Weak Arithmetic Completeness of Object-Oriented First-Order Assertion Networks [⋆]

Stijn de Gouw[2,3], Frank de Boer[2,3], Wolfgang Ahrendt[1], and Richard Bubel[4]

[1] Chalmers University, Göteborg, Sweden
[2] CWI, Amsterdam, The Netherlands
[3] Leiden University, The Netherlands
[4] Technische Universität Darmstadt, Germany

Abstract. We present a completeness proof of the inductive assertion method for object-oriented programs extended with auxiliary variables. The class of programs considered are assumed to compute over structures which include the standard interpretation of Presburger arithmetic. Further, the assertion language is first-order, i.e., quantification only ranges over basic types like that of the natural numbers, Boolean and Object.

1 Introduction

In [5], Cook introduced a general condition of completeness of Hoare logics in terms of the expressibility of the weakest precondition. Harel defined in [9] a general class of (first-order) structures which include the standard interpretation of Peano Arithmetic. For this class standard coding techniques suffice to express the weakest precondition. This is *not* the case for programs with general abstract data structures as studied by Tucker and Zucker in [18]. They prove therefore expressibility of the weakest precondition in a weak second-order language which contains quantification over finite sequences.

In this paper we study arithmetic completeness of inductive assertion networks [7] for proving correctness of object-oriented programs. Our main contribution shows that the inductive assertion method is complete for a class of programs which compute over *weak* arithmetic structures. Such structures include the standard interpretation of Presburger arithmetic. Though multiplication can be simulated in the programming language by repeated addition using a while loop, omitting multiplication limits the expressiveness of the assertion language severely, as can be seen by the following argument. In a Turing complete programming language, any recursively enumerable set is the weakest precondition of some program. But by Presburger's result [16], formulas of Presburger arithmetic define only recursive sets, and hence, cannot express the weakest precondition (nor strongest postcondition) of arbitrary programs.

[⋆] This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu/`)

We show however that the strongest postcondition *is* expressible using only Presburger arithmetic for object-oriented programs, when appropriately instrumented with auxiliary array variables. In particular we demonstrate that treating arrays as objects allows a direct representation of a computation at the abstraction level of both the programming language and the first-order logic and enables us to express arbitrary properties of the heap in first-order logic. As a practical consequence, any first-order logic theorem prover can be used to prove verification conditions of instrumented object-oriented programs. In contrast to second-order logic (as used by Tucker and Zucker) or recursive predicates (separation logic) which are traditionally used to express heap properties, first-order logic has desirable proof-theoretical properties: first-order logic is the strongest logic satisfying completeness and the Löwenheim-Skolem theorem [10]. Our approach is tool supported by a special version of KeY [4], a state-of-the-art prover for Java. On the theoretical side we show that the above expressiveness result implies completeness: for any valid pre-/postcondition specification of an object-oriented program there is an inductive assertion network of the program extended with *auxiliary variables.*

Finally, using auxiliary variables allows us to restrict the network to *recursive* assertions in case the given pre- and postcondition are recursive. This possibility is of fundamental practical importance as recursive assertions are effectively computable and, hence, we can use them for run-time checking of programs.

Related work. In [6] completeness of the inductive assertion method has been studied for recursive programs only and without the use of auxiliary variables. The absence of auxiliary variables made it necessary to resort to an infinite collection of intermediate assertions. Apt showed in [2] that recursive assertions are complete for while programs extended with auxiliary variables. In this paper we combine and extend on the above results by showing that recursive assertions are complete for object-oriented programs extended with auxiliary variables.

Completeness of Hoare logics for object-oriented programs is also formally proven e.g. in [15]. This completeness result however is based on the expressibility of the strongest postcondition in a weak second-order language which contains quantification over finite sequences. In [3] completeness for an object-oriented core language without object creation is proven assuming the standard interpretation of Peano arithmetic for the expressibility of the weakest precondition. We are not aware of any other completeness result based on weak arithmetic structures using only Presburger arithmetic and an assertion language which only contains quantification over basic types., i.e., integer, Boolean and Object.

