Mutable Lists and Call-by-reference in Equational Logic

Olivier Ponsini Carine Fédèle
Laboratoire I3S - UNSA - CNRS
2000 route des lucioles
B.P. 121
06 903 Sophia Antipolis Cedex, FRANCE
{ponsini, carine}@i3s.unice.fr

ABSTRACT

As the interest for formal methods grows within industry, the need for convenient and automated tools grows too. *SOS*Sub$_{\mathcal{C}}$ is an attempt to help the development of certified programs. It allows formal reasoning about imperative programs by translating programs written in Sub$_{\mathcal{C}}$, 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 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.

Categories and Subject Descriptors

F.3.1 [**Logics And Meanings Of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics And Meanings Of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics, program analysis*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines, data types and structures*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, correctness proofs*

General Terms

Verification, Languages

1. INTRODUCTION

In computer science, formal methods provide a mathematical framework to logically reason about computer programs and systems. They contribute both theories and tools to the specification and verification of these systems. Their rigorous approach to programming, and more widely to software engineering, gives an invaluable understanding of programs and allows to prove that an implementation meets its specification.

Thanks to formal methods, designers gain an incomparable degree of confidence in their critical applications where human safety, material security or financial costs are involved. 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.

From the beginning of this work [4], we think that *equational logic* is well suited to serve as the mathematical foundations underlying formal methods. And it would have the advantage of being well known by industry engineers. Equational logic is well understood and existing tools are mature enough to conduct proofs within it, possibly without user interaction. Since 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.

When associated with a theorem prover or a proof checker, *SOS*Sub$_{\mathcal{C}}$ is a framework within which developpers can program and prove properties of their programs. *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.

In the preceding version of *SOS*Sub$_{\mathcal{C}}$ [14], 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);
  - 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 the C language. Section 2 presents an example which motivates the introduction of the new constructs in Sub$_{\mathcal{C}}$ and discusses their use. In Section 3, 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 Section 4. Section 5 presents the equations generated by our system for the example of Section 2. Finally, Section 6 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 Sub$_{\mathcal{C}}$ programs with side effects on lists and function parameters.

Program `MergeSort`, excerpted from Foundations of Computer Science [1, Section 2.8 (page 84)], will serve as a running example to illustrate how *mutable lists* and *reference parameters* are handled in *SOS*Sub$_{\mathcal{C}}$. Indeed, this program makes an extensive use of side effects on lists. The program in [1] is written in the C language; in Figure 1, only the syntax has changed, it is now the one of the Sub$_{\mathcal{C}}$ language (*e.g.*, we use `element(L)` instead of `L->element` to read the element at the head of a list `L`).

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. 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 [9], is often found tedious by programmers and rarely used in practice. With *SOS*Sub$_{\mathcal{C}}$, we propose to translate the Sub$_{\mathcal{C}}$ program into conditional equations. The equations

```
list split(list & L);
list merge(list & L1, list & L2);

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(List) == NULL)
    return NULL;
  else {
    pSecondCell = next(L);
    L->next = next(pSecondCell);
    pSecondCell->next =
      split(next(pSecondCell));
    return pSecondCell;
  }
}
```

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

resulting from this translation are showed in Figures 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.* Section 5):

$$permutation(l, mergeSort(l)) = true$$

and

$$sorted(mergeSort(l)) = true$$

In Sub$_{\mathcal{C}}$ however, up to now, the programmer was only allowed to manipulate single values—integers and functional lists—and modify their state through assignment (*cf.* [14]). 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 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 Sub$_{\mathcal{C}}$, we add the following two features: mutable lists and reference parameters.

We introduce *mutable lists* in 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 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 `element` of the cell referenced by `L` (*i.e.*, the head of `L`), while `next(L)` returns the value of the `next` field of the cell referenced by `L` (*i.e.*, the tail of `L`);
- operator `add(elt, L)` allows to dynamically create a new cell whose `element` is `elt` and `next` field is `L`;
- 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 the `MergeSort` program.

We enrich the language Sub$_{\mathcal{C}}$ with a *call-by-reference* parameter passing mode denoted by an ampersand before the name of a parameter. Depending on the type of the parameter, this means:

- either an integer passed by reference, *i.e.*, modifications of the value of the parameter are propagated to the corresponding object in the calling function;
- either a mutable list whose elements can be modified by the callee (but the reference contained in the actual parameter cannot be modified).

Thanks to these additions, the 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 Sub$_{\mathcal{C}}$. For instance, there is no generic pointer type, nor pointer arithmetic. Nevertheless, even this restrictive usage of pointers in 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 [11]. There are two ways to create aliases in 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`; `L1 = next(L2)` creates an alias between `L1` and field `next` of `L2`.
- Through call-by-reference formal 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, consequently 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. RELATED WORK

We designed the programming language Sub$_{\mathcal{C}}$ as a subset of the imperative language `C`. The syntax and constructs are very similar though limited in Sub$_{\mathcal{C}}$. The principle of our approach for proving properties of imperative programs is to translate source code into conditional equations. *SOS*Sub$_{\mathcal{C}}$ is the system which expresses the semantics of Sub$_{\mathcal{C}}$ programs in equational logic and performs the translation.

