Towards Verification of SubC Programs with Side Effects

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

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

Abstract: - As the interest for formal methods grows within industry, the need for convenient and automated tools grows too. SOSSubC is an attempt to help the development of certified programs. It allows formal reasoning about imperative programs by translating programs written in SubC, a simple imperative language, into equations. Programs are then axioms of a logical system within which proofs can be carried out. In this paper, we describe how SOSSubC deals with side effects on lists and procedure parameters.

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. [2] 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 [3]. Indeed, equational logic is well understood, amenable to automation and existing tools are mature enough to conduct proofs within it, possibly without user interaction. It also has the advantage of being well known by industry engineers. And since the imperative paradigm is the most widespread among industrial languages, we developed SOSSubC to fill the gap between imperative programs and equational logic.

SOSSubC 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. We designed the programming language SubC as a subset of the imperative language C. The syntax and constructs are very similar though limited in SubC. The principle of our approach for proving properties of imperative programs is to translate the SubC source code into conditional equations. SOSSubC is the system which performs the translation automatically (under the assumptions discussed in Sect. 4.3). This process is called the axiomatization and expresses the semantics of SubC programs in equational logic. 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. The proof is conducted within a proof system. The whole process is illustrated by Fig. 1.

In a preceding version of SOSSubC [11], the SubC 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 SubC: 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 SubC programs using these constructs.

In this paper, we assume some familiarity with equational logic and the C language. Section 2 presents an example which motivates the introduction of the new constructs in SubC. In Sect. 3, we briefly discuss the background and main results which conduct to the method implemented in SOSSubC. The method itself is explained in Sect. 4. Finally, Sect. 6 concludes and gives some perspectives of this work.
2 Motivation

We want to be able to reason about imperative programs written in Sub$_C$. In practice, this means being able to prove properties of Sub$_C$ programs with, possibly, side effects on lists and procedure parameters.