2 The Programming and Specification Language

We introduce now our *core object-oriented language.* The language is strongly typed and contains the primitive types Presburger and Boolean. The only operations provided by Presburger are those of Presburger arithmetic (0, successor and addition). The only operations allowed on Booleans are those of Boolean algebra. Additionally there are user-defined class types C, predefined class types $T[\,]$ of *unbounded* arrays in which the elements are of type T and a union type Object. Arrays can be dynamically allocated and are indexed by natural numbers. Multi-dimensional arrays are modeled (as in Java) as arrays of arrays. We assume a transitive reflexive subtype relation between types with Object being the supertype of any class type. Our language can be statically type checked.

2.1 Syntax

Expressions of our language are side-effect free and generated by the grammar:

$$e ::= u \,|\, e.x \,|\, \mathsf{null} \,|\, e_1 = e_2 \,|\, \mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi} \,|\, \mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi} \,|\, e_1[e_2] \,|\, f(e_1, \ldots, e_n) \,|\, C(e)$$

Variables are indicated by u while x denotes a typical field. The Boolean expression $e_1 = e_2$ denotes the test for equality between the values of e_1 and e_2. For object expressions we use Java reference semantics, i.e., to be equal e_1 and e_2 must denote the same object identity. The expression $\mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi}$ has value e if the Boolean expression b is true, otherwise it has an arbitrary value. This expression allows a systematic approach to proving properties about partial functions. A conditional expression is denoted by $\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi}$. The motivation for including it in our core language is that it significantly simplifies treatment of aliasing. If e_1 is an expression of type $T[\,]$ and e_2 is an expression of type Presburger then $e_1[e_2]$ is an expression of type T, also called a subscripted variable. Here T itself can be an array type. For example, if a is an array variable of type Presburger[ ][ ] then the expression $a[0]$ denotes an array of type Presburger[ ]. The function $f(e_1, ..., e_n)$ denotes a Presburger arithmetic or Boolean operation of arity n. For class types C the Boolean expression $C(e)$ is true if and only if the dynamic type of e is C. Dynamic binding can be simulated in our core language with such expressions. Expressions of a class type can only be compared for equality, dereferenced, accessed as an array if the object is of an array type, or appear as arguments of a class predicate, if-expression, or conditional expression.

The language of statements is generated by the following grammar:

$$\begin{aligned} s ::= {} & s_1; s_2 \mid \mathsf{if}\, b\, \mathsf{then}\, s_2\, \mathsf{else}\, s_3\, \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \mid \mathsf{abort} \mid \\ & e_0.m(e_1, ..., e_n) \mid u := \mathsf{new} \mid u := e \mid e_1[e_2] := e \mid e_1.x := e \end{aligned}$$

The abort statement causes a failure. A statement $u := \mathsf{new}$ assigns to the program variable u a newly created object of the declared type (possibly an array type) of u. Objects are never destroyed. We assume every statement and expression to be well-typed. A *program* in our language consists of a *main statement* together with sets for variable-, field- and method declarations (respectively Var and F_{C}, M_{C} for every class C).

Assertions are generated by the following *first-order* language:

$$\phi ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi \rightarrow \phi_2 \mid \exists l : \phi \mid \forall l : \phi$$

Here, b is a Boolean expression and l is a logical variable of any type.

2.2 Semantics

The basic notion underlying the semantics of both the programming language and the assertion language is that of a many-sorted structure of the form

$$(dom(\text{Presburger}), \{\mathsf{true}, \mathsf{false}\}, dom(T_1), \ldots, dom(T_n), I)$$

where T_i for $1 \leq i \leq n$, denotes a class type, array type or some abstract data type and I denotes an interpretation I of the non-logical function and predicate symbols (i.e. arithmetic and logical operations). The non-logical symbols include at least (i) variables declared in Var; (ii) for every class C, its fields declared in F_{C}; (iii) for every class C a unary predicate C of type Object $\rightarrow$ Boolean; and (iv) for each array type $T[\,]$ an access function $[\,]_{T[\,]}$ of type Presburger $\rightarrow (T[\,] \rightarrow T)$. The domains of different class types are assumed to be disjoint. There is no need for a separate sort for Object, semantically this set is simply the union of all the sorts for the class types (which includes array types). It is crucial here that the structure fixes the standard interpretation of both the types Presburger and Boolean, and the arithmetical and logical operations defined on them. The interpretation of the other sorts and operations are user-defined (i.e. not fixed).

