User Manual for QFIS – A Verifier for the Theory of Quantifier-Free Integer Sequences

Carlo A. Furia
caf@inf.ethz.ch

v. 1.1 – December 2012*

1 What is qfis

qfis (also QFIS) is a static program verifier for imperative programs working on sequences of integers and annotated with formulas in the quantifier-free fragment of the theory of integer sequences. qfis generates verification conditions from an annotated program by backward reasoning. The verification conditions are translated into SMT input files and CVC3 tries to discharge them. qfis reports the results of verification back to users.

If you are familiar with program verifiers such as Boogie [4, 5] or Why [1, 3], using qfis is a very similar experience, except that qfis does not work on generic programs and first-order annotations but is targeted to the theory of quantifier-free integer sequences described in [2]. You can try out qfis online without installing any software through COMCOM at:

`http://cloudstudio.ethz.ch/comcom/#QFIS`

2 Downloading and installing

qfis is implemented in Eiffel and available for download in source and pre-compiled form from:

`http://bitbucket.org/caf/qfis/`

Since version 1.1, qfis works on both GNU/Linux and Windows environments. It is distributed under GNU GPL v. 3 or later versions.

After you have downloaded it, you can install qfis following these simple instructions (which use Unix shell syntax and commands by default).

1. Move the downloaded package (e.g., `qfis.tgz`) to a directory of your choice (e.g., `/home/user/tools/`):

```
mv qfis.tgz /home/user/tools/
```

*The first release was v. 0.1, on 24 March 2011.

2. Move to the installation directory and unpack the package:

```
cd /home/user/tools ; gzip -cd qfis.tgz | tar xvf -
```

   Unpacking will create a directory (e.g., `/home/user/tools/qfis/`) with examples, prelude files, and a binary for your architecture.

3. If you want to compile qfis (you can get the sources from the same repository), you have to install the EiffelStudio compiler. See the instructions on:

   `http://www.eiffel.com`

   qfis has been developed and compiled with EiffelStudio 7.0. To compile the binary, launch the script

   `./build-linux.sh` or `build-windows.bat`

   according to your platform, in the installation directory. You may have to edit `qfis.ecf` if the compiler can't locate the libraries needed.

4. Add the path to the `bin` directory to the environment variable `$PATH` (`%PATH%` in Windows environments):

   ```
   export PATH=$PATH:/home/user/tools/qfis/bin/
   ```

5. Set the environment variable `$QFIS_PRELUDE` (`%QFIS_PRELUDE%` in Windows environment) to the `prelude` directory:

   ```
   export QFIS_PRELUDE=/home/user/tools/qfis/prelude/
   ```

   Notice that the path stored by the environment variable must use *forward slashes* even in Windows environments. If you don't set the variable, qfis will look for background theories in its working directory.

6. Install CVC3 and make sure it is reachable from the program path. This version of qfis was tested with CVC3 v. 2.2. To install CVC3 under a Debian-based distribution you can type:

   ```
   sudo apt-get install cvc3
   ```

   If you use Emacs, you may want to install the package `cvc3-el` as well.

7. To have syntax highlighting with any text editor supporting GTK source view (such as *gedit*), copy the file `qfis.lang` in the directory `grammar` to the appropriate directory. For Ubuntu and other similar distributions:

   ```
   sudo cp qfis.lang /usr/share/gtksourceview-2.0/language-specs/
   ```

3 Input language

qfis's input language includes both a simple modular imperative language, to describe computations on sequences of integer variables, and an annotation language, very close to the quantifier-free theory of integer sequences [2].

In the following BNF grammars, the meta-characters are "::=" (rule definition), "⌊" (alternative), "⟨···⟩" (optional part); square and round brackets and all other special characters are terminals; $^+$ denotes one or more repetitions.

3.1 Differences with the theory of sequences

There are some differences between the theory of integer sequences as defined in [2] and the formulas allowed in qfis. The major differences are the following:

- The following table summarizes the way operators on sequences are encoded in ASCII in qfis. This manual uses, whenever possible, pretty printing, but input files must use the ASCII encoding.

