Artificial Intelligence 235 (2016) 1–25

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

A formalization of programs in first-order logic with a discrete linear order

Fangzhen Lin

Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

ARTICLE INFO

Article history:
Received 17 December 2014
Received in revised form 23 August 2015
Accepted 28 January 2016
Available online 11 February 2016

Keywords:
Program semantics
Reasoning about programs
First-order logic

ABSTRACT

We consider the problem of representing and reasoning about computer programs, and propose a translation from a core procedural iterative programming language to first-order logic with quantification over the domain of natural numbers that includes the usual successor function and the "less than" linear order, essentially a first-order logic with a discrete linear order. Unlike Hoare's logic, our approach does not rely on loop invariants. Unlike the typical temporal logic specification of a program, our translation does not require a transition system model of the program, and is compositional on the structures of the program. Some non-trivial examples are given to show the effectiveness of our translation for proving properties of programs.

1. Introduction

In computer science, how to represent and reason about computer programs effectively has been a major concern since the beginning. For imperative, non-concurrent programs that we are considering here, notable approaches include Dijkstra's calculus of weakest preconditions [1,2], Hoare's logic [3], dynamic logic [4], and separation logic [5]. For the most part, these logics provide rules for proving assertions about programs. In particular, for proving assertions about iterative loops, these logics rely on what have been known as Hoare's loop invariants. In this paper, we propose a way to translate a program to a first-order theory with quantification over natural numbers. The properties that we need about natural numbers are that they have a smallest element (zero), are linearly ordered, and each of them has a successor (plus one). Thus we are essentially using first-order logic with a predefined discrete linear order. This logic is closely related to linear temporal logic, which is a main formalism for specifying concurrent programs [6].

Given a program, we translate it to a first-order theory that captures the relationship between the input and output values of the program variables, independent of what one may want to prove about the program. For instance, trivially, the following assignment

```
X = X+Y
```

can be captured by the following two axioms:

$$X' = X + Y,$$
$$Y' = Y,$$

E-mail address: flin@cs.ust.hk.

http://dx.doi.org/10.1016/j.artint.2016.01.014
0004-3702/© 2016 Elsevier B.V. All rights reserved.

where X and Y denote the initial values of the corresponding program variables and X' and Y' their values after the statement is performed. Obviously, the question is how the same can be done for loops. This is where quantification over natural numbers comes in. Consider the following while loop

```
 while X < M do { X = X+1 }
```

It can be captured by the following set of axioms:

$$
\begin{aligned}
&M' = M,\\
&X \geq M \rightarrow X' = X,\\
&X < M \rightarrow X' = X(N),\\
&X(0) = X,\\
&\forall n.X(n+1) = X(n) + 1,\\
&X(N) \geq M,\\
&\forall n.n < N \rightarrow X(n) < M,
\end{aligned}
$$

where N is a natural number constant denoting the total number of iterations that the loop runs to termination, and $X(n)$ the value of X after the nth iteration. Thus the third axiom says that if the program enters the loop, then the output value of the program variable X, denoted by X', is $X(N)$, the value of X when the loop exits.

The purpose of this paper is to describe how this set of axioms can be systematically generated, and show by some examples how reasoning can be done with this set of axioms. Without going into details, one can already see that unlike Hoare's logic, our axiomatization does not make use of loop invariants. One can also see that unlike typical temporal logic specification of a program, we do not need a transition system model of the program, and do not need to keep track of program execution traces. We will discuss related work in more detail later.

2. Preliminaries

We use a typed first-order language. We assume a type for natural numbers (non-negative integers). Depending on the programs, other types such as integers may be used. For natural numbers, we use constant 0, linear ordering relation $<$ (and $\leq$), successor function $n+1$, and predecessor function $n-1$. We follow the convention in logic to use lower case letters, possibly with subscripts, for logical variables. In particular, we use m and n for natural number variables, and x, y, and z for generic variables. The variables in a program will be treated as functions in logic, and written as either upper case letters or strings of letters.

We use the following shorthands. The conditional expression:

$$e_1 = \text{ if } \varphi \text{ then } e_2 \text{ else } e_3$$

is a shorthand for the conjunction of the following two sentences:

$$
\begin{aligned}
&\forall \vec{x}.\varphi \rightarrow e_1 = e_2,\\
&\forall \vec{x}.\neg\varphi \rightarrow e_1 = e_3,
\end{aligned}
$$

where $\vec{x}$ are all the free variables in φ and e_i, $i = 1, 2, 3$. Typically, all free variables in φ occur in e_1.

Our most important shorthand is the following expression which says that e is the smallest natural number that satisfies $\varphi(n)$:

$$smallest(e, n, \varphi)$$

is a shorthand for the following formula:

$$\varphi(n/e) \wedge \forall m.m < e \rightarrow \neg\varphi(n/m),$$

where n is a natural number variable in φ, m a new natural number variable not in e or φ, $\varphi(n/e)$ the result of replacing n in φ by e, similarly for $\varphi(n/m)$. For example, $smallest(M, k, k < N \wedge found(k))$ says that M is the smallest natural number such that $M < N \wedge found(M)$:

$$M < N \wedge found(M) \wedge \forall n.n < M \rightarrow \neg(n < N \wedge found(n)).$$

Finally, we use the convention that free variables in a displayed sentence are implicitly universally quantified from outside. For instance, the following displayed formula

$$n < M \rightarrow \neg(n < N \wedge found(n))$$

stands for $\forall n.n < M \rightarrow \neg(n < N \wedge found(n))$, where the universal quantification is over the domain of natural numbers as n is a natural number variable. Notice however, in the macro $smallest(M, k, k < N \wedge found(k))$, k is not a free variable.

The following are some useful properties about the smallest macro.

Proposition 1. *Let $\vec{x}$ be the free variables other than n in $\varphi(n)$, and m a variable not in $\varphi(n)$. We have that*

$$\forall\vec{x}.\exists n\varphi(n) \rightarrow \exists m.smallest(m, n, \varphi(n)).$$

Proposition 2. *Let $\vec{x}$ be the free variables other than n in $\varphi(n)$, and m a variable not in $\varphi(n)$. We have that*

$$\forall\vec{x}, m.[smallest(m, n, \varphi(n)) \wedge m > 0] \rightarrow [\varphi(m) \wedge \neg\varphi(m-1)].$$

Furthermore,

$$\forall\vec{x}\{\exists n[(\forall k.k > n \rightarrow \varphi(k)) \wedge (\forall k.k \leq n \rightarrow \neg\varphi(k))] \rightarrow \\ \forall m[smallest(m, n, \varphi(n)) \equiv \varphi(m) \wedge \neg\varphi(m-1)]\},$$

where k is a variable not in $\varphi(n)$.

Proof. For any given $\vec{x}$ and m, suppose $smallest(m, n, \varphi(n)) \wedge m > 0$. Then $\varphi(m)$ and $\forall k.k < m \rightarrow \neg\varphi(k)$. Since $m > 0$, thus $\neg\varphi(m-1)$. Now suppose that $\varphi(m) \wedge \neg\varphi(m-1)$ and for some M,

$$[(\forall k.k > M \rightarrow \varphi(k)) \wedge (\forall k.k \leq M \rightarrow \neg\varphi(k))].$$

This means that $M = m - 1$, and $smallest(m, n, \varphi(n))$. □

To motivate our next proposition, consider again the following loop

```
while X < M do { X = X+1 }
```

Given input M, the number $N(M)$ of the iterations for this loop to exit is captured by the formula $smallest(N(M), n, \neg X(n) < M)$. It can be seen that $N(M+1) = N(M) + 1$, i.e. the number of iterations before the loop exits on input $M+1$ is one more than that of on input M. This can be proved using the following proposition.

Proposition 3. *Let $\vec{x}$ be the free variables other than n in $\varphi_1(n)$ and $\varphi_2(n)$, and m_1, m_2, k and t variables not in φ_1 or φ_2. We have*

$$\forall\vec{x}, t, m_1, m_2.[\forall k(\varphi_1(k) \equiv \varphi_2(k+t)) \wedge \forall k(k < t \rightarrow \neg\varphi_2(k)) \wedge \\ smallest(m_1, n, \varphi_1(n)) \wedge smallest(m_2, n, \varphi_2(n))] \rightarrow \\ m_2 = m_1 + t$$

3. A simple class of programs

Consider the following simple class of programs P constructed from a set of array identifiers array, a set of functions operator, and a set of Boolean operators boolean-op:

```
E ::= array(E,...,E) |
      operator(E,...,E)
B ::= E = E |
      boolean-op(B,...,B)
P ::= array(E,...,E) = E |
      if B then P else P |
      P; P |
      while B do P
```

Here E denotes expressions, B boolean expressions, and P programs. Notice that instead of, for example "array[i][j]" commonly used in programming languages to refer to an array element, we use the notation "array(i,j)" more commonly used in mathematics and logic.

As one can see, programs here are constructed using assignments, sequences, if-then-else, and while loops. Other constructs such as if-then and for-loop can be defined using these constructs. For instance, "if B then P" can be defined as "if B then P else X = X".

We assume a base first-order language $\mathcal{L}$ that contains functions and predicates that are static in the sense that their semantics are fixed and cannot be changed by programs. They include functions that correspond to operator, predicates that correspond to `boolean-op`, and possibly other functions and predicates for formalizing the domain knowledge. In the following, we call $\mathcal{L}$ the base language.

Given a program P, we extend the base language $\mathcal{L}$ by functions to represent program variables in the program. These functions are dynamic in that their values may be changed during the execution of a program. We assume that program variables are new, not already used in $\mathcal{L}$. We also assume that there is no overloading so that two different program variables cannot have the same name but different arities. Thus we can use the same program variables as functions in our first-order language. Specifically, if V is a program variable for an n-ary array, then we add V and V' as new n-ary functions to $\mathcal{L}$: $V(x_1, ..., x_n)$ and $V'(x_1, ..., x_n)$ denote the values of the $(x_1, ..., x_n)$th cell in V at the input and the output, respectively, of the program P. For their values during the execution of P, we'll introduce temporary function symbols to denote them. These temporary function symbols can be systematically named using statement labels (see Section 6 below) and are useful when one is interested about properties that hold during the execution of a program. For now, we assume that we are interested only in the program outputs.

