Using a Set Constraint Solver for Program Verification

Maximiliano Cristiá
Universidad Nacional de Rosario and CIFASIS, Rosario, Argentina
cristia@cifasis-conicet.gov.ar

Gianfranco Rossi
Università di Parma, Parma, Italy
gianfranco.rossi@unipr.it

Claudia Frydman
Aix Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296,13397, France
claudia.frydman@lsis.org

$\{log\}$ is a constraint solver for an expressive theory of finite unbounded sets and binary relations. It is a complete solver for an important fragment of formulas based on operators such as equality, membership, union, domain, composition, etc. where terms are finite unbounded extensional sets and binary relations. It is also a complete solver for formulas based on equality and membership where terms are extensional and restricted intensional sets. As such $\{log\}$ can automatically prove a number of theorems of the theory of finite unbounded sets and relations. In turn, the theory of finite sets and relations is known to be a very good specification language for many programs. In this paper we show how $\{log\}$ can be used as an effective tool for automatically discharging verification conditions produced during the formal functional verification of imperative programs. A case study shows the application of $\{log\}$ to the verification of several list functions.

1 Introduction

$\{log\}$ (pronounced 'setlog') is a freely available constraint solver for an expressive theory of finite unbounded sets and binary relations, implemented in Prolog [24]. With the constraint language supported by $\{log\}$ it is possible to write a large subset of the formulas that can be expressed in formal specification languages such as Z [27] and B [1] (basically all set and relational operators, except for the transitive closure, are supported). Besides, $\{log\}$ is a complete solver for a non-trivial fragment of its input language (i.e. it can compute a finite representation of all the solutions of a given formula). As such it behaves as a specialized SMT solver for a theory of finite sets and relations. Therefore, $\{log\}$ is able to automatically prove theorems of set theory and relation algebra.

Formal specification languages based on set theory, such as Z and B, have been proposed as ideal vehicles for the formal specification of large classes of programs. The underlying idea is that set-based specifications are concise and accurate abstractions of many program behaviors. Widespread used data structures such as lists, arrays and maps can be easily described with sets and set operators.

Usually, these specifications give the pre- and post-conditions the implementation must verify. The same language can be used to state loop invariants and program assertions in general, in a Hoare-Floyd-like verification framework. As a consequence, the verification conditions (or proof obligations) derived from such a framework will take the form of theorems of set theory. Observe that finite unbounded sets and binary relations accurately represent the state of programs managing data structures such as lists, because they are finite but unbounded (in the sense that the program can keep adding nodes to the list).

In this paper we show how $\{log\}$ can be used as an effective tool to discharge such verification conditions. In our framework, programs are written in an object-oriented abstract programming language and specifications and loop invariants are written in a language similar to the base logics underlying Z and B.

Submitted to:
HCVS 2017

Programmers are assumed to provide specifications and loop invariants and to produce the verification conditions; while $\{log\}$ will automatically prove them. The automatic derivation of verification conditions needs to be further investigated. The core of the paper is a case study where we use $\{log\}$ to prove the partial functional correctness of an abstract data type (ADT) encoding lists and of six subroutines using instances of this ADT (such as list equality and membership, mapping a function over a list, etc.).

The original contribution of this paper is to show how the latest version of $\{log\}$ (including binary relations and restricted intensional sets [10, 11]) can be used as an effective tool to support set-based program verification. We do not claim that our solution is somewhat "better" than other known solutions; rather, we claim that using $\{log\}$ represents a viable challenging alternative to other known approaches. In particular, we show how set unification and set theory as supported by $\{log\}$ allow to give precise yet concise formulas capturing the semantics of several important instructions of imperative programming languages. Furthermore, in the case of $\{log\}$ these formulas fall inside the decision procedures implemented by the tool thus making it possible to automatically discharge verification conditions.

Using set constraints for program analysis and verification can be traced back at least to the nineties [20, 2, 22]. The differences between these proposals and $\{log\}$ have been already analyzed [25, 16, 17]. Basically, we have chosen to use $\{log\}$ because: *(i)* it provides general forms of set terms (e.g. nested and partially specified sets) and a relatively rich collection of set-theoretical operations (including set unification, and basic relational operators); *(ii)* all these features are provided as first-class citizens of the language, all endowed with a precise set-theoretical semantics; *(iii)* a large fragment of the formulas that can be expressed in $\{log\}$ can be decided using $\{log\}$'s constraint solver; and *(iv)* when $\{log\}$ provides a decision procedure it actually provides a complete solver which is able to compute solutions, i.e. assignments of values to the free variables of the input set formula.

Specifically, in this paper we apply $\{log\}$ to programs accessing *dynamic data structures*, such as linked lists. Separation logic [23] has been proposed and used with outstanding results to verify programs using dynamic data structures. There are several powerful tools implementing—in different ways and for different purposes—a range of analysis based on separation logic. Some of them rest on first-class theorem provers such as Coq [9, 6]; others implement some decision procedures or heuristics or with extra input from the user require less user work during the proofs [4, 29]; and yet others perform fully automatic analysis although their scope is limited to detect some specific errors (e.g. memory safety) [8]. The kind of specifications that we use in our case study bear some relation to separation logic as we predicate only about the state of the data structure being implemented (see Section 5.2). However, our specifications are quantifier-free first order formulas over a theory of finite sets, and we use general-purpose set and relational operators instead of the more specific separation logic connectives (e.g. separating implication).

Other logics are used to verify program correctness even at the presence of dynamic data structures. Zee et al. [30] use the Jahob system to prove full functional correctness of linked data structures. Jahob specifications are based on classical higher-order logic including sets and relations. However, many specifications fall inside of an undecidable fragment of higher-order logic. Hence, discharging some verification conditions require user help. Attempts to reduce the specification language fail because it would not be possible to express some properties. As our case study suggests, $\{log\}$ might overcome these limitations by providing an expressive enough specification language whose formulas fall inside a decision procedure. As proof back-end, Jahob relies on a range of first-order and SMT provers. Droos and Moy [19] use the SPARK technology to automatically prove that an efficient imperative implementation is a correct refinement of the abstract specification. In this setting, specifications are based on abstract data structures such as sets and maps. As Jahob, the SPARK verifier also rests on automatic provers such as Alt-Ergo, CVC4 and Z3. $\{log\}$'s input language bears some relation to relation algebra. Berghammer et

al. [5] verify imperative programs implementing relation-based discrete structures by combining relation algebra with automated theorem proving.

As can be seen, SMT solvers are routinely used for diverse forms of program verification [15]. For example, VS^3 [28] uses Z3 to discover program invariants. Lahiri and Qadeer [21] propose a logic, implemented over Simplify and Z3, that facilitates precise, automated, and efficient reasoning about many heap-intensive programs. The logic is based on bounded quantification over interpreted sets. Although $\{log\}$ does not directly support bounded quantification over interpreted sets, they can be encoded by means of Restricted Intensional Sets (RIS) [11]. In general, SMT solvers do not directly support interpreted sets or if they do the admissible formulas are not as expressive as $\{log\}$'s (e.g., they do not support dom, ran, etc., as in [3]).

Besides SMT solvers and general proof assistants, there are approaches to program verification based on representing programs as Horn clauses. For example, solvers such as Duality, HSF, SeaHorn, and μZ, are used to transform imperative programs into Horn clauses [7]. De Angelis et al. [14] use Constraint Handling Rules to transform CLP programs that manipulate integer arrays for verifying properties of imperative programs. All these proposals, however, do not exploit the high-level expressiveness power of sets and set-based formulas.

2 Introduction to $\{log\}$

In this section we introduce the constraint language provided by $\{log\}$ and show some examples of how $\{log\}$ can be used as a constraint solver for set theory formulas. Full and formal accounts of the tool and the theory behind it can be found elsewhere [16, 10, 11].

$\{log\}$ is a *Constraint Logic Programming (CLP)* language, whose constraint domain is that of *hereditarily finite sets*—i.e., finitely nested sets that are finite at each level of nesting. $\{log\}$ allows sets to be *nested* and *partially specified*—e.g., set elements can contain unbound variables, and it is possible to operate with sets that have been only partially specified. $\{log\}$ provides a collection of primitive constraint predicates, sufficient to represent all the most commonly used set-theoretic and relation operations—e.g., union, intersection, domain, composition.

$\{log\}$ admits the following set terms: variables; the empty set ($\emptyset$); and the extensional set constructor ($\{\cdot \sqcup \cdot\}$). Sets are untyped, i.e. they can contain elements of any sort; in particular they can be numbers, strings, ordered pairs, and set terms. Literals are formed by several set and relation operators including equality ($=$), set membership ($\in$), set union ($un(A,B,C)$ interpreted as $C = A \cup B$), domain of a relation ($dom(R,A)$, i.e. $\mathrm{dom}\, R = A$), etc. The negation of each operator is also available: $\neq$, $\notin$, *nun* (negation of *un* constraints), *ndom* (negation of *dom* constraints), and so forth. $\{log\}$ constraint formulas are formed by conjunctions ($\wedge$) and disjunctions ($\vee$) of literals.

The first parameter of the extensional set constructor is a set element and the second parameter (called *set part*) is a set term. Then, $\{a_1 \sqcup \ldots \{a_n \sqcup A\} \ldots\}$ is interpreted as $\{a_1, \ldots, a_n\} \cup A$. The term $\{a_1 \sqcup \ldots \{a_n \sqcup A\} \ldots\}$ is written concisely as $\{a_1, \ldots, a_n \sqcup A\}$ and when A is the empty set we write $\{a_1, \ldots, a_n\}$. Any set element and the set part can be variables. This implies that extensional sets in $\{log\}$ are finite but unbounded (because the set part can be a variable).

When a formula is entered in $\{log\}$ it attempts to find all its solutions. If the formula lays inside of one of the implemented decision procedures, then the tool will return a compact representation of all its possible solutions; if the formula is unsatisfiable, $\{log\}$ will return *false*. In particular, $\{log\}$ implements *set unification* [18]. Specifically, $\{log\}$ provides a form of E-unification, called $(Ab)(Cl)$-unification, where (Ab) (Absorption) and (Cl) (Commutativity on the left) are the identities of the underlying equa-

tional theory that capture the properties of the set constructor $\{\cdot \sqcup \cdot\}$, i.e. the fact that the ordering and repetitions of elements in a set are immaterial. $(Ab)(Cl)$-unification allows to compute a complete set of E-unifiers for equalities involving general set terms of the form $\{a_1, \ldots a_n \sqcup A\}$.

Example 2.1 *Given the equality $\{\{x,1\} \sqcup A\} = \{y \sqcup B\}$, where x, y, A and B are variables, $\{log\}$ returns the following four independent solutions that constitute the minimal complete set of E-unifiers for the given unification problem: (i) $y = \{x,1\}, B = A$; (ii) $A = \{\{x,1\} \sqcup B\}, y = \{x,1\}$; (iii) $y = \{x,1\}, B = \{\{x,1\} \sqcup A\}$; (iv) $A = \{y \sqcup N\}, B = \{\{x,1\} \sqcup N\}$, where N is a new fresh set variable. Note that if, for instance, $y \neq \{x,1\} \wedge y \notin A$ is conjoined, then the answer of $\{log\}$ is false.*

$\{log\}$ can be used as an effective automated theorem prover for (finite) set theory (including binary relations). Given that $\{log\}$ is a satisfiability solver, if you want to prove that a formula is a theorem, you have to negate it and wait for a *false* answer.

Example 2.2 *$\{log\}$ can be used to prove that $\cup$ is commutative (i.e. $A \cup B = B \cup A$) whose negation in $\{log\}$ writes as $un(A,B,C) \wedge nun(B,A,C)$ (for which it returns false).* □

In $\{log\}$ (finite) binary relations are just set of ordered pairs. Hence, binary relations and sets can be freely combined. In particular, set operators can be applied to binary relations and there are relation operators that take sets as parameters. This feature leverages the expressiveness of the language.

Example 2.3 *The equality $(A \lhd R)[B] = R[A \cap B]$ (where A and B are sets, R is a binary relation, $\lhd$ is domain restriction, and $\cdot[\cdot]$ is relational image), is a known theorem of binary relations [26]. Its negation can be written in $\{log\}$ as follows:*

$$dres(A,R,N_1) \wedge rimg(N_1,B,N_2) \wedge inters(A,B,N_3) \wedge nrimg(R,N_3,N_2) \tag{1}$$

for which $\{log\}$ answers false immediately. □

$\{log\}$ can tell by itself when a set is a binary relation by looking at the constraints where it participates in. However, if users want to force this constraint $\{log\}$ offers the $rel(R)$ literal which forces R to be a binary relation (i.e. a set of ordered pairs). Besides, $\{log\}$ offers the $pfun(f)$ constraint which forces f to be a partial function (i.e. a set of ordered pairs where no first component is in more than one pair).

Note that the ASCII version of $\{log\}$ input language somewhat differs from the notation used across this paper. We do so because we think a more mathematical notation would help readers to better understand our work.

Example 2.4 *The ASCII of the formula $un(\{(x, \text{'hello world'}) \sqcup A\}, b, c) \wedge c \neq \emptyset$ is:*

```
un({[X,hello_world] / A},B,C) & C neq {}
```

where variables must start with a capital letter while constants start with a lower letter, as in Prolog. □

3 The Programming Language

In this paper we show how {log} can be used as an effective tool for proving partial functional correctness of imperative programs dealing with lists. We use an abstract programming language, simply called $\mathscr{P}$, whose syntax and semantics will be given informally. Some sample programs can be found in Figures 2 and 4 and in Appendix A.1 and A.2.

