A Verifier for Functional Properties of Sequence-Manipulating Programs

Carlo A. Furia

ETH Zurich, Switzerland
caf@inf.ethz.ch

Abstract. Many programs operate on data structures whose models are sequences, such as arrays, lists, and queues. When specifying and verifying functional properties of such programs, it is convenient to use an assertion language and a reasoning engine that incorporate sequences natively. This paper presents qfis, a program verifier geared to sequence-manipulating programs. qfis is a command-line tool that inputs annotated programs, generates the verification conditions that establish their correctness, tries to discharge them by calls to the SMT-solver CVC3, and reports the outcome back to the user. qfis can be used directly or as a back-end of more complex programming languages.

1 Overview

Many programs use data structures whose functional properties are expressible in terms of *sequences* of values from a certain domain. For example, lists, queues, and stacks are all modeled by sequences accessed according to specific patterns. To specify and reason about such programs, it is convenient to use first-order languages that support sequences natively, and that are amenable to automated reasoning.

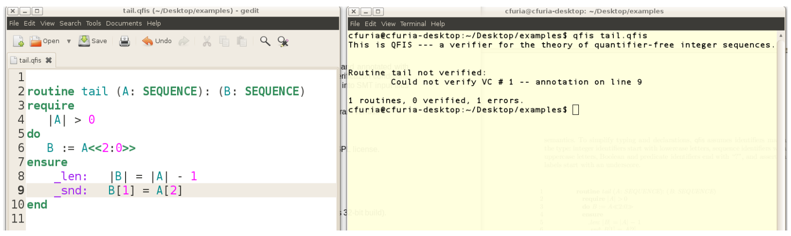
In previous work [2], we introduced a first-order theory of integer sequences $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ whose quantifier-free fragment is decidable. $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ also includes Presburger arithmetic on sequence elements, and it is sufficiently expressive to specify several functional properties of sequence-manipulating programs. The present paper describes qfis, an automated verifier for programs annotated with $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ formulas. qfis inputs programs written in a simple imperative Algol-like procedural language, supporting integers and sequences as primitive types. Each routine may include a functional specification in the form of pre- and postcondition, written in a logic language including native functions and predicates on sequences—such as the concatenation and length functions.

The overall usage of qfis is similar to that of general-purpose program verifiers such as Dafny [3] or Why [4]. First, the user writes the program as a collection of routines with pre- and postconditions. Whenever useful, she also provides a collection of logic axioms (also expressed in the theory of integer sequences $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$) that define the semantics of predicates mentioned in the specification. For example, when proving the correctness of a sorting algorithm, it is customary to introduce a predicate *sorted*?(X) with the expected meaning. When called on the input file, qfis generates the *verification conditions* (VC): a set of first-order formulas whose validity entails the correctness of the program with respect to its specification. qfis encodes the VC in the input language of the SMT-solver CVC3 [1], and calls the solver to discharge the VC. Then, qfis filters back CVC3's output and reports the outcome to the user. In case of unsuccessful verification, qfis points to specific annotations in the input program that could not be verified. Figure 1 shows a screenshot of qfis, with the input program displayed in the editor on the left, and the verifier's output in the shell on the right.

qfis is written in Eiffel, distributed under GPL, and available for download at:

`http://se.inf.ethz.ch/people/furia/software/qfis.html`

The download page includes pre-compiled binaries for Linux; the source code for compilation; a user manual with installation instructions and a tutorial examples; a demo video; a collection of annotated example programs; and a syntax-highlighting GTK source-view specification of its input language (used in Figure 1).


tail.qfis (~/Desktop/examples) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
tail.qfis
cfuria@cfuria-desktop: ~/Desktop/examples
File Edit View Terminal Help

```
 1
 2 routine tail (A: SEQUENCE): (B: SEQUENCE)
 3 require
 4    |A| > 0
 5 do
 6    B := A<<2:0>>
 7 ensure
 8     _len:   |B| = |A| - 1
 9     _snd:   B[1] = A[2]
10 end
11
```

```
cfuria@cfuria-desktop:~/Desktop/examples$ qfis tail.qfis
This is QFIS --- a verifier for the theory of quantifier-free integer sequences.


Routine tail not verified:
          Could not verify VC # 1 -- annotation on line 9

1 routines, 0 verified, 1 errors.
cfuria@cfuria-desktop:~/Desktop/examples$ 
```

Fig. 1. qfis verifying routine *tail*.

2 Using qfis

qfis's input language features sequences as a primitive type; unlike other verifiers that also support sequence types (e.g. Dafny [3]), qfis's sequences are usable in both specification and imperative constructs.