Given a program P and a set $\vec{X}$ of program variables including all variables used in P, we define inductively the set of axioms for P and $\vec{X}$, written $\Pi_P^{\vec{X}}$, as follows:

- If P is

  ```
  V(E1,...,Ek) = E
  ```

  then $\Pi_P^{\vec{X}}$ consists of following axioms that say that only the value of `V(E1,...,Ek)` is possibly changed:

  $$\begin{aligned} V'(\vec{x}) = {} & \text{if } (x_1 = E_1 \wedge \cdots \wedge x_k = E_k) \text{ then } E \\ & \text{else } V(\vec{x}), \\ X'(\vec{y}) = {} & X(\vec{y}), \quad X \in \vec{X} \text{ and } X \text{ different from } V \end{aligned}$$

  where $\vec{x} = (x_1, ..., x_k)$, and k is the arity of the program variable (array) V. We assume here that for each program expression `E`, there is a corresponding term E in our first-order language. Recall that by our convention, these variables are universally quantified. The domains of these variables depend on the type of the program variable V.
- If P is

  ```
  if B then P1 else P2
  ```

  then $\Pi_P^{\vec{X}}$ is constructed from $\Pi_{P_1}^{\vec{X}}$ and $\Pi_{P_2}^{\vec{X}}$ as follows:

  $$B \to \varphi, \text{ for each } \varphi \in \Pi_{P_1}^{\vec{X}},$$
  $$\neg B \to \varphi, \text{ for each } \varphi \in \Pi_{P_2}^{\vec{X}}.$$

  We assume here that for each boolean expression `B`, there is a corresponding formula B in our first-order language.
- If P is

  ```
  P1; P2
  ```

  then $\Pi_P^{\vec{X}}$ is constructed from $\Pi_{P_1}^{\vec{X}}$ and $\Pi_{P_2}^{\vec{X}}$ by connecting the outputs of P_1 with the inputs of P_2 as follows:

  $$\varphi(\vec{X}'/\vec{Y}), \text{ for each } \varphi \in \Pi_{P_1}^{\vec{X}},$$
  $$\varphi(\vec{X}/\vec{Y}), \text{ for each } \varphi \in \Pi_{P_2}^{\vec{X}},$$

  where $\vec{Y} = (Y_1, ..., Y_k)$ is a tuple of new function symbols such that each Y_i is of the same arity as X_i in $\vec{X}$, $\varphi(\vec{X}'/\vec{Y})$ is the result of replacing in φ each occurrence of X'_i by Y_i, and similarly for $\varphi(\vec{X}/\vec{Y})$. The new function symbols in $\vec{Y}$ are called temporary functions and used to denote the values of program variables during the execution of the program. By our inductive construction, $\Pi_{P_1}^{\vec{X}}$ and $\Pi_{P_2}^{\vec{X}}$ may already have some temporary function symbols introduced this way. Furthermore, if P_1 and/or P_2 have loops, then they may also have some new natural number constants (see below for how axioms are constructed for while loops). By renaming if necessary, we assume here that $\Pi_{P_1}^{\vec{X}}$ and $\Pi_{P_2}^{\vec{X}}$ do not share any of these temporary symbols. In other words, we assume that $\Pi_{P_1}^{\vec{X}}$ and $\Pi_{P_2}^{\vec{X}}$ have only common symbols from $\mathcal{L} \cup \vec{X} \cup \vec{X}'$.
- If P is

  ```
  while B do P1
  ```

Then $\Pi_P^{\vec{X}}$ is constructed by adding an index parameter n to all dynamic functions in $\Pi_{P_1}^{\vec{X}}$ to record their values after the body P_1 has been executed n times. Formally, it consists of the following axioms:

$$\varphi[n], \text{ for each } \varphi \in \Pi_{P_1}^{\vec{X}},$$

$$X_i(\vec{x}) = X_i(\vec{x}, 0), \text{ for each } X_i \in \vec{X}$$

$$smallest(N, n, \neg B[n]),$$

$$X_i'(\vec{x}) = X_i(\vec{x}, N), \text{ for each } X_i \in \vec{X}$$

where n is a new natural number variable not already in φ, and N a new natural number constant not already used in $\Pi_{P_1}^{\vec{X}}$ and for each formula or term α, $\alpha[n]$ denotes the value of α after the body P_1 has been executed n times, and is obtained from α as follows:

1. $(\exists x\alpha)[n]$ is $\exists x(\alpha[n])$, $(\alpha_1 \vee \alpha_2)[n]$ is $\alpha_1[n] \vee \alpha_2[n]$, and $(\neg\alpha)[n]$ is $\neg(\alpha[n])$.
2. $F(e_1, ..., e_k)[n]$ is $F(e_1[n], ..., e_k[n])$ if F is a predicate or a function in the base first-order language $\mathcal{L}$. In particular, $(e_1 = e_2)[n]$ is $e_1[n] = e_2[n]$.
3. $X_i'(e_1, ..., e_k)[n]$ is $X_i(e_1[n], ..., e_k[n], n+1)$, if X_i is in $\vec{X}$.
4. $V(e_1, ..., e_k)[n]$ is $V(e_1[n], ..., e_k[n], n)$, if V is a non-primed function not in $\mathcal{L}$.

While we have used $\Pi_P^{\vec{X}}$ to denote "the" set of axioms for P and $\vec{X}$, the construction above does not yield a unique set of axioms as the temporary functions introduced when constructing axioms for program sequences and while-loops are not unique. However, $\Pi_P^{\vec{X}}$ is unique up to the renaming of these new functions. In particular, any two different sets of these axioms are logically equivalent when considering only program variables from $\vec{X}$, i.e. when the temporary functions are "forgotten". More precisely, given two theories Σ_1 and Σ_2, we say that they are equivalent when considering a subset Ω of their vocabularies if any model M_1 of Σ_1 can be modified into a model M_2 of Σ_2 such that M_1 and M_2 agree on Ω, and conversely any model of Σ_2 can be similarly modified into a model of Σ_1.

Appendix A proves the correctness of $\Pi_P^{\vec{X}}$ under an operational semantics. In the following, we give some simple properties about our axiomatization.

The following proposition says that a program is local, in that it has effects only on variables in it.

Proposition 4. *Let $\vec{Y}$ be a tuple of program variables that are not in P and not used in $\Pi_P^{\vec{X}}$. Then considering only $\vec{X} \cup \vec{Y}$, $\Pi_P^{\vec{X} \cup \vec{Y}}$ is equivalent to the union of $\Pi_P^{\vec{X}}$ and the set of following "frame axioms":*

$$Y'(\vec{y}) = Y(\vec{y}), \textit{ for each } Y \in \vec{Y}$$

The construction rule for a sequence $P; Q$ can also be modified so that temporary functions only need to be introduced for those that occur in both P and Q.

Proposition 5. *Let $\vec{X}$ be a tuple of program variables including those used in either P or Q, and $\vec{V} = (V_1, ..., V_k)$ the tuple of program variables used in both P and Q (thus a subset of $\vec{X}$). When considering only $\vec{X}$, $\Pi_{P;Q}^{\vec{X}}$ is equivalent to the set of following axioms:*

$$\varphi(\vec{V}'/\vec{Y}), \textit{ for each } \varphi \in \Pi_P^{\vec{X}},$$
$$\varphi(\vec{V}/\vec{Y}), \textit{ for each } \varphi \in \Pi_Q^{\vec{X}},$$

where $\vec{Y} = (Y_1, ..., Y_k)$ is a tuple of temporary functions such that each Y_i is of the same arity as V_i in $\vec{V}$. Again we assume that, by renaming if necessary, $\Pi_P^{\vec{X}}$ and $\Pi_Q^{\vec{X}}$ have no common function symbols other than those in $\vec{X}$ or in the base language $\mathcal{L}$.

The following important property about our axiomatization says that we do not need to wait until we have the full set of axioms to do simplification. During the construction of the axioms for a program, we can simplify first the axioms for its subprograms. This greatly simplifies the above recursive procedure for constructing axioms of a program.

Proposition 6. *Let $\vec{X}$ be a tuple of program variables, including all those that occur in program P. For any subprogram P', if T is equivalent to $\Pi_{P'}^{\vec{X}}$ when considering only $\vec{X}$, then if we use T instead of $\Pi_{P'}^{\vec{X}}$ in computing $\Pi_P^{\vec{X}}$, the resulting theory is equivalent to $\Pi_P^{\vec{X}}$ when considering only $\vec{X}$ as well.*

Notice that in the above proposition, when we use T instead of $\Pi_{P'}^{\vec{X}}$ in computing $\Pi_P^{\vec{X}}$, we assume that we will also rename temporary function symbols when necessary to avoid name conflicts. For example, if P is $P_1; P_2$, and a theory equivalent to $\Pi_{P_1}^{\vec{X}}$ is

$$X' = Y \wedge Y = X + 1. \qquad (1)$$

If $\Pi^{\vec{X}}_{P_2}$ also uses the temporary function symbol Y, then we need to rename either the Y in (1) or the Y in $\Pi^{\vec{X}}_{P_2}$ when constructing $\Pi^{\vec{X}}_{P}$.

Before we consider more interesting examples, we illustrate our construction of $\Pi^{\vec{X}}_{P}$ using two simple programs.

3.1. A simple sequence

Consider the following program P and two program variables X_1 and X_2 (notice that X_1 is used in P, but X_2 is not):

```
X1 = 1; X1 = X1+1
```

$\Pi^{(X_1,X_2)}_{X1=1}$ is the set of the following two sentences

$$X'_1 = 1,$$
$$X'_2 = X_2$$

and $\Pi^{(X_1,X_2)}_{X1=X1+1}$ the set of following two sentences:

$$X'_1 = X_1 + 1,$$
$$X'_2 = X_2$$

Thus $\Pi^{(X_1,X_2)}_{P}$ is

$$Y_1 = 1,$$
$$Y_2 = X_2,$$
$$X'_1 = Y_1 + 1,$$
$$X'_2 = Y_2$$

Eliminating the temporary constants Y_1 and Y_2, we get $X'_1 = 2$ and $X'_2 = X_2$.

3.2. A simple loop

Consider the following program P with a simple loop.

```
while I < N do
  if X < A(I) then X = A(I);
  I = I+1
```

Notice that the program variables are X, A, I, and N. Among them, A is unary (a list), and the rest are 0-ary (constants).

