A Formalization of Programs in First-Order Logic with a Discrete Linear Order[*]

Fangzhen Lin (flin@cs.ust.hk)
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

We consider the problem of representing and reasoning about computer programs, and propose a translator 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 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.

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 (Dijkstra 1976; Dijkstra and Scholten 1990), Hoare's logic (Hoare 1969), dynamic logic (Harel 1979), and separation logic (Reynolds 2002). 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 (Pnueli 1981).

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:

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

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 come 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.

[*]This work was supported in part by HK RGC under GRF 616013.

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:

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

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,

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

stands for $\forall n.n < M \rightarrow \neg(n < N \wedge found(n))$. Notice however, that in the macro $smallest(M, k, k < N \wedge found(k))$, k is not a free variable.

The following two useful properties about the smallest macro are easy to prove.

Proposition 1 *If $\exists n\varphi(n)$, then $\exists m.smallest(m, n, \varphi(n))$.*

Proposition 2 *If $smallest(N, n, \varphi(n)) \wedge N > 0$, then $\varphi(N) \wedge \neg\varphi(N-1)$. Furthermore, if $\varphi(n)$ has the following property*

$$\exists n[(\forall m.m > n \rightarrow \varphi(m)) \wedge (\forall m.m \leq n \rightarrow \neg\varphi(m))] \quad (1)$$

then

$$smallest(N, n, \varphi(n)) \equiv \varphi(N) \wedge \neg\varphi(N-1).$$

Proof: If $smallest(N, n, \varphi(n))$, then $\varphi(N)$ and $\forall m.m < N \rightarrow \neg\varphi(m)$. Since $N > 0$, thus $\neg\varphi(N-1)$. Now suppose that $\varphi(N) \wedge \varphi(N-1)$. By (1), for some natural number M,

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

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

A simple class of programs

Consider the following simple class of programs:

```
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 arrays are program variables, and operators are functions. The difference between the two is that the former can occur in the left hand side of an assignment and their values be changed while the latter cannot. Notice that instead of, for example in the case of two dimensional array, the notation of "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".

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 $A(P, \vec{X})$, as follows:

- If P is

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

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

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

  where $\vec{x} = (x_1, ..., x_k)$, and k is the arity of the program variable (array) V. Recall that by our convention, these variables are universally quantified.

- If P is

  `if B then P1 else P2`

  then $A(P, \vec{X})$ is constructed from $A(P_1, \vec{X})$ and $A(P_2, \vec{X})$ as follows:

  $$B \rightarrow \varphi, \text{ for each } \varphi \in A(P_1, \vec{X}),$$
  $$\neg B \rightarrow \varphi, \text{ for each } \varphi \in A(P_2, \vec{X}).$$

- If P is

  `P1; P2`

  Then $A(P, \vec{X})$ is constructed from $A(P_1, \vec{X})$ and $A(P_2, \vec{X})$ by connecting the outputs of P_1 with the inputs of P_2 as follows:

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

where $\vec{Y} = (Y_1, ..., Y_k)$ is a tuple of new program variables (array names) 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 $\varphi(\vec{X}/\vec{Y})$. By renaming if necessary, we assume here that $A(P_1, \vec{X})$ and $A(P_2, \vec{X})$ have no common program variables except those in $\vec{X}$.

- If P is