We write $M(s)$ instead of $I(s)$ for the interpretation of the non-logical symbol s and $M(T)$ for the sort $dom(T)$ in a structure M of our language.

If u is declared in Var as a variable of type T, it is interpreted as an individual of the sort $M(T)$. A field $x \in F_{\text{C}}$ of type T is interpreted as a unary function $M(\text{C}) \rightarrow M(T)$. Array access functions $[\,]_{T[\,]}$ are interpreted as binary functions $M(\text{Presburger}) \rightarrow (M(T[\,]) \rightarrow M(T))$. Thus array indices can be seen as fields.

Semantics of Expressions and Statements. The meaning of an expression e of type T is a total function $[\![e]\!]$ that maps a structure M to an individual of $M(T)$. This function is defined by induction on e. Here are the main cases:

- $[\![e_1.x]\!](M) = M(x)([\![e_1]\!](M))$.
- $[\![e_1[e_2]]\!](M) = M([\,]_{T[\,]})([\![e_2]\!](M))([\![e_1]\!](M))$
  where e_1 has the array type $T[\,]$ and e_2 has type Presburger.
- $[\![\text{C}(e_1)]\!](M) = \mathsf{true}$ iff $[\![e_1]\!](M) \in M(\text{C})$

As the meaning function of our semantics is total, some meaning is assigned to the expression $\mathsf{null}.x$. However, in the execution of programs their meaning is given operationally by executing the abort statement.

Statements in our language are deterministic and can fail (abort) or diverge. We define the meaning of a statement in terms of a small-step operational semantics, and use the (quite common) notation

$$\langle s, M \rangle \longrightarrow \langle s', M' \rangle$$

to express that executing s in structure M, results after one step in the statement s' and structure M'. We use $\longrightarrow^*$ for the reflexive transitive closure of this transition relation. We omit s' if s immediately terminates from M. Since calls can appear in statements, the definition of the above transition relation depends in general on the method declarations. Note that throughout execution, assignments to variables and fields change the structure in the interpretation of the variables and fields respectively. The interpretation of the array access function changes due to assignments to subscripted variables. Moreover during executing, the sorts $dom(\text{C}_i)$ containing instances of C_i are extended with new objects by object creations $u := \mathsf{new}$. Statements do not affect the sorts Presburger, Boolean, and the interpretation of the other non-logical symbols.

The meaning of normal assignments, conditional statements and while loops is defined in the standard way. Hence, we focus on the semantics of array creation. First define for each type a default value: $init_{\text{Presburger}} = 0, init_{\text{Boolean}} = \mathsf{false}$ and $init_{\text{C}} = \mathsf{null}$. For the selection of a new object of class C we use a choice function ν on a structure M and class C to get a fresh object $\nu(M, \text{C})$ of class C which satisfies $\nu(M, \text{C}) \notin M(T)$ for any type T (in particular, $\nu(M, \text{C}) \notin M(\text{C})$). Clearly, without loss of generality we may assume that $\nu(M, \text{C})$ only depends on $M(\text{C})$ in the sense that this choice function preserves the deterministic nature of our core language (formally: $\nu(M, \text{C}) = \nu(M', \text{C})$ if $M(\text{C}) = M(\text{C})$). Non-deterministic (or random) selection of a fresh object would require reasoning semantically up to a notion of isomorphic models which would unnecessarily complicate proofs.