Let P_1 be the body of the loop. $\Pi^{(X,A,I,N)}_{P_1}$ is equivalent to the set of following sentences (up to the choice of temporary names Y_1, Y_2, Y_3, Y_4):

$$Y_1 = \text{ if } X < A(I) \text{ then } A(I) \text{ else } X,$$
$$Y_2(x) = \text{ if } X < A(I) \text{ then } A(x) \text{ else } A(x),$$
$$Y_3 = \text{ if } X < A(I) \text{ then } I \text{ else } I,$$
$$Y_4 = \text{ if } X < A(I) \text{ then } N \text{ else } N,$$
$$X' = Y_1,$$
$$A'(x) = Y_2(x),$$
$$I' = Y_3 + 1,$$
$$N' = Y_4.$$

Instead of using this set to compute $\Pi^{(X,A,I,N)}_{P}$, by Proposition 6, we can simplify it first by eliminating Y_1, Y_2, Y_3, Y_4, and get the following equivalent set of axioms:

$$\begin{aligned}
&X' = \text{ if } X < A(I) \text{ then } A(I) \text{ else } X,\\
&A'(x) = A(x),\\
&I' = I + 1,\\
&N' = N.
\end{aligned}$$

Thus $\Pi_P^{(X,A,I,N)}$ is

$$\begin{aligned}
&X(0) = X,\\
&A(x,0) = A(x),\\
&I(0) = I,\\
&N(0) = N,\\
&X(n+1) = \text{ if } X(n) < A(I(n),n) \text{ then } A(I(n),n)\\
&\qquad\quad \text{else } X(n),\\
&A(x,n+1) = A(x,n),\\
&I(n+1) = I(n) + 1,\\
&N(n+1) = N(n),\\
&\mathit{smallest}(M, n, \neg I(n) < N(n)),\\
&X' = X(M),\\
&A'(x) = A(x, M),\\
&I' = I(M),\\
&N' = N(M).
\end{aligned}$$

Clearly $A(x)$ and N do not change: $A(x,n) = A(x)$ and $N(n) = N$. So we get the following sentences by expanding the *smallest* macro:

$$\begin{aligned}
&X(0) = X,\\
&I(0) = I,\\
&X(n+1) = \text{ if } X(n) < A(I(n)) \text{ then } A(I(n))\\
&\qquad\quad \text{else } X(n),\\
&I(n+1) = I(n) + 1,\\
&I(M) \geq N,\\
&n < M \rightarrow I(n) < N,\\
&X' = X(M),\\
&A'(x) = A(x),\\
&I' = I(M),\\
&N' = N.
\end{aligned}$$

Now suppose that initially $I = 0$. Solving the recurrence:

$$\begin{aligned}
&I(0) = 0,\\
&I(n+1) = I(n) + 1
\end{aligned}$$

we have $I(n) = n$. Thus we have

$$\begin{aligned}
&M \geq N,\\
&n < M \rightarrow n < N,
\end{aligned}$$

which imply that $M = N$. So we can eliminate $I(n)$ and M and get the following axioms:

$$X(0) = X,$$

$$X(n+1) = \text{ if } X(n) < A(n) \text{ then } A(n)$$

$$\text{else } X(n),$$

$$X' = X(N),$$

$$A'(x) = A(x),$$

$$I' = N,$$

$$N' = N.$$

An example assertion to prove about the program is the following

$$0 \leq n < N \rightarrow X' \geq A(n), \tag{2}$$

which is equivalent to

$$0 \leq n < N \rightarrow X(N) \geq A(n),$$

which can be proved by induction on N. The base case of $N = 0$ is trivial. For the inductive case, suppose the result holds for $N = K$. Let $N = K + 1$. There are two cases to consider: $X(K) < A(K)$ and $X(K) \geq A(K)$. We show the first case here. The second case is similar. In the first case, $X(K + 1) = A(K)$ and we need to show that

$$0 \leq n < K + 1 \rightarrow A(K) \geq A(n).$$

Two cases for $0 \leq n < K + 1$: $0 \leq n < K$ or $n = K$. The second case is trivial. For the first case, $A(K) \geq A(n)$ follows from the inductive assumption and that $X(K) < A(K)$.

3.3. Partial and total correctness

A program is partially correct w.r.t. a specification if the program satisfies the specification when it terminates. It is totally correct if it is partially correct and terminates.

In our framework, a program P with variables $\vec{X}$ is represented by a set of sentences, $\Pi_P^{\vec{X}}$. Whatever properties that one wants to show about P are proved using this set of sentences. A partial correctness result corresponds to proving a sentence about $\vec{X}$ and $\vec{X}'$ from $\Pi_P^{\vec{X}}$. An example is the assertion (2) above for the simple loop. On the other hand, termination of a program is proved by showing that the new natural number constants introduced by the loops and used in the *smallest* macro expressions are well-defined, which in logic means that the resulting theory $\Pi_P^{\vec{X}}$ is consistent, thus there is a model where the new constants are mapped to natural numbers. For instance, for the above simple loop, the smallest macro is $smallest(M, n, \neg I(n) < N(n))$. By Proposition 1 and the fact that $I(N) \geq N$ holds, it can be verified that the theory is consistent because there is indeed a natural number M that satisfies this macro expression.

If a loop does not terminate, then its smallest macro will cause a contradiction. For instance, consider the following loop:

```
while I < M do
  if I>0 then I = I+1.
```

If initially $I = 0$ and $M > 0$, then it will loop forever. Our axioms for the loop are:

$$I' = I(N) \wedge M' = M,$$

$$I(0) = I,$$

$$I(n+1) = \text{ if } I(n) > 0 \text{ then } I(n) + 1 \text{ else } I(n),$$

$$n < N \rightarrow I(n) < M,$$

$$I(N) \geq M.$$

If we add $I = 0 \wedge 0 < M$ to these axioms, we will conclude $\forall n.I(n) < M$, which contradicts the last axiom $I(N) \geq M$. Of course in logic, this also means that the axioms for the loops will entail that $\neg(I = 0 \wedge 0 < M)$, which can be taken as a pre-condition of the loop.

4. Related work

Our formalization of the simple loop above also illustrates the difference between our approach and Hoare's logic, arguably the dominant approach for reasoning about non-parallel imperative computer programs. To begin with, an assertion like (2) would be represented by a triple like

$$\{I = 0\}P\{\forall m(0 \le m < N \rightarrow X \ge A(m))\}$$

in Hoare's logic. To prove this assertion, one would need to find a suitable "loop invariant", a formula that if true initially will continue to be true after each iteration. In general, there are infinite number of such loop invariants. The key is to find one that, in conjunction with the negation of the loop condition, can entail the postcondition in the assertion. For this simple loop, the following is such a loop invariant:

$$\forall m(I_0 \le m < I \rightarrow X \ge A(m)).$$

Finding suitable loop invariants is at the heart of Hoare's logic, and it is not surprising that there has been much work on discovering loop invariants (e.g. [7–11]).

In comparison, our proof of (2) uses ordinary mathematical induction and recurrences on $I(n)$ and $X(n)$. See Appendix B for more details.

Another difference between our approach and Hoare's logic is that Hoare's logic is a set of general rules about program assertions, while we provide a translation from programs to first-order theories with quantification over natural numbers. Once the translation is done, assertions about it are proved with respect to the translated first-order theory, without reference to the original program. This is similar to Pnueli's temporal logic approach to program semantics [6]. According to a common classification used in the formal methods community (cf. [12,13]): approaches like Hoare's logic and dynamic logic are *exogenous* in that they have programs explicitly in the language, while in the temporal logic approach, program semantics is typically *endogenous* in that a fixed program is often assumed and a program execution counter is part of the specification language. Our approach is certainly not exogenous. It is a little endogenous as we use natural numbers to keep track of loop iterations, but not as endogenous as typical temporal logic specifications which requires program counters to be part of states. In particular, our mapping from programs to theories is compositional, built up from the structure of the program. Barringer et al. [14] proposed a compositional approach using temporal logic, but only in the style of Hoare's logic, using Hoare triples. However, a caveat is that so far the temporal logic approach to program semantics have been mainly for concurrent programs, while what we have proposed is for non-parallel programs. Given the close relationship between temporal logics and first-order logic with a linear order, if there are no nested loops, then our translation can be reformulated in a temporal logic. It is hard to see how this can be done when there are nested loops, as this will lead to nested time lines, modeled here by predicates with multiple natural number arguments. Of course, one can always construct a transition graph of a program, and model it in a temporal logic. But then the structure of the original program is lost.

We are certainly not the first to use first-order logic with a linear order to model dynamic systems. For instance, it has been used to model Turing machines in the proof of Trakhtenbrot's theorem in finite model theory (see, e.g. [15]).

A closely related work is Charguéraud's characteristic formulas for functional programs [16,17]. However, these formulas are higher-order formulas that reformulate Hoare's rules by quantifying over preconditions and postconditions.

Our use of natural numbers as "indices" to model iterations is similar to Wallace's use of natural numbers to model rule applications in his semantics for logic programs [18].

While we use natural numbers to formalize loops, Levesque et al. [19] used second-order logic to capture Golog programs with loops in the situation calculus. Recently, Lin [20] showed that under the foundational axioms of the situation calculus, Golog programs can be defined in first-order logic as well. However, the crucial difference between the work here and the work in the situation calculus on Golog is that our axioms try to capture the changes of states in terms of values of program variables, while the semantics of Golog programs is more about defining legal sequences of executions. To illustrate the difference here, consider a program that consists of assignments that make no change (*nil* actions). For this program, it would still be non-trivial to define sequences of legal executions, although it does not matter which sequences are legal as none of them change the values of program variables. Another difference is that we consider only assignments and deterministic programs, while Golog programs allow any actions that can be axiomatized by successor state axioms, and can have nondeterministic choices.