  ```
  while B do P1
  ```

  Then $A(P, \vec{X})$ is constructed by adding an index parameter n to all functions and predicates in $A(P_1, \vec{X})$ to record their values after the body P_1 has been executed n times. Formally, it consists of the following axioms:

  $$\begin{array}{l}\varphi[n], \text{ for each } \varphi \in A(P_1, \vec{X}),\\ X_i(\vec{x}) = X_i(\vec{x}, 0), \text{ for each } X_i \in \vec{X}\\ smallest(N, n, \neg B[n]),\\ X'_i(\vec{x}) = X_i(\vec{x}, N), \text{ for each } X_i \in \vec{X}\end{array}$$

  where n is a new natural number variable not already in φ, and N a new constant not already used in $A(P_1, \vec{X})$, 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 by performing the following recursive substitutions:
  - for each X_i in $\vec{X}$, replace all occurrences of $X'_i(e_1, ..., e_k)$ by $X_i(e_1[n], ..., e_k[n], n+1)$, and
  - for each program variable X in α, replace all occurrences of $X(e_1, ..., e_k)$ by $X(e_1[n], ..., e_k[n], n)$. Notice that this replacement is for every program variable X, including those temporary program variables not in $\vec{X}$, but introduced during the construction of $A(P_1, \vec{X})$.

While we have used $A(P, \vec{X})$ to denote "the" set of axioms for P and $\vec{X}$, the construction above does not yield a unique set of axioms as the new program variables (functions in our first-order language) introduced when constructing axioms for program sequences are not unique. However, $A(P, \vec{X})$ is unique upto the renaming of these new program variables. In particular, any two different sets of these axioms are logically equivalent when considering only program variables from $\vec{X}$, i.e. when the new program variables 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.

It is easy to see the following "local" property of our construction, similar to the "frame rule" in separation logic.

Proposition 3 *Let $\vec{Y}$ be a tuple of program variables that are not in P and not used in $A(P, \vec{X})$. Then considering only $\vec{X} \cup \vec{Y}$, $A(P, \vec{X} \cup \vec{Y})$ is equivalent to the union of $A(P, \vec{X})$ 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 new program variables only need to be introduced for those that occur in both P and Q.

Proposition 4 *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 $\vec{X}$, $A(P; Q, \vec{X})$ is equivalent to the set of following axioms:*

$$\begin{array}{l}\varphi(\vec{V'}/\vec{Y}), \textit{ for each } \varphi \in A(P, \vec{X}),\\ \varphi(\vec{V}/\vec{Y}), \textit{ for each } \varphi \in A(Q, \vec{X}),\end{array}$$

where $\vec{Y} = (Y_1, ..., Y_k)$ is a tuple of new program variables such that each Y_i is of the same arity as V_i in $\vec{V}$. Again we assume that, by renaming if necessary, $A(P, \vec{X})$ and $A(Q, \vec{X})$ have no common program variables other than those in $\vec{X}$.

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 5 *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 $A(P', \vec{X})$ when considering only $\vec{X}$, then if we use T instead of $A(P', \vec{X})$ in computing $A(P, \vec{X})$, the resulting theory is equivalent to $A(P, \vec{X})$ when considering only $\vec{X}$ as well.*

Notice that in the above proposition, when we use T instead of $A(P', \vec{X})$ in computing $A(P, \vec{X})$, we assume that we will also rename program variables when necessary to avoid name conflicts. For example, if P is $P_1; P_2$, and a theory equivalent to $A(P_1, X)$ is

$$X' = Y \wedge Y = X + 1. \qquad (2)$$

If $A(P_2, X)$ also mentions the temporary variable Y, then we need to rename either the Y in (2) or the Y in $A(P_2, X)$ when constructing $A(P, X)$.

Before we consider more interesting examples, we illustrate our construction of $A(P, \vec{X})$ using two simple programs.

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
```

$A(X1 = 1, (X_1, X_2))$ is the set of the following two sentences

$$\begin{array}{l}X'_1 = 1,\\ X'_2 = X_2\end{array}$$

and $A(X1 = X1+1, (X_1, X_2))$ the set of following two sentences:

$$X_1' = X_1 + 1,$$
$$X_2' = X_2$$

Thus $A(P, (X_1, X_2))$ is

$$Y_1 = 1,$$
$$Y_2 = X_2,$$
$$X_1' = Y_1 + 1,$$
$$X_2' = Y_2$$

Eliminating the helper names (temporary program variables) Y_1 and Y_2, we get $X_1' = 2$ and $X_2' = X_2$.