Description	Native	ASCII	Pretty-printed
Concatenation	$\circ$	`++`	$\circ$
Sequence equality	$\doteq$	`==`	$\doteq$
Sequence inequality	$\not\doteq$	`=/=`	$\not\doteq$
Implication	$\Rightarrow$	`==>`	$\Longrightarrow$
Iff	$\Leftrightarrow$	`<==>`	$\Longleftrightarrow$
Element selection	$S[3]$	`S[3]`	$S[3]$
Subrange	$S[3,0]$	`S<<3:0>>`	$S\ll 3{:}0\gg$
True, False	$\top, \bot$	`True`, `False`	**True**, **False**

  Ranges with mixed endpoints (negative and positive) or empty interval (e.g. $S[2{:}1]$) are currently not supported. In a range or element selection, 1 denotes the first element, 2 the second; 0 denotes the last element, -1 the second-to-last, etc.

- Regular expressions are currently disallowed in qfis.

- Arithmetic is not applicable directly to sequences in qfis, but only to singletons (which are handled as integers). In practice, this means that every sequence appearing in an arithmetic operation or relation must be projected (e.g., $X[1] + 2 < Y[1]$ instead of $X + 2 < Y$). In this respect, notice that the empty sequence is *not* treated as zero, and not accepted at all in arithmetic expressions.

- qfis annotations allow for full integer arithmetic (including multiplication and integer division) and comparison and other operations between two integer variables. For example, you can specify that two sequences have the same length.

- qfis axioms also allow for quantifiers.

The extensions beyond the quantifier-free fragment entail that the resulting verification conditions are in general undecidable, unlike the original theory [2]. Overall, this seemed a better trade-off for users, given that the underlying SMT solving technology includes, on the one hand, several optimization that introduce incompleteness, but works reasonably well, on the other hand, to

prove the validity of formulas (as opposed to finding out for sure that they are invalid).

3.2 Variable naming

qfis is case sensitive. Every identifier starts with a letter, followed by letters, digits, or the special characters _ and #. In addition, variables must be named according to the following convention:

Integer variables: identifier starts with lowercase letter

Sequence variables: identifier starts with uppercase letter

Boolean variables: identifier ends with the extra character ?

Labels (of assertions): identifier starts with the extra character _ and ends with the extra character :

Variables are declared with the following syntax:

VariableDeclaration	::=	Declaration ⌊ Declaration ; VariableDeclaration
Declaration	::=	IntegerIdentifierList : *INTEGER*
	⌊	SequenceIdentifierList : *SEQUENCE*
	⌊	BooleanIdentifierList : *BOOLEAN*

IntegerIdentifierList, SequenceIdentifierList, and BooleanIdentifierList are comma-separated non-empty lists of integer, sequence, and Boolean identifiers respectively.

3.3 Declarations and background theories

Annotations can mention logic functions. Every logic function must be declared; the signature lists the function argument's types, according to the syntax:

FunctionDeclaration	::=	Identifier ⟨(TypeList)⟩ : Type ⟨**in** "*path*"⟩
TypeList	::=	Type ⌊ Type , TypeList
Type	::=	*INTEGER* ⌊ *SEQUENCE* ⌊ *BOOLEAN*

The variable naming convention for variables extends to functions: a function returning an integer (respectively, sequence, Boolean) must have a name suitable for an integer (respectively, sequence, Boolean) variable.

A function with only a declaration is used as *uninterpreted*. If the **in** clause is supplied, the CVC3 file *path* is included as theory of the function declared. qfis performs no check that the theory in *path* (if any) is consistent with the function declaration. There are two other ways to introduce axioms for a declared function.

Declarations directory. You can set a declarations directory, where qfis searches for theories for declared functions. The declarations directory can be set either by setting the environment variable `QFIS_PRELUDE` or with the `-d=path` flag when calling qfis. See the installation instructions above for recommend settings for the declarations directory; if you don't set it as suggested, concatenation and length are also used as uninterpreted.

For each function declared **function** *fun* (*T1*, *T2*, ..., *Tn*): (*T*), qfis will look for a file named `declaration_fun_t_t1_t2_..._tn_.cvc` in the declarations directory, where each `ti` is 0 if *Ti* is *SEQUENCE*, 1 if *Ti* is *BOOLEAN*, and 2 if *Ti* is *INTEGER*.