In this programming language programmers can define ADTs by giving the subroutines in their interfaces. The syntax for defining ADT's is sketched in Appendix A.2 because is not needed here. ADTs can be parameterized by a type (such as C++ templates). If T is an ADT or a primitive type, then

add(T e)	Adds e to the end of the list.
fst()	The internal cursor is set to the left of the (possible) first element of the list.
T next()	Returns the next element in the list, if the iterator has been initialized (i.e. fst() has been called) and if there is such an element, and increments the internal cursor by one; otherwise the behavior is not specified. The client is responsible for initializing the iterator and for checking that there are more element ahead before calling next().
more()	Returns true iff there are more elements in the list w.r.t. the current position of the internal cursor; or if the list is not empty and the internal cursor has not been initialized yet (i.e. fst() has not been called). When the list is empty it returns false.
rpl(T e)	Replaces the last element returned by next(), if any, with e; otherwise it does nothing.
del()	Deletes the list; i.e. it becomes the empty list.

Figure 1: Public interface of the List ADT

T x declares x as a variable of type T. If T is an ADT, then x needs to be bound to an *instance* of T by means of an assignment statement and the instance creator: x := **new** T. If f(...) is a subroutine in the interface of T, depending on zero or more parameters, then x.f(...) has the same semantics given by object-oriented languages. Variables whose type are ADTs are always treated as references (like in Java). null is a value for any variable of an ADT type indicating that no instance has been bound to it.

Besides, the language has the following standard control and procedural abstractions: assignment (:=), **skip**, **if** – **then** – **else** – **end if** (where the **else** part is optional), **while** – **do** – **end while**, **function** – **end function**, **procedure** – **end procedure** and **return**, with their intuitive semantics. **function** blocks are used to define subroutines that return a value of some type (either ADT or primitive) that is indicated after this keyword and before the function's name. The **return** keyword is allowed only inside a **function** block. **procedure** blocks are used when the subroutine is not meant to return anything to the caller.

3.1 An ADT for lists

The ADT for lists that we will use is called List and it is assumed to be parameterized by some type T which is the type of the elements stored in the list. Since the actual parameter does not play an important role in relation with proving partial correctness of the programs using List, we will not mention it explicitly whenever possible.

The (public) interface of List is constituted by the subroutines given in Figure 1. The formal specification of these subroutines is discussed in Section 5.2. fst(), next() and more() work together as a sequential iterator. Internally, List maintains a cursor pointing to the left of the next element to be iterated over. Note that the client must call more() and fst() before calling next() for the first time, and more() before successive calls to it; and that it must call next() before calling rpl(). In other words, a typical correct usage of the iterator is: fst() → more() → next() → more() → next() → ... until more() returns false, while calls to rpl() are also possible after the first call to next(); and fst() can be called at any moment to reset the iterator. Otherwise the behavior of the caller could be erroneous.

When del() is called, more() will return false until add() is called. Whenever an assignment of the form s := **new** List is executed, an empty list is bound to s. When an instance of List is created, more() returns false (and thus the behavior of next() is not specified).

As can be seen, the only way programmers have to scan a list is by means of the abstract iterator defined by List. Since iterating over lists in the correct way is one of the main issues when proving

correctness of programs dealing with them, we will discuss a simple but important property of iterators, on which our correctness conditions will depend. When an iterator is used to iterate over a list, the list can be divided into two parts: the leftmost one has already been iterated over and thus its elements have already been processed, we call it the *processed* part; and the rightmost part whose elements are (possibly) going to be iterated over, we call it the *remaining* part. In particular when fst() is called, the processed part is empty and the remaining part equals the whole list; and when more() returns false, the remaining part is empty and the processed part is equal to the list.

4 The Specification Language

As we have said, we aim at proving functional properties of programs using instances of List. We do it by proving that programs verify their specifications. Specifications are given as formulas over the theory of sets supported by $\{log\}$. Firstly, we write these formulas using the more or less standard language of set theory and, secondly, we translate them into the constraint language of $\{log\}$ (see several examples of this translation in Section 5, Appendix A and in [13]). We do so because we assume readers are more familiarized with the language of set theory. As we will see, this translation is straightforward.

More precisely, the atomic predicates of our specification language are set equality ($=$), set membership ($\in$) and their negations ($\neq$ and $\notin$). In turn, set expressions are formed with variables; the empty set ($\emptyset$); extensional sets $\{a_1,\ldots,a_n\}$ and $\{a_1,\ldots,a_n \sqcup A\}$, where A is a variable; set union ($\cup$), intersection ($\cap$); domain (dom) and range (ran) of a binary relation, and so forth. If f is a set representing a partial function, then $f(x)$ denotes function application. Set elements can be ground elements, variables, ordered pairs (noted $(\cdot,\cdot)$ or $\cdot \mapsto \cdot$) and other finite sets. Sets are untyped; that is they can contain elements of different sorts. Formulas in this specification language are formed as usual with the standard logical symbols: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Longrightarrow$) and equivalence ($\Longleftrightarrow$). The language includes a special symbol (*ret*) denoting the value returned by functions when this value is the result of evaluating the expression of a **return** statement.

Now we are going to explain how instances of List are modeled at specification level. In set theory a list can be modeled as a partial function whose domain is an interval of natural numbers starting at 1.

Definition 4.1 *Let $\mathbb{N}_1 \nrightarrow X$ be the set of all partial functions from $\mathbb{N}_1 \triangleq \mathbb{N} \setminus \{0\}$ onto the set X. Then, the set List defined as:*

$$s \in List \iff s \in \mathbb{N}_1 \nrightarrow X \wedge \exists n \in \mathbb{N} : \mathrm{dom}\, s = [1,n] \tag{2}$$

represents the set of all possible lists whose elements belong to X.

Note that the empty list is the empty set as $n = 0$ satisfies (2). In this way lists are sets of ordered pairs. For example, the list $\langle e_1,e_2,e_3\rangle$ corresponds to the set $\{(1,e_1),(2,e_2),(3,e_3)\}$. In this representation of lists, the first components of the ordered pairs are called *indexes*.

With this representation we can easily use all the set and relational operators defined in our specification language. For example, list equality becomes trivial as it is reduced down to set equality; list membership becomes set membership as $(\_,e) \in s$ or $e \in \mathrm{ran}\, s$; and so forth.

Besides, when specifying programs using instances of List we can assume that List's interface is correct (see Section 5.2) thus making modeling even easier. For example, the property that a list can be divided into two parts as it is iterated over with the abstract iterator, can simply be modeled as $s = s_p \cup s_r$ (or $s = \{a_1,\ldots,a_n \sqcup s_r\}$ when s_p is known to be $\{a_1,\ldots,a_n\}$, $n \geq 1$), where s_p represents the processed part and s_r the remaining part. At specification level it is not always necessary to precisely characterize

PRE-CONDITION $true$
function Bool listEq(List s, t)
  s.fst(); t.fst()
  INVARIANT $s \in \_ \nrightarrow \_ \wedge t \in \_ \nrightarrow \_ \wedge s = s_p \cup s_r \wedge s_p \parallel s_r \wedge t = t_p \cup t_r \wedge t_p \parallel t_r \wedge s_p = t_p$
  while s.more() $\wedge$ t.more() $\wedge$ s.next() = t.next() **do**
    skip
  end while
  return $\neg$s.more() $\wedge \neg$t.more()
end function
POST-CONDITION $ret \iff s = t$

Figure 2: listEq returns true if and only if s and t are equal

those parts, if we work under the assumption that the interface is correct. In effect, under this assumption we know for sure that the list can be divided into these two parts and that they have some properties. In turn, if at specification level we do not need these properties (for instance to prove that some formula is a loop invariant), then we do not write them down. These are some examples of List's properties that users may use when needed:

- s, s_p and s_r are partial functions.
- s_p and s_r are disjoint (i.e. $s_p \parallel s_r$ and $disj(s_p, s_r)$ in $\{log\}$).
- Assuming next() is correct, we do not need to characterize the next element to be processed. We need to state that there is an element z to be processed ($z \in s_r$ or $s_r = \{z \sqcup s_r^1\}$) and, some times, we need to identify its components ($(x,y) \in s_r$ or $s_r = \{(x,y) \sqcup s_r^1\}$).

5 $\{log\}$ for Functional Program Verification: Case Study

In this section we show part of a case study we conducted in order to evaluate $\{log\}$ as a program verifier (the remaining part is in the Appendix). The case study includes the verification of all subroutines in List's interface (as described in Section 3.1) and six subroutines processing instances of List. All the code is written in $\mathscr{P}$ (Section 3). The external subroutines include a **while** loop which iterates over one or two lists by means of the abstract iterator provided by the List interface. For each subroutine we give its specification in the form of pre- and post-conditions and the loop invariants, all in the specification language outlined in Section 4. Then, assuming the implementation of List is correct, we prove that the invariants are preserved and that the post-conditions are met. Secondly, we prove that the implementation of each subroutine in the interface of List is correct, by following a similar methodology. All these proofs are made by feeding $\{log\}$ with the verification conditions.

5.1 List equality

One of the six subroutines making use of the List ADT is a function implementing list equality between two instances of List (Figure 2). As can be seen, the pre- and post-conditions and the loop invariant are given as formulas of our specification language (highlighted in gray). These are annotations to the program. We will not discuss the code as we assume is straightforward. Instead we will discuss some issues regarding the relation between the specification and the code.

If v is a program variable then v is the abstraction of this variable at the specification level. As the specification language is untyped, variables of type T are abstracted as variables without specifying any sort for them; and variables of type List are abstracted as variables of sort set as discussed in Section 4.

Pre-condition Observe that since the programming language enforces a type system, a subroutine cannot be called by passing to it an actual parameter of the wrong type. This makes unnecessary to enforce those types at specification level in order to prove the *functional* correctness of these programs. In other words, the correctness of these programs cannot be jeopardized by receiving a parameter of the wrong type. Therefore, for example, in the pre-condition of listEq we do not need to state that s satisfies (2) because it will, no matter what, since this is enforced by the compiler. Hence, in this example the pre-condition is simply *true*.

Loop invariant The loop invariant uses the property yielded by the iterator discussed in Section 4. That is, both s and t are divided into two parts which in this case verify that $s_p = t_p$. In effect, the loop keeps running as long as the corresponding elements are equal, so the processed part of s, in each iteration, must be equal to the processed part of t. In this case, however, we need to state that both s and t are partial functions and that s and t are partitioned by s_p, s_r and t_p, t_r, respectively.

Post-condition The post-condition makes use of the special variable *ret*. The meaning of this variable is equivalent to have assigned $\neg$s.more() $\wedge \neg$t.more() to variable ret and then returning this variable instead of the expression (i.e. **return** ret).

Verifying the specification Now we are going to explain how we prove that listEq verifies its specification (in the context of partial correctness). In general we follow the Hoare-Floyd approach to proving partial correctness. Then, we assert pre- and post-conditions for each statement and prove whether or not pre-conditions imply post-conditions. In order to shorten the presentation we will do some simplifications when the assertions are simple and when proofs are trivial (in general we will focus on proving invariant preservation and post-conditions after loop termination).

The post-condition after the first two calls to fst() is: $s = s_p \cup s_r \wedge s_p = \emptyset \wedge t = t_p \cup t_r \wedge t_p = \emptyset$, which trivially implies the loop invariant.

Next, we must prove that the loop invariant is preserved after each iteration if the loop condition holds. In this case the loop condition is a little bit tricky because the clause s.next() = t.next() produces a side-effect consisting in advancing both iterators (note that this clause is executed only when the first two clauses hold). Therefore, the verification condition is as follows:

$$\begin{array}{ll}
s_r = \{(x,y_1) \sqcup s_r^1\} \wedge t_r = \{(x,y_2) \sqcup t_r^1\} \wedge y_1 = y_2 & \text{[condition holds]} \\
\wedge\, s \in _ \nrightarrow _ \wedge t \in _ \nrightarrow _ \wedge s = s_p \cup s_r \wedge s_p \parallel s_r \wedge t = t_p \cup t_r \wedge t_p \parallel t_r \wedge s_p = t_p & \text{[inv. before]} \\
\Longrightarrow s \in _ \nrightarrow _ \wedge s = \{(x,y_1) \sqcup s_p\} \cup s_r^1 \wedge \{(x,y_1) \sqcup s_p\} \parallel s_r^1 & \text{[inv. after]} \\
\quad \wedge t \in _ \nrightarrow _ \wedge t = \{(x,y_2) \sqcup t_p\} \cup t_r^1 \wedge \{(x,y_2) \sqcup t_p\} \parallel t_r^1 \wedge \{(x,y_1) \sqcup s_p\} = \{(x,y_2) \sqcup t_p\} &
\end{array} \tag{3}$$

If we want to use $\{log\}$ to discharge this verification condition we need to translate its negation while

writing the implication as a disjunction, resulting in:

$$\begin{array}{l} s_r = \{(x,y_1) \sqcup s_r^1\} \wedge t_r = \{(x,y_2) \sqcup t_r^1\} \wedge y_1 = y_2 \\ \wedge pfun(s) \wedge pfun(t) \wedge un(s_p,s_r,s) \wedge disj(s_p,s_r) \wedge un(t_p,t_r,t) \wedge disj(t_p,t_r) \wedge s_p = t_p \\ \wedge (npfun(s) \vee nun(\{(x,y_1) \sqcup s_p\}, s_r^1, s) \vee ndisj(\{(x,y_1) \sqcup s_p\}, s_r^1) \\ \quad \vee npfun(t) \vee nun(\{(x,y_2) \sqcup t_p\}, t_r^1, t) \vee ndisj(\{(x,y_2) \sqcup t_p\}, t_r^1) \vee \{(x,y_1) \sqcup s_p\} \neq \{(x,y_2) \sqcup t_p\}) \end{array} \tag{4}$$

which, when fed into $\{log\}$, makes it return *false* (i.e. (3) holds).

