cdot e_{L:2}\cdot l_{L:3})))\\
\\
\mathit{split}(\mathrm{NULL}) &= \mathrm{NULL}\\
\mathit{split}(e_{L:1}\cdot \mathrm{NULL}) &= \mathrm{NULL}\\
\mathit{split}(e_{L:1}\cdot e_{L:2}\cdot l_{L:3}) &= e_{L:2}\cdot \mathit{split}(l_{L:3})\\
\mathit{split}_L(\mathrm{NULL}) &= \mathrm{NULL}\\
\mathit{split}_L(e_{L:1}\cdot \mathrm{NULL}) &= e_{L:1}\cdot \mathrm{NULL}\\
\mathit{split}_L(e_{L:1}\cdot e_{L:2}\cdot l_{L:3}) &= e_{L:1}\cdot \mathit{split}_L(l_{L:3})
\end{aligned}
$$

Fig. 5: `mergeSort` and `split` equations.

$$
\begin{aligned}
\mathrm{merge}(\mathrm{NULL}, L2) &= L2\\
\mathrm{merge}_{L1}(\mathrm{NULL}, L2) &= \mathrm{NULL}\\
\mathrm{merge}_{L2}(\mathrm{NULL}, L2) &= L2\\
\mathrm{merge}(e_{L1:1}\cdot l_{L1:2}, \mathrm{NULL}) &= e_{L1:1}\cdot l_{L1:2}\\
\mathrm{merge}_{L1}(e_{L1:1}\cdot l_{L1:2}, \mathrm{NULL}) &= e_{L1:1}\cdot l_{L1:2}\\
\mathrm{merge}_{L2}(e_{L1:1}\cdot l_{L1:2}, \mathrm{NULL}) &= \mathrm{NULL}
\end{aligned}
$$

Fig. 6: `merge` equations (part 1).

# 7 Conclusion

In this paper, we have presented an extension of the Sub$_C$ language which introduces the concepts of pointer and reference in a limited way. The language grants access to references through the *call-by-reference* mechanism and operators on *mutable lists*. We have described the implementation of these new constructs and the underlying semantics.

We also have presented how the *SOS*Sub$_C$ system translated these constructs into first-order conditional equations. We have given an example of this process with the equations obtained from the non trivial program `MergeSort`. Thanks to these equations we proved that this program was sound, with respect to its specification, by showing that the desired properties were inductive theorems of the program equations.

$$e_{L1:1} \leq e_{L2:1} \Rightarrow \text{merge}(e_{L1:1} \cdot l_{L1:2}, e_{L2:1} \cdot l_{L2:2}) = e_{L1:1} \cdot \text{merge}(l_{L1:2}, e_{L2:1} \cdot l_{L2:2})$$

$$e_{L1:1} \leq e_{L2:1} \Rightarrow \text{merge}_{L1}(e_{L1:1} \cdot l_{L1:2}, e_{L2:1} \cdot l_{L2:2}) = e_{L1:1} \cdot \text{merge}(l_{L1:2}, e_{L2:1} \cdot l_{L2:2})$$

$$e_{L1:1} \leq e_{L2:1} \Rightarrow \text{merge}_{L2}(e_{L1:1} \cdot l_{L1:2}, e_{L2:1} \cdot l_{L2:2}) = \text{merge}_{L2}(l_{L1:2}, e_{L2:1} \cdot l_{L2:2})$$

$$e_{L1:1} > e_{L2:1} \Rightarrow \text{merge}(e_{L1:1} \cdot l_{L1:2}, e_{L2:1} \cdot l_{L2:2}) = e_{L2:1} \cdot \text{merge}(e_{L1:1} \cdot l_{L1:2}, l_{L2:2})$$

$$e_{L1:1} > e_{L2:1} \Rightarrow \text{merge}_{L1}(e_{L1:1} \cdot l_{L1:2}, e_{L2:1} \cdot l_{L2:2}) = \text{merge}_{L1}(e_{L1:1} \cdot l_{L1:2}, l_{L2:2})$$

$$e_{L1:1} > e_{L2:1} \Rightarrow \text{merge}_{L2}(e_{L1:1} \cdot l_{L1:2}, e_{L2:1} \cdot l_{L2:2}) = e_{L2:1} \cdot \text{merge}(e_{L1:1} \cdot l_{L1:2}, l_{L2:2})$$

Fig. 7: `merge` equations (part 2).

We have discussed what is needed to ensure the correctness of the translation. We have showed how undecidability of aliasing and design choices lead to assumptions on the creation and use of aliases in Sub$_\mathcal{C}$ programs. Unfortunately, the assumptions can not be decided automatically and we could not enforce these rules in the Sub$_\mathcal{C}$ syntax. The introduction of the assumptions in the method does not invalidate our approach to formal program correctness. Simply, the proof that the assumptions hold are a prerequisite to proving other properties of programs. Obviously, this proof cannot benefit from the equational definition of the program produced by *SOS*Sub$_\mathcal{C}$. This has to be done using other suitable approaches, *e.g.*, traditional deductive reasoning if automated alias analyses failed. We believe that the cost of proving that the assumptions are not violated is low comparatively to the benefit from reasoning in equational logic for all the other program properties.

Our method is successful in providing a formal approach to reason about mutable lists and call-by-reference parameters for a restricted, yet interesting, class of imperative programs. In comparison to more specialized approaches considering larger program classes, ours integrates these aspects in the same setting – equational logic – used for other program properties. Future work should concentrate on refining the conditions of validity of *SOS*Sub$_\mathcal{C}$. The goal is to accept a larger class of programs and make the conditions easier to check. Techniques applied in alias analysis are a good starting point to go beyond these restrictions in particular cases.

The Sub$_\mathcal{C}$ language has still to be improved. Some extensions, as array data type, are on hand. Indeed, arrays can already be implemented over the list model. Others, as offering a greater control to programmers over references (*e.g.* through mechanisms like pointer of pointer), require further study.

*References:*


[1] V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. *ACM SIGPLAN Notices*, 38(1):115–125, Jan. 2003.

[2] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 232–245, Charleston, 1993. ACM Press.

[3] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

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

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

[6] M. Hind. Pointer analysis: haven't we solved this problem yet? In *ACM Workshop on Program analysis for software tools and engineering*, pages 54–61. ACM Press, June 2001.

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

[8] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *ACM Conf. on Programming Language Design and Implementation*, pages 28–40. ACM Press, 1989.

[9] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *ACM Conf. on Programming Language Design and Implementation*, pages 221–231. ACM Press, 2001.

[10] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th Internat. Workshop on Computer Science Logic*, pages 1–19. Springer-Verlag, 2001.

[11] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Internat. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[12] O. Ponsini, C. Fédèle, and E. Kounalis. Rewriting of imperative programs into logical equations. *Science of Computer Programming*, 56(3):363–401, 2005.