Let u be of type $T[\,]$. The semantics of an array creation is modeled by:

$$\langle u := \mathsf{new}, M\rangle \longrightarrow M'$$

where M' is changed from M as follows: Let o denote the object identity chosen by $\nu(M, T[\,])$, i.e, $o = \nu(M, T[\,])$ then

1. $M'(T[\,]) = M(T[\,]) \cup \{o\}$.
2. $M'([\,]_{T[\,]})(n)(o) = init_T$ for all $n \in M(\text{Presburger})$.
3. $M'(u) = o$.

The second clause states that all array elements have initially their default value.

The operational semantics of a program is given by the one of its main statement, executed in the *initial structure* M_0. In M_0, for every class type C no objects other than null_{C} exist, and all variables have their default value.

Semantics of Assertions. The semantics of assertions is defined by the usual Tarski truth definition. Interestingly, even though we allow quantification over array objects, all assertions are *first-order* formulas (interpreted over arbitrary structures obeying the first-order Presburger and Boolean algebra axioms, including non-standard interpretations). This is because of a subtle difference in meaning between modeling arrays as sequences (not first-order), or as pointers to sequences (first-order [17, 12]): In case s ranges over (finite) sequences $\exists s : s[0] = 0$ expresses that there exists a sequence s of natural numbers, of which the first is 0. This sequence itself is not an element of the domain of a structure for our many-sorted dynamic logic language, but rather a sequence of elements of the domain Presburger. In this interpretation the above formula is valid. In this paper we model arrays as pointers to data-structures in the heap (e.g., structure) as in Java. If a is a logical variable of type Presburger[ ] then $\exists a : a[0] = 0$ asserts the existence of an array object (an individual of the sort for Presburger[ ]) in which currently the first element is 0. This formula is not valid, for it is false in all structures in which no such array exists. Note that the *extensionality* axiom $\forall a, b, n : a[n] = b[n] \rightarrow a = b$ for arrays is also not valid.

3 Inductive Assertion Networks

We extend Floyd's inductive assertion method to object-oriented programs. An inductive assertion network is a labelled transition system where transitions are labelled with conditional assignments of (local) variables. A labelled transition may fire if its (pre-)condition is satisfied. The corresponding assignment is executed and the state updated accordingly. Obviously control-flow graphs of imperative programs fall into the class of these transition systems. The labelled transition system is extended to an assertion network by assigning each state a (set of) assertions. An assertion network is called inductive if and only if whenever $M(\phi)$ holds for a structure (state) M and the condition of a transition is satisfied, then the assertion ϕ' assigned to the resulting structure holds as well.

We extend Floyd's notion of an inductive assertion network to object-oriented programs: besides basic assignments, transitions can be labelled with object creations and assignments to fields and subscripted variables. This requires a corresponding extension for computing verification conditions, taking for example aliasing into account. Finally we need in general auxiliary variables to describe the object structures in the heap. Because, for instance, first-order logic itself cannot express reachability in linked lists (see Section 5).

As one main feature of our semantics is to model object creation as extension of the underlying structure's domain, the rule for deleting assignments to auxiliary variables as introduced in Owicki and Gries [14] for reasoning about shared variable concurrency is not sound anymore. Clearly we cannot remove the dynamic allocation of the auxiliary variable u even if u does not appear in the assertions (an assignment $u := \mathsf{new}$ in fact may validate an assertion $\exists l : \phi$, where the logical variable is of the same type as u). To obtain a complete inductive assertion method we allow method signatures extended with auxiliary formal parameters.

A basic assertion network of a program extended with auxiliary variables associates assertions with each (sub)statement of the program. A (finite) set of verification conditions for this annotated program is then generated fully automatically by means of the weakest precondition calculus defined in [1] extended with a substitution for the (dynamic) creation of arrays. The verification conditions of the pre- and postcondition of a method call are defined in the standard way in terms of the pre- and postcondition of the method body, modeling parameter passing by substitution.

4 Expressiveness

In this section we first investigate the expressiveness of auxiliary variables. This leads to the following main result: the set of reachable states at each control point of a general class of instrumented programs is expressible in a first-order assertion language with equality, unbounded arrays and addition. This forms the basis for the completeness of our object-oriented inductive assertion method.