Finally, we must prove that upon termination of the loop its invariant implies the post-condition:

$$\begin{array}{ll} (s_r = \emptyset \vee t_r = \emptyset \vee s_r = \{(x,y_1) \sqcup s_r^1\} \wedge t_r = \{(x,y_2) \sqcup t_r^1\} \wedge y_1 \neq y_2) & \text{[termination]} \\ \wedge\, s \in \_ \nrightarrow \_ \wedge s = s_p \cup s_r \wedge s_p \parallel s_r \wedge t \in \_ \nrightarrow \_ \wedge t = t_p \cup t_r \wedge t_p \parallel t_r \wedge s_p = t_p & \text{[invariant]} \\ \implies ((s_r = \emptyset \wedge t_r = \emptyset) \iff s = t) & \text{[postcondition]} \end{array} \tag{5}$$

whose negation translated into $\{log\}$ is:

$$\begin{array}{l} (s_r = \emptyset \vee t_r = \emptyset \vee s_r = \{(x,y_1) \sqcup s_r^1\} \wedge t_r = \{(x,y_2) \sqcup t_r^1\} \wedge y_1 \neq y_2) \\ \wedge pfun(s) \wedge un(s_p,s_r,s) \wedge disj(s_p,s_r) \wedge pfun(t) \wedge un(t_p,t_r,t) \wedge disj(t_p,t_r) \wedge s_p = t_p \\ \wedge (s_r = \emptyset \wedge t_r = \emptyset \wedge s \neq t \vee s = t \wedge (s_r \neq \emptyset \vee t_r \neq \emptyset)) \end{array} \tag{6}$$

which again is found to be *false* by the tool.

5.2 Proving List's Implementation is Correct

In this section we show how $\{log\}$ can be used to prove the partial functional correctness of the implementation of the List ADT. Given that in this case the verification conditions are rather easy, the example shows how the expressiveness of the constraint language supported by $\{log\}$, in particular set unification, really helps to generate a simple, accurate and concise specification of the ADT.

Due to space restrictions, we focus the analysis on the add() subroutine (the rest can be found in Appendix A.2). Figure 4 shows the implementation of add() along with its specification and the assertions generated as the result of a Hoare-Floyd-like verification. List has been implemented as a singly-linked list. The nodes of this list are instances of an ADT called Node, whose interface is described in Figure 3 (and whose implementation and specification are given in Appendix A.2). List declares four internal (member) variables of type Node: s, holding the first node of the list; f, holding the last node of the list (to make add() to run in constant time); and c and p for the internal cursor.

At specification level, we introduce the following notions and notation.

Definition 5.1 *The* state of an instance of List *is a triple* $\langle s,a,s_m \rangle$ *where: s is a partial function representing the part of the memory holding the list (i.e. the* heap*); a is the function that maps program variables to the memory locations they are referencing (i.e. a* stack-allocated variable store*); and* s_m *is a set holding the memory locations of s whose nodes have been already iterated over. Moreover, the first node of s coincides with the node referenced by* s *and the last node with that referenced by* f.

Definition 5.2 *Let s be the first component of the state of an instance of* List. *Then, the* specification variables exported by List, s_p *and* s_r, *are defined as:* $s_p = s_m \lhd s$ *and* $s_r = s \setminus s_p$.

For s to represent a linked list we use the following representation.

setElem(T e)	e is the element stored in the node.
setNext(Node n)	n is the next node.
T getElem()	Returns the element stored in the node.
Node getNext()	Returns the next node.

Figure 3: Public interface of the Node ADT

```
1:  PRE-CONDITION true
2:  procedure add(T e)
3:      if s = null then                          ▷ at specification level condition writes: s = ∅
4:          s := new Node
5:          s.setElem(e)
6:          f := s
7:      else                                                          ▷ i.e. s ≠ ∅
8:          ASSERTION s = {a(f) ↦ (y,z) ⊔ s1}          ▷ set unification singles out last element
9:          Node n := new Node
10:         ASSERTION s' = {c ↦ (null,null), a(f) ↦ (y,z) ⊔ s1} ∧ c ∉ dom s ∧ c ≠ null
11:         n.setElem(e)
12:         ASSERTION s' = {c ↦ (null,e), a(f) ↦ (y,z) ⊔ s1}     ▷ n transient, not in a; c is used
13:         f.setNext(n)
14:         ASSERTION s' = {c ↦ (null,e), a(f) ↦ (c,z) ⊔ s1}
15:         f := n
16:         ASSERTION a' = a ⊕ {f ↦ c}
17:     end if
18: end procedure
19: POST-CONDITION s = ∅ ∧ s' = {c ↦ (null,e)} ∧ a' = a ⊕ {f ↦ c} ∧ c ≠ null
20:                ∨ s = {a(f) ↦ (y,z) ⊔ s1} ∧ c ∉ dom s ∧ c ≠ null
21:                  ∧ s' = {c ↦ (null,e), a(f) ↦ (c,z) ⊔ s1} ∧ a' = a ⊕ {f ↦ c}
```

Figure 4: add() appends e to the list

$$\{P\}\ \mathsf{v} := \textbf{new}\ \mathsf{T}\ \{P \wedge m' = \{c \mapsto \tau \sqcup m\} \wedge a' = a \oplus \{\mathsf{v} \mapsto c\} \wedge c \notin \mathrm{dom}\, m \wedge c \neq null\} \quad (7)$$

$$\{P\}\ \mathsf{v} := \mathsf{w}\ \{P \wedge m' = m \wedge a' = a \oplus \{\mathsf{v} \mapsto a(\mathsf{w})\}\}, \text{ if the type of v and w is an ADT} \quad (8)$$

$$\{P\}\ \textbf{end procedure}\ \{P \wedge m' = m \wedge a' = \{\mathsf{v}_1, \ldots, \mathsf{v}_n\} \lhd a\},\ \mathsf{v}_1, \ldots, \mathsf{v}_n \text{ all local variables} \quad (9)$$

$$\{P\}\ \textbf{end function}\ \{P \wedge m' = m \wedge a' = \{\mathsf{v}_1, \ldots, \mathsf{v}_n\} \lhd a\},\ \mathsf{v}_1, \ldots, \mathsf{v}_n \text{ all local variables} \quad (10)$$

Figure 5: Axioms for **new**, the assignment of ADT variables and **end procedure** and **end function**. P is a predicate; τ is the abstraction of the T ADT.

Definition 5.3 *If* $\{c_i\}_{i\in[1,n]} \cup \{null\}$ *are* $n+1$ *different constant symbols, then*

$$\{c_1 \mapsto (c_2, e_1), c_2 \mapsto (c_3, e_2), \ldots, c_n \mapsto (null, e_n)\} \tag{11}$$

represents the list $\langle e_1, e_2, \ldots, e_n\rangle$.

In this representation, c_i in $c_i \mapsto (c_{i+1}, e_i)$ is the memory location of the Node (c_{i+1}, e_i); c_{i+1} points to the next node of the list and e_i is the element stored in this node. Note that such a set is indeed a partial function as memory locations are unique. The main reason for choosing this representation is that it allows to specify all properties of List with formulas inside the decision procedures implemented in $\{log\}$.

Now we analyze with some detail the specification of add() and how it can be proved that the implementation verifies it. In particular we want to show how set unification and $\{log\}$'s constraint language provide a suitable vehicle for this task. Part of this analysis depends on the axioms given in Figure 5 which complement the standard rules of the Hoare-Floyd framework. We will consider only the **else** branch as the other one is trivial although we introduce some simplifications due to space reasons (see the full account in Appendix A.2). In Figure 4, if v is a variable, v' represents its value in the after state. When a specification variable is not mentioned in the assertion following a statement it is assumed to remain unchanged (i.e. $v' = v$).

Initially (line 8) we know the list is not empty. Then we know that the last node belongs to it. We are interested in the last node because it will be linked to the new node that is going to be appended. As the last node is referenced by variable f, we know that $a(\mathsf{f})$ is its location. Hence, we know that $a(\mathsf{f}) \mapsto (y, z)$ (for some variables y and z) belongs to s, which may have an unknown number of other nodes which is represented by a set variable s_1. So the initial assertion is $s = \{a(\mathsf{f}) \mapsto (y, z) \sqcup s_1\}$, which shows how set unification can be used to concisely express these facts. Now we apply a simplified version of the axiom for the **new** instruction (7). When applied to Node, this rule states that a new instance of Node is added to s, that the location of this instance (c) is not being used in s and is different from *null*. Again, we use set unification and $\{log\}$'s constraints to state all this in line 10. In effect, $c \mapsto (null, null)$ is added to s thus giving the value for s' and $c \notin \mathrm{dom}\, s$ states that c is new w.r.t. s.

Next the program sets e as the element to be stored by n. The effect of this statement is given by the post-condition of Node :: setElem() (shown in Appendix A.2) which states that the second component of the node in c must be e. Then, we use set unification once more to state this fact in line 12. Note that since n is a transient variable we do not store it in a in order to make the presentation shorter. Now we must apply Node :: setNext() specification (Appendix A.2) which states that the first component of the node pointed to by the Node variable on which the subroutine is called must be set to the address of parameter n. Again, set unification comes handy because we can change y by c as we have identified the node pointed to by f (line (14)).

Finally, f is pointed to the last node which is the one just appended to the list (line 15). Given that f and n are two variables whose type is an ADT, axiom (8) is applied. That is, a statement such as f := n changes the location to which f points to—note that is not that f points to n but that f points to what n is pointing to. Then, such a statement does not change s but just a. In this case we use the $\oplus$ operator instead of set unification just to show that relational operators can be used, too. The $\oplus$ operator updates the second component of an ordered pair in a partial function by looking up the first component. So after line 15, a' is equal to a except in f which now points to c.

As can be seen, set unification and set theory allow to give precise yet concise formulas capturing the semantics of several important instructions of imperative programming languages. Further, in the case of $\{log\}$ these formulas fall inside the decision procedures implemented by the tool thus making it possible to automatically discharge verification conditions.

5.3 Proving state invariants

In the loop invariant of the listEq() function (Figure 2) we include predicates such as $s \in \_ \nrightarrow \_$, $s_p \parallel s_r$, etc. Without these predicates, the correctness of listEq() cannot be proved. Hence, part of the verification effort includes proving that these predicates are indeed consequences of the specification of the List interface. $\{log\}$ can be used to automatically prove such properties, too.

In particular, the predicates included in the loop invariant mentioned above are *state invariants* of the specification of List's interface. Therefore, in order to be able to use them as valid properties we must prove that every subroutine specification preserves them.

Definition 5.4 *Let $\mathscr{S}_{\mathsf{m}}$ be the specification of subroutine* m. *Then the state predicate $\mathscr{I}$ is an* invariant of m *iff the following verification condition holds:*

$$\mathscr{I} \wedge \mathscr{S}_{\mathsf{m}} \implies \mathscr{I}' \tag{12}$$

where $\mathscr{I}'$ is the predicate resulting from substituting in $\mathscr{I}$ all the state variables by their primed counterparts. $\mathscr{I}$ is an invariant of the M ADT *iff $\mathscr{I}$ is an invariant for all subroutines in its interface.*

As an example, we use $\{log\}$ to prove that $s \in \_ \nrightarrow \_$ is an invariant of add(). In this case (12) is:

$$\begin{array}{l} s \in \_ \nrightarrow \_ \\ \wedge\, (s = \emptyset \wedge c \neq null \wedge s' = \{c \mapsto (null, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \\ \quad \vee\, s = \{a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s \wedge c \neq null \\ \quad\quad \wedge\, s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c,z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\}) \\ \implies s' \in \_ \nrightarrow \_ \end{array} \tag{13}$$

whose negation translates into $\{log\}$ as:

$$\begin{array}{l} pfun(s) \\ \wedge\, (s = \emptyset \wedge c \neq null \wedge s' = \{(c, (null, e))\} \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \\ \quad \vee\, apply(a, \mathsf{f}, m_2) \wedge s = \{(m_2, (y,z)) \sqcup s_1\} \wedge dom(s, m_1) \wedge c \notin m_1 \wedge c \neq null \\ \quad\quad \wedge\, apply(a, \mathsf{f}, m_2) \wedge s' = \{(c, (null, e)), (m_2, (c,z)) \sqcup s_1\} \\ \quad\quad \wedge\, oplus(a, \{(\mathsf{f}, c)\}, a')) \\ \wedge\, npfun(s') \end{array} \tag{14}$$

which again is found to be unsatisfiable by $\{log\}$.

The proofs of several different invariants for all the subroutines can be found in Appendix A.3.