2.1 Input Language

qfis inputs text files containing a collection of routines (functions and procedures) and declarations of global variables (accessible from any routine), predicates (usable in annotations), and axioms (defining the semantics of user-defined predicates). For example:

```
1    routine tail (A: SEQUENCE): (B: SEQUENCE)
2      require |A| > 0
3      do B := A≪2:0≫
4      ensure
5        _len: |B| = |A| − 1
6        _snd: B[1] = A[2]
7    end
```

is a partially-specified function *tail* that returns the input sequence without the first element (line 3, i.e., from the second element to the last one). *tail*'s precondition (line 2) requires that the input sequence A has positive length; the postcondition has two clauses (lines 5 and 6) that assert that the returned sequence B has one less element than A, and that B's first element equals A's second element.

qfis annotations include axioms, pre- and postconditions (**require**, **ensure**), intermediate assertions (**assert**, **assume**), loop invariants (**invariant**), and frame clauses (**modify**) to specify the effect of routines on global variables. The assertions themselves share the same language of Boolean expressions also usable in imperative statements. It is very similar to the quantifier-free fragment of the theory $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ with some restrictions (for performance in the current implementation) but also some additions. In particular, it includes element ($A[3]$) and range ($A\ll 2{:}5 \gg$) selection, sequence length ($|A|$), concatenation ($A \circ B$) and sequence equality ($A \doteq B$), as well as full integer arithmetic among sequence elements and lengths. Postconditions may also include the **old** keyword to refer to the values of global variables before invocation of the current routine. Axioms may also include arbitrary quantifiers (**forall**, **exist**) to define predicate semantics. To simplify typing and declarations, qfis assumes different identifier styles according to the type: integer identifiers start with lowercase letters, sequence identifiers with uppercase letters, Boolean and predicate identifiers end with "?", and assertion labels start with an underscore.

2.2 Verification Condition Generation

Given a collection of annotated input routines, qfis generates *verification conditions* by weakest precondition calculation (performed by visiting the AST of the input). The backward propagated assertions hold references to their source line numbers in the input program; this information is used when some VC cannot be discharged, to trace back the error to the location in the source. For example, the backward substitution of *tail*'s postcondition clause *_len* determines the VC $|A| > 0 \Longrightarrow |A\ll 2{:}0 \gg| = |A| - 1$, which can be proven valid.

Similarly to other program provers, qfis performs *modular reasoning*: the semantics of a call to some routine *foo* within another routine is entirely determined by *foo*'s precondition P, postcondition Q, and frame F: check that P holds before the call (**assert** P); nondeterministically assign values to variables in *foo*'s frame (and arguments) **havoc** F; constrain the nondeterministic assignment to satisfy Q (**assume** Q).

2.3 SMT Encoding

qfis does not implement the decision procedure for the quantifier-free fragment of $\mathcal{T}_{\mathsf{seq}(\mathbb{Z})}$ presented in [2], but it directly encodes the VC in the input language of the SMT-solver CVC3. This design choice provides overall more flexibility and a more robust implementation, relying on a carefully engineered tool such as CVC3. The input language is deliberately relaxed to include undecidable components (full-fledged integer arithmetic and unrestricted quantifiers in axioms), but this is not much of a problem in practice thanks to the powerful instantiation heuristics provided by SMT-solvers—as long as the departure from the basic decidable kernel is reasonably restricted.

The translation of VC to CVC3 uses a list **DATATYPE** definition to encode sequences. A set of standard axioms provides an axiomatization of the concatenation function *cat* applied on lists; for example, an axiom asserts that **nil** is the neutral element of the concatenation function *cat* with formulas such as $cat(x, \mathbf{nil}) = x$. The CVC3 encoding of expressions involving element selection and subranges uses *unrolled* definitions that are handled efficiently by the reasoning engine.

3 Examples

Table 1 lists 11 programs verified using qfis with CVC3 2.2 as back-end, running on a GNU/Linux Ubuntu box (kernel 2.6.32) with an Intel Quad-Core2 CPU at 2.40 GhZ with 4 GB of RAM. For each program, the table shows the number of routines in the input file (# R), the number of user-defined predicates and specification functions (# P), of user-written axioms (# A), the total lines of the input (# L), and the real time (in seconds) taken by verification, including both the VC generation and the call to CVC3. All the programs are included in the qfis distribution.

The most complex program is the second version of *merge sort*, which required an increase of the standard timeout of 10 seconds (per VC, before the SMT solver gives up proving validity). *tail* is the very simple program of Section 2.1, and it is also the only program where verification fails (as shown in Figure 1); qfis reports that it cannot verify the postcondition clause *_snd*: $B[1] = A[2]$: in fact, if A has only one element (which is possible, because the precondition only requires that it is not empty), $A[2]$ is undefined.

PROGRAM	# R	# P	# A	# L	TIME [S]
binary search	2	3	11	87	3.1
linear search	1	3	5	41	2.3
linked list	15	4	6	208	15.6
merge sort (v. 1)	1	1	4	49	12.3
merge sort (v. 2)	2	1	4	67	31.0
quick sort	2	3	11	87	6.8
reversal	1	1	2	24	4.0
stack reversal	1	2	4	37	5.5
sum & max (v. 1)	3	2	8	83	6.8
sum & max (v. 2)	1	2	9	52	3.8
tail	1	0	0	11	2.2

Table 1. Programs verified with qfis.

Acknowledgements. Work partially supported by SNF project 200021-134976 (Automated Support for Invariant Inference) and by the Hasler foundation on related projects.

References

1. Barrett, C., Tinelli, C.: CVC3. In: CAV. LNCS, vol. 4590, pp. 298–302. Springer (2007)
2. Furia, C.A.: What's decidable about sequences? In: ATVA. LNCS, vol. 6252, pp. 128–142. Springer (2010)
3. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR (Dakar). LNCS, vol. 6355, pp. 348–370. Springer (2010)
4. Why3: Where programs meet provers. `http://why3.lri.fr`