Axiom declarations. You can introduce axioms that are translated into CVC3. qfis allows quantifiers in axioms but not in annotation formulas. The syntax for axioms is then:

AxiomDeclaration	::=	**axiom** QuantifiedFormula
QuantifiedFormula	::=	Formula
	⌊	Quantifier (VariableDeclaration) (QuantifiedFormula)
Quantifier	::=	**forall** ⌊ **exist**

where quantifier-free formulas are defined as:

Formula	::=	AtomicFormula ⌊ (Formula)
	⌊	**not** Formula ⌊ Formula Connective Formula
Connective	::=	**and** ⌊ **or** ⌊ $\Longrightarrow$ ⌊ $\Longleftrightarrow$
AtomicFormula	::=	**True** ⌊ **False** ⌊ BooleanQuery
BooleanQuery	::=	BooleanIdentifier ⌊ **old** BooleanIdentifier
	⌊	BooleanIdentifier (IdentifierList)
	⌊	Sequence SequenceComparison Sequence
	⌊	Integer ArithmeticComparison Integer
SequenceComparison	::=	$\doteq$ ⌊ $\not\doteq$
ArithmeticComparison	::=	$=$ ⌊ $\neq$ ⌊ $<$ ⌊ $\leq$ ⌊ $>$ ⌊ $\geq$
Sequence	::=	SequenceIdentifier ⌊ **old** SequenceIdentifier
	⌊	SequenceIdentifier (IdentifierList)
	⌊	**e** ⌊ Integer ⌊ (Sequence) ⌊ Sequence $\circ$ Sequence
	⌊	Subrange
Subrange	::=	Sequence $\ll$ Integer : Integer $\gg$
Integer	::=	IntegerConstant ⌊ ElementSelection
	⌊	IntegerIdentifier ⌊ **old** IntegerIdentifier
	⌊	IntegerIdentifier [IdentifierList]
	⌊	|Sequence| ⌊ ArithmeticCompound
IntegerConstant	::=	$\langle - \rangle$ Digit$^+$
Digit	::=	0 ⌊ 1 ⌊ ... ⌊ 9
ElementSelection	::=	SequenceIdentifier [IntegerConstant]
	⌊	(Sequence) [IntegerConstant]
ArithmeticCompound	::=	Integer Operator Integer
Operator	::=	$+$ ⌊ $-$ ⌊ $*$ ⌊ $/$
IdentifierList	::=	Identifier ⌊ Identifier , IdentifierList
Identifier	::=	IntegerIdentifier ⌊ SequenceIdentifier ⌊ BooleanIdentifier

Notice that integer functions use square, rather than round, brackets to list actual arguments. Operator precedence is as follows: $\circ$, arithmetic operators (first $*$, $/$, then $+$, $-$), arithmetic comparisons, **not**, **and**, **or**, $\Longrightarrow$, $\Longleftrightarrow$. Implication is right-associative and all other binary operators are left-associative. The **old** keyword can only appear in postconditions, directly in front of global variable identifiers (of any type).

3.4 Instructions

qfis includes instructions for assignment, conditional (or nondeterministic) branch, loop, intermediate assertions (**assert**, **assume**), nondeterministic shuffling (**havoc**), routine call. **split** is the only non-conventional instruction, designed specifically for sequences: **split** *S* **into** *S1*, *S2*, ..., *Sn* is equivalent to

havoc *S1*, *S2*, ..., *Sn* ; **assume** $S \doteq S1 \circ S2 \circ ... \circ Sn$

Compound instructions sit on multiple line, or on the same line separated by a semicolon. Similarly, multiple loop invariant clauses are either separated by labels or semicolons. A condition (in a conditional or loop) <?> denotes nondeterministic choice.

Instruction	::=	**skip**
	⌊	**assert** Formula
	⌊	**assume** Formula
	⌊	**havoc** IdentifierList
	⌊	**split** Sequence **into** SequenceIdentifierList
	⌊	IntegerIdentifier := Integer
	⌊	SequenceIdentifier := Sequence
	⌊	BooleanIdentifier := Formula
	⌊	Conditional ⌊ Loop ⌊ RoutineCall
Conditional	::=	**if** Condition **then** Compound ⟨**else** Compound ⟩ **end**
Loop	::=	**until** Condition ⟨**invariant** ⟨ClauseList⟩⟩ **loop** Compound **end**
Condition	::=	Formula ⌊ <?>
ClauseList	::=	Formula ⌊ ClauseList ; Formula ⌊ ClauseList Label Formula
RoutineCall	::=	**call** ⟨IdentifierList :=⟩ RoutineIdentifier ⟨(IdentifierList)⟩
Compound	::=	Instruction ⌊ Instruction ⟨;⟩ Compound

3.5 Routines

Each routine has a signature, with input and output arguments, and optionally a precondition, a postcondition, a declaration of local variables, a frame. The input arguments are assumed read-only. Pre and postconditions must mention only input and arguments variables, or global variables. Global variables that may be modified by the routine must be listed in its frame (**modify**).