5.4 Summary of the case study

We have formally verified the partial functional correctness of all the subroutines in List's interface plus six subroutines using this ADT. In verifying the subroutines in List's interface we only proved 22 specification invariants because verification conditions arising from the application of Hoare-Floyd rules are trivial. Instead when verifying the other six subroutines we only proved loop invariant preservation and implication of post-condition. Besides, we proved the three properties used in the loop invariant of Figure 2. Table 1 summarizes these results, where: VC is the number of verification conditions in each group; TIME is the time spent by $\{log\}$ in proving all the VC's; and AVG is the average time (both in milliseconds). $\{log\}$ 494-13 was used for all the proofs. The ASCII version of the VC's can be downloaded from [12].

This case study provides concrete evidence that $\{log\}$ is an appealing tool for program verification.

GROUP	VC	TIME	AVG
Loop invariant	6	1,639 ms	271 ms
Post-condition	6	37 ms	6 ms
Specification invariant	22	1,170 ms	53 ms
Other properties	3	6 ms	2 ms
TOTALS	37	2,843 ms	76 ms

Table 1: Summary of $\{log\}$'s performance

6 Conclusions

In this paper we have shown that $\{log\}$'s set constraint language permits to concisely and accurately specify properties of object-oriented programs. Then, we have shown that $\{log\}$ can automatically discharge the proofs obligations generated when a Hoare-Floyd framework is applied to verify the functional partial correctness of these programs. We have shown that it is possible to write high-level, set-based specifications (similar to the logic underlying Z and B) that later can be translated into the constraint language of $\{log\}$. As the case study shows, the specification methodology we have applied follows the modularization of the software design. In effect, the specification of the interface of an ADT is used to give the semantics and specifications of the components that use the ADT. The case study shows concrete evidence that $\{log\}$ can deal with classic verification problems.

However, there are many open problems. One of the most important is to integrate $\{log\}$ in a tool chain where many of the steps that in this paper are manual become more automatic. Other line of work is to evaluate $\{log\}$ on proving total correctness.

Acknowledgements Part of the work of M. Cristiá is supported by ANPCyT's grant PICT-2014-2200.

References

[1] J.-R. Abrial (1996): *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA.

[2] Alexander Aiken (1994): *Set Constraints: Results, Applications, and Future Directions*. In Alan Borning, editor: *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings*, *Lecture Notes in Computer Science* 874, Springer, pp. 326–335, doi:10.1007/3-540-58601-6_110. Available at `https://doi.org/10.1007/3-540-58601-6_110`.

[3] Kshitij Bansal, Andrew Reynolds, Clark W. Barrett & Cesare Tinelli (2016): *A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT*. In Nicola Olivetti & Ashish Tiwari, editors: *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, *Lecture Notes in Computer Science* 9706, Springer, pp. 82–98, doi:10.1007/978-3-319-40229-1_7. Available at `https://doi.org/10.1007/978-3-319-40229-1_7`.

[4] Josh Berdine, Cristiano Calcagno & Peter W. O'Hearn (2005): *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem P. de Roever, editors: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, *Lecture Notes in Computer Science* 4111, Springer, pp. 115–137, doi:10.1007/11804192_6. Available at `https://doi.org/10.1007/11804192_6`.

[5] Rudolf Berghammer, Peter Höfner & Insa Stucke (2014): *Automated Verification of Relational While-Programs*. In Peter Höfner, Peter Jipsen, Wolfram Kahl & Martin Eric Müller, editors: *Relational and Algebraic Methods in Computer Science - 14th International Conference, RAMiCS 2014, Marienstatt, Germany, April 28-May 1, 2014. Proceedings*, *Lecture Notes in Computer Science* 8428, Springer, pp. 173–190, doi:10.1007/978-3-319-06251-8_11. Available at `http://dx.doi.org/10.1007/978-3-319-06251-8_11`.

[6] Lennart Beringer, Adam Petcher, Katherine Q. Ye & Andrew W. Appel (2015): *Verified Correctness and Security of OpenSSL HMAC*. In Jaeyeon Jung & Thorsten Holz, editors: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, USENIX Association, pp. 207–221. Available at `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer`.

[7] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner & Wolfram Schulte, editors: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, *Lecture Notes in Computer Science* 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2. Available at `https://doi.org/10.1007/978-3-319-23534-9_2`.

[8] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick & Dulma Rodriguez (2015): *Moving Fast with Software Verification*. In Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, *Lecture Notes in Computer Science* 9058, Springer, pp. 3–11, doi:10.1007/978-3-319-17524-9_1. Available at `https://doi.org/10.1007/978-3-319-17524-9_1`.

[9] Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich & Daniel Ziegler (2017): *Certifying a file system using crash hoare logic: correctness in the presence of crashes*. *Commun. ACM* 60(4), pp. 75–84, doi:10.1145/3051092. Available at `http://doi.acm.org/10.1145/3051092`.

[10] Maximiliano Cristiá & Gianfranco Rossi (2016): *A Decision Procedure for Sets, Binary Relations and Partial Functions*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, *Lecture Notes in Computer Science* 9779, Springer, pp. 179–198, doi:10.1007/978-3-319-41528-4_10. Available at `http://dx.doi.org/10.1007/978-3-319-41528-4_10`.

[11] Maximiliano Cristiá & Gianfranco Rossi (2017): *A Decision Procedure for Restricted Intensional Sets*. In Leonardo de Moura, editor: *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, *Lecture Notes in Computer Science* 10395, Springer, pp. 185–201, doi:10.1007/978-3-319-63046-5_12. Available at `https://doi.org/10.1007/978-3-319-63046-5_12`.

[12] Maximiliano Cristiá & Gianfranco Rossi (2017): *Verification conditions of case study (HCVS 2017)*. Available at `https://www.dropbox.com/s/8860soq450n7lor/vcHCVS.zip?dl=0`.

[13] Maximiliano Cristiá, Gianfranco Rossi & Claudia S. Frydman (2013): *{log} as a Test Case Generator for the Test Template Framework*. In Robert M. Hierons, Mercedes G. Merayo & Mario Bravetti, editors: *SEFM*, *Lecture Notes in Computer Science* 8137, Springer, pp. 229–243. Available at `http://dx.doi.org/10.1007/978-3-642-40561-7_16`.

[14] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2017): *Program Verification using Constraint Handling Rules and Array Constraint Generalizations*. *Fundam. Inform.* 150(1), pp. 73–117, doi:10.3233/FI-2017-1461. Available at `https://doi.org/10.3233/FI-2017-1461`.

[15] Leonardo De Moura & Nikolaj Bjørner (2011): *Satisfiability Modulo Theories: Introduction and Applications*. *Commun. ACM* 54(9), pp. 69–77, doi:10.1145/1995376.1995394. Available at `http://doi.acm.org/10.1145/1995376.1995394`.

[16] Agostino Dovier, Carla Piazza, Enrico Pontelli & Gianfranco Rossi (2000): *Sets and constraint logic programming*. *ACM Trans. Program. Lang. Syst.* 22(5), pp. 861–931. Available at http://doi.acm.org/10.1145/365151.365169.

[17] Agostino Dovier, Carla Piazza & Gianfranco Rossi (1999): *Relating Set Constraints and Constraint Logic Programming with Finite Sets*. Technical Report 196, Dipartimento di Matematica, Universit di Parma.

[18] Agostino Dovier, Enrico Pontelli & Gianfranco Rossi (2006): *Set Unification*. *Theory Pract. Log. Program.* 6(6), pp. 645–701, doi:10.1017/S1471068406002730. Available at http://dx.doi.org/10.1017/S1471068406002730.

[19] Claire Dross & Yannick Moy (2016): *Abstract Software Specifications and Automatic Proof of Refinement*. In Thierry Lecomte, Ralf Pinger & Alexander Romanovsky, editors: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*, *Lecture Notes in Computer Science* 9707, Springer, pp. 215–230, doi:10.1007/978-3-319-33951-1_16. Available at https://doi.org/10.1007/978-3-319-33951-1_16.

[20] Nevin Heintze & Joxan Jaffar (1990): *A Decision Procedure for a Class of Set Constraints (Extended Abstract)*. In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, IEEE Computer Society, pp. 42–51, doi:10.1109/LICS.1990.113732. Available at https://doi.org/10.1109/LICS.1990.113732.

[21] Shuvendu K. Lahiri & Shaz Qadeer (2008): *Back to the future: revisiting precise program verification using SMT solvers*. In George C. Necula & Philip Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 171–182, doi:10.1145/1328438.1328461. Available at http://doi.acm.org/10.1145/1328438.1328461.

[22] Leszek Pacholski, Wieslaw Szwast & Lidia Tendera (1997): *Complexity of Two-Variable Logic with Counting*. In: *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, IEEE Computer Society, pp. 318–327, doi:10.1109/LICS.1997.614958. Available at https://doi.org/10.1109/LICS.1997.614958.

[23] John C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, IEEE Computer Society, pp. 55–74, doi:10.1109/LICS.2002.1029817. Available at https://doi.org/10.1109/LICS.2002.1029817.

[24] Gianfranco Rossi (2008): $\{log\}$. Available at http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html.

[25] Gianfranco Rossi (2011): *From set unification to set constraints*. *Intelligenza Artificiale* 5(1), pp. 157–161, doi:10.3233/IA-2011-0020. Available at https://doi.org/10.3233/IA-2011-0020.

[26] Mark Saaltink (1997): *The Z/EVES Mathematical Toolkit Version 2.2 for Z/EVES Version 1.5*. Technical Report, ORA Canada.

[27] J. M. Spivey (1992): *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[28] Saurabh Srivastava, Sumit Gulwani & Jeffrey S. Foster (2009): *VS3: SMT Solvers for Program Verification*. In Ahmed Bouajjani & Oded Maler, editors: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, *Lecture Notes in Computer Science* 5643, Springer, pp. 702–708, doi:10.1007/978-3-642-02658-4_58. Available at https://doi.org/10.1007/978-3-642-02658-4_58.

[29] Gijs Vanspauwen & Bart Jacobs (2015): *Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications*. In Radu Calinescu & Bernhard Rumpe, editors: *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Pro-*

ceedings, *Lecture Notes in Computer Science* 9276, Springer, pp. 53–68, doi:10.1007/978-3-319-22969-0_4. Available at https://doi.org/10.1007/978-3-319-22969-0_4.

[30] Karen Zee, Viktor Kuncak & Martin C. Rinard (2008): *Full functional verification of linked data structures*. In Rajiv Gupta & Saman P. Amarasinghe, editors: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, ACM, pp. 349–361, doi:10.1145/1375581.1375624. Available at http://doi.acm.org/10.1145/1375581.1375624.

A Case Study: Complete Version

This appendix contains all the code, specifications and verification conditions of the case study based on the List ADT and six subroutines processing instances of List.

A.1 Programs that use List

For all the following functions and procedures we give the verification conditions that guarantee that the loop invariant is preserved and that upon termination of the loop the post-condition holds. These verification conditions are given both in our specification language and in $\{log\}$.

A.1.1 List membership

listIn(s,x) returns true if and only if x is an element of s.

The specification of this example uses the relational operator that takes the range of a binary relation. We use ran for the specification language and $ran(R,A)$ in $\{log\}$, which corresponds to $\text{ran}\,R = A$.

PRE-CONDITION $true$
function Bool listIn(List s, T x)
  Bool found := false
  s.fst()
  INVARIANT $s = s_p \cup s_r \wedge (found = true \iff x \in \text{ran}\, s_p)$
  while s.more() $\wedge \neg$found **do**
    if x := s.next() **then**
      found := true
    end if
  end while
  return found
end function
POST-CONDITION $found = true \iff x \in \text{ran}\, s$

Verification conditions in specification language

Loop invariant

$$\begin{array}{l} s_r = \{(z,y) \sqcup s_r^1\} \wedge found = false \\ \wedge\, s = s_p \cup s_r \wedge (found = true \iff x \in \text{ran}\, s_p) \\ \wedge\, (x \neq y \wedge found' = found \vee x = y \wedge found' = true) \qquad \text{[\textbf{if} post-condition]} \\ \implies s = \{(z,y) \sqcup s_p\} \cup s_r^1 \wedge (found' = true \iff x \in \text{ran}\{(z,y) \sqcup s_p\}) \end{array} \tag{15}$$

Post-condition

$$\begin{array}{l} (s_r = \emptyset \vee found = true) \\ \wedge\, s = s_p \cup s_r \wedge (found = true \iff x \in \text{ran}\, s_p) \\ \implies (found = true \iff x \in \text{ran}\, s) \end{array} \tag{16}$$

Verification conditions in $\{log\}$

Loop invariant

$$\begin{array}{l} s_r = \{(z,y) \sqcup s_r^1\} \wedge found = false \\ \wedge\, un(s_p, s_r, s) \wedge ran(s_p, m_1) \wedge (found = false \vee x \in m_1) \wedge (found = true \vee x \notin m_1) \\ \wedge\, (x \neq y \wedge found' = found \vee x = y \wedge found = true) \\ \wedge\, ran(\{(z,y) \sqcup s_p\}, m_2) \wedge (nun(\{(z,y) \sqcup s_p\}, s_r^1, s) \vee found' \wedge x \notin m_2 \vee found = false \wedge x \in m_2) \end{array} \tag{17}$$

Post-condition

$$\begin{array}{l} (s_r = \emptyset \vee found = true) \\ \wedge\, un(s_p, s_r, s) \wedge ran(s_p, m_1) \wedge (found = false \vee x \in m_1) \wedge (found = true \vee x \notin m_1) \\ \wedge\, ran(s, m_2) \wedge (found = true \wedge x \notin m_2 \vee found = false \wedge x \in m_2) \end{array} \tag{18}$$

A.1.2 Map over a list

listMap(s, o) applies o.f to every element of list s returning the result in a new list t.