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 [10], 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 [7] for instance. A store is an abstraction which associates indices to memory cells. 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.

We address this problem by introducing a *naming scheme* for memory cells dynamically allocated (*cf.* Section 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 as in [12, 6]. 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, 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 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 is a difficult problem. Indeed, this is known to be undecidable [11, 5] for languages, such as Sub$_\mathcal{C}$, with dynamic memory, assignment, conditional and loop statements. All the existing alias analysis methods (*cf.* [3, 2, 15] and [8] 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.

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 that allows to take into account alias relations (with the restrictions which are discussed in Section 4.5). 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. TRANSLATION INTO EQUATIONS

The translation process, called *axiomatisation*, is a static analysis of the source code. The goal of the axiomatisation 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 ϕ 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 axiomatisation, we keep a state of the program variables in what we call an *environment*. Environments synthesize the current state of the computation in all the execution paths of the program (*cf.* [14] for a full description). The Sub$_\mathcal{C}$ semantics is expressed as transformations of the environment.

4.1 Mutable lists representation

A list value is a sequence of elements denoting the chaining of the elements in a 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 an unknown list element, whether this is an integer or a list. This occurs in a function with list parameters because we do not know the value of these parameters in the function. We decompose a list parameter in as many elements as needed by the function statements. List elements are separated by a dot in our notation.

For instance, if we need to access `next(next(L))`, we will decompose parameter `L` in $L = e_{L:1} \cdot e_{L:2} \cdot l_{L:3}$. This means that `L` contains at least two elements, $e_{L:1}$ and $e_{L:2}$, which can be integer values or lists. Then, $l_{L:3}$ denotes `next(next(L))`; this is the tail of the list L, which could be further decomposed to show more elements; it can also match the empty list (NULL).

In the case where `L` would be a list of lists and $e_{L:1}$ would be a list which needs to be decomposed, we would simply append another number to denote the decomposition: $e_{L:1}$ would become $e_{L:1:1} \cdot l_{L:1:2}$. We read $l_{L:1:2}$ as the second element of the first element of `L`. This naming scheme allows to represent any combination of list of lists.

A Sub$_\mathcal{C}$ variable of type list is a reference to a *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.

```
1 void f(list & L) {
2   list La, Lb;
3
4   La = next(L);
5   Lb = add(1, add(2, La));
6
7   La->element = 3;
8   L->next = next(La);
9 }
```

Figure 2: Mutable lists.

We will use the listing of Figure 2 to illustrate how we handle mutable lists in *SOS*Sub$_\mathcal{C}$. In the translation environment, we are interested in three sets of values:

- the set of function variables;
- the set of cells;

- the set of the decompositions of function parameters.

At line 3, we have:

variables: $L = ref_L$ $La = \text{NULL}$ $Lb = \text{NULL}$
cells: $ref_L = l_{L:1}$

Thus, L is decomposed as $L = l_{L:1}$. Then, at line 6:

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

And L is decomposed as $L = e_{L:1} \cdot l_{L:2}$.

4.2 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 Figure 2, we would obtain at line 9:

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

And L is decomposed as $L = e_{L:1} \cdot e_{L:2} \cdot l_{L:3}$.

We can now reconstitute the value of the lists by following the linking; if at the moment of the call we had:

$$L = e_{L:1} \cdot e_{L:2} \cdot l_{L:3}$$

then at the end of the function, we have:

$$L = ref_L = e_{L:1} \cdot ref_4 = e_{L:1} \cdot l_{L:3}$$
$$La = 3 \cdot l_{L:3}$$
$$Lb = 1 \cdot 2 \cdot 3 \cdot l_{L:3}$$

4.3 Call-by-reference parameters

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, o is an integer variable of the calling function and this latter will be modified. If p is of type list, then o is the cell referenced by a list variable of the calling function: the cell o is modified but not the list variable of the calling function that reference it. There is a different equational function for each reference parameter of the procedure. And in the case of a $\text{Sub}_\mathcal{C}$ function, as opposed to a procedure, we generate one more equational function for the $\text{Sub}_\mathcal{C}$ function return value. Each execution path in the procedure will generate a conditional equation. The condition is built by merging the path conditions and the list decompositions. Indeed, the list decompositions express the requirements on the structure of the lists (*e.g.*, how many elements) so that the procedure runs correctly.

For instance, $SOS\text{Sub}_\mathcal{C}$ generates two functions for the $\text{Sub}_\mathcal{C}$ function `split` of Figure 1: *split* for the return value, and $split_{list}$ for the reference parameter as shown in Figure 4.