5. Cohen's integer division algorithm

For a more complex example, consider the following program P which implements the well-known Cohen's integer division algorithm [21] (our program below is adapted from [11]). It has two loops, one nested inside another. The program variables are A, B, Q, R, X, Y, where X and Y are inputs, and Q is the output. Let $\vec{X} = (A, B, Q, R, X, Y)$. There are two loops. Let's name the inner loop *Inner*, and outer loop *Outer*. When computing $\Pi_P^{\vec{X}}$, we again consider only equivalence under $\vec{X}$ and use Proposition 6 to simplify the process.

```
// X and Y are two input integers; Y > 0
  Q=0; // quotient
  R=X; // remainder
  while (R >= Y) do {
    A=1; // A and B are some that at any time for
    B=Y; // some n, A=2^n and B=2^n*Y
    while (R >= 2*B) do {
      A = 2*A;
      B = 2*B;
    }
    R = R-B;
    Q = Q+A
  }
//  return Q = X/Y;
```

It is easy to see that $\Pi_P^{\vec{X}}$ is equivalent to $\Pi_{Outer}^{\vec{X}} \cup \{Q = 0, R = X\}$. To compute $\Pi_{Outer}^{\vec{X}}$, we compute first $\Pi_{Inner}^{\vec{X}}$, which is equivalent to the set of following sentences:

$$
\begin{aligned}
&A(n+1) = 2A(n),\\
&B(n+1) = 2B(n),\\
&Q(n+1) = Q(n),\\
&R(n+1) = R(n),\\
&X(n+1) = X(n),\\
&Y(n+1) = Y(n),\\
&A(0) = A,\\
&B(0) = B,\\
&Q(0) = Q,\\
&R(0) = R,\\
&X(0) = X,\\
&Y(0) = Y,\\
&\mathit{smallest}(N, n, R(n) < 2B(n)),\\
&A' = A(N),\\
&B' = B(N),\\
&Q' = Q(N),\\
&R' = R(N),\\
&X' = X(N),\\
&Y' = Y(N).
\end{aligned}
$$

Solving the recurrences, we have

$$
\begin{aligned}
&A(n) = 2^n A,\\
&B(n) = 2^n B,\\
&Q(n) = Q,\\
&R(n) = R,\\
&X(n) = X,\\
&Y(n) = Y\\
&\mathit{smallest}(N, n, R < 2^{n+1} B),\\
&A' = 2^N A,\\
&B' = 2^N B,
\end{aligned}
$$

$$Q' = Q,$$
$$R' = R,$$
$$X' = X,$$
$$Y' = Y.$$

We can now eliminate terms like $A(n)$ and $B(n)$, expand the smallest macro expression, and obtain $\Pi^{\vec{X}}_{Inner}$ as the set of following sentences:

$$R < 2^{N+1}B,$$
$$m < N \rightarrow R \geq 2^{m+1}B,$$
$$A' = 2^N A,$$
$$B' = 2^N B,$$
$$Q' = Q,$$
$$R' = R,$$
$$X' = X,$$
$$Y' = Y.$$

Thus the set of sentences for the body of the loop *Outer* is equivalent to the set of the following sentences:

$$R < 2^{N+1}Y,$$
$$m < N \rightarrow R \geq 2^{m+1}Y,$$
$$A' = 2^N,$$
$$B' = 2^N Y,$$
$$Q' = Q + A',$$
$$R' = R - B',$$
$$X' = X,$$
$$Y' = Y.$$

Thus $\Pi^{\vec{X}}_{Outer} \cup \{Q = 0, R = X\}$ is equivalent to

$$R(n) < 2^{N(n)+1}Y(n),$$
$$m < N(n) \rightarrow R(n) \geq 2^{m+1}Y(n),$$
$$A(n+1) = 2^{N(n)},$$
$$B(n+1) = 2^{N(n)}Y(n),$$
$$Q(n+1) = Q(n) + 2^{N(n)},$$
$$R(n+1) = R(n) - 2^{N(n)}Y(n),$$
$$X(n+1) = X(n),$$
$$Y(n+1) = Y(n),$$
$$A(0) = A,$$
$$B(0) = B,$$
$$Q(0) = 0,$$
$$R(0) = X,$$
$$X(0) = X,$$
$$Y(0) = Y,$$
$$smallest(M, n, R(n) < Y(n)),$$
$$A' = A(M),$$

$$\begin{aligned}
&B' = B(M),\\
&Q' = Q(M),\\
&R' = R(M),\\
&X' = X(M),\\
&Y' = Y(M).
\end{aligned}$$

Now get rid of $X(n)$ and $Y(n)$ as they do not change: $X(n) = X$ and $Y(n) = Y$, get rid of A and B as they are irrelevant now, and expand the smallest macro expression, we obtain $\Pi_P^{\vec{X}}$ as the set of following sentences:

$$\begin{aligned}
&R(n) < 2^{N(n)+1}Y,\\
&m < N(n) \rightarrow R(n) \geq 2^{m+1}Y,\\
&Q(n+1) = Q(n) + 2^{N(n)},\\
&R(n+1) = R(n) - 2^{N(n)}Y,\\
&Q(0) = 0,\\
&R(0) = X,\\
&R(M) < Y,\\
&m < M \rightarrow R(m) \geq Y,\\
&Q' = Q(M),\\
&R' = R(M).
\end{aligned}$$

From these axioms, we can show the partial correctness of Cohen's algorithm by proving the following two properties, under the precondition that $X \geq 0$ and $Y \geq 1$:

$$\begin{aligned}
&0 \leq R' < Y,\\
&X = Q'Y + R'.
\end{aligned}$$

For the first property, $R' < Y$ trivially follows from the condition of the *Outer* loop. For $R' \geq 0$, we have $R' = R(M) = R(M-1) - 2^{N(M-1)}Y$. By the axiom

$$m < N(n) \rightarrow R(n) \geq 2^{m+1}Y,$$

let $n = M-1$ and $m = N(M-1)-1$, we have $R(M-1) \geq 2^{(N(M-1)-1)+1}Y = 2^{N(M-1)}Y$. Thus $R' \geq 0$. For the second property, we have

$$\begin{aligned}
&Q'Y + R'\\
&= Q(M)Y + R(M)\\
&= (Q(M-1) + 2^{N(M-1)})Y + R(M-1) - 2^{N(M-1)}Y\\
&= Q(M-1)Y + R(M-1)\\
&= \cdots = Q(0)Y + R(0) = X.
\end{aligned}$$

Again this is partial correctness. To prove the termination, we need to show that the new terms introduced by the smallest macro expressions are all well-defined. For this program, it means that M (the outer loop counter) is bounded, and for every n, $N(n)$ (the inner loop counter for each outer loop iteration) is bounded. By Proposition 1, these can be proved by showing the following two properties:

$$\begin{aligned}
&\exists m.R(m) < Y,\\
&\forall n \exists m.R(n) < 2^{m+1}Y.
\end{aligned}$$

Notice that these properties must be proved without those axioms about M and $N(n)$. Since $R(n+1)$ is inductively defined in terms of $N(n)$, we prove the second property by induction on n, thus showing that $N(n)$ is well-defined. Since $R(0) = X$ and $Y > 1$, $\exists m.R(0) < 2^{m+1}Y$ is easy to see: we can let $m = X$. Thus $N(0)$ is well-defined. Inductively, suppose $N(k)$ is well-defined and $\exists m.R(k) < 2^{m+1}Y$. Since $R(k+1) < R(k)$, we have $\exists m.R(k+1) < 2^{m+1}Y$ as well. Thus $N(k+1)$ is well-defined. Now to show the first property $\exists m.R(m) < Y$, observe that $R(m) \leq X - mY$, thus $\exists m.R(m) < 0 < Y$.

It may not seem obvious how properties like these can be proved in general. As we mentioned, logical consistency is what we meant for terms like $N(n)$ to be well-defined. Thus all one needs to show is that the set of axioms is consistent under the assumption that $Y > 0$ and $X \geq 0$. Using Proposition 1 is just one way of showing this: if the axioms that do not mention N are consistent and entail $\exists n.\varphi(n)$, then adding $smallest(N, n, \varphi(n))$ to the axioms will also be consistent.

Again we remark that we relied on mathematical induction in our proof and made no use of loop invariants. Notice also that our proof actually shows that for integer division, any program of the following form is correct:

```
// X and Y are two input integers; Y > 0
  Q=0; // quotient
  R=X; // remainder
  while (R >= Y) do {
    A=1;
    B=Y;
    while (R >= k*B) do {
      A = k*A;
      B = k*B;
    }
    R = R-B;
    Q = Q+A
  }
//  return Q = X/Y;
```

where $k > 1$ can be any constant.

6. Properties of programs during execution

As we mentioned, our proposed translation to first order logic has been tailored for the program behaviors in terms of input and output conditions. Sometimes one may be interested in properties of a program during its execution. We have been using temporary function symbols to denote the values of program variables during the execution of a program. So to reason about properties of a program during its execution, all we need to do is to give these temporary functions explicit names. One way to do this is to label program statements and use these labels as the point of reference. Consider the following class of labeled programs:

```
E ::= array(E,...,E) |
      operator(E,...,E)
B ::= E = E |
      boolean-op(B,...,B)
P ::= L: array(E,...,E) = E |
      L: if B then P else P |
      L: while B do P |
      P; P
```

Here L is a label, typically a natural number. Notice that there is no label in front of a sequence. In general, a program P is a sequence of statements:

$$(L_1 : P_1); (L_2 : P_2); \cdots ; (L_k : P_k)$$

where P_i is either an assignment, a conditional or a while loop. We call P_k the last statement of P, and the output of P is the same as the output of P_k.

Again assume that program variable names are unique and not in the base language $\mathcal{L}$. Now given a program P, for each program variable V and label L, we add functions V and V^L to $\mathcal{L}$. Again, $V(\vec{x})$ denotes the value at the input, where $\vec{x}$ are the indices of the corresponding array. The value at the end of a statement L is now denoted by $V^L(\vec{x})$. Of course, V' is V^L when L is the label of the last statement in the program.