The specification of this function makes use of the *restricted intensional sets* (RIS) provided by $\{log\}$. RIS are the core of [11] which is going to be presented at CADE 2017. A RIS term is noted $\{e[\vec{x}] : D \mid \Psi \bullet \tau[\vec{x}]\}$, where e (*control term*) is a $\{log\}$ term and $\vec{x} \triangleq \langle x_1, \ldots, x_n \rangle$, $n > 0$, are all the variables occurring in it; D (*domain*) is a set term; Ψ (*filter*) is a $\{log\}$ formula; and τ (*pattern*) is a $\{log\}$ term containing $\vec{x}$. When the filter is *true* or the pattern is the control term itself they can be omitted but at least one must be present. The form of RIS terms is borrowed from the form of set comprehension expressions available in Z.

A RIS term $\{e[\vec{x}] : D \mid \Psi[\vec{x}, \vec{v}] \bullet \tau[\vec{x}, \vec{v}]\}$, where $\vec{v}$ is a vector of *free* variables, is interpreted as the set $\{y : \exists \vec{x}(e[\vec{x}] \in D \wedge \Psi[\vec{x}, \vec{v}] \wedge y = \tau[\vec{x}, \vec{v}])\}$. Note that $x_1, \ldots, x_n$ are bound variables whose scope is the RIS itself.

If control terms and patterns bear some specific relations and if filters verify some conditions on the variables they include, $\{log\}$ provides a complete solver for formulas including this fragment of RIS terms (see [11] for all the technical details). So far, RIS terms can participate in equality and membership constraints and their negations.

This example is included in [11].

It would have been possible to specify a program where the mapping is done over the same input list. However in order to discharge some verification conditions $\{log\}$ would need to implement set union for RIS terms, which is not implemented yet.

Note that in the specification and loop invariant we simply say $f(y)$ instead of $o.f(y)$ given that the important thing here is that f is applied.

```
PRE-CONDITION true
function List listMap(List s, Object o)
    List t := new List
    s.fst()
    INVARIANT s = s_p ∪ s_r ∧ t = {(x,y) : s_p • (x, f(y))}
    while s.more() do
        t.add(o.f(s.next()))
    end while
end function
POST-CONDITION t = {(x,y) : s • (x, f(y))}
```

Verification conditions in specification language

Loop invariant

$$\begin{array}{l} s_r = \{(x,y) \sqcup s_r^1\} \\ \wedge\, s = s_p \cup s_r \wedge t = \{(v,w) \in s_p \bullet (v, f(w))\} \\ \quad \Longrightarrow\ s = \{(x,y) \sqcup s_p\} \cup s_r^1 \wedge \{(x, f(y)) \sqcup t\} = \{(v,w) \in \{(x,y) \sqcup s_p\} \bullet (v, f(w))\}) \end{array} \tag{19}$$

Post-condition

$$\begin{array}{l} s_r = \emptyset \\ \wedge\, s = s_p \cup s_r \wedge t = \{(x,y) : s_p \bullet (x, f(y))\} \\ \quad \Longrightarrow\ t = \{(x,y) : s \bullet (x, f(y))\} \end{array} \tag{20}$$

Verification conditions in $\{log\}$

Loop invariant

$$\begin{array}{l} s_r = \{(x,y) \sqcup s_r^1\} \\ \wedge\, un(s_p, s_r, s) \wedge t = \{(v,w) \in s_p \bullet (v, f(w))\} \\ \wedge\, (nun(\{(x,y) \sqcup s_p\}, s_r^1, s) \vee \{(x, f(y)) \sqcup t\} \neq \{(v,w) \in \{(x,y) \sqcup s_p\} \bullet (v, f(w))\}) \end{array} \tag{21}$$

Post-condition

$$\begin{array}{l} s_r = \emptyset \\ \wedge\, un(s_p, s_r, s) \wedge t = \{(x,y) : s_p \bullet (x, f(y))\} \\ \wedge\, t \neq \{(x,y) : s \bullet (x, f(y))\} \end{array} \tag{22}$$

A.1.3 List to set

listToSet(s) stores all the elements of s in set c (so possible repeated elements in s are stored only once in c), and c is returned.

For this example we assume the existence of an ADT called Set providing the concept of finite set (i.e. no repetitions, no order). As List, Set is parameterized by type T. Set provides an abstract iterator as List does; its interface includes (at least) the add() subroutine to add elements to the set. Note that is Set :: add() who eliminates repetitions (i.e. is not the responsibility of listToSet() to check for them).

In order to discharge the verification conditions of this example we need the negation of the *ran* constraint, called *nran*.

PRE-CONDITION $true$
function Set listToSet(List s)
  Set c := **new** Set
  s.fst()
  INVARIANT $s = s_p \cup s_r \wedge c = \text{ran}\, s_p$
  while s.more() **do**
    c.add(s.next())
  end while
  return c
end function
POST-CONDITION $c = \text{ran}\, s$

Verification conditions in specification language

Loop invariant

$$\begin{array}{l} s_r = \{(x,y) \sqcup s_r^1\} \\ \wedge\, s = s_p \cup s_r \wedge c = \text{ran}\, s_p \\ \implies s = \{(x,y) \sqcup s_p\} \cup s_r^1 \wedge \{y \sqcup c\} = \text{ran}(\{(x,y) \sqcup s_p\}) \end{array} \tag{23}$$

Post-condition

$$\begin{array}{l} s_r = \emptyset \\ \wedge\, s = s_p \cup s_r \wedge c = \text{ran}\, s_p \\ \implies c = \text{ran}\, s \end{array} \tag{24}$$

Verification conditions in $\{log\}$

Loop invariant

$$\begin{array}{l} s_r = \{(x,y) \sqcup s_r^1\} \\ \wedge\, un(s_p, s_r, s) \wedge ran(s_p, c) \\ \wedge\, (nun(\{(x,y) \sqcup s_p\}, s_r^1, s) \vee nran(\{(x,y) \sqcup s_p\}, \{y \sqcup c\})) \end{array} \tag{25}$$

Post-condition

$$\begin{array}{l} s_r = \emptyset \\ \wedge\, un(s_p, s_r, s) \wedge ran(s_p, c) \\ \wedge\, nran(s, c) \end{array} \tag{26}$$

A.1.4 List filtering

listFilter(s, elem) filters all the elements of s that belong to set elem and returns the resulting list in t.

The specification of this example uses the relational operator know as range restriction, noted $\rhd$. If A is a set and R is a binary relation, then:

$$R \rhd A = \{(x,y) : R \mid y \in A\}$$

In $\{log\}$ it is noted $rres(R,A,S)$ whose interpretation is $S = R \rhd A$. In turn, its negation is noted $nrres(R,A,S)$.

PRE-CONDITION $true$
function List listFilter(List s, Set elem)
    T x
    List t := **new** List
    s.fst()
    INVARIANT $s = s_r \cup s_p \wedge t = s_p \rhd elem$
    while s.more() **do**
        x := s.next()
        if elem.in(x) **then**
            t.add(x)
        end if
    end while
    return t
end function
POST-CONDITION $t = s \rhd elem$

Verification conditions in specification language

Loop invariant

$$\begin{array}{l} s_r = \{(x,y) \sqcup s_r^1\} \\ \wedge\, s = s_p \cup s_r \wedge t = s_p \rhd elem \\ \wedge\, (x \in elem \wedge t' = \{(x,y) \sqcup t\} \vee x \notin elem \wedge t' = t) \qquad \text{[\textbf{if} post-condition]} \\ \quad \Longrightarrow\ s = \{(x,y) \sqcup s_p\} \cup s_r^1 \wedge t' = \{(x,y) \sqcup s_p\} \rhd elem \end{array} \tag{27}$$

Post-condition

$$\begin{array}{l} s_r = \emptyset \\ \wedge\, s = s_p \cup s_r \wedge t = s_p \rhd elem \\ \quad \Longrightarrow\ t = s \rhd elem \end{array} \tag{28}$$

Verification conditions in $\{log\}$

Loop invariant

$$\begin{array}{l} s_r = \{(x,y) \sqcup s_r^1\} \\ \wedge\, un(s_p, s_r, s) \wedge rres(s_p, elem, t) \\ \wedge\, (x \in elem \wedge t' = \{(x,y) \sqcup t\} \vee x \notin elem \wedge t' = t) \\ \wedge\, (un(\{(x,y) \sqcup s_p\}, s_r^1, s) \vee nrres(\{(x,y) \sqcup s_p\}, elem, t')) \end{array} \tag{29}$$

Post-condition

$$\begin{aligned} & s_r = \emptyset \\ & \wedge un(s_p, s_r, s) \wedge rres(s_p, elem, t) \\ & \wedge nrres(s, elem, t) \end{aligned} \tag{30}$$

A.1.5 Length of a list

listLen(s) returns the length of s (which is a value of type Nat).

For this example we assume s.next() is a legal statement of the language (i.e. although next() is a function we do not assign its return value).

The specification of this problem requires the introduction of the cardinality operator symbolized $|\cdot|$ in our specification language and $size(A,n)$ in $\{log\}$, with the intuitive meaning.

PRE-CONDITION *true*
function Nat listLen(List s)
  Nat n := 0
  s.fst()
  INVARIANT $s = s_p \cup s_r \wedge s_p \parallel s_r \wedge n = |s_p|$
  while s.more() **do**
    s.next()
    n := n + 1
  end while
  return n
end function
POST-CONDITION $n = |s|$

Verification conditions in specification language

Loop invariant

$$\begin{aligned} & s_r = \{e \sqcup s_r^1\} \\ & \wedge s = s_p \cup s_r \wedge s_p \parallel s_r \wedge n = |s_p| \\ & \implies s = \{e \sqcup s_p\} \cup s_r^1 \wedge n = |\{e \sqcup s_p\}| \end{aligned} \tag{31}$$

Post-condition

$$\begin{aligned} & s_r = \emptyset \\ & \wedge s = s_p \cup s_r \wedge n = |s_p| \\ & \implies n = |s| \end{aligned} \tag{32}$$

Verification conditions in $\{log\}$ Discharging the verification conditions generated in this example shows some limitations of $\{log\}$ when it comes to solve formulas including integer constraints. $\{log\}$ uses CLP(FD) as the integer arithmetics constraint solver, so it can solve the same formulas and with the same limitations of CLP(FD). In particular, for some formulas, $\{log\}$ needs that all integer variables be bound to a finite domain. If the user does not establish these bindings and $\{log\}$ really needs them, it

will let the user know by printing a suitable message. The bindings are set by means of constraints of the form $n \in [c_1, c_2]$, where c_1 and c_2 are two integer constants. Hence, if $\{log\}$ does no find a solution for such a formula it does not mean the formula is unsatisfiable (because it might have a solution in a bigger domain). On the other hand, specifying very large domains may cause unacceptable execution times because the CLP(FD) solver may need to explore all values of the domain to detect the unsatisfiability. In this example, only the first goal needs the cardinality of s_p to be bound to a finite domain

Loop invariant

$$\begin{array}{l} n \in [0,200] \\ \wedge\, s_r = \{e \sqcup s_r^1\} \\ \wedge\, un(s_p, s_r, s) \wedge disj(s_p, s_r) \wedge size(s_p, n) \\ \wedge\, (nun(\{e \sqcup s_p\}, s_r^1, s) \vee k_1 \; is \; n+1 \wedge size(\{e \sqcup s_p\}, k) \wedge k \neq k_1) \end{array} \tag{33}$$

Post-condition

$$\begin{array}{l} s_r = \emptyset \\ \wedge\, un(s_p, s_r, s) \wedge size(s_p, n) \\ \wedge\, size(s, k) \wedge k \neq n \end{array} \tag{34}$$

A.2 List's Implementation and Verification

In this section we show how $\{log\}$ is used to prove the partial correctness of the subroutines in the interface of List. Given that all verification conditions are trivial, except for those shown in Section 5.2, we only provide the assertions after each statement. Note that none of these subroutines uses **while** or **if** statements (except add() which is shown in Section 5.2).

A.2.1 Node specification

The implementation of List makes use of the Node ADT. For the sake of completeness we give its definition below. Besides, we give the specification of each subroutine in its interface. We would like to comment on these specifications briefly.

Any instance of Node lives in memory and (at least initially) it is referenced by some variable. Then, when a Node subroutine is called it is so by telling which variable is referencing the instance. Since at the ADT level we do not have the name of the variable used to make the call we will assume its name is v. Now, the specification of Node is based on a partial function, called a, which represents the function mapping program variables to the instances they are referencing (i.e. a is a *stack-allocated variable store*)[1]; and a partial function, called m, representing the memory used by the program (i.e. the *heap* or *dynamically-allocated memory*). In turn, an instance of Node is represented as an ordered pair (n,e), where n corresponds to next and e to elem. So, if an instance of Node is in memory location x, then $m(x) = (n,e)$; if x is referenced by v, then $a(\mathsf{v}) = x$; and then, $m(a(\mathsf{v})) = (n,e)$.

Memory locations belong to $M \cup \{null\}$, where M is some underspecified finite set. Actually we do not need to use M in the specifications.

[1] Recall that variables whose type is an ADT are always references to their instances.

When a state variable is not mentioned in the post-condition it is assumed to remain unchanged (i.e. $v' = v$).