When a procedure with reference parameters is called in a $\text{Sub}_\mathcal{C}$ function, 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 assigned a fresh variable, but it is dealt with as any other expression in $\text{Sub}_\mathcal{C}$, *i.e.*, there is no specific substitution.

For instance, in function `mergeSort` of Figure 4, after the call `secondList = split(L)`, we will add in the environment two fresh variables: $l_{v_1:1} = split_L(L)$ and $l_{v_2:1} = split(L)$.

$split_L(L)$ 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 $ref_{v_1} = l_{v_1:1}$ and we set $L = ref_{v_1}$. For the reference parameter, we update $ref_2 = l_{v_2:1}$ provided that $secondList = ref_2$.

4.4 *while* loops

Loops in $\text{Sub}_\mathcal{C}$ are treated as special recursive functions. They are special in two ways:

- They can return several values: one for each variable modified by an assignment in the loop body.
- Each variable in the current context becomes a parameter of the functions; moreover, if it is a list, it becomes a reference parameter. This is justified because the value of any variable may be use in a `while` loop, and any list may be modified by side effects.

Therefore, we obtain two families of functions. The first one is composed of functions which give the new value, after the loop, of the variables in the current context. There is one function for each variable modified by an assignment in the loop body. The second one is made up of functions which give the new value, after the loop, of lists (*i.e.*, of their cells). There is one function per variable of type list in the current context.

For instance, the `while` loop of Figure 3 will generate three function definitions:

- the definition of $Loop_i(L, i)$ which denotes the value of variable i after the loop;

```
void g(void) {
  list L;
  int i;
  .
  .
  .
  while(element(L) != 0) {
    i = i+1;
    L->element = 0;
    L = next(L);
  }
  .
  .
  .
}
```

Figure 3: A *while* loop.

- the definition of $Loop_L(L, i)$ which denotes the value of variable L after the loop;
- the definition of $Loop_{ref_L}(L, i)$ which denotes the value, after the loop, of the list that was referenced by L before the loop.

4.5 Assumptions

In our method, undecidability of alias analysis (*cf.* [11, 5]) involves 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 Sub$_\mathcal{C}$:

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 `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, but contrary to Sub$_\mathcal{C}$ functions, they can modify the reference contained in list variables. This leads to Assumptions 3 and 4.

These assumptions do not restrict the scope of the algorithms which can be implemented in Sub$_\mathcal{C}$, even though expert programmers would occasionally find them restricting the way they can express the algorithms. On the other side, these assumptions can be seen as good practice rules for programming.

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

Figure 4: `mergeSort` and `split` equations.

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

Figure 5: `merge` equations (part 1).

5. EXAMPLE

We developed in *SOS*Sub$_\mathcal{C}$ the algorithms designed from the principles described in this paper. The checking of the four assumptions presented in Section 4.5 has to be done manually, but the translation into conditional equations is automatic. *SOS*Sub$_\mathcal{C}$ gives for the `MergeSort` program of Figure 1 the equations of Figures 4, 5 and 6. These very 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.

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

Figure 6: `merge` equations (part 2).

We used the verification system PVS [13] to prove with these equations several properties of the `MergeSort` program. 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, if these latter are sorted, then the result is sorted. Finally, we proved that the `MergeSort` program actually sorts any list given to it. To this end, we proved the two formulae: $permutation(l, mergeSort(l)) = true$ and $sorted(mergeSort(l)) = true$ which state that the list returned by *mergeSort(l)* is an ordered permutation of l. Functions *permutation* and *sorted* are also defined by equations.

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 *SOS*Sub$_C$ system translated these constructs into semantically equivalent 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.

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 restrictions can not be decided automatically and we could not enforce these rules in the Sub$_C$ syntax.

Future work should concentrate on refining the conditions of validity of *SOS*Sub$_C$. 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$_C$ 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.

7. REFERENCES

[1] A. V. Aho and J. D. Ullman. *Foundations of Computer Science : C edition*. W. H. Freeman & Co., 1994.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, may 1994.

[3] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41. ACM Press, 1979.

[4] F. C. and K. E. Automatic proofs of properties of simple $C--$ modules. In *14th IEEE International Conference on Automated Software Engineering*, October 1999.

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

[6] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, Jan. 10–13, 1993. ACM Press.

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

[8] M. Hind. Pointer analysis: haven't we solved this problem yet? In *ACM SIGPLAN–SIGSOFT workshop on Program analysis for software tools and engineering: June 18–19, 2001, Snowbird, Utah, USA: PASTE'01*, pages 54–61. ACM Press, 2001.

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

[10] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 28–40. ACM Press, 1989.

[11] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.

[12] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31. ACM Press, 1988.

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

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

[15] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. ACM Press, Jan. 1996.