RoutineDeclaration	::=	**routine** RoutineIdentifier ⟨Signature⟩ Declarations
Signature	::=	⟨(VariableDeclaration)⟩ ⟨: (VariableDeclaration)⟩
Declarations	::=	⟨Precondition⟩ ⟨Modify⟩ ⟨Locals⟩ Body ⟨Postcondition⟩ **end**
Precondition	::=	**require** ⟨ClauseList⟩
Postcondition	::=	**ensure** ⟨ClauseList⟩
Modify	::=	**modify** IdentifierList
Locals	::=	**local** VariableDeclaration
Body	::=	**do** ⟨Compound⟩

A qfis input file consists of declarations of functions, axioms, global variables, and routines.

Program	::=	ProgramElement$^+$
ProgramElement	::=	FunctionDeclaration ⌊ AxiomDeclaration
	⌊	GlobalDeclaration ⌊ RoutineDeclaration
GlobalDeclaration	::=	**global** VariableDeclaration

4 VC generation

qfis uses a weakest-precondition calculus defined on primitive instructions as follows, where $\phi[x \mapsto y]$ is ϕ with every occurrence of x replaced by y, and v' is a fresh variable of the same type as v.

1 wp(**skip**, ϕ) = ϕ
2 wp(**assert** F, ϕ) = F **and** ϕ
3 wp(**assume** F, ϕ) = $F \Longrightarrow \phi$
4 wp(**havoc** V, ϕ) = $\phi[V \mapsto V']$
5 wp($X := E$, ϕ) = $\phi[X \mapsto E]$

The weakest-preconditions of compound and derived instructions is as follows.

1 wp(A ; B, ϕ) = wp(A, wp(B, ϕ))
2 wp(**split** S **into** S_1, ..., S_m, ϕ) =
3 wp(**havoc** S_1, ..., S_m ; **assume** $S \doteq S_1 \circ ... \circ S_m$, ϕ)
4 wp(**if** F **then** T **else** E **end**, ϕ) =
5 ($F \Longrightarrow$wp(T, ϕ)) **and** (**not** $F \Longrightarrow$wp(E, ϕ))
6 wp(**if** <?> **then** T **else** E **end**, ϕ) = wp(T, ϕ) **and** wp(E, ϕ)
7 wp(**until** F **invariant** I **loop** L **end**, ϕ) = I

If a loop invariant is not specified, assume $I =$ **True**.

For routine calls, assume *foo* has m input arguments $A_1, \ldots, A_m$, n output arguments $B_1, \ldots, B_n$, precondition *Pre* (**True** if not specified), postcondition *Post* (**True** if not specified), and frame $G_1, \ldots, G_h$. Then, the weakest precondition for a call to *foo* uses fresh variables for the output arguments to handle the cases where some input and output actuals coincide, and saves the value of global variables before the call in other fresh variables Old_local_k (reused in every routine call):

1 wp(**call** V_1, ..., V_n := *foo* (U_1, ..., U_m), ϕ) =
2 wp(**assert** *Pre*[A_1, ..., $A_m \mapsto U_1$, ..., U_m] ;
3 $Old\_local_1 := G_1$; ... ; $Old\_local_h := G_h$
4 **havoc** G_1, ..., G_h ;
5 **assume** *Post*[A_1, ..., A_m, B_1, ..., B_n, **old** G_1, ..., **old** $G_h \mapsto$
6 U_1, ..., U_m, V'_1, ..., V'_n, Old_local_1, ..., Old_local_h]
7 $V_1 := V'_1$; ... ; $V_n := V'_n$, ϕ)