---

adt Node(T)
  private
    T elem
    Node next
  end private
  public
    PRE-CONDITION *true*
    procedure setElem(T e)
      elem := e
    end procedure
    POST-CONDITION $m' = m \oplus \{a(\mathsf{v}) \mapsto (y, e)\} \wedge m(a(\mathsf{v})) = (y, \_)$

    PRE-CONDITION $a(\mathsf{n}) = x$ ▷ x can be *null*
    procedure setNext(Node n)
      next := n
    end procedure
    POST-CONDITION $m' = m \oplus \{a(\mathsf{v}) \mapsto (x, z)\} \wedge m(a(\mathsf{v})) = (\_, z)$

    PRE-CONDITION *true*
    function T getElem()
      return elem
    end function
    POST-CONDITION $ret = z \wedge m(a(\mathsf{v})) = (\_, z)$

    PRE-CONDITION *true*
    function Node getNext()
      return next
    end function
    POST-CONDITION $ret = y \wedge m(a(\mathsf{v})) = (y, \_)$
  end public
end adt

A.2.2 List specification

Now we give the skeletal definition of the List ADT. As can be seen, it declares four member variables all of type Node: s, holding the first node of the list; f, holding the last node of the list[2]; c, holding the next element to iterate over; and p, holding the last element returned by next().

[2] f is maintained to make add() to run in constant time.

```
adt List(T)
    private
        Node s, f, c, p
        ASSERTION a' = {s ↦ null, f ↦ null, c ↦ null, p ↦ null}
    end private
    public
        procedure List()                ▷ ADT's constructor; called when new instance is created
            ...
        end procedure
        ......
                the rest of the interface is given below
        ......
    end public
end adt
```

Below the reader can find the implementation and specification of each subroutine of List. For each subroutine we also give all the assertions valid after each statement. In general these assertions are obvious because there are no loops in these subroutines. However, in Section A.3 we show a number of verification conditions that should be proved of the specifications and how $\{log\}$ discharges them automatically.

At specification level, the state of an instance of List is given by three sets: s, the partial function representing the part of the memory holding the list; a, the function mapping variables to the memory locations they are referencing; and s_m, holding the memory locations of s whose nodes have been already iterated over. Then, s_p and s_r (i.e. the specification variables exported by List), are defined as follows: $s_p \triangleq s_m \lhd s$ and $s_r \triangleq s \setminus s_p$. The first node of s coincides with the node referenced by s and the last node with that referenced by f.

When a state variable is not mentioned in the assertion following a statement it is assumed to remain unchanged (i.e. $v' = v$); the same for the post-condition of subroutines.

add() appends e to the list The first subroutine is add(). We have introduced an error in one of the assertions and in the post-condition to show how it becomes evident when some properties are proved (see Section A.3.5).

Note that all the assertions are directly derived from Node's specification and from the axioms of Figure 6. For instance, in the **then** branch, when s.setElem(e) is executed, the second component of the Node instance referenced by s is changed by e. That is, in this case variable v used in the specification of Node becomes s because the call is made on it.

On the other hand, two assertions deserve some attention. Firstly, the semantics of **new** is that it stores a new instance of an ADT (Node in this case) in the memory area of interest (s in this case). When it does so, it stores the instance in an unused location. The formal semantic rule is given in axiom (35) in Figure 6. In the **then** branch a simplified version of (35) is applied because s is empty at beginning. Besides, we avoid conjoining $c \neq null$ because this is the error we want to later spot when some properties fail to be proven. However, in the **else** branch we conjoin $c \notin \mathrm{dom}\, s$ although we do not update a because n is a local transient variable which will be destroyed right after the **end procedure** instruction, so we keep the presentation shorter. Nonetheless, note that axiom (37) can be applied right after **end procedure**, yielding the same net result.

$$\{P\}\ \mathsf{v} := \textbf{new}\ \mathsf{T}\ \{P \wedge m' = \{c \mapsto \tau \sqcup m\} \wedge a' = a \oplus \{\mathsf{v} \mapsto c\} \wedge c \notin \mathrm{dom}\, m \wedge c \neq null\} \quad (35)$$

$$\{P\}\ \mathsf{v} := \mathsf{w}\ \{P \wedge m' = m \wedge a' = a \oplus \{\mathsf{v} \mapsto a(\mathsf{w})\}\}\text{, if the type of v and w is an ADT} \quad (36)$$

$$\{P\}\ \textbf{end procedure}\ \{P \wedge m' = m \wedge a' = \{\mathsf{v}_1, \ldots, \mathsf{v}_n\} \lhd a\},\ \mathsf{v}_1, \ldots, \mathsf{v}_n \text{ all local variables} \quad (37)$$

$$\{P\}\ \textbf{end function}\ \{P \wedge m' = m \wedge a' = \{\mathsf{v}_1, \ldots, \mathsf{v}_n\} \lhd a\},\ \mathsf{v}_1, \ldots, \mathsf{v}_n \text{ all local variables} \quad (38)$$

Figure 6: Axioms for **new**, the assignment of ADT variables and **end procedure**. P is a predicate; τ is the abstraction of the T ADT.

Secondly, statements of the form v := w (where both are variables whose type is an ADT) modify a but not s. In effect, the semantics of such a statement is that v now references the same memory location referenced by w. The formal axiom is given in equation (36). This can be seen, in the **then** branch, after f := s (note that is not that f points to s but that f points to what s is pointing to).

PRE-CONDITION $true$
procedure add(T e)
  if s = null **then** ▷ at specification level condition writes: $s = \emptyset$
    ASSERTION $s = \emptyset$
    s := **new** Node
    ASSERTION $s' = \{c \mapsto (null, null)\} \wedge a' = a \oplus \{\mathsf{s} \mapsto c\}$
    s.setElem(e)
    ASSERTION $s' = \{a(\mathsf{s}) \mapsto (null, e)\} \wedge s(a(\mathsf{s})) = (null, \_)$
    f := s
    ASSERTION $a' = a \oplus \{\mathsf{f} \mapsto a(\mathsf{s})\}$
  else ▷ i.e. $s \neq \emptyset$
    ASSERTION $s = \{a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\}$ ▷ set unification singles out last element
    Node n := **new** Node
    ASSERTION $s' = \{c \mapsto (null, null), a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s$ ▷ c unused location
    n.setElem(e)
    ASSERTION $s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\}$ ▷ n transient, not in a; c is used
    f.setNext(n)
    ASSERTION $s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c,z) \sqcup s_1\}$
    f := n
    ASSERTION $a' = a \oplus \{\mathsf{f} \mapsto c\}$
  end if
end procedure
POST-CONDITION $s = \emptyset \wedge s' = \{c \mapsto (null, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\}$
$\vee\ s = \{a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s$
$\wedge\ s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c,z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\}$

The remaining subroutines are simple, so we will not comment on them.

fst() **initializes the iterator**

PRE-CONDITION $true$
procedure fst()
    c := s
    ASSERTION $a' = a \oplus \{\mathsf{c} \mapsto a(\mathsf{s})\} \wedge s'_m = \emptyset$
end procedure
POST-CONDITION $a' = a \oplus \{\mathsf{c} \mapsto a(\mathsf{s})\} \wedge s'_m = \emptyset$

---

next() **returns the current element and moves the cursor one position ahead**

PRE-CONDITION $a(\mathsf{c}) \neq null$
function T next()
    p := c
    ASSERTION $a' = a \oplus \{\mathsf{p} \mapsto a(\mathsf{c})\}$
    c := c.getNext()
    ASSERTION $a' = a \oplus \{\mathsf{c} \mapsto n\} \wedge s(a(\mathsf{c})) = (n, \_) \wedge s'_m = \{a(\mathsf{p}) \sqcup s_m\} \wedge s(a(\mathsf{p})) = (y, z)$
    return p.getElem()
    ASSERTION $ret = z \wedge s(a(\mathsf{p})) = (\_, z)$
end function
POST-CONDITION $ret = z \wedge s(a(\mathsf{c})) = (y, z) \wedge a' = a \oplus \{\mathsf{p} \mapsto a(\mathsf{c}), \mathsf{c} \mapsto y\}$
$\wedge\, s'_m = \{a(\mathsf{c}) \sqcup s_m\}$

---

more() **returns** true **if and only if there are more element in the list**

PRE-CONDITION $true$
procedure more()
    return c $\neq$ null
end procedure
POST-CONDITION $ret \iff a(\mathsf{c}) \neq null$

---

rpl() **replaces the last element returned by** next() **with** e

---

PRE-CONDITION $a(\mathsf{p}) \neq null$
procedure rpl(T e)
    p.setElem(e)
    ASSERTION $s' = s \oplus \{a(\mathsf{p}) \mapsto (y, e)\} \wedge s(a(\mathsf{p})) = (y, \_)$
end procedure
POST-CONDITION $s' = s \oplus \{a(\mathsf{p}) \mapsto (y, e)\} \wedge s(a(\mathsf{p})) = (y, \_)$

---

del() **empties the list**

PRE-CONDITION $true$
procedure del()
  free(s)
  ASSERTION $s' = s'_m = \emptyset \wedge a' = a \oplus \{\mathsf{s} \mapsto null\}$
  $\mathsf{f} := \mathsf{c} := \mathsf{p} := \mathsf{null}$
  ASSERTION $a' = a \oplus \{\mathsf{f} \mapsto null, \mathsf{c} \mapsto null, \mathsf{p} \mapsto null\}$
end procedure
POST-CONDITION $\mathrm{ran}\, a' = \{null\} \wedge s' = s'_m = \emptyset$

A.3 Proving Properties of List's Specification

In this section we use the same technique shown in Section 5.3 to prove more properties of the List specification. Some of these properties are necessary to establish some loop invariants and in general is an standard verification activity to gain confidence in the correctness of the specification.

For each subroutine we show the verification condition in our specification language followed by the negation written in $\{log\}$.

A.3.1 Invariant: s is a partial function

- add()

$$\begin{array}{l} s \in \_ \nrightarrow \_ \\ \wedge\, (s = \emptyset \wedge s' = \{c \mapsto (null, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m \\ \quad \vee\, s = \{a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s \\ \quad\quad \wedge\, s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c,z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m) \\ \implies s' \in \_ \nrightarrow \_ \end{array} \tag{39}$$

$$\begin{array}{l} pfun(s) \\ \wedge\, (s = \emptyset \wedge s' = \{(c, (null, e))\} \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m \\ \quad \vee\, apply(a, \mathsf{f}, m_2) \wedge s = \{(m_2, (y,z)) \sqcup s_1\} \wedge dom(s, m_1) \wedge c \notin m_1 \\ \quad\quad \wedge\, apply(a, \mathsf{f}, m_2) \wedge s' = \{(c, (null, e)), (m_2, (c,z)) \sqcup s_1\} \\ \quad\quad \wedge\, oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m) \\ \wedge\, npfun(s') \end{array} \tag{40}$$

- fst()

$$\begin{array}{l} s \in \_ \nrightarrow \_ \\ \wedge\, a' = a \oplus \{\mathsf{c} \mapsto a(\mathsf{s})\} \wedge s'_m = \emptyset \wedge s' = s \\ \implies s' \in \_ \nrightarrow \_ \end{array} \tag{41}$$

$$\begin{array}{l} pfun(s) \\ \wedge\, apply(a, \mathsf{s}, m_1) \wedge oplus(a, \{(\mathsf{c}, m_1)\}, a') \wedge s'_m = \emptyset \wedge s' = s \\ \wedge\, npfun(s') \end{array} \tag{42}$$

- next()

$$
\begin{array}{l}
s \in \_ \nrightarrow \_ \\
\land\, a(\mathsf{c}) \neq null \land ret = z \land s(a(\mathsf{c})) = (y,z) \land a' = a \oplus \{\mathsf{p} \mapsto a(\mathsf{c}), \mathsf{c} \mapsto y\} \\
\land\, s'_m = \{a(\mathsf{c}) \sqcup s_m\} \\
\quad \Longrightarrow\ s' \in \_ \nrightarrow \_
\end{array}
\tag{43}
$$

$$
\begin{array}{l}
pfun(s) \\
\land\, ret = z \land apply(a, \mathsf{c}, m_1) \land m_1 \neq null \land apply(s, m_1, (y,z)) \land oplus(a, \{(\mathsf{p}, m_1), (\mathsf{c}, y)\}, a') \\
\land\, s'_m = \{m_1 \sqcup s_p\} \land s' = s \\
\land\, npfun(s')
\end{array}
\tag{44}
$$

- rpl()

$$
\begin{array}{l}
s \in \_ \nrightarrow \_ \\
\land\, a(\mathsf{p}) \neq null \land s' = s \oplus \{a(\mathsf{p}) \mapsto (y,e)\} \land s(a(\mathsf{p})) = (y,\_) \\
\quad \Longrightarrow\ s' \in \_ \nrightarrow \_
\end{array}
\tag{45}
$$

$$
\begin{array}{l}
pfun(s) \\
\land\, apply(a, \mathsf{p}, m_1) \land m_1 \neq null \land apply(s, m_1, (y,\_)) \\
\land\, oplus(s, \{m_1 \mapsto (y,e)\}, s') \\
\land\, npfun(s')
\end{array}
\tag{46}
$$

