GUIDED CORRECTNESS PROOFS OF LOGIC PROGRAMS

Emmanuil I. Marakakis
Science Department
Technological Educational Institute of Crete,
Stavromenos
714 10 Heraklion, Crete, Greece
email: mmmarak@es.teicrete.gr

ABSTRACT
This paper presents a correctness method of logic programs based on proof schemes. A proof scheme is a systematic plan of proof actions. A set of correctness proof schemes is presented. It is shown that the structure of a general logic program constructed by our schema-based method is reflected in the structure of its correctness proof. For each program schema of our method a proof scheme is proposed. The proof schemes corresponding to the design schemata are followed for the correctness of logic programs. Correctness proofs in this approach are guided by the constructed logic programs.

KEY WORDS
Correctness of logic programs, proof schemes, specifications, transformations, program schemata, verification.

1 Introduction

This paper presents a method for proving the correctness of logic programs which are constructed by a schema-based method.

This schema-based method constructs general typed, moded logic programs by stepwise top-down design using five program schemata, data types (DTs) and modes [1], [2]. A program is constructed by this method by successively refining undefined predicates. The lowest refinement level involves refinement by DT operations. The programs which are constructed by this method are polymorphic many-sorted programs. They satisfy declared input-output modes when run using the standard left-right depth-first computation rule. The logic programs which are constructed by this method are called Schema-Instance Programs or SI-Programs.

In this paper, the problem of proving the correctness of an SI-program with respect to a specification is considered. Given a specification there are two main approaches to proving the correctness of a program. In the first approach, correctness is ensured during construction of the program [3], [4]. In this case a specification is successively refined into a program. The construction process consists either of smaller predefined development steps whose correctness has been proved or of equivalence preserving transformations. In the second approach, a program is first constructed by any means and then it is proved to be correct with respect to its specification [5].

We follow the second approach. That is, first a program is constructed by using this schema-based method and then its correctness is proved with respect to its logic specification. The highly structured form of SI-programs facilitates the formulation of correctness proofs. In this paper, it is claimed that the structure of an SI-program is reflected in the structure of its correctness proof. In other words, there are "proof schemes" corresponding to design schemata which are followed in the correctness proof. The main features of this correctness method are the following. 1) Specifications are transformed into a structured form which facilitates correctness proofs. 2) This correctness method provides guidance to proofs through the proof schemes which correspond directly to design schemata. 3) The structure of an SI-program is reflected in the structure of its correctness proof. 3) The correctness of the DT operations is assumed. This results in correctness proofs shorter than proofs which have to show the correctness of DT operations.

The classification of different approaches to proving correctness in related work is based on the form of specifications. We distinguish three approaches. In the first approach, specifications are expressed as sets of examples and properties. In the second approach, specifications are expressed as first-order logic formulas. In the third approach, schemata are enhanced with specifications which characterize the problem domain.

Specifications in [6], [7] are expressed by examples of behavior of the constructed procedure. Programs in [7] are specified by sets of positive and negative examples. Verification is discussed with respect to these sets of examples. In a similar direction, programs in [6], are specified by sets of examples and properties. Verification is discussed with respect to these sets of examples and properties.

Specifications in [3], [4] are expressed as formulas in first-order logic and correctness is ensured by construction. It is shown in [3] that the logic descriptions are correct by construction. Logic descriptions in [3] have a structured form similar to the structured form of logic specifications of our approach. The development method in [4] produces programs which are partially correct with respect to a first-order logic specification. It is also shown that there exists a
specification called the implemented specification which is stronger than the actual specification. The completed definition of the implemented relation is complete with respect to the implemented specification. Correctness theorems are proved by equivalence preserving transformations as in our approach.

Program schemas in [8] are defined as open first order logic programs. A special kind of correctness for open programs is defined which is called steadfastness. The derived programs from the program schemas inherit the schema correctness (steadfastness) provided that their parameters are correctly computed. That is, the derived programs in [8] are correct by construction.

Specifications in our approach are expressed in typed first-order logic with equality. Types are an important component of specifications. Typed specifications are more expressive than untyped ones. The main advantage of our method over these methods is the guidance which is offered by the proof schemes. In addition, correctness proofs are facilitated by types.

Initially, an overview of the five program schemata and the construction process of our schema-based method [1], [2] are presented. Then, the correctness method is presented. Next, the proof schemes are proposed. After that, the application of proof schemes and two examples are presented. Finally, conclusions are discussed.

2 Five Schemata and the Construction Process

A (program design) schema is a problem-independent algorithm design strategy. A design schema contains a fixed list of subparts or sub-strategies that have different instantiations for each problem to which they are applied.

Definition 3.1. A typed, moded program schema is a typed program schema together with a mode schema for each predicate variable.

Five typed, moded program schemata have been defined in this method which are problem-independent algorithm design strategies. Their classification is based on the algorithm design strategy that each schema represents. That is, Incremental, Divide-and-conquer, Subgoal, Case and Search. These schemata can be applied to problems with different representations because they are defined to be independent of particular representations. Logic programs in this method are constructed by composing instances of the 5 schemata.

The notation that is used in this paper is as follows. The lower case letters \( u \) and \( v \), possibly subscripted, are schema argument variables. Identifiers beginning with a capital letter are predicate variables. The lower case Greek letter \( \alpha \) possibly subscripted stands for parameter variables. The lower case Greek letter \( \tau \) possibly subscripted stands for type variables.

The Incremental schema assumes that the input data \( \tau \) includes a constructor operation which builds an element of \( \tau \) from another element of \( \tau \) and some other data item. The Incremental schema processes one by one each piece of the input data and composes the results for each one to construct the solution. The schema is as follows. Note that mode schemata are omitted because are not used in program correctness.

**Type Schemata**

- **Type(Incr) :** \( \alpha_1 \times \alpha_2 \)
- **Type(Terminating) :** \( \alpha_1 \)
- **Type(Initial_result) :** \( \alpha_1 \times \alpha_2 \)
- **Type(Decomstruction) :** \( \alpha_1 \times \alpha_3 \times \alpha_1 \)
- **Type(Non_initial_result) :** \( \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_2 \)

**Clause Schemata**

- \( \text{Incr}(u_1, u_2) \rightarrow \text{Terminating}(u_1) \land \text{Initial_result}(u_1, u_2) \)
- \( \text{Incr}(u_1, u_2) \rightarrow \neg \text{Terminating}(u_1) \land \text{Decomstruction}(u_1, e_1, e_2) \land \text{Incr}(e_2, e_3) \land \text{Non_initial_result}(u_1, e_1, e_2, u_2) \)

The Divide-and-conquer schema decomposes the problem representation in two subproblems of similar form to the initial one. The solution of the problem is constructed by composing the solutions of the subproblems together.

The Subgoal schema reduces the problem into a conjunction of