With the weakest precondition, we generate the set VC of verification conditions for *foo* as follows. Let $X_1, \ldots, X_n$ be the global variables mentioned in *foo*'s postcondition with the **old** syntax. In the generation of verification conditions, we prepend some assignments to the routine body that store the **old** values of $X_1, \ldots, X_n$ in fresh local variables Old_X_1, ..., Old_X_n:

$$SaveOld \triangleq Old\_X_1 := X_1 \ ; \ ... \ ; \ Old\_X_n := X_n$$

Then, a reference **old** X_k to any such variable in the postcondition becomes a reference to the alias local variable Old_X_k:

$$PostOld \triangleq Post[\textbf{old } X_1, ..., \textbf{old } X_n \mapsto Old\_X_1, ..., Old\_X_n]$$

Correspondingly, we have the following rules.

1 $\mathsf{VC}(\textbf{routine } foo \textbf{ require } Pre \textbf{ do } Body \textbf{ ensure } Post \textbf{ end}) =$
2 $\quad \mathsf{VC}(Body, PostOld) \cup \{Pre \Longrightarrow \mathsf{wp}(SaveOld\ ;\ Body, PostOld)\}$
3 $\mathsf{VC}(A\ ;\ B, \phi) = \mathsf{VC}(B, \phi) \cup \mathsf{VC}(A, \mathsf{wp}(B, \phi))$
4 $\mathsf{VC}(\textbf{until } F \textbf{ invariant } I \textbf{ loop } L \textbf{ end}, \phi) =$
5 $\quad \{I \textbf{ and not } F \Longrightarrow \mathsf{wp}(L, I)\} \cup \{I \textbf{ and } F \Longrightarrow \phi\}$
6 $\mathsf{VC}(\textbf{until } <?> \textbf{ invariant } I \textbf{ loop } L \textbf{ end}, \phi) =$
7 $\quad \{I \Longrightarrow \mathsf{wp}(L, I)\} \cup \{I \Longrightarrow \phi\}$

and, for all other instructions *Inst*, $\mathsf{VC}(Inst, \phi) = \emptyset$.

5 CVC3 axiomatization

qfis maps integer sequences to Lisp-like lists in CVC3:

DATATYPE IntList = **nil** | **cons**(**head**: INT, **tail**: IntList) **END**;

The singleton sequence n for an integer term n becomes **cons**(n, **nil**).

5.1 Concatenation

Concatenation is the function *cat*: (IntList , IntList) → IntList with the following axioms (Tarski's editor axioms is needed only in the most complex inference, hence commented out for efficiency by default).

%% Congruence
∀ (h: INT, t, o: IntList): $cat(\textbf{cons}(h, t), o) = \textbf{cons}(h, cat(t, o))$;
%% TC1
∀ (l: IntList): $cat(\textbf{nil}, l) = l$;
∀ (l: IntList): $cat(l, \textbf{nil}) = l$;
%% TC2
∀ (x, y, z: IntList): $cat(x, cat(y, z)) = cat(cat(x, y), z)$;
%% TC3: Tarski's editor axiom
∀ (x, y, u, v: IntList): $cat(x, y) = cat(u, v) \Rightarrow \exists$ (w: IntList):
($cat(x, w) = u \wedge cat(w, v) = y) \vee (cat(u, w) = x \wedge cat(w, y) = v)$;
%% TC4−6
∀ (x, y: IntList): $cat(x, y) = x \Leftrightarrow y = \textbf{nil}$;
∀ (x, y: IntList): $cat(x, y) = y \Leftrightarrow x = \textbf{nil}$;
∀ (x, y: IntList): $cat(x, y) = \textbf{nil} \Rightarrow x = \textbf{nil} \wedge y = \textbf{nil}$;

5.2 Length

Length has signature *length*: IntList → NONNEGATIVE_INT and the usual inductive definition (where NONNEGATIVE_INT are the nonnegative integers) and congruence relation with respect to concatenation.