- del()

$$
\begin{array}{l}
s \in \_ \nrightarrow \_ \\
\land\, \mathrm{ran}\, a' = \{null\} \land s' = \emptyset \land s'_m = \emptyset \\
\quad \Longrightarrow\ s' \in \_ \nrightarrow \_
\end{array}
\tag{47}
$$

$$
\begin{array}{l}
pfun(s) \\
\land\, ran(a', \{null\}) \land s' = \emptyset \land s'_m = \emptyset \\
\land\, npfun(s')
\end{array}
\tag{48}
$$

A.3.2 Invariant $s_m \subseteq \mathrm{dom}\, s$

- add()

$$
\begin{array}{l}
s_m \subseteq \mathrm{dom}\, s \\
\land\, (s = \emptyset \land s' = \{c \mapsto (null, e)\} \land a' = a \oplus \{\mathsf{f} \mapsto c\} \land s'_m = s_m \\
\quad \lor\, s = \{a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\} \land c \notin \mathrm{dom}\, s \\
\qquad \land\, s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c,z) \sqcup s_1\} \land a' = a \oplus \{\mathsf{f} \mapsto c\} \land s'_m = s_m) \\
\quad \Longrightarrow\ s'_m \subseteq \mathrm{dom}\, s'
\end{array}
\tag{49}
$$

$$
\begin{array}{l}
dom(s,m_7) \wedge subset(s_m,m_7) \\
\wedge\, (s=\emptyset \wedge s' = \{(c,(null,e))\} \wedge oplus(a,\{(\mathsf{f},c)\},a') \wedge s'_m = s_m \\
\quad \vee\, apply(a,\mathsf{f},m_2) \wedge s = \{(m_2,(y,z)) \sqcup s_1\} \wedge dom(s,m_1) \wedge c \notin m_1 \\
\qquad \wedge\, apply(a,\mathsf{f},m_2) \wedge s' = \{(c,(null,e)),(m_2,(c,z)) \sqcup s_1\} \\
\qquad \wedge\, oplus(a,\{(\mathsf{f},c)\},a') \wedge s'_m = s_m) \\
\wedge\, dom(s',m_6) \wedge nsubset(s'_m,m_6)
\end{array}
\tag{50}
$$

- $\mathsf{fst}()$

$$
\begin{array}{l}
s_m \subseteq \mathrm{dom}\, s \\
\wedge\, a' = a \oplus \{\mathsf{c} \mapsto a(\mathsf{s})\} \wedge s'_m = \emptyset \wedge s' = s \\
\implies s'_m \subseteq \mathrm{dom}\, s'
\end{array}
\tag{51}
$$

$$
\begin{array}{l}
dom(s,m_7) \wedge subset(s_m,m_7) \\
\wedge\, apply(a,\mathsf{s},m_1) \wedge oplus(a,\{(\mathsf{c},m_1)\},a') \wedge s'_m = \emptyset \wedge s' = s \\
\wedge\, dom(s',m_6) \wedge nsubset(s'_m,m_6)
\end{array}
\tag{52}
$$

- $\mathsf{next}()$

$$
\begin{array}{l}
s_m \subseteq \mathrm{dom}\, s \\
\wedge\, a(\mathsf{c}) \neq null \wedge ret = z \wedge s(a(\mathsf{c})) = (y,z) \wedge a' = a \oplus \{\mathsf{p} \mapsto a(\mathsf{c}), \mathsf{c} \mapsto y\} \\
\wedge\, s'_m = \{a(\mathsf{c}) \sqcup s_m\} \\
\implies s'_m \subseteq \mathrm{dom}\, s'
\end{array}
\tag{53}
$$

$$
\begin{array}{l}
dom(s,m_7) \wedge subset(s_m,m_7) \\
\wedge\, ret = z \wedge apply(a,\mathsf{c},m_1) \wedge m_1 \neq null \wedge apply(s,m_1,(y,z)) \wedge oplus(a,\{(\mathsf{p},m_1),(\mathsf{c},y)\},a') \\
\wedge\, s'_m = \{m_1 \sqcup s_p\} \wedge s' = s \\
\wedge\, dom(s',m_6) \wedge nsubset(s'_m,m_6)
\end{array}
\tag{54}
$$

- $\mathsf{rpl}()$

$$
\begin{array}{l}
s_m \subseteq \mathrm{dom}\, s \\
\wedge\, a(\mathsf{p}) \neq null \wedge s' = s \oplus \{a(\mathsf{p}) \mapsto (y,e)\} \wedge s(a(\mathsf{p})) = (y,\_) \wedge s'_m = s_m \\
\implies s'_m \subseteq \mathrm{dom}\, s'
\end{array}
\tag{55}
$$

$$
\begin{array}{l}
dom(s,m_7) \wedge subset(s_m,m_7) \\
\wedge\, apply(a,\mathsf{p},m_1) \wedge m_1 \neq null \wedge apply(s,m_1,(y,\_)) \wedge oplus(s,\{m_1 \mapsto (y,e)\},s') \wedge s'_m = s_m \\
\wedge\, dom(s',m_6) \wedge nsubset(s'_m,m_6)
\end{array}
\tag{56}
$$

- del()

$$\begin{array}{l} s_m \subseteq \mathrm{dom}\, s \\ \wedge\, \mathrm{ran}\, a' = \{null\} \wedge s' = \emptyset \wedge s'_m = \emptyset \\ \implies s'_m \subseteq \mathrm{dom}\, s' \end{array} \tag{57}$$

$$\begin{array}{l} dom(s, m_7) \wedge subset(s_m, m_7) \\ \wedge\, ran(a', \{null\}) \wedge s' = \emptyset \wedge s'_m = \emptyset \\ \wedge\, dom(s', m_6) \wedge nsubset(s'_m, m_6) \end{array} \tag{58}$$

A.3.3 A well-formedness invariant

Now we prove that one of the many well-formedness properties of s is an invariant, namely:

$$w \in \mathrm{dom}(\mathrm{ran}\, s) \wedge w \neq null \implies w \in \mathrm{dom}\, s \tag{59}$$

That is, the node in s are chained through the first component of the image of each location. For example:

$$s = \{c_1 \mapsto (c_2, e_1), c_2 \mapsto (c_3, e_2), c_3 \mapsto (null, e_3)\}$$

Then, for instance, c_3 is the first component of the image of one element of s and so it must be the location of another node of s.

- add()

$$\begin{array}{l} w \in \mathrm{dom}(\mathrm{ran}\, s) \wedge w \neq null \implies w \in \mathrm{dom}\, s \\ \wedge\, (s = \emptyset \wedge s' = \{c \mapsto (null, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m \\ \quad \vee\, s = \{a(\mathsf{f}) \mapsto (y, z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s \\ \qquad \wedge\, s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c, z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m) \\ \implies (w \in \mathrm{dom}(\mathrm{ran}\, s') \wedge w \neq null \implies w \in \mathrm{dom}\, s') \end{array} \tag{60}$$

$$\begin{array}{l} ran(s, m_5) \wedge dom(m_5, m_6) \wedge dom(s, m_7) \wedge (w \notin m_6 \vee w = null \vee w \in m_7) \\ \wedge\, (s = \emptyset \wedge s' = \{(c, (null, e))\} \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m \\ \quad \vee\, apply(a, \mathsf{f}, m_2) \wedge s = \{(m_2, (y, z)) \sqcup s_1\} \wedge dom(s, m_1) \wedge c \notin m_1 \\ \qquad \wedge\, apply(a, \mathsf{f}, m_2) \wedge s' = \{(c, (null, e)), (m_2, (c, z)) \sqcup s_1\} \\ \qquad \wedge\, oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m) \\ \wedge\, ran(s', m_8) \wedge dom(m_8, m_9) \wedge dom(s', m_0) \wedge w \in m_9 \wedge w \neq null \wedge w \notin m_0 \end{array} \tag{61}$$

- fst()

$$\begin{array}{l} w \in \mathrm{dom}(\mathrm{ran}\, s) \wedge w \neq null \implies w \in \mathrm{dom}\, s \\ \wedge\, a' = a \oplus \{\mathsf{c} \mapsto a(\mathsf{s})\} \wedge s'_m = \emptyset \wedge s' = s \\ \implies (w \in \mathrm{dom}(\mathrm{ran}\, s') \wedge w \neq null \implies w \in \mathrm{dom}\, s') \end{array} \tag{62}$$

$$\begin{array}{l} ran(s, m_5) \wedge dom(m_5, m_6) \wedge dom(s, m_7) \wedge (w \notin m_6 \vee w = null \vee w \in m_7) \\ \wedge\, apply(a, \mathsf{s}, m_1) \wedge oplus(a, \{(\mathsf{c}, m_1)\}, a') \wedge s'_m = \emptyset \wedge s' = s \\ \wedge\, ran(s', m_8) \wedge dom(m_8, m_9) \wedge dom(s', m_0) \wedge w \in m_9 \wedge w \neq null \wedge w \notin m_0 \end{array} \tag{63}$$

- next()

$$\begin{array}{l} w \in \mathrm{dom}(\mathrm{ran}\, s) \wedge w \neq \mathit{null} \implies w \in \mathrm{dom}\, s \\ \wedge\, a(\mathsf{c}) \neq \mathit{null} \wedge \mathit{ret} = z \wedge s(a(\mathsf{c})) = (y,z) \wedge a' = a \oplus \{\mathsf{p} \mapsto a(\mathsf{c}), \mathsf{c} \mapsto y\} \\ \wedge\, s'_m = \{a(\mathsf{c}) \sqcup s_m\} \\ \implies (w \in \mathrm{dom}(\mathrm{ran}\, s') \wedge w \neq \mathit{null} \implies w \in \mathrm{dom}\, s') \end{array} \tag{64}$$

$$\begin{array}{l} \mathit{ran}(s, m_5) \wedge \mathit{dom}(m_5, m_6) \wedge \mathit{dom}(s, m_7) \wedge (w \notin m_6 \vee w = \mathit{null} \vee w \in m_7) \\ \wedge\, \mathit{ret} = z \wedge \mathit{apply}(a, \mathsf{c}, m_1) \wedge m_1 \neq \mathit{null} \wedge \mathit{apply}(s, m_1, (y,z)) \wedge \mathit{oplus}(a, \{(\mathsf{p}, m_1), (\mathsf{c}, y)\}, a') \\ \wedge\, s'_m = \{m_1 \sqcup s_p\} \wedge s' = s \\ \wedge\, \mathit{ran}(s', m_8) \wedge \mathit{dom}(m_8, m_9) \wedge \mathit{dom}(s', m_0) \wedge w \in m_9 \wedge w \neq \mathit{null} \wedge w \notin m_0 \end{array} \tag{65}$$

- rpl()

$$\begin{array}{l} w \in \mathrm{dom}(\mathrm{ran}\, s) \wedge w \neq \mathit{null} \implies w \in \mathrm{dom}\, s \\ \wedge\, a(\mathsf{p}) \neq \mathit{null} \wedge s' = s \oplus \{a(\mathsf{p}) \mapsto (y,e)\} \wedge s(a(\mathsf{p})) = (y,\_) \wedge s'_m = s_m \\ \implies (w \in \mathrm{dom}(\mathrm{ran}\, s') \wedge w \neq \mathit{null} \implies w \in \mathrm{dom}\, s') \end{array} \tag{66}$$

$$\begin{array}{l} \mathit{ran}(s, m_5) \wedge \mathit{dom}(m_5, m_6) \wedge \mathit{dom}(s, m_7) \wedge (w \notin m_6 \vee w = \mathit{null} \vee w \in m_7) \\ \wedge\, \mathit{apply}(a, \mathsf{p}, m_1) \wedge m_1 \neq \mathit{null} \wedge \mathit{apply}(s, m_1, (y,\_)) \wedge \mathit{oplus}(s, \{m_1 \mapsto (y,e)\}, s') \wedge s'_m = s_m \\ \wedge\, \mathit{ran}(s', m_8) \wedge \mathit{dom}(m_8, m_9) \wedge \mathit{dom}(s', m_0) \wedge w \in m_9 \wedge w \neq \mathit{null} \wedge w \notin m_0 \end{array} \tag{67}$$

- del()

$$\begin{array}{l} w \in \mathrm{dom}(\mathrm{ran}\, s) \wedge w \neq \mathit{null} \implies w \in \mathrm{dom}\, s \\ \wedge\, \mathrm{ran}\, a' = \{\mathit{null}\} \wedge s' = \emptyset \wedge s'_m = \emptyset \\ \implies (w \in \mathrm{dom}(\mathrm{ran}\, s') \wedge w \neq \mathit{null} \implies w \in \mathrm{dom}\, s') \end{array} \tag{68}$$

$$\begin{array}{l} \mathit{ran}(s, m_5) \wedge \mathit{dom}(m_5, m_6) \wedge \mathit{dom}(s, m_7) \wedge (w \notin m_6 \vee w = \mathit{null} \vee w \in m_7) \\ \wedge\, \mathit{ran}(a', \{\mathit{null}\}) \wedge s' = \emptyset \wedge s'_m = \emptyset \\ \wedge\, \mathit{ran}(s', m_8) \wedge \mathit{dom}(m_8, m_9) \wedge \mathit{dom}(s', m_0) \wedge w \in m_9 \wedge w \neq \mathit{null} \wedge w \notin m_0 \end{array} \tag{69}$$

A.3.4 Invariant: $a(\mathsf{f}) \in \mathrm{dom}\, s$

Actually the invariant is a little bit more complex:

$$s \neq \emptyset \wedge a(\mathsf{f}) \in \mathrm{dom}\, s \vee s = \emptyset \wedge a(\mathsf{f}) = \mathit{null} \tag{70}$$

Then, we have the following proof obligations.