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. $A(P_1, (X, A, I, N))$ is the set of following sentences (Y_1, Y_2, Y_3, Y_4 are helper names):

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

Instead of using this set to compute $A(P, (X, A, I, N))$, by Proposition 5, we can simplify it first by eliminating the helper names Y_1, Y_2, Y_3, Y_4, and get the following equivalent set of axioms:

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

Thus $A(P, (X, A, I, N))$ is

$$X(0) = X,$$
$$A(x, 0) = A(x),$$
$$I(0) = I,$$
$$N(0) = N,$$
$$X(n+1) = \text{ if } X(n) < A(I(n), n) \text{ then } A(I(n), n) \text{ else } X(n),$$
$$A(x, n+1) = A(x, n),$$
$$I(n+1) = I(n) + 1,$$
$$N(n+1) = N(n),$$
$$smallest(M, n, \neg I(n) < N(n)),$$
$$X' = X(M),$$
$$A'(x) = A(x, M),$$
$$I' = I(M),$$
$$N' = N(M).$$

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:

$$X(0) = X,$$
$$I(0) = I,$$
$$X(n+1) = \text{ if } X(n) < A(I(n)) \text{ then } A(I(n)) \text{ else } X(n),$$
$$I(n+1) = I(n) + 1,$$
$$I(M) \geq N,$$
$$n < M \to I(n) < N,$$
$$X' = X(M),$$
$$A'(x) = A(x),$$
$$I' = I(M),$$
$$N' = N.$$

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

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

we have $I(n) = n$. It is also easy to see 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

$$0 \leq n < N \to X' \geq A(n), \tag{3}$$

which is equivalent to

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

which is easily proved by induction on N using the recurrence about $X(n+1)$.

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, $A(P, \vec{X})$. Whatever properties that one wants to show about P are proved using this set of sentences. A partial correctness corresponds to proving a sentence about $\vec{X}$ and $\vec{X}'$ from $A(P, \vec{X})$. An example is the assertion (3) 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. For instance, for the above simple loop, the smallest macro is $smallest(M, n, \neg I(n) < N(n))$. The fact that there is indeed a natural number M that satisfies this macro expression follows from Proposition 1 and the fact that $I(N) \geq N$ holds.

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:

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

If we add $I = 0 \wedge 0 < M$ to these axioms, we will conclude $\forall n. I(n) < M$, which contradicts the last axiom $I(N) \geq M$.

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 (3) would be represented by a triple like

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

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:

$$\forall m(I_0 \leq m < I \rightarrow X \geq A(m)).$$

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. (Wegbreit 1974; Bjørner, Browne, and Manna 1997; Ernst et al. 2001; 2007; Nguyen et al. 2012)).

In comparison, our proof of (3) uses ordinary mathematical induction and recurrences on $I(n)$ and $X(n)$.

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 (Pnueli 1981). According to a common classification used in formal method community (cf. (Kozen and Tiuryn 1990; Emerson 1990)): approaches like Hoare's logic and dynamic logic are *exogenous* in that they have programs explicitly in the language, while temporal logic approach to program semantics is typically *endogenous* in that a fixed program is often assumed and a program execution counter 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, build up from the structure of the program. Barringer *et al.* (1984) 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 temporal logic approach to program semantics has 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. (Libkin 2004)).

A closely related work is Charguéraud's characteristic formulas for functional programs (Charguéraud 2010; 2011). 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 (Wallace 1993).

While we use natural numbers to formalize loops, Levesque *et al.* (1997) used second-order logic to capture Golog programs with loops in the situation calculus. Recently, Lin (2014) 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 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.