$\forall$ (l: IntList): $length(l) = 0 \Leftrightarrow\ l = \mathbf{nil}$;
$\forall$ (l: IntList): $l \neq \mathbf{nil} \Rightarrow length(l) = 1 + length(\mathbf{tail}(l))$;
$\forall$ (x, y: IntList): $length(cat(x,\ y)) = length(x) + length(y)$;

5.3 Element selection

For every element selection $A[p]$ with $p > 0$ appearing in the qfis formulas, the translation defines a position function *atp*: IntList $\rightarrow$ INT with the axioms:

$\forall$ (x: IntList): $\Gamma(x,\ p) \Rightarrow atp(x) = \mathbf{head}(\mathbf{tail}^{p-1}(x))$;
$\forall$ (x, y: IntList): $\Gamma(x,\ p) \Rightarrow at2(cat(x,\ y)) = at2(x)$;
%% For each integer a, b such that a + b = p, a >0, b >0
$\forall$ (x, y: IntList):
$\Gamma(x,\ a) \wedge \mathbf{tail}^{a}(x) = \mathbf{nil} \wedge \Gamma(y,\ b) \Rightarrow at2(cat(x,\ y)) = atb(y)$

$\Gamma(x, p)$ is a shorthand for the formula denoting: "x has at least p elements":

$$\Gamma(x,p) \ = \ \begin{cases} x \neq \mathbf{nil} & p = 1 \\ \mathbf{tail}^{p-1}(x) \neq \mathbf{nil} \wedge \Gamma(x,\ p-1) & p > 1 \end{cases}$$

and $\mathbf{tail}^n$ denotes the function **tail** applied n times (and $\mathbf{tail}^0(x)$ is just x).

For every element selection $A[p]$ with $p \leq 0$ appearing in the qfis formulas, the translation defines a position function *at_bottomq*: IntList $\rightarrow$ INT where $q = -p$ with the axioms:

$\forall$ (x: IntList): $\Gamma(x,\ q{+}1) \wedge \mathbf{tail}^{q+1}(x) = \mathbf{nil} \Rightarrow at\_bottomq(x) = \mathbf{head}(x)$;
$\forall$ (x: IntList): $\Gamma(x,\ q{+}2) \Rightarrow at\_bottomq(x) = at\_bottomq(\mathbf{tail}(x))$;

5.4 Subranges

For every subrange $A \ll a,\ b \gg$ with $1 \leq a < b$ appearing in the qfis formulas, the translation defines a function *rangea_b*: IntList $\rightarrow$ IntList with the axioms:

$\forall$ (x: IntList): $\Lambda(x,\ b) \Rightarrow rangea\_b(x) = \mathbf{nil}$;
$\forall$ (x: IntList): $\Gamma(x,\ b) \Rightarrow rangea\_b(x) = \Pi(x,\ a,\ b)$;

$\Lambda(x, p)$ is a shorthand for the formula denoting: "x has less than p elements":

$$\Lambda(x,p) \ = \ \begin{cases} x = \mathbf{nil} & p = 1 \\ \Lambda(x,\ p-1) \vee \mathbf{tail}^{p-1}(x) = \mathbf{nil} & p > 1 \end{cases}$$

$\Pi(x, p, q)$ is a term corresponding to the subrange from p to q, constructed over the list datatype:

$$\Pi(x,p,q) \ = \ \begin{cases} \mathbf{cons}(\mathbf{head}(\mathbf{tail}^{p-1}(x)),\ \mathbf{nil}) & p = q \\ \mathbf{cons}(\mathbf{head}(\mathbf{tail}^{p-1}(x)), \Pi(p{+}1,\ q)) & p < q \end{cases}$$

For every subrange $A \ll a,\ b \gg$ with $0 \geq a > b$ appearing in the qfis formulas, the translation defines a function *range_bottomp_q*: IntList $\rightarrow$ IntList where $p = -a$, $q = -b$ with the axioms:

$\forall$ (x: IntList): $\Lambda(x,\ q{+}1) \Rightarrow$*range_bottomp_q*$(x) =$ **nil**;
$\forall$ (x: IntList): $\Gamma(x,\ q{+}1) \wedge$ **tail**$^{q+1}(x) =$ **nil**
$\Rightarrow$ *range_bottomp_q*$(x) = \Pi(x,\ 1,\ q{-}p)$;
$\forall$ (x: IntList): $\Gamma(x,\ q{+}2) \Rightarrow$*range_bottomp_q*$(x) =$ *range_bottomp_q*(**tail**(x));

For every subrange $A{\ll}a,\ 0{\gg}$ with $1 \leq a$ appearing in the qfis formulas, the translation defines a function *rangea_0*: IntList $\rightarrow$ IntList with the axioms:

$\forall$ (x: IntList): $\Lambda(x,\ b) \Rightarrow$ *rangea_0*(x) = **nil**;
$\forall$ (x: IntList): $\Gamma(x,\ b) \Rightarrow$ *rangea_0*(x) = **tail**$^{b-1}(x)$;

Finally, a subrange $A{\ll}a,a{\gg}$ is simply treated as the element selection $A[a]$.