Given a program P and a set $\vec{X}$ of program variables including all variables used in P, we again use $\Pi_P^{\vec{X}}$ to denote the set of axioms for P and $\vec{X}$:

- If P is

  ```
  L: V(E1,...,Ek) = E
  ```

  then $\Pi_P^{\vec{X}}$ consists of following axioms:

$$\begin{aligned} V^L(\vec{x}) &= \text{if } (x_1 = E_1 \wedge \cdots \wedge x_k = E_k) \text{ then } E \\ &\quad\ \text{else } V(\vec{x}), \\ X^L(\vec{y}) &= X(\vec{y}), \quad X \in \vec{X} \text{ and } X \text{ different from } V \end{aligned}$$

- If P is

  ```
  L: if B then P1 else P2
  ```

  then $\Pi_P^{\vec{X}}$ is the union of $\Pi_{P_1}^{\vec{X}}$, $\Pi_{P_2}^{\vec{X}}$ and the set of the following axioms: for each $X \in \vec{X}$,

  $$\begin{aligned} &B \rightarrow X^L(\vec{x}) = X^{L_1}(\vec{x}), \\ &B \rightarrow X^L(\vec{x}) = X^{L_2}(\vec{x}), \end{aligned}$$

  where L_1 and L_2 are the labels of the last statements in P_1 and P_2, respectively.
- If P is

  ```
  P1; P2
  ```

  then $\Pi_P^{\vec{X}}$ is the union of $\Pi_{P_1}^{\vec{X}}$ and the set of the following axioms:

  $$\varphi(\vec{X}/\vec{X^{L_1}}), \ \text{for each } \varphi \in \Pi_{P_2}^{\vec{X}},$$

  where L_1 is the label of the last statement in P_1.
- If P is

  ```
  L: while B do P1
  ```

  Then $\Pi_P^{\vec{X}}$ is constructed from $\Pi_{P_1}^{\vec{X}}$ as follows:

  $$\begin{aligned} &\varphi[n], \ \text{for each } \varphi \in \Pi_{P_1}^{\vec{X}}, \\ &X_i^L(\vec{x}, 0) = X_i(\vec{x}), \ \text{for each } X_i \in \vec{X} \\ &X_i^L(\vec{x}, n+1) = X_i^{L_1}(\vec{x}, n), \\ &smallest(N, n, \neg B[n]), \\ &X_i^L(\vec{x}) = X_i^L(\vec{x}, N), \ \text{for each } X_i \in \vec{X} \end{aligned}$$

  where L_1 is the label of the last statement of the loop body P_1, n is a new natural number variable not already in φ, and N a new natural number constant not already used in $\Pi_{P_1}^{\vec{X}}$ and for each formula or term α, $\alpha[n]$ is defined similarly as before:
  1. $(\exists x\alpha)[n]$ is $\exists x(\alpha[n])$, $(\alpha_1 \vee \alpha_2)[n]$ is $\alpha_1[n] \vee \alpha_2[n]$, and $(\neg\alpha)[n]$ is $\neg(\alpha[n])$.
  2. $F(e_1, ..., e_k)[n]$ is $F(e_1[n], ..., e_k[n])$ if F is a predicate or a function in the base first-order language $\mathcal{L}$.
  3. $X(e_1, ..., e_k)[n]$ is $X^L(e_1[n], ..., e_k[n], n)$, for each $X \in \vec{X}$,
  4. for each label t in P_1, $X^t(e_1, ..., e_k)[n]$ is $X^t(e_1[n], ..., e_k[n], n)$, for each $X \in \vec{X}$.

As an example, consider the simple loop in Section 3.2, with labels added:

```
1:   while I < N do
2:     if X < A(I) then
3:        X = A(I);
4:     I = I+1
```

The axioms for the body of the loop are (we ignore $A(x)$ and N as they do not change):

$$\begin{aligned} &X^4 = X^2 \wedge I^4 = I^2 + 1, \\ &X^2 = \text{ if } X < A(I) \text{ then } X^3 \text{ else } X, \\ &I^2 = \text{ if } X < A(I) \text{ then } I^3 \text{ else } I, \\ &X^3 = A(I) \wedge I^3 = I. \end{aligned}$$

Thus the axioms for the program are:

$$X^4(n) = X^2(n),$$

$$I^4(n) = I^2(n) + 1,$$

$$X^2(n) = \text{ if } X^1(n) < A(I^1(n)) \text{ then } X^3(n) \text{ else } X^1(n),$$

$$I^2(n) = \text{ if } X^1(n) < A(I^1(n)) \text{ then } I^3(n) \text{ else } I^1(n),$$

$$X^3(n) = A(I^1(n)),$$

$$I^3(n) = I^1(n),$$

$$X^1(0) = X \wedge I^1(0) = I,$$

$$X^1(n+1) = X^4(n),$$

$$I^1(n+1) = I^4(n),$$

$$n < M \rightarrow I^1(n) < N,$$

$$\neg I^1(M) < N,$$

$$X^1 = X^4(M) \wedge I^1 = I^4(M).$$

This set of axioms looks more complicated, which is natural as it has more information. One could query it about the values of program variables at any point during the execution of the program. For instance, to say that statement 2 does not change the value of I during the execution, we write $\forall n.I^2(n) = I^1(n)$. Notice that $I^1(n)$ denotes the value of I at the beginning of the nth iteration of the loop.

7. Functions

One may ask how general our proposed approach is. Can it be done for programs with more complex structures like pointers, functions, classes, concurrency? We believe so. We have extended it to pointers and functions. Classes should present no problem as they are basically user defined types. We are currently working on extending it to handle Java-like threads. In this section, we describe how the same approach can be used to axiomatize programs with user defined functions. We consider pointers in the next section.

In practice, a program consists of a set of functions. To illustrate how we can handle functions, including recursive functions, consider the following class of programs:

```
E ::= array(E,...,E) |
      operator(E,...,E) |
      function(E,...,E) |
B ::= E = E |
      boolean-op(B,...,B)
Body ::= array(E,...,E) = E |
         if B then P else P |
         P; P |
         while B do P |
         return E
F ::= function(variable,...,variable)
      { Body }
P ::= F | P; P
```

Thus a program is a collection of functions. Presumably, one of them is the "main" function, the one that will be executed first when the program is run. In some programming languages, these functions can communicate by sharing some global variables. To simplify things, we assume here that there are no global variables, and that all program variables in the body of a function must occur in the parameter list of the function.

If P is $F_1; \cdots ; F_k$, then the set of axioms for P is the union of the sets of axioms for F_i, $1 \leq i \leq k$, with renaming of program variables in them if needed to avoid conflict of names.

Given a function definition $f(\vec{X})\{Body\}$, we first capture the return value of the function on input $\vec{X}$ by using a special keyword *Result*. Then the function is defined by universally quantifying over $\vec{X}$. More precisely, the set of sentences for f, written Π_f, consists of the following ones:

$$\forall \vec{x} \varphi(\vec{X}/\vec{x})(Result'/f(\vec{x})), \varphi \in \Pi_{Body}^{\vec{X} \cup \{Result\}},$$

where

- $\varphi(\vec{X}/\vec{x})(Result'/f(\vec{x}))$ is the result of replacing in φ each X_i in $\vec{X}$ by x_i, X'_i by a new function name $g(\vec{x})$, and $Result'$ by $f(\vec{x})$. We assume that *Result* is a reserved word used to denote the value of the function. Notice that once we replace each X_i by a variable x_i, X'_i, the value of X_i when the function exits, is no longer relevant. Here we just replace it by a dummy new function g.
- $\Pi_{Body}^{\vec{X}\cup\{Result\}}$ is defined as before, except that when *Body* is `return E`, the axioms are

$$Result' = E,$$

$$X'_i(\vec{x}) = X_i(\vec{x}),\ X_i \text{ is a program variable.}$$

Notice that according to our axiomatization here, while the body of a function may execute the return statement multiple times, only the last time matters. For example, given

```
foo() { return 1; return 2 }
```

only the second return statement is meaningful because $Result'$ from the first return statement is replaced by a temporary variable when constructing axioms for the sequence, and thus discarded. So the function is captured by the axiom $foo() = 2$. One could argue that it does not make sense for more than one instance of the return statement to be executed, and it is the programmer's responsibility to make sure that this does not happen. Alternatively, one can assume that as soon as a return statement is executed, the function exits. This can be modeled by introducing a special flag *Exit*, and replace each return statement by

```
if -Exit then {return E; Exit = true}
```

For a more meaningful example, consider the following two mutually defined functions *isEven* and *isOdd*:

```
isEven(N) {
  if N=0 then return true
         else return isOdd(N-1) }
isOdd(N) {
  if N=0 then return false
         else return isEven(N-1) }
```

Suppose that we denote the body of *isEven*(N) by *Body*1, and that of *isOdd*(N) by *Body*2. Then $\Pi_{Body1}^{(N,Result)}$ consists of the following axioms:

$$N' = N,$$

$$Result' = \text{ if } N = 0 \text{ then } true \text{ else } isOdd(N-1)$$

and similarly for $\Pi_{Body2}^{(N,Result)}$:

$$N' = N,$$

$$Result' = \text{ if } N = 0 \text{ then } false \text{ else } isEven(N-1)$$

Thus $\Pi_{isOdd} \cup \Pi_{isEven}$ is

$$f(x) = x,$$

$$isEven(x) = \text{ if } x = 0 \text{ then } true \text{ else } isOdd(x-1),$$

$$g(x) = x,$$

$$isOdd(x) = \text{ if } x = 0 \text{ then } false \text{ else } isEven(x-1),$$

where f and g are two new functions used to denote the values of x when the functions *isEven*(x) and *isOdd*(x), respectively, return. They are irrelevant, so the two corresponding axioms can be deleted. By induction on n, it is easy to prove that the following hold for all $n \geq 0$:

$$isEven(2n) = true,$$

$$isOdd(2n) = false,$$

$$isEven(2n+1) = false,$$

$$isOdd(2n+1) = true.$$

Now consider the following program with a type definition:

```
List ::= [] | a::List

length(X:List) {
 if X=[] then return 0
   else return length(tail(X))+1}
tail(X:List) {
 if X=[] then return []
   else if X=a::X1 then return X1}
append(X:List, Y:List) {
 if X=[] then return Y
   else if X=a::X1
         then return a::append(X1,Y)}
```

where `a::List` is list concatenation: the new list is obtained by adding `a` into `List` as the first element.