4.1 Instrumentation

Below we show how the computation history of instrumented programs can be stored in auxiliary variables in a canonical manner. The instrumentation must be faithful to the original program:

Definition 1 (Faithful Instrumentation). *Given a set of auxiliary variables, an instrumented program is* faithful *to the original program if neither the value of the normal (non-auxiliary) variables nor the termination behavior is affected by the instrumentation.*

Intuitively the instrumentation adds auxiliary array variables to the original program which record only the changes to the values of variables and fields of the program (including those of an array type). In comparison to storing the full state at each computation step, this allows for a fairly simple update mechanism for the auxiliary variables. Faithful instrumentations allow the expression of properties *of the original program* which cannot be expressed in first-order logic formulas directly. We now list the auxiliary variables, along with a description how they are set during the execution of instrumented programs, assuming a unique line number for each (sub)statement:

- A one-dimensional array variable pc of Presburger[ ] to record the history of the program counter. The intention is that if $pc[i] = j$, then line j was executed in the i-th step of the computation.
- A variable $|pc|$ of type Presburger containing the number of completed computation steps.
- For each variable u of a type T an array variable u' with content type T, and an array variable u'' of type Boolean. If in the i-th step of the computation the value v is assigned to u, then $u'[i] = v$, and $u''[i] = \mathsf{true}$. If the i-th step does not involve an assignment to variable u, we have $u'[i] = init_T$, where T is the type of u, and $u''[i] = \mathsf{false}$, which is the default Boolean value.
- For each field x an array x' of pairs $< o, v >$ (where o is an object identity and v a value)[5], and an array x'' of Boolean. In analogy to the two arrays storing the changes to variables, these two arrays store the changes to the field. The extra object identity is needed to identify the object whose field was changed.
- For each array type T occurring in the program, a one-dimensional array variable Arr'_T of Boolean storing the computation steps in which the interpretation of an array object of type T was changed, and a one-dimensional array Arr_T of triples $< o, n, v >$ storing the new values of the changed element in that array (o is an array object, n an array index and v a value).
- A method parameter loc of type Presburger, which stores the line number on which the call was made.

As two examples we show how the instrumentation of the basic assignment and method call is performed in Figures 1 and 2. The control structures are

[5] Such a type can be easily defined in our language as a class with two fields

```
      $pc[|pc|] := j$ ;
      $u'[|pc|] := e$ ;
$j$ :   $u := e$ ;
      $u''[|pc|] := \text{true}$ ;
      $|pc| := |pc| + 1$ ;
```

Fig. 1. Instrumentation of the statement $u := e$ on line number j

```
      $pc[|pc|] := j$ ;
      $x'[|pc|] :=< e, e' >$ ;
$j$ :   $e.x := e'$ ;
      $x''[|pc|] := \text{true}$ ;
      $|pc| := |pc| + 1$ ;
```

Fig. 2. Instrumentation of the statement $e.x := e'$ on line number j

simply instrumented by updates to the variable pc to record the flow of control. Additionally in a call we pass the line number as a parameter which is used upon return. Given a line number j, by next(j) we denote the line number of the statement which will be executed in the next step of the computation.

To instrument a program with a main statement s_{main} and method bodies $B_1, \ldots, B_n$, label first each program statement uniquely. Then apply the instrumentation given above to s_{main} to obtain s'_{main} and to each method body B_i to obtain the statement B'_i. Next, define a statement init which creates new objects for the auxiliary array variables, and sets $|pc| := 0$. The final instrumented program is given by the main statement $init; s'_{main}$ and method bodies B'_i.

Theorem 1. *The above instrumentation is faithful to the original program.*

4.2 Weak Arithmetic Completeness

Completeness of basic inductive assertion networks has been proven in [6, 11], provided that suitable intermediate assertions exist in the assertion language. Here we demonstrate how to find such assertions for the class of instrumented object-oriented programs as defined previously.