- add()

$$
\begin{array}{l}
(s \neq \emptyset \wedge a(\mathsf{f}) \in \mathrm{dom}\, s \vee s = \emptyset \wedge a(\mathsf{f}) = null) \\
\wedge (s = \emptyset \wedge s' = \{c \mapsto (null, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m \\
\quad \vee s = \{a(\mathsf{f}) \mapsto (y,z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s \\
\qquad \wedge s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c,z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m) \\
\implies (s' \neq \emptyset \wedge a'(\mathsf{f}) \in \mathrm{dom}\, s' \vee s' = \emptyset \wedge a'(\mathsf{f}) = null)
\end{array}
\tag{71}
$$

$$
\begin{array}{l}
(s \neq \emptyset \wedge apply(a, \mathsf{f}, m_6) \wedge dom(s, m_7) \wedge m_6 \in m_7 \vee s = \emptyset \wedge apply(a, \mathsf{f}, null)) \\
\wedge (s = \emptyset \wedge s' = \{(c, (null, e))\} \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m \\
\quad \vee apply(a, \mathsf{f}, m_2) \wedge s = \{(m_2, (y,z)) \sqcup s_1\} \wedge dom(s, m_1) \wedge c \notin m_1 \\
\qquad \wedge apply(a, \mathsf{f}, m_2) \wedge s' = \{(c, (null, e)), (m_2, (c,z)) \sqcup s_1\} \\
\qquad \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m) \\
\wedge (s' = \emptyset \vee apply(a', \mathsf{f}, m_8) \wedge dom(s', m_9) \wedge m_8 \notin m_9) \\
\wedge (s' \neq \emptyset \vee napply(a', \mathsf{f}, null))
\end{array}
\tag{72}
$$

- fst()

$$
\begin{array}{l}
(s \neq \emptyset \wedge a(\mathsf{f}) \in \mathrm{dom}\, s \vee s = \emptyset \wedge a(\mathsf{f}) = null) \\
\wedge a' = a \oplus \{\mathsf{c} \mapsto a(\mathsf{s})\} \wedge s'_m = \emptyset \wedge s' = s \\
\implies (s' \neq \emptyset \wedge a'(\mathsf{f}) \in \mathrm{dom}\, s' \vee s' = \emptyset \wedge a'(\mathsf{f}) = null)
\end{array}
\tag{73}
$$

$$
\begin{array}{l}
(s \neq \emptyset \wedge apply(a, \mathsf{f}, m_6) \wedge dom(s, m_7) \wedge m_6 \in m_7 \vee s = \emptyset \wedge apply(a, \mathsf{f}, null)) \\
\wedge apply(a, \mathsf{s}, m_1) \wedge oplus(a, \{(\mathsf{c}, m_1)\}, a') \wedge s'_m = \emptyset \wedge s' = s \\
\wedge (s' = \emptyset \vee apply(a', \mathsf{f}, m_8) \wedge dom(s', m_9) \wedge m_8 \notin m_9) \\
\wedge (s' \neq \emptyset \vee napply(a', \mathsf{f}, null))
\end{array}
\tag{74}
$$

- next()

$$
\begin{array}{l}
(s \neq \emptyset \wedge a(\mathsf{f}) \in \mathrm{dom}\, s \vee s = \emptyset \wedge a(\mathsf{f}) = null) \\
\wedge a(\mathsf{c}) \neq null \wedge ret = z \wedge s(a(\mathsf{c})) = (y,z) \wedge a' = a \oplus \{\mathsf{p} \mapsto a(\mathsf{c}), \mathsf{c} \mapsto y\} \\
\wedge s'_m = \{a(\mathsf{c}) \sqcup s_m\} \\
\implies (s' \neq \emptyset \wedge a'(\mathsf{f}) \in \mathrm{dom}\, s' \vee s' = \emptyset \wedge a'(\mathsf{f}) = null)
\end{array}
\tag{75}
$$

$$
\begin{array}{l}
(s \neq \emptyset \wedge apply(a, \mathsf{f}, m_6) \wedge dom(s, m_7) \wedge m_6 \in m_7 \vee s = \emptyset \wedge apply(a, \mathsf{f}, null)) \\
\wedge ret = z \wedge apply(a, \mathsf{c}, m_1) \wedge m_1 \neq null \wedge apply(s, m_1, (y,z)) \wedge oplus(a, \{(\mathsf{p}, m_1), (\mathsf{c}, y)\}, a') \\
\wedge s'_m = \{m_1 \sqcup s_p\} \wedge s' = s \\
\wedge (s' = \emptyset \vee apply(a', \mathsf{f}, m_8) \wedge dom(s', m_9) \wedge m_8 \notin m_9) \\
\wedge (s' \neq \emptyset \vee napply(a', \mathsf{f}, null))
\end{array}
\tag{76}
$$

- rpl()

$$\begin{array}{l}(s \neq \emptyset \wedge apply(a,\mathsf{f},m_6) \wedge dom(s,m_7) \wedge m_6 \in m_7 \vee s = \emptyset \wedge apply(a,\mathsf{f},null)) \\ \wedge a(\mathsf{p}) \neq null \wedge s' = s \oplus \{a(\mathsf{p}) \mapsto (y,e)\} \wedge s(a(\mathsf{p})) = (y,\_) \wedge s'_m = s_m \\ \implies (s' \neq \emptyset \wedge a'(\mathsf{f}) \in \mathrm{dom}\, s' \vee s' = \emptyset \wedge a'(\mathsf{f}) = null)\end{array} \tag{77}$$

$$\begin{array}{l}dom(s,m_7) \wedge subset(s_m,m_7) \\ \wedge apply(a,\mathsf{p},m_1) \wedge m_1 \neq null \wedge apply(s,m_1,(y,\_)) \wedge oplus(s,\{m_1 \mapsto (y,e)\},s') \wedge s'_m = s_m \\ \wedge (s' = \emptyset \vee apply(a',\mathsf{f},m_8) \wedge dom(s',m_9) \wedge m_8 \notin m_9) \\ \wedge (s' \neq \emptyset \vee napply(a',\mathsf{f},null))\end{array} \tag{78}$$

- del()

$$\begin{array}{l}(s \neq \emptyset \wedge apply(a,\mathsf{f},m_6) \wedge dom(s,m_7) \wedge m_6 \in m_7 \vee s = \emptyset \wedge apply(a,\mathsf{f},null)) \\ \wedge \mathrm{ran}\, a' = \{null\} \wedge s' = \emptyset \wedge s'_m = \emptyset \\ \implies (s' \neq \emptyset \wedge a'(\mathsf{f}) \in \mathrm{dom}\, s' \vee s' = \emptyset \wedge a'(\mathsf{f}) = null)\end{array} \tag{79}$$

$$\begin{array}{l}dom(s,m_7) \wedge subset(s_m,m_7) \\ \wedge ran(a',\{null\}) \wedge s' = \emptyset \wedge s'_m = \emptyset \\ \wedge (s' = \emptyset \vee apply(a',\mathsf{f},m_8) \wedge dom(s',m_9) \wedge m_8 \notin m_9) \\ \wedge (s' \neq \emptyset \vee napply(a',\mathsf{f},null))\end{array} \tag{80}$$

A.3.5 Other properties

Three very simple sets properties to confirm that $s_p \parallel s_r$, $s = s_p \cup s_r$ and $s_p, s_r \in \_ \nrightarrow \_$. Recall that we define $s_p \triangleq s_m \lhd s$ and $s_r \triangleq s \setminus s_p$. As always written first in our specification language followed by the negation in $\{log\}$.

$$s_p = s_m \lhd s \wedge s_r = s \setminus s_p \implies s_p \parallel s_r \tag{81}$$

$$dres(s_m,s,s_p) \wedge diff(s,s_p,s_r) \wedge ndisj(s_p,s_r) \tag{82}$$

$$s_p = s_m \lhd s \wedge s_r = s \setminus s_p \implies s = s_p \cup s_r \tag{83}$$

$$dres(s_m,s,s_p) \wedge diff(s,s_p,s_r) \wedge nun(s_p,s_r,s) \tag{84}$$

Given that we have already proved that s is a partial function, we can assume this to prove that s_p and s_r are also partial functions.

$$s_p = s_m \lhd s \wedge s_r = s \setminus s_p \wedge s \in \_ \nrightarrow \_ \implies s_p \in \_ \nrightarrow \_ \wedge s_r \in \_ \nrightarrow \_ \tag{85}$$

$$dres(s_m, s, s_p) \wedge diff(s, s_p, s_r) \wedge pfun(s) \wedge (npfun(s_p) \vee npfun(s_r)) \tag{86}$$

This is another well-formedness invariant on s but we prove it just for add() because is the only subroutine that modifies s in a non-trivial way. The invariant goes as follows: in a non-empty list, *null* is the first component of the image of the last node.

$$\begin{array}{l}(w \in \mathrm{dom}(\mathrm{ran}\, s) \wedge w \notin \mathrm{dom}\, s \implies w = null) \\ \wedge (s = \emptyset \wedge s' = \{c \mapsto (null, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m \\ \quad \vee s = \{a(\mathsf{f}) \mapsto (y, z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s \\ \quad\quad \wedge s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c, z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m) \\ \implies (w \in \mathrm{dom}(\mathrm{ran}\, s') \wedge w \notin \mathrm{dom}\, s' \implies w = null)\end{array} \tag{87}$$

The negation in $\{log\}$:

$$\begin{array}{l}(ran(s, m_0) \wedge dom(m_0, m_9) \wedge dom(s, m_5) \wedge (w \notin m_9 \vee w \in m_5 \vee w = null) \\ \wedge (s = \emptyset \wedge s' = \{(c, (null, e))\} \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m \\ \quad \vee apply(a, \mathsf{f}, m_2) \wedge s = \{(m_2, (y, z)) \sqcup s_1\} \wedge dom(s, m_1) \wedge c \notin m_1 \\ \quad\quad \wedge apply(a, \mathsf{f}, m_2) \wedge s' = \{(c, (null, e)), (m_2, (c, z)) \sqcup s_1\} \\ \quad\quad \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m) \\ \wedge ran(s', m_6) \wedge dom(m_6, m_7) \wedge dom(s', m_8) \wedge w \in m_7 \wedge w \notin m_8 \wedge w \neq null\end{array} \tag{88}$$

Another well-formedness invariant on s is that *null* does not belong to the domain of s (i.e. *null* never points to a node). Again we prove it just for add().

$$\begin{array}{l}null \notin \mathrm{dom}\, s \\ \wedge (s = \emptyset \wedge s' = \{c \mapsto (null, e)\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m \\ \quad \vee s = \{a(\mathsf{f}) \mapsto (y, z) \sqcup s_1\} \wedge c \notin \mathrm{dom}\, s \\ \quad\quad \wedge s' = \{c \mapsto (null, e), a(\mathsf{f}) \mapsto (c, z) \sqcup s_1\} \wedge a' = a \oplus \{\mathsf{f} \mapsto c\} \wedge s'_m = s_m) \\ \implies null \notin \mathrm{dom}\, s'\end{array} \tag{89}$$

The negation in $\{log\}$:

$$\begin{array}{l}dom(s, m_0) \wedge null \notin m_0 \\ \wedge (s = \emptyset \wedge s' = \{(c, (null, e))\} \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m \\ \quad \vee apply(a, \mathsf{f}, m_2) \wedge s = \{(m_2, (y, z)) \sqcup s_1\} \wedge dom(s, m_1) \wedge c \notin m_1 \\ \quad\quad \wedge apply(a, \mathsf{f}, m_2) \wedge s' = \{(c, (null, e)), (m_2, (c, z)) \sqcup s_1\} \\ \quad\quad \wedge oplus(a, \{(\mathsf{f}, c)\}, a') \wedge s'_m = s_m) \\ \wedge dom(s', m_7) \wedge null \in m_7\end{array} \tag{90}$$

But the proof fails! Part of the answer returned by $\{log\}$ is:

$$s = \emptyset \wedge s' = \{null \mapsto (null, e)\}$$

that is, $\{log\}$ finds that x, the location returned by **new**, can be *null*. Check add() assertions.

The reason is that we missed the fact that **new** always delivers non-null locations. By adding this fact in the assertion following a **new** instruction, we can propagate it right through the post-condition of the subroutine. In this case the $\{log\}$ formula becomes:

$$\begin{array}{l} dom(s,m_0) \wedge null \notin m_0 \\ \wedge\, (s = \emptyset \wedge s' = \{(c,(null,e))\} \wedge oplus(a,\{(\mathtt{f},c)\},a') \wedge s'_m = s_m \wedge \; x \neq null \\ \quad \vee\, apply(a,\mathtt{f},m_2) \wedge s = \{(m_2,(y,z)) \sqcup s_1\} \wedge dom(s,m_1) \wedge c \notin m_1 \\ \qquad \wedge\, apply(a,\mathtt{f},m_2) \wedge s' = \{(c,(null,e)),(m_2,(c,z)) \sqcup s_1\} \\ \qquad \wedge\, oplus(a,\{(\mathtt{f},c)\},a') \wedge s'_m = s_m \wedge \; x \neq null \;) \\ \wedge\, dom(s',m_7) \wedge null \in m_7 \end{array} \tag{91}$$

which is solved by $\{log\}$ in no time.

Observe that this change in add()'s post-condition requires to redo all the proofs involving this subroutine.