To model the data type `List`, we introduce a corresponding *List* sort in our first-order language, and write $(x : List)$ to mean that x is of sort *List*. In first-order terms, the definition of *List* yields the following axioms:

$$(\forall x : List).x = [] \vee \exists a(\exists y : List)x = a :: y,$$
$$\forall a, b(\forall x, y : List).a :: x = b :: y \rightarrow (a = b \wedge x = y),$$
$$\forall a(\forall x : List)[] \neq a :: x,$$

and the three functions yield the following axioms:

$$\begin{aligned}&(\forall x : List).length(x) = \text{ if } x = [] \text{ then } 0\\ &\qquad \text{else } length(tail(x)) + 1,\\ &(\forall x : List).tail(x) = \text{ if } x = [] \text{ then } []\\ &\qquad \text{else } \text{ if } \exists a(\exists y : List)x = a :: y \text{ then } y,\\ &(\forall x, y : List).append(x, y) = \text{ if } x = [] \text{ then } y\\ &\qquad \text{else if } \exists a(\exists x_1 : List)x = a :: x_1\\ &\qquad\quad \text{then } a :: append(x1, y).\end{aligned}$$

With these axioms, one can prove, for example $length(a :: b :: []) = 2$. However, they are not sufficient for proving general properties like the following simple one:

$$(\forall x, y : List)length(append(x, y)) = length(x) + length(y).$$

To prove properties like this, we need induction on lists. This can be done by using a second-order axiom on sort *List*, similar to the one on natural numbers. However, since we already have natural numbers, this is not necessary. We can introduce lists of n elements, and define a list to be a list of n elements, for some n. This way, we can use mathematical induction on natural numbers to prove inductive properties about lists. We show how this is done here. We introduce a binary predicate $List(x, n)$, meaning that x is a list with exactly n elements:

$$(\forall x : List)\exists n.List(x, n),$$
$$List(x, 0) \equiv x = [],$$
$$List(x, n + 1) \equiv (\exists a)(\exists y : List).x = a :: y \wedge List(y, n)$$

We first show that if x is a list, then there is a unique n such that $List(x, n)$ holds:

$$List(x, n) \wedge List(x, m) \rightarrow m = n. \tag{3}$$

Suppose x is a list, and $List(x, m)$ and $List(x, n)$ are true. We do simultaneous induction on n and m. If $n = 0$, then $x = []$. If $m \neq 0$, then for some k, $m = k + 1$ and $x = [] = a :: y$ for some a and list y, a contradiction with one of our axioms about lists. Thus $m = 0$ as well. Similarly, if $m = 0$, then $n = 0$ as well. Suppose $n = k_1 + 1$ and $m = k_2 + 1$, and suppose inductively that for any $i, j < max\{m, n\}$, we have that

$$List(y, i) \wedge List(y, j) \rightarrow i = j$$

for any list y. We then have $x = y_1 :: z_1$ for some list z_1 such that $List(z_1, k_1)$ holds, and $x = y_2 :: z_2$ for some list z_2 such that $List(z_2, k_2)$ holds. From $y_1 :: z_1 = y_2 :: z_2$, we have $z_1 = z_2$, thus by the inductive assumption, $k_1 = k_2$. So $m = n$. This concludes the inductive step, thus the proof of (3).

Using (3), we can then prove the induction schema on lists: for any formula $\varphi(x)$,

$$\varphi([]) \wedge \forall a(\forall x : List)(\varphi(x) \rightarrow \varphi(a :: x)) \rightarrow (\forall x : List)\varphi(x).$$

Suppose the premise is true and for some list x, $\neg\varphi(x)$. By (3) there is a unique n such that $List(x, n)$. Suppose x is a shortest such list: for any list y, if $List(y, m) \wedge m < n$, then $\varphi(y)$ holds. If $n = 0$, then $x = []$, which satisfies φ, a contradiction. Suppose $n = m + 1$, then there are some a and y such that $x = a :: y \wedge List(y, m)$. By our assumption about x, $\varphi(y)$ holds. By the premise, $\varphi(a :: y)$ holds as well, a contradiction.

The same idea can be used to axiomatize in first-order logic other inductively defined data structures such as trees.

For recursive functions, a challenge is to distinguish between cycles and undefined values. Consider the following example.

```
foo(X) { if X=0 then return foo(X)
           else if x=1 then return 1}
```

With our axiomatization, the set of axioms for $foo(x)$ is equivalent to a single fact $foo(1) = 1$. It leaves completely open the possible values for $foo(x)$ when $x \neq 1$. One could argue whether this is a right formalization. But operationally, there is a difference between function calls $foo(0)$ and $foo(2)$: calling $foo(0)$ will cause a cycle, but calling foo(2) will terminate without any value being returned. The former causes stack overflow and the latter abnormal exit.

In the following, we provide an axiomatization of functions that can differentiate these two cases. The key idea is to keep a counter of the number of times a recursive function has been called.

Let $f_1, f_2, ..., f_k$ be functions that are mutually defined recursively: $f_i(X_1, ..., X_m)\{B_i\}$. Extend these functions with one more argument:

$$f_i(X_1, ..., X_m, M) \text{ \{if } M = 0 \text{ then } B_{i0} \text{ else } B_{i1}\}$$

where

- B_{i0} is the result of replacing each function call $f_j(T_1, ..., T_m)$ in B_i by $Cycle$, and
- B_{i1} is the result of replacing each function call $f_j(T_1, ..., T_m)$ in B_i by $f_j(T_1, ..., T_m, M - 1)$.
- M is a natural number, and $Cycle$ is a new constant.

The set Π_{f_i} of axioms for f_i is then

$$f_i(\vec{x}) = y \equiv \exists n \forall m \geq n. f_i(\vec{x}, m) = y.$$

This axiomatization is similar to the iterated version of the least fixed-point semantics for recursive functions: B_{i0} is the base case and B_{i1} is the inductive case.

Consider again function $foo()$ defined above. We have

```
foo(X,M) { if M=0 then
  {if X=0 then return Cycle else
    if X=1 then return 1} else
  {if X=0 then return foo(X,M-1) else
    if X=1 then return 1}
```

and the following axioms for $foo(X)$ and $foo(X, M)$:

$$foo(x) = y \equiv \exists m \forall n \geq m. foo(x, n) = y,$$

$$foo(0, 0) = Cycle,$$

$$foo(1, 0) = 1,$$

$$foo(0, n + 1) = foo(0, n),$$

$$foo(1, n + 1) = 1$$

Thus $\forall n. foo(0, n) = Cycle$ and $\forall n. foo(1, n) = 1$. So $foo(0) = Cycle$ and $foo(1) = 1$. The axioms again leave open the possible values for $foo(x)$ when x is not equal to 0 or 1.

8. Pointers

To illustrate how this approach can handle pointers and reference variables, we consider here a language with some simple pointer operations similar to those in C. In particular, for a program variable `X`, we use `&X` to refer to the address of the memory location assigned to `X`, and for a pointer variable `L`, use `#L` to refer to the value in the location pointed to by `L`. Thus for a variable `X`, `#(&X)` and `X` return the same value when evaluated in an expression.

```
E ::= IE | PE | B
IE ::= id |
      operator(E,...,E) |
      #PE
PE ::= pointer | &id | pointer-op(E,...,E)
B ::= E = E |
      boolean-op(E,...,E)
P ::= id = IE |
      pointer = PE |
      #PE = E |
      if B then P else P |
      P; P |
      while B do P
```

Here `id` is an integer program variable, and `pointer` a pointer program variable. For simplicity, we do not consider arrays here. `operator` is a function that returns an integer value, `pointer-op` a pointer value, and `boolean-op` a truth value.

Our axiomatization of this class of programs will model directly how the compiler works. We assume a set of storage locations which can hold a value which is either an integer or the location of another storage cell. A program variable will be assigned to a location by the compiler at the beginning and this will not be changed during the execution of the program. The value of a program variable will be the value stored at the location.

Thus we assume a *location* sort in our language. In our axiomatization above, we represent a program variable `V` by a function (of the same name) in our language. The value of this function denotes the value of the program variable in the program. Here, we are going to make a conceptual shift: we are going to represent a program variable by a function of *location* sort so that the value of the function denotes the location assigned to the program variable by the compiler. So for a program variable `V`, while before its corresponding function V in the first-order language was dynamic as its value changes during the execution of the program, now V is static as the location assigned to this program variable by the compiler will not change. What is changing during the program execution is the values stored in the memory locations, and this will be modeled by a dynamic function *val*:

$$val : location \to int \cup location.$$

Thus if `V` is an integer variable, then $val(V)$ will be an integer. If V is a pointer, then $val(V)$ will be a location.

To summarize, given a program and a program variable `V`, instead of using V and V' to denote its values at the input and output of the program, respectively, we now use V to denote a memory location and $val(V)$ and $val'(V)$ to denote these two values, respectively. We need to do a similar translation from an expression `E` in the program to a term $val(E)$ (and similarly $val'(E)$) in our first-order language:

1. val(&id) is id, if `id` is an integer program variable.
2. val(#PE) is $val(val(PE))$, if `PE` is a pointer expression.
3. $val(f(E_1, ..., E_k))$ is $f(val(E_1), ..., val(E_k))$, if `f` is either an `operator`, `pointer-op`, or `boolean-op`, assuming that we have a corresponding function f in our language.

Given a program P, our axioms for it, denoted Π_P, will be in terms of *val* and *val*′.

- If P is

  ```
  V = E
  ```

  where `V` is an (integer or pointer) program variable and `E` an (integer or pointer) expression, then we have the following axiom:

  $$val'(x) = \text{ if } x = V \text{ then } val(E) \text{ else } val(x),$$

  where x ranges over locations.
- If P is

  ```
  #PE = E
  ```

  where `PE` is a pointer expression, then the axioms are as follows:

  $$val'(x) = \text{ if } x = val(PE) \text{ then } val(E) \text{ else } val(x).$$

- If P is

  ```
  P1;P2
  ```

  then Π_P is constructed from Π_{P_1} and Π_{P_2} as follows:

  $$\varphi(val'/tmp),\ \text{for each}\ \varphi \in \Pi_{P_1},$$
  $$\varphi(val/tmp),\ \text{for each}\ \varphi \in \Pi_{P_2},$$

  where tmp is a new function of the same arity as val and val' and not already used in Π_{P_1} and Π_{P_2}.
- The cases for conditionals and while loops are similar as before.

Consider the program

```
L = &V;
#L = 1
```

where V is an integer variable and L a pointer. Given that $val(\&V) = V$, we have the following axioms:

$$tmp(x) = \text{ if } x = L \text{ then } V \text{ else } val(x),$$
$$val'(x) = \text{ if } x = tmp(L) \text{ then } 1 \text{ else } tmp(x).$$

Assuming that $L \neq V$ (they are assigned different locations by the compiler), we have $val'(V) = 1$, $val'(L) = V$, and

$$x \neq V \wedge x \neq L \supset val'(x) = val(x).$$

Now consider the following program:

```
while X < Y do
  if Max < #next(A,X) then Max = #next(A,X);
  X = X+1
```

where A is a pointer variable and next(A,X) is a pointer pointing to the Xth location after the one pointed to by A. In C notation, next(A,X) would be A+X and "+" is addition in pointer arithmetic. We use next in order to distinguish it from the normal addition operator arithmetic. We assume the following unique names axioms on locations:

$$\forall n.(X \neq Y \neq Max \neq next(A,n)).$$

Again we compute the axioms for the body of the loop first, which can be simplified into the following axioms under the above unique names axioms:

$$val'(X) = val(X) + 1,$$
$$val(Max) < val(val(next(A, X))) \rightarrow$$
$$\quad val'(Max) = val(val(next(A, X))),$$
$$x \neq X \wedge (x \neq Max \vee \neg val(Max) < val(val(next(A, X)))) \rightarrow$$
$$\quad val'(x) = val(x).$$

Thus the axioms for the program are

$$val(x, 0) = val(x),$$
$$val(X, n+1) = val(X, n) + 1,$$
$$val(Max, n) < val(val(next(A, X), n), n) \rightarrow$$
$$\quad val(Max, n+1) = val(val(next(A, X), n), n),$$
$$x \neq X \wedge (x \neq Max \vee \neg val(Max) < val(val(next(A, X), n), n)) \rightarrow$$
$$\quad val(x, n+1) = val(x, n),$$
$$n < M \rightarrow val(X, n) < val(Y, n),$$
$$\neg val(X, M) < val(Y, M),$$
$$val'(x) = val(x, M).$$

From these axioms, we can deduce that

$$val(X,n) = val(X,0) + n \wedge val(Y,n) = val(Y,0).$$

Thus if we assume that $val(X,0) = 0$ and $val(Y,0) = N$ for a natural number N, then we have $M = N$ and

$$\begin{aligned}
& val(x,0) = val(x), \\
& val(Max,n) < val(val(next(A,X),n),n) \rightarrow \\
& \qquad val(Max,n+1) = val(val(next(A,X),n),n), \\
& x \neq X \wedge (x \neq Max \vee \neg val(Max) < val(val(next(A,X),n),n)) \rightarrow \\
& \qquad val(x,n+1) = val(x,n), \\
& val'(x) = val(x,N).
\end{aligned}$$

Compared to the formalization in Section 3, the one here looks more compact as it quantifies over program variables which are locations. This representation is more low level. For instance, some axioms about the *next* function will be needed before they can be used to infer anything interesting about the program. For more details on how this can be done, see [22].