Recall that programs start executing in a fixed initial structure (see Section 2 on semantics of statements). Hence from a purely semantic viewpoint, the intermediate assertion at location l can simply be chosen as the set of all structures reachable in l from the initial structure. Such reachability predicates are reminiscent of the most general correctness formulae introduced by Gorelick in [8] to show completeness for a Hoare logic for recursive programs.

Definition 2. *Let P be a program with statement s on line number l. The reachability predicate $\mathbb{R}_l$ denotes the set of states $\{M | \langle P, M_0 \rangle \longrightarrow^* \langle s; s', M \rangle\}$, where M_0 is a standard model (the initial structure, see Section 2), and s' is the remainder of the program to be executed.*

It remains to show that our first-order assertion language which only assumes the standard interpretation of Presburger arithmetic is expressive enough to define the above reachability predicates *syntactically.* This is indeed the case for instrumented programs. For such programs, the state-based encoding of the computation allows recovering the computation of the instrumented program in the assertion language without using a Gödel encoding (which relies on the presence of multiplication in the assertion language). Our results are not restricted to the specific instrumentation defined in the previous section. In general any faithful instrumentation which allows recovering the computation in the assertion language can be used.

We now describe how the computation of instrumented programs can be recovered in the assertion language. Given an *uninstrumented* program P and a computation step i (i.e. a number), define an assertion $\mathrm{COMP}_{P,i}$ which completely describes the state change induced by the i-th computation step in the instrumented version of P. For an assignment $u := e$ with line number j we define $\mathrm{COMP}_{P,i}$ by

$$pc[i]=j\rightarrow(pc[i+1]=\mathrm{next}(j)\wedge u'[i+1]=\mathrm{Val}(e,i)\wedge u''[i+1]=\mathsf{true}\wedge \mathrm{nochange}_j(i))$$

Assignments to fields or subscripted variables can be handled similarly to the variable assignment above. The expression $\mathrm{Val}(e, i)$ stands for the value of the expression e after the i-th computation step. The interesting case is when e is a (subscripted) variable or field. We show the case when e is a variable u of type T:

$$\begin{aligned}\mathrm{Val}(u,i)=z\leftrightarrow\ &(init_T = z\wedge\forall n\leq|pc|:u''[n]=\mathsf{false})\vee\\ &(\exists n\leq i:u''[n]\wedge u'[n]=z\wedge\forall k(n<k<|pc|):u''[n]=\mathsf{false})\end{aligned}$$

The first disjunct asserts that $\mathrm{Val}(u, i)$ is determined by the last assignment to u which occurred before or on computation step i. The second disjunct asserts that if there was no such assignment, the variable has retained its initial value. The value of a Boolean condition in a given computation step can also be determined easily using the Val function.

The predicate $\mathrm{nochange}_j(i)$ asserts that only the auxiliary variables representing the l.h.s. of the assignment with line number j are affected by the i-th computation step, i.e., all the other auxiliary variables indicate at the i-th step that their represented program variables have not changed. For example, for a program variable u of type T this is expressed simply by the assertion $u'[i] = init_T \wedge u''[i] = \mathsf{false}$; and for arrays of type T this is expressed by $\mathrm{Arr}_T[i] = \mathsf{null}\wedge\mathrm{Arr}'_T[i] = \mathsf{false}$. Note that we make use of the initial default values of the auxiliary variables. We denote by $\mathrm{nochange}(i)$ that *all* auxiliary variables indicate at the i-th step that their program variables have not changed.