6 A tutorial example

In this tutorial example, we are proving the (partial) correctness of an implementation of MergeSort working on sequences of integers. The complete example is available qfis's distribution, in the subdirectory `examples` with several other examples. You are welcome to submit your own examples to the author.

MergeSort takes any integer sequence as input and returns the same sequence sorted. In this example we only specify that the output is sorted and that it has the same length as the input. Correspondingly, we introduce the following declaration in qfis.

routine *merge_sort* (*A*: *SEQUENCE*): (*Result*: *SEQUENCE*)
  do
  ensure
    _sorted: *sorted?* (*Result*)
    _same_size: $|A| = |Result|$
  end

We declare the predicate *sorted?* as a logic function with the right signature:

function *sorted?* (*SEQUENCE*): *BOOLEAN*

We also give *sorted?* the intended semantics with an inductive definition. Notice that we should also specify how *sorted?* works on concatenations: the concatenation of two sequences is sorted iff both sequences are sorted and the first sequence's last element and the second sequence's first elements are sorted.

axiom *sorted?*(**e**)
axiom forall (*i*: *INTEGER*) (*sorted?*(i))
axiom forall (*L*: *SEQUENCE*)
  ($|L| > 1 \Longrightarrow$(*sorted?*$(L) \iff L[1] \leq L[2]$ **and** *sorted?*$(L{\ll}2{:}0{\gg})$))
axiom forall (X, Y: *SEQUENCE*)
  ($|X| > 0$ **and** $|Y| > 0 \Longrightarrow$
    (*sorted?*(X) **and** *sorted?*(Y) **and** *sorted?*$(X[0] \circ Y[1]) \iff$ *sorted?*$(X \circ Y)$))

Now, let us add a basic implementation of *merge_sort*. If the input has zero or one element, it is already sorted. Otherwise, the algorithm works recursively: it splits the input into two sequences, sorts each sequence, and then merges the result. Notice that the way the input is split into two subsequences is not relevant for partial correctness but only for progress (and performance).

routine *merge_sort* (*A*: *SEQUENCE*): (*Result*: *SEQUENCE*)
  local *L*, *R*: *SEQUENCE*
  do
    if $|A| \leq 1$ **then**
      Result := *A*
    else
      split *A* **into** *L*, *R*
      call *L* := *merge_sort* (*L*)
      call *R* := *merge_sort* (*R*)
      call *Result* := *merge*(*L*, *R*)
    end
  ensure
    _sorted: *sorted*? (*Result*)
    _same_size: $|A| = |Result|$
end

merge takes two sorted sequences and returns a merged sorted sequence:

routine *merge* (*X*, *Y*: *SEQUENCE*): (*M*: *SEQUENCE*)
  require
    _sorted_X: *sorted*?(*X*)
    _sorted_Y: *sorted*?(*Y*)
  do
  ensure
    _sorted: *sorted*? (*M*)
    _same_size: $|M| = |X| + |Y|$
end

Calling qfis on the current program results in a successful verification of *merge_sort* and a failed verification of *merge*, whose implementation we haven't provided yet: qfis stops at the first postcondition of *merge* which it cannot establish.

```
> qfis merge_sort.qfis

This is QFIS --- a verifier for the theory of quantifier-free integer sequences.

Routine merge_sort verified successfully.

Routine merge not verified:
Could not verify VC # 2 -- annotation on line 63

2 routines, 1 verified, 1 errors.
```

Let us go on and provide an implementation of *merge*: at every iteration of the loop the smallest of the elements in *L* and *R* (initialized to *X* and *Y* because input arguments are read-only) is added at the end of *M*. Finally, if any of the two sequences is not empty after the loop, it is also added at the end of *M*.

routine *merge* (*X*, *Y*: *SEQUENCE*): (*M*: *SEQUENCE*)
  local
    L, *R*: *SEQUENCE*
  do
    L := *X* ; *R* := *Y* ; *M* := **e**
    until $|L| = 0$ **or** $|R| = 0$
    loop
      if $L[1] > R[1]$ **then**