9. Concluding remarks

Our goal is to construct a translator from a full programming language like C or Java to first-order logic. Once this is done, reasoning about programs can then be done in logic using techniques including but not limited to induction and loop invariants. In this paper, we show how this is possible for a core procedural programming language with loops, functions, and simple pointers. We have extended it to more complex mutable data structures including lists and trees and are working on extending it to thread-based concurrency.

The complexity of the translated first-order theory of a given program depends on what the program is about. If all program variables are propositional, then the resulting first-order theory is decidable for proving both partial and total correctness of the program with respect to any given propositional specification. If the program is about natural numbers and involves addition and multiplication, then we may need full arithmetic to reason about it. If the program is about predicting the trajectory of a planet, then a theory of physics is needed in order to prove anything interesting about it. How to integrate logical reasoning with a domain theory has long been a challenge in AI as well as in computer science.

Acknowledgements

I thank Yin Chen, Shing-Chi Cheung, Yongmei Liu, Pritom Prajkhowa, Yidong Shen, Bo Yang, Charles Zhang, Mingyi Zhang, Yan Zhang, and Yi Zhou for many useful discussions related to the subject of this paper. I also thank the anonymous reviewers for both KR-2014 and this journal for sharing their insights on the topic of this paper and for their valuable comments on earlier versions of this paper. This work was supported in part by HK RGC under GRF 616013.

Appendix A. Correctness under an operational semantics

In this appendix we provide an operational semantics to the language in Section 3 and show the correctness of our axiomatization with respect to this semantics.

Given a program, we define its models to be sequences of states from its executions. We represent states by first-order structures. As before, we assume a base language that contains functions and predicates corresponding to build-in functions and operators. Given a program P, and a tuple $\vec{X}$ of functions that includes all program variables used in P, we extend the base language to a new language $\mathcal{L}_{\vec{X}}$ by adding functions in $\vec{X}$. Notice that this means that if V is a program variable in P for an n-ary array, then we assume that $V \in \vec{X}$ is an n-ary function. Again, we assume that the program variable names are unique and there is no overloading of names.

For the class of programs that we consider here, executing a program in a state either does not terminate or yields a finite sequence of assignments. This can be defined in a standard way. Now a finite sequence $[M_1, \cdots, M_n]$ of $\mathcal{L}_{\vec{X}}$ structures is a model of P if when executed in M_1, P terminates with a sequence of assignments $\alpha_1, \cdots, \alpha_{n-1}$ such that for each $1 \leq i < n$, M_{i+1} is the result of executing α_i in M_i: given a structure M, M' is the result of executing the assignment $V(t_1,...,t_k) = e$ in M if M and M' have the same domains, same interpretation for predicates, same interpretation for functions except V, and for V, its value in M' is defined as follows:

$$V^{M'}(u_1,...,u_k) = \begin{cases} e^M, & \text{if } (u_1,...,u_k) = (t_1^M,...,t_k^M) \\ V^M(u_1,....,u_k), & \text{otherwise} \end{cases}$$

Now consider our translation $\Pi_P^{\vec{X}}$. It uses a language that is an extension of $\mathcal{L}_{\vec{X}}$. In particular, for each variable V, it has a new "primed" function V'. Given a model M of $\Pi_P^{\vec{X}}$, we can project it on $\mathcal{L}_{\vec{X}}$ as usual. Furthermore, we say that a structure I of $\mathcal{L}_{\vec{X}}$ is the primed-projection of M if for any symbol τ in $\mathcal{L}_{\vec{X}}$, if τ does not have a primed version in $\Pi_P^{\vec{X}}$, then its interpretation in I is the same as its in M, but if τ has a primed version, then its interpretation in I is according to τ' in M. We have

Proposition 7. *If M is a model of $\Pi_P^{\vec{X}}$, then there is a model $[M_1, ..., M_k]$ of P such that*

$$M_1 \text{ is the projection of } M \text{ on } \mathcal{L}_{\vec{X}}, \tag{A.1}$$

$$M_k \text{ is the primed-projection of } M \text{ on } \mathcal{L}_{\vec{X}}. \tag{A.2}$$

Conversely, if $[M_1, ..., M_k]$ is a model of P, then there is a model M of $\Pi_P^{\vec{X}}$ such that (A.1) and (A.2) hold.

Proof. We prove the first half of the proposition. The second half is similar and easier. We prove by induction on P. The base case is when P is an assignment

```
V(E1,...,Ek) = E
```

Recall that $\Pi_P^{\vec{X}}$ consists of following axioms:

$$\begin{aligned} V'(\vec{x}) = {} & \text{if } (x_1 = E_1 \wedge \cdots \wedge x_k = E_k) \text{ then } E \\ & \text{else } V(\vec{x}), \\ X'(\vec{y}) = {} & X(\vec{y}). \ \ X \in \vec{X} \text{ and } X \text{ different from } V \end{aligned}$$

It is easy to see that M is a model of $\Pi_P^{\vec{X}}$ if M_1, M_2, the projection and the primed projection of M on $\mathcal{L}_{\vec{X}}$, respectively, is a model of P.

Inductively suppose the result holds for the subprograms of P. There are three cases: conditional statements, sequences and loops. Suppose P is

```
if B then P1 else P2
```

then a sequence $[M_1, ..., M_k]$ of states is a model of P iff either B is true in M_1 and $[M_1, ..., M_k]$ is a model of P_1 or B is false in M_1 and $[M_1, ..., M_k]$ is a model of P_2.

Recall that $\Pi_P^{\vec{X}}$ is constructed from $\Pi_{P_1}^{\vec{X}}$ and $\Pi_{P_2}^{\vec{X}}$ as follows:

$$B \to \varphi, \text{ for each } \varphi \in \Pi_{P_1}^{\vec{X}},$$

$$\neg B \to \varphi, \text{ for each } \varphi \in \Pi_{P_2}^{\vec{X}}.$$

Observe that a model M of $\Pi_P^{\vec{X}}$ satisfies B iff the projection of M on $\mathcal{L}_{\vec{X}}$, $M_{\vec{X}}$, satisfies B. Thus M is a model of $\Pi_P^{\vec{X}}$ iff either B is true in $M_{\vec{X}}$ and M is a model of $\Pi_{P_1}^{\vec{X}}$, or B is false in $M_{\vec{X}}$ and M is a model of $\Pi_{P_2}^{\vec{X}}$. By inductive assumption, suppose now M is a model of $\Pi_P^{\vec{X}}$. Then either B is true in $M_{\vec{X}}$ and for some M_i, $[M_{\vec{X}}, M_1, ..., M_k, M_{\vec{X}'}]$ is a model of P_1 or B is false in $M_{\vec{X}}$ and for some M_i, $[M_{\vec{X}}, M_1, ..., M_k, M_{\vec{X}'}]$ is a model of P_2, where $M_{\vec{X}'}$ is the primed projection of M on $\mathcal{L}_{\vec{X}}$. In either case, for some M_i, $M_{\vec{X}}, M_1, ..., M_k, M_{\vec{X}'}$ is a model of P.

Suppose P is

```
P1; P2
```

then a sequence of states $[M_1, ..., M_k]$ is a model of P iff for some $1 < i < k$, $[M_1, ..., M_i]$ is a model of P_1 and $[M_i, ..., M_k]$ is a model of P_2.

Recall that $\Pi_P^{\vec{X}}$ is constructed from $\Pi_{P_1}^{\vec{X}}$ and $\Pi_{P_2}^{\vec{X}}$ by connecting the outputs of P_1 with the inputs of P_2 as follows:

$$\varphi(\vec{X}'/\vec{Y}), \text{ for each } \varphi \in \Pi_{P_1}^{\vec{X}},$$

$$\varphi(\vec{X}/\vec{Y}), \text{ for each } \varphi \in \Pi_{P_2}^{\vec{X}},$$

where $\vec{Y} = (Y_1, ..., Y_k)$ is a tuple of new function symbols such that each Y_i is of the same arity as X_i in $\vec{X}$. Now suppose M is a model of $\Pi_P^{\vec{X}}$. Construct two models M^1 and M^2 from M as follows: M^1 is the same as M except that for each X_i', its interpretation in M^1 is the same as the interpretation of Y_i in M; and M^2 is the same as M except that for each X_i, its interpretation in M^2 is the same as the interpretation of Y_i in M. Then M^i is a model of $\Pi^{\vec{X}}_{P_i}$, $i = 1, 2$. Notice that this is because $\vec{Y}$ is a tuple of new functions not in $\Pi^{\vec{X}}_{P_i}$, $i = 1, 2$. By the inductive assumption, there is a model $[M^i_1, ..., M^i_{k_i}]$ of P_i, $i = 1, 2$, such that M^i_1 and $M^i_{k_o}$ is the projection and the primed projection of M^i, respectively. By our construction, $M^1_{k_1} = M^2_1$. Thus $[M^1_1, ..., M^1_{k_1}, M^2_2, ..., M^2_{k_2}]$ is a model of P. Notice that we also assume that $\Pi^{\vec{X}}_{P_1}$ and $\Pi^{\vec{X}}_{P_2}$ do not share any temporary variables. This assumption is not needed here but needed for the second half of the proposition.