To express the reachability predicate at a location l, one must further assert that the current values of the normal (non-auxiliary) variables, fields and array access function are those stored in the auxiliary variables at l (but before the statement at l is executed). Let us abbreviate such an assertion by $\mathrm{aux}(l)$. To see how $\mathrm{aux}(l)$ can be defined in our assertion language, note that for a variable u it simply reduces to the assertion $u = \mathrm{Val}(u, l)$ as defined above. For arrays, a universal quantification ranging over all array indices is necessary. For instance, $\forall n : a[n] = Val(a[n], l)$ characterizes the full contents of the array a at location l. Thus $\mathrm{aux}(l)$ can now be expressed as the (finite) conjunction of such assertions for all variables, fields and arrays. The reachability predicates of instrumented programs can now be readily defined:

Theorem 2. *Let P be an arbitrary program, and let $P*$ be the instrumented version of P. Then the reachability predicate $\mathbb{R}_l$ of $P*$ is defined by the assertion:*

$$aux(l) \wedge pc[|pc|] = l \wedge \forall 0 \leq i < |pc| : COMP_{P,i} \wedge \text{nochange}$$

where nochange *stands for* $\text{nochange}(0) \wedge \forall i > |pc| : pc[i] = 0 \wedge \text{nochange}(i)$.

The next theorem now follows by construction of the reachability predicates:

Theorem 3. *Let $P*$ be a program instrumented with auxiliary variables as described above, and let P' be its annotation with at each location l an assertion which defines $\mathbb{R}_l$. A partial correctness formula $\{p\}P{*}\{q\}$ is true in the initial model if and only if all generated verification conditions for the assertion network $\{p\}P'\{q\}$ are true.*

The above theorem states that we can derive any true correctness formula in first-order logic. For proving the generated verification conditions we can take as axioms all first-order sentences true in the initial model[6] (and use any off-the-shelve theorem prover for first-order logic). This normally results in ineffective proof systems, since by Gödels incompleteness theorem the axioms are typically not recursively enumerable. However by excluding multiplication from the assertion language, it follows from Presburgers result that the set arithmetical axioms is recursive. Thus if the other types are also interpreted in such a way that their first-order theory is recursive, the truth of the generated verification conditions is decidable.

5 Example: Expressing and Verifying Reachability

In the previous section the reachability predicates were defined *uniformly*. In this section we show how to reduce the complexity of the instrumentation significantly by exploiting the structure of a given program and property to prove.

Consider a queue data structure where items of type Presburger can be added to the beginning of the queue and removed from the end of the queue. The queue is backed up by a linked list using a *next* field which points to the next item in the queue. The public interface of such a queue contains of (i) two global variables pointing to its $first$ and $last$ element, (ii) an $enqueue(v)$ method which adds v to the beginning of the queue and (iii) a $dequeue$ method which removes the last item from the queue. Figure 3 visualises the result of the method call $first.enqueue(2)$, where $first$ initially (i.e. before executing the call) points to an item with value 3, and $last$ points to an item with value 25. Let $\text{reach}(f, l, a, n)$ abbreviate the assertion

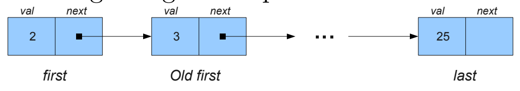

Fig. 3. Queue resulting from $first.enqueue(2)$

$$n \geq 0 \wedge a[0] = f \neq null \neq a[n] = l \wedge l.next = null$$
$$\wedge \forall j(0 \leq j < n) : a[j] \neq null \wedge a[j].next = a[j+1]$$

Intuitively this assertion specifies that an array a stores the linked list, and that l is reachable from f by repeated dereferencing of field $next$.

[6] This set is sometimes called the first-order theory of a structure

Using an auxiliary array b to store the new linked list, we can now express that if this reachability property was initially true then it holds again after executing $enqueue(v)$:

$\{\exists a, n : \text{reach}(first, last, a, n)\}$
z := new; z.next := first ; z.val := v; first := z
b := new; b[0] := first ; i := 0;
while b[i] $\neq$ last **do** b[i+1] := b[i].next; i := i+1 **od**;
$\{\exists a, n : \text{reach}(first, last, a, n)\}$

Strictly speaking this is a property of this particular instrumented version of $enqueue(v)$: the original version does not even have the auxiliary array b. However as the above instrumentation is faithful to the original version, it follows that $last$ is reachable from $first$ in the original program (by repeated dereferencing of $next$), for otherwise $\exists a, n : \text{reach}(first, last, a, n)$ would not hold in *any* faithful instrumentation.