However, perhaps the best way to show the correctness of our axiomatization is to connect it with Golog in the situation calculus. Given a program P, and a tuple $\vec{X}$ of program variables that include all those used in P, we can consider each X in $\vec{X}$ as a functional fluent in the situation calculus, and write successor state axioms for these fluents, assuming that the assignments used in P are the only actions. We can then show that M is a model of $\mathcal{A}(P, \vec{X})$ iff there is a situation calculus model W such that for each $X \in \vec{X}$, $X^M = X^W(S_0)$, and $(X')^M = X^W(S_1)$, where S_1 is such that $W \models Do(P, S_0, S_1)$ (a program P here can be considered as a Golog complex action in a straightforward way). We omit the detail here as this should present no conceptual difficulties.

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 (Cohen 1990) (our program below is adapted from (Nguyen et al. 2012)). It has two loops, one nested inside another.

```
// X and Y are two input integers
  Q=0; // quotient
  R=X; // remainder
  while (R >= Y) do {
    A=1;
    B=Y;
    while (R >= 2*B) do {
      A = 2*A;
      B = 2*B;
    }
    R = R-B;
    Q = Q+A
  }
//  return Q = X/Y;
```

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 $\mathcal{A}(P, \vec{X})$, we again consider only equivalence under $\vec{X}$ and use Proposition 5 to simplify the process.

It is easy to see that $\mathcal{A}(P, \vec{X})$ is equivalent to the union of $\mathcal{A}(Outer, \vec{X})$ and $\{Q = 0 \wedge R = X\}$. To compute $\mathcal{A}(Outer, \vec{X})$, we compute first $\mathcal{A}(Inner, \vec{X})$, which is equivalent to the set of following sentences:

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

Solving the recurrences, we have

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

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

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

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

$$\begin{aligned}
&R < 2^{N+1}Y,\\
&m < N \rightarrow R \geq 2^{m+1}Y,
\end{aligned}$$

$$\begin{aligned}&A' = 2^N,\\&B' = 2^N Y,\\&Q' = Q + A',\\&R' = R - B',\\&X' = X,\\&Y' = Y.\end{aligned}$$

Thus $A(Outer, \vec{X}) \cup \{Q = 0, R = X\}$ is equivalent to

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

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 $A(P, \vec{X})$ as the set of following sentences:

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

From these axioms, we can show the correctness of Cohen's algorithm by proving the following two properties:

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

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

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

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:

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

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
  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.

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. While we have not done it for concurrency, we believe it can be done as well given that we were able to provide a first-order axiomatization of ConGolog (Lin 2014). In this section, we describe how the same approach can be used to axiomatize programs with user defined functions.

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 a bit, 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 $f(\vec{X})\{Body\}$, the set of sentences for it, written $A(f)$, is

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

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.
- $A(Body, \vec{X} \cup \{Result\})$ 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, and the function is captured by the axiom $foo() = 2$. One could argue that it does not make sense for more than one instances 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 $Body1$, and that of $isOdd(N)$ by $Body2$. Then $A(Body1, (N, Result))$ consists of the following axioms:

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

and similarly for $A(Body2, (N, Result))$:

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

Thus $A(isOdd) \cup A(isEven)$ 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)}
```

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:

$$\begin{aligned}&(\forall x\!:\!List).x = [] \vee \exists a(\exists y\!:\!List)x = a :: y,\\ &\forall a, b(\forall x, y\!:\!List).a :: x = b :: y \to (a = b \wedge x = y),\\ &\forall a(\forall x\!:\!List)[] \neq a :: x,\end{aligned}$$

and the three functions yield the following axioms:

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

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:

$$\begin{aligned}&(\forall x\!:\!List)\exists n.List(x, n),\\ &List(x, 0) \equiv x = [],\\ &List(x, n + 1) \equiv (\exists a)(\exists y\!:\!List).x = a :: y \wedge List(y, n)\end{aligned}$$

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

$$List(x, n) \wedge List(x, m) \to m = n. \qquad (4)$$

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

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

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 inductively assumption, $k_1 = k_2$. So $m = n$. This concludes the inductive step, thus the proof of (4).

Using (4), we can then prove the induction schema on lists: for any formula $\varphi(x)$,

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

Suppose the premise is true and for some list x, $\neg\varphi(x)$. Suppose x is a shortest such list: if $List(x, n)$ then for any list y, if $List(y, m) \wedge m < n$, then $\varphi(y)$ holds. Notice that the existence of such an x follows from (4). Suppose $List(x, n)$. 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 address. 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 $A(f_i)$ of axioms for f_i is then

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

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 $foo(X, M)$:

$$\begin{aligned}&foo(x) = y \equiv \exists m \forall n \geq m.foo(x, n) = y,\\ &foo(0, 0) = Cycle,\\ &foo(1, 0) = 1,\\ &foo(0, n + 1) = foo(0, n),\\ &foo(1, n + 1) = 1\end{aligned}$$

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.

Concluding remarks

My goal is to have a translator from a full programming language like C or Java to first-order logic. In this paper, I show how this is possible for a core procedural programming language with loops and functions. Instead of loop invariants used in Hoare's logic, the approach relies on mathematical induction and recurrences. I show that even for programs with nested while loops such as Cohen's integer division, typical properties about them can be proved effectively using their corresponding first-order theories.

The complexity of the translated first-order theory from a program depends on the domain that 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. I hope that with this work, more KR researchers will take up this challenge and start to contribute to program verification. As a step in this direction, we plan to develop a reasoner for programs about mathematics by integrating a logic solver with Mathematica[1], which has a programmable set of powerful tools for solving problems in mathematics. For instance, it can easily solve the recurrences that we have seen in this paper.

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

References

Barringer, H.; Kuiper, R.; and Pnueli, A. 1984. Now you may compose temporal logic specifications. In *STOC*, 51–63.

Bjørner, N.; Browne, A.; and Manna, Z. 1997. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.* 173(1):49–87.

Charguéraud, A. 2010. Program verification through characteristic formulae. In *ACM Sigplan Notices*, volume 45 (9), 321–332. ACM.

Charguéraud, A. 2011. Characteristic formulae for the verification of imperative programs. In *ACM SIGPLAN Notices*, volume 46 (9), 418–430. ACM.

Cohen, E. 1990. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer-Verlag.

Dijkstra, E. W., and Scholten, C. S. 1990. *Predicate Calculus and Program Semantics*. New York: Springer-Verlag.

Dijkstra, E. 1976. *A Discipline of Programming*. Englewood Cliffs, N.J.: Prentice Hall.

Emerson, E. A. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 995–1072. Elsevier.

Ernst, M. D.; Cockrell, J.; Griswold, W. G.; and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on* 27(2):99–123.

Ernst, M. D.; Perkins, J. H.; Guo, P. J.; McCamant, S.; Pacheco, C.; Tschantz, M. S.; and Xiao, C. 2007. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69(1):35–45.

Harel, D. 1979. *First-Order Dynamic Logic*. New York: Springer-Verlag: Lecture Notes in Computer Science 68.

Hoare, C. 1969. An axiomatic basis for computer programming. *Comm. ACM*.

Kozen, D., and Tiuryn, J. 1990. Logics of programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. Elsevier. 789–840.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming, Special issue on Reasoning about Action and Change* 31:59–84.

Libkin, L. 2004. *Elements of Finite Model Theory*. Springer.

Lin, F. 2014. A first-order semantics for Golog and ConGolog under a second-order induction axiom for situations. In *Proceedings of KR 2014*.

Nguyen, T.; Kapur, D.; Weimer, W.; and Forrest, S. 2012. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of 34th International Conference on Software Engineering (ICSE 2012)*, 683–693. IEEE.

Pnueli, A. 1981. The temporal semantics of concurrent programs. *Theor. Comput. Sci.* 13:45–60.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. IEEE.

Wallace, M. G. 1993. Tight, consistent, and computable completions for unrestricted logic programs. *Journal of Logic Programming* 15:243–273.

Wegbreit, B. 1974. The synthesis of loop predicates. *Communications of the ACM* 17(2):102–113.