Our last case is loops. Suppose that P is

```
while B do P1
```

Then $[M_1, ..., M_k]$ is a model of P iff there are some $Q \geq 0$, some $1 = k_0 \leq k_1 < \cdots < k_Q = k$ such that

- $[M_{k_i}, ..., M_{k_{i+1}}]$ is a model of P_1, $0 \leq i < Q$.
- $M_{k_i} \models B$, $0 \leq i < Q$.
- $M_{k_Q} \models \neg B$.

Recall that $\Pi^{\vec{X}}_P$ consists of the following axioms:

$$\begin{aligned}
&\varphi[n], \text{ for each } \varphi \in \Pi^{\vec{X}}_{P_1},\\
&X_i(\vec{x}) = X_i(\vec{x}, 0), \text{ for each } X_i \in \vec{X}\\
&\mathit{smallest}(N, n, \neg B[n]),\\
&X'_i(\vec{x}) = X_i(\vec{x}, N), \text{ for each } X_i \in \vec{X}
\end{aligned}$$

where n is a new natural number variable not already in φ, and N a new natural number constant not already used in $\Pi^{\vec{X}}_{P_1}$.

Let $\mathcal{L}_P$ be the language of $\Pi^{\vec{X}}_P$, and $\mathcal{L}_{P_1}$ the language of $\Pi^{\vec{X}}_{P_1}$. Notice that every symbol in $\mathcal{L}_{P_1}$ not in the base language is also in $\mathcal{L}_P$ but extended by a natural number argument, as described in the construction of $\varphi[n]$.

Suppose M is a model of $\Pi^{\vec{X}}_P$. Let Q be the value of N in M (notice that N is a constant in the language and Q a natural number in the domain). For all $0 \leq i < Q$, $M \models B[i]$ and $M \models \neg B[Q]$. For each natural number i, let M^i be constructed from M as follows: M^i is the same as M except that for each symbol X in $\Pi^{\vec{X}}_{P_1}$ that has been extended by a natural number parameter, the interpretation of X in M^i is the same as the interpretation of $X(i)$ in M and the interpretation of X' in M^i is the same as the interpretation of $X(i+1)$ in M. Notice that M is a structure for language $\mathcal{L}_P$ and M^i a structure for $\mathcal{L}_{P_1}$. It can be seen that for all i, M^i is a model of $\Pi^{\vec{X}}_{P_1}$. By our inductive assumption, for each i, there is a sequence $M^i_1, ..., M^i_{k_i}$ of states that is a model of P_1 and that M^i_1 and $M^i_{k_i}$ are the projection and primed projection of M^i on $\mathcal{L}_{\vec{X}}$. Observe that $M^i_{k_i} = M^{i+1}_1$, it is not hard to see then that the sequence $[M^0_1, ..., M^0_{k_0}, M^1_2, ..., M^1_{k_1}, ..., M^{Q-1}_2, ..., M^{Q-1}_{k_{Q-1}}]$ is a model of P. □

Appendix B. Loop invariants

Our approach translates programs to first-order theories. Once the translation is done, properties of programs can be proved using whatever methods that are valid in first-order logic. In particular, for programs with loops, one can use loop invariants.

Consider a loop of the form `while C do P`. A condition φ is a loop invariant if whenever φ and C are true initially, φ will continue to hold after P is performed. In our notation, this means that the theory corresponding to the program entails the following sentence:

$$\forall n. C[n] \wedge \varphi[n] \rightarrow \varphi[n+1].$$

Now if a postcondition Q can be proved using the invariant φ:

$$\neg C \wedge \varphi \rightarrow Q,$$

then we can prove in our theory that Q' holds as Q' is $Q[N]$ for the N that satisfies $\mathit{smallest}(N, n, \neg C[n])$.

Consider the simple example of the following loop for computing factorials:

```
F=1;
I=0;
while I<X do
  I=I+1;
  F = I*F
```

Given a non-negative integer input X, the output value of F is the factorial of X: $F' = \mathit{fact}(X)$.

To prove the correctness of this program, we first need to assume a definition of factorial, which can be defined inductively as: $\mathit{fact}(0) = 1$ and $\forall n.\mathit{fact}(n+1) = n \times \mathit{fact}(n)$.

One can see that the following is a loop invariant:

$$I \leq X \wedge F = \mathit{fact}(I).$$

Given that this condition is true when the loop initiates, if the loop terminates, then we have:

$$\neg(I < X) \wedge I \leq X \wedge F = \mathit{fact}(I) \tag{B.1}$$

which implies $I = X$ and $F = \mathit{fact}(X)$.

We have implemented a translator[1] for programs in Section 3. The direct translation of the program without any simplification gives the following axioms:

$$\begin{aligned}
&F1 = 1,\\
&I1 = I,\\
&X1 = X,\\
&I2 = 0,\\
&F2 = F1,\\
&X2 = X1,\\
&F(0) = F2,\\
&I(0) = I2,\\
&X(0) = X2,\\
&n < N1 \to I(n) < X(n),\\
&I(N1) \geq X(N1),\\
&I4(n) = I(n) + 1,\\
&F4(n) = F(n),\\
&X4(n) = X(n),\\
&F(n+1) = I4(n) * F4(n),\\
&I(n+1) = I4(n),\\
&X(n+1) = X4(n),\\
&F' = F(N1),\\
&I' = I(N1),\\
&X' = X(N1)
\end{aligned}$$

From this set of equations, it is easy to verify the loop invariant:

$$\begin{aligned}
&I(n) < X(n) \wedge (I(n) \leq X(n) \wedge F(n) = \mathit{fact}(I(n))) \to\\
&\qquad I(n+1) \leq X(n+1) \wedge F(n+1) = \mathit{fact}(I(n+1)).
\end{aligned}$$

Thus

$$n < N1 \to I(n+1) \leq X(n+1) \wedge F(n+1) = \mathit{fact}(I(n+1))$$

Instantiate $n = N1 - 1$ in the above equation, we have $I(N1) \leq X(N1)$. Thus $I(N1) = X(N1)$. So $F' = F(N1) = \mathit{fact}(I(N1)) = \mathit{fact}(X(N1)) = \mathit{fact}(X')$.

Our translator simplifies the translated theory as much as possible by getting rid of temporary variables and making use of Mathematica[2] to solve recurrences as much as possible. For the factorial program, it generates the following set Π of formulas:

[1] This system is implemented by Pritom Prajkhowa and is available upon request.

[2] http://www.wolfram.com/mathematica/.

$$F(0) = 1,$$

$$n < N1 \to I(n) < X(n),$$

$$I(N1) \geq X(N1),$$

$$F(n+1) = (n+1) \times F(n),$$

$$I' = N1,$$

$$F' = F(N1),$$

$$X' = X$$

Notice that $I(n)$ has been eliminated: from the recurrences $I(0) = 0$ and $I(n+1) = I(n) + 1$, Mathematica computes the closed form solution $I(n) = n$. Thus the loop invariant (B.1) cannot be used anymore.

Looking at the formulas in Π, it is clear that $F(n) = \mathit{fact}(n)$ (e.g. this can be verified using Mathematica). Thus to show that $F' = \mathit{fact}(X)$, it all comes down to proving that $N1 = X$, which also proves that the program terminates.

By the definition of the smallest macro, to prove that $N1 = X$, we need to prove the following two assertions:

$$n \leq X - 1 \to n < X,$$

$$\neg(X < X)$$

which are obvious – they can be easily verified using, e.g. Mathematica.

References

[1] E. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, N.J., 1976.
[2] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Springer-Verlag, New York, 1990.
[3] C. Hoare, An axiomatic basis for computer programming, Commun. ACM (1969) 576–580.
[4] D. Harel, First-Order Dynamic Logic, Lecture Notes in Computer Science, vol. 68, Springer-Verlag, New York, 1979.
[5] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science, IEEE, 2002, pp. 55–74.
[6] A. Pnueli, The temporal semantics of concurrent programs, Theor. Comput. Sci. 13 (1981) 45–60.
[7] B. Wegbreit, The synthesis of loop predicates, Commun. ACM 17 (2) (1974) 102–113.
[8] N. Bjørner, A. Browne, Z. Manna, Automatic generation of invariants and intermediate assertions, Theor. Comput. Sci. 173 (1) (1997) 49–87, http://dx.doi.org/10.1016/S0304-3975(96)00191-0.
[9] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Trans. Softw. Eng. 27 (2) (2001) 99–123.
[10] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, C. Xiao, The daikon system for dynamic detection of likely invariants, Sci. Comput. Program. 69 (1) (2007) 35–45.
[11] T. Nguyen, D. Kapur, W. Weimer, S. Forrest, Using dynamic analysis to discover polynomial and array invariants, in: Proceedings of 34th International Conference on Software Engineering, ICSE 2012, IEEE, 2012, pp. 683–693.
[12] D. Kozen, J. Tiuryn, Logics of programs, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), Elsevier, 1990, pp. 789–840.
[13] E.A. Emerson, Temporal and modal logic, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), Elsevier, 1990, pp. 995–1072.
[14] H. Barringer, R. Kuiper, A. Pnueli, Now you may compose temporal logic specifications, in: STOC, 1984, pp. 51–63.
[15] L. Libkin, Elements of Finite Model Theory, Springer, 2004.
[16] A. Charguéraud, Program verification through characteristic formulae, ACM SIGPLAN Not. 45 (9) (2010) 321–332.
[17] A. Charguéraud, Characteristic formulae for the verification of imperative programs, ACM SIGPLAN Not. 46 (9) (2011) 418–430.
[18] M.G. Wallace, Tight, consistent, and computable completions for unrestricted logic programs, J. Log. Program. 15 (1993) 243–273.
[19] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, R. Scherl, GOLOG: a logic programming language for dynamic domains, J. Logic Program. 31 (1997) 59–84, special issue on Reasoning about Action and Change.
[20] F. Lin, A first-order semantics for Golog and ConGolog under a second-order induction axiom for situations, in: Proceedings of KR 2014, 2014.
[21] E. Cohen, Programming in the 1990s: An Introduction to the Calculation of Programs, Springer-Verlag, 1990.
[22] F. Lin, B. Yang, Reasoning about mutable data structures in first-order logic with arithmetic: lists and binary trees, Technical report, Department of Computer Science, Hong Kong University of Science and Technology, http://www.cs.ust.hk/faculty/flin/papers/dsw2015.pdf.