As our semantics are based on *abstract object creation* (not yet created objects play no role in structures, and are not referable in assertions), a corresponding proof theory is needed to verify the above instrumented program. Based on [1], we have extended their approach to support dynamically created arrays. The new rules are fully implemented in a special version of KeY, available at http://keyaoc.hats-project.eu. Only one interaction with KeY is required to verify the specified reachability property for method $enqueue(v)$, namely the provision of a loop invariant, everything else was fully automatic.

6 Conclusions

Scope of the Programming Language. We want to stress that our core language contains all necessary primitive constructs from which more intricate features can be handled by a completely mechanical translation. The features to which this transformational approach applies include failures and *bounded* arrays. Inheritance and dynamic binding have been addressed in [3]. These transformations allow us to treat object creation orthogonally to such features, and thereby indicates our approach scales up to modern languages.

Expressibility of Weakest Precondition. General results of Olderog [13] show there is a certain symmetry between the expressibility of strongest postconditions and weakest preconditions. To prove the result, Olderog makes a constant domain assumption which requires a different modeling of object creation than in our case where we support *abstract* object creation. Hence one cannot in general refer to the final values of the variables in assertions evaluated in an initial state. Consequently Olderog's result does not apply here: object creation breaks the symmetry between strongest postconditions and weakest preconditions.

Recursive Assertions. Apt et al. [2] prove that even for recursive preconditions and postconditions, the intermediate assertions cannot be chosen recursively for general programs, but only for a class of suitably instrumented programs. Our assertions defining reachability are currently not recursive due to unbounded quantification over array indices (see Section 4.2). However, if we restrict to bounded arrays then we only need *bounded* quantification in the expression of the reachability predicates. The trade-off is a significantly more complicated instrumentation as at each computation step a reallocation of the auxiliary (array) variables becomes necessary.

References

1. W. Ahrendt, F. S. de Boer, and I. Grabe. Abstract object creation in dynamic logic. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *LNCS*, pages 612–627. Springer, 2009.
2. K. R. Apt, J. A. Bergstra, and L. G. L. T. Meertens. Recursive assertions are not enough - or are they? *Theor. Comput. Sci.*, 8:73–87, 1979.
3. K. R. Apt, F. S. de Boer, E.-R. Olderog, and S. de Gouw. Verification of object-oriented programs: A transformational approach. *JCSS*, 78(3):823 – 852, 2012.
4. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
5. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
6. J. de Bakker and L. Meertens. On the completeness of the inductive assertion method. *Journal of Computer and System Sciences*, 11(3):323 – 357, 1975.
7. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symposia in Applied Mathematics*, pages 19–32. AMS, 1967.
8. G. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Univ. of Toronto, 1975.
9. D. Harel. Arithmetical completeness in logics of programs. In G. Ausiello and C. Böhm, editors, *ICALP*, volume 62 of *LNCS*, pages 268–288. Springer, 1978.
10. P. Lindström. On extensions of elementary logic. *Theoria*, 35(1):1–11, 1969.
11. Z. Manna. Mathematical theory of partial correctness. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 252–269. Springer, 1971.
12. J. McCarthy. Towards a mathematical science of computation. In *IFIP*, pages 21–28. North-Holland, 1962.
13. E.-R. Olderog. On the notion of expressiveness and the rule of adaption. *Theor. Comput. Sci.*, 24:337–347, 1983.
14. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.
15. C. Pierik. *Validation Techniques for Object-Oriented Proof Outlines.* PhD thesis, Universiteit Utrecht, 2006.
16. M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes Rendus du I congrs de Mathmaticiens des Pays Slaves*, pages 92–101, 1929.
17. N. Suzuki and D. Jefferson. Verification decidability of presburger array programs. *J. ACM*, 27(1):191–205, Jan. 1980.
18. J. Tucker and J. Zucker. *Program correctness over abstract data types, with error-state semantics.* Elsevier Science Inc., 1988.