Program MergeSort will serve as a running example to illustrate how mutable lists and reference parameters are handled in SOS(Sub$_C$). Indeed, this program makes an extensive use of side effects on lists. The program in Fig. 2 is written in the Sub$_C$ language.

```c
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) {
    list secondList = split(L);
    if(L1 == NULL) return NULL;
    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. 2: The MergeSort program in Sub$_C$.

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 we would also need 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 as introduced by [6], is often found tedious by programmers and rarely used in practice.

With SOS(Sub$_C$), we propose to translate the Sub$_C$ program into conditional equations. The equations resulting from this translation are showed in Fig. 4, 5 and 6. 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. 5): permutation(l, mergeSort(l)) = true and sorted(mergeSort(l)) = true.

In Sub$_C$, 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 the MergeSort. This way of proceeding is very efficient and typical of imperative language 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 SubC, we add the following two features: mutable lists and reference parameters.

3 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 [9] 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 [8] 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.

Separation logic or PAL are specifically designed for reasoning about pointer data structures. With SOSSubC, our ambition is to deal with more general program properties, which may include assertions about data structures. As a consequence, we are not interested in an ad hoc logic. Indeed, we want to describe side effects 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, we introduce dynamic memory and, as stated in [7], 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 [4] for instance. A store is an abstraction which associates values to indices. However, 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. 4). 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. 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 SubC, the only dynamically allocated structures are lists, possibly lists of lists. This is expressive enough, if not convenient, to represent all kind of data structures while keeping the naming scheme simple.

• The representation language of programs is equational logic whose semantics is dynamic through recursion. Thus, unbounded recursive structures 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 is a difficult problem. Indeed, this is known to be undecidable [1] for languages, such as SubC, with dynamic memory, assignment, conditional and loop statements. All the existing alias analysis methods (cf. [5] for a survey) are approximations of the actual aliases. Alias analysis has long been studied in order to perform compiling optimization, consequently, methods usually provide safe, from a compiler perspective, approximations. Even so, an approximation does not meet our requirements and would lead to erroneous translation of programs. In order to cope with aliasing, we introduce some assumptions on the SubC-
4 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$_C$ function $f$ of the source program, an equational definition of a function transfer $f^I$ from input terms to output terms. Let $\phi$ be an isomorphism between values in Sub$_C$ and terms in equational logic, the translation must ensure that $f$ with input $I$ gives output $O$ if and only if $f^I(\phi(I)) = \phi(O)$.

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. [11] for a full description). The Sub$_C$ semantics is expressed as transformations of the environment.

This paper focuses on a method to address the translation into equational logic of imperative programs 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 that allows to take into account alias relations (with the restrictions which are discussed in Sect. 4.3). Thanks to this representation of lists, side effects on lists can be naturally expressed.

- Call-by-reference is dealt with by generating specific functions describing the value of each parameter after the call.

4.1 Mutable Lists

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. 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 while analysing the function and we deal with them as unknown inputs. So, when we need to access a particular cell in an unknown 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

\[ L \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n \rightarrow \text{tail} \]

next(next(L)), we will decompose$^1$ L in $L = e_{1L} \cdot e_{2L} \cdot l_L$ (see Fig. 3). This means that L contains at least two elements, $e_{1L}$ and $e_{2L}$, 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 $e_{1L}$, $e_{2L}$, and $l_L$ named list elements. This naming scheme allows to represent any combination of list of lists.

A Sub$_C$ variable of type list is a reference to a list 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.

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. And so it is for the modification of the next field of a list cell.

4.2 Call-by-Reference Parameters

In the case of a Sub$_C$ function without reference parameters, the semantics of the Sub$_C$ function is expressed by the definition of an equational function. Each execution path in the Sub$_C$ function will generate a conditional equation (see [11]). 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 Sub$_C$ function.

For instance, SOSSub$_C$ generates two functions for the Sub$_C$ function split of Fig. 2: split for the return value, and split$_L$ for the reference parameter as shown in Fig. 4.

On the side of a calling Sub$_C$ 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.

---

$^1$List elements are separated by a dot in our notation.
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 \texttt{mergeSort} of Fig. 2, after the call to \texttt{split} in \texttt{secondList = split(L)}, we will add in the environment two fresh variables: \( t_{v_1} = \text{split}_L(L) \) and \( t_{v_2} = \text{split}(L) \).

### 4.3 Assumptions

In our method, undecidability of alias analysis and unknown function call contexts involve losing alias relation through function calls and \texttt{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 \texttt{SubC}:

1. Different call-by-reference parameters should not be aliases and if they are lists, their elements should not be aliases neither; 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 \texttt{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 \texttt{while} loop and in the following code.

4. The statements in a \texttt{while} loop body should not create aliases; or there should be no ambiguous use of the concerned variables after the \texttt{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 \texttt{while} loops are treated as recursive functions, but contrary to \texttt{SubC} functions, they can modify the reference contained in list variables. 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.

### 5 Example

We report here our experience on using the \texttt{SOSSubC} system to prove the correctness of the \texttt{MergeSort} program of Fig. 2. First, we have to check that the four assumptions presented in Sect. 4.3 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 with \texttt{SOSSubC}.

We used the verification system PVS [10] to prove with these equations several properties of the \texttt{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 \texttt{MergeSort}. On simpler examples, an automatic theorem prover may be used. In our case, we had to introduce several lemmas, for instance, we proved that \texttt{split} divides into two partitions the list which is passed to it. Finally, we could prove that the \texttt{MergeSort} program actually sorts any list given to it. To this end, we proved the two formulae: \texttt{permutation(mergeSort(l)) = true} and \texttt{sorted(mergeSort(l)) = true} which state that the list returned by \texttt{mergeSort(l)} is an ordered permutation of \( l \). Functions \texttt{permutation} and \texttt{sorted} are also defined.
by equations.

6 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 SOSSub$_C$ system translated these constructs into first-order conditional equations. We have illustrated 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.

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$_C$ programs. Unfortunately, the assumptions can not be decided automatically and we could not enforce these rules in the Sub$_C$ syntax. 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.

Future work should concentrate on refining the conditions of validity of SOSSub$_C$. 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.

References:


\[ e_{L11} \leq e_{L21} \Rightarrow \text{merge}(e_{L11} \cdot L_{11}, e_{L21} \cdot L_{21}) = e_{L11} \cdot \text{merge}(L_{11}, e_{L21} \cdot L_{21}) \]
\[ e_{L11} \leq e_{L21} \Rightarrow \text{merge}_{L1}(e_{L11} \cdot L_{12}, e_{L21} \cdot L_{22}) = e_{L11} \cdot \text{merge}(L_{12}, e_{L21} \cdot L_{22}) \]
\[ e_{L11} \leq e_{L21} \Rightarrow \text{merge}_{L2}(e_{L11} \cdot L_{12}, e_{L21} \cdot L_{22}) = \text{merge}_{L1}(e_{L11} \cdot L_{12}, e_{L21} \cdot L_{22}) \]
\[ e_{L11} > e_{L21} \Rightarrow \text{merge}(e_{L11} \cdot L_{12}, e_{L21} \cdot L_{22}) = e_{L21} \cdot \text{merge}(e_{L11} \cdot L_{12}, L_{22}) \]
\[ e_{L11} > e_{L21} \Rightarrow \text{merge}_{L1}(e_{L11} \cdot L_{12}, e_{L21} \cdot L_{22}) = \text{merge}_{L1}(e_{L11} \cdot L_{12}, L_{22}) \]
\[ e_{L11} > e_{L21} \Rightarrow \text{merge}_{L2}(e_{L11} \cdot L_{12}, e_{L21} \cdot L_{22}) = e_{L21} \cdot \text{merge}(e_{L11} \cdot L_{12}, L_{22}) \]

Fig. 6: merge equations (part 2).