$M := M \circ R[1]$; $R := R \ll 2{:}0 \gg$
else
$M := M \circ L[1]$; $L := L \ll 2{:}0 \gg$
end
end
if $|L| > 0$ **then** $M := M \circ L$ **else** $M := M \circ R$ **end**
end

qfis still cannot verify *merge* because we have provided no loop invariants. In particular, the first error it encounters:

```
Routine merge not verified:
Could not verify closing of loop on line 38 -- annotation on line 55
```

means that it cannot establish that the loop invariant is strong enough to guarantee the postcondition.

Let us add a loop invariant: M, L, R are always sorted, and additionally the first element of both L and R is no smaller than the last element of M. Another clause states the invariance of the combined length of M, L, and R.

until $|L| = 0$ **or** $|R| = 0$
invariant
_sorted_M: *sorted?*(M)
_sorted_L: *sorted?*(L)
_sorted_R: *sorted?*(R)
_L_smaller: $M[0] \leq L[1]$
_R_smaller: $M[0] \leq R[1]$
_size: $|X| + |Y| = |M| + |L| + |R|$
loop
...
end

Verification still fails, now the errors reported are about the loop invariant clauses *L_smaller* and *R_smaller* for which qfis cannot verify consecution (inductiveness). This is because the clauses must be evaluated before and after a generic iteration of the loop to verify consecution, but M is empty in the first iteration (hence $M[0]$ is undefined) and L or R is empty after the last iteration of the loop (hence $L[1]$ or $R[1]$ is undefined). We weaken the two clauses to:

_L_smaller: $|M| > 0$ **and** $|L| > 0 \Longrightarrow M[0] \leq L[1]$
_R_smaller: $|M| > 0$ **and** $|R| > 0 \Longrightarrow M[0] \leq R[1]$

qfis can verify the latest version correctly:

```
> qfis merge_sort.qfis

This is QFIS --- a verifier for the theory of quantifier-free integer sequences.

2 routines, 2 verified, 0 errors.
```

A final consistency check is the smoke test, which tries to see if the axioms (of *sorted?*) and the background theories used are consistent (i.e., **False** cannot be inferred).

```
> qfis -s merge_sort.qfis

This is QFIS --- a verifier for the theory of quantifier-free integer sequences.

Smoke test successful: the axioms and the background theories seem consistent.
```

7 Command line options

The help screen of qfis is the following:

```
This is QFIS --- a verifier for the theory of quantifier-free integer sequences.

Usage: qfis [options] <input_file>
where [options] is one or more of the following.
   -h  display this notice.
   -s  perform a smoke test (consistency of the axioms).
   -w=<wdir>  set <wdir> as the work directory (default: current dir).
   -d=<ddir>  set <ddir> as the declarations directory
              (default: /home/user/tools/qfis/prelude/-- $QFIS_PRELUDE is set).
   -t=N  give up after N seconds for each verification condition (default: 10).
   -v    parse and typecheck only (no verification).
```

Notice that the declarations directory is set as recommended in the installation instructions. We already showed the smoke test feature in the tutorial example.

Work directory. The work directory is where qfis stores the generated CVC3 input files. The files are not removed after verification, so that you can inspect them (for lower-level debugging or to see exactly the problem with a failed verification attempt).

Timeout. The timeout is important for non-trivial verifications, because qfis treats a timed out run of CVC3 as a failed verification. This is consistent with the observation that CVC3 is usually rather efficient to determine validity but it often saturates all the available memory when trying to establish "Invalid" (or "Unknown"). If you go for complex verification conditions you may have to give CVC3 some more time to try to prove validity.

References

[1] Jean-Christophe Filliâtre. *The WHY verification tool*, 2009. Version 2.18, `http://proval.lri.fr`.

[2] Carlo A. Furia. What's decidable about sequences? In Ahmed Bouajjani and Wei-Ngan Chin, editors, *Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA'10)*, volume 6252 of *Lecture Notes in Computer Science*, pages 128–142. Springer, September 2010.

[3] Claude Marché Jean-Christophe Filliâtre. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[4] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008.

[5] K. Rustan M. Leino. Specification and verification of object-oriented software. Marktoberdorf International Summer School 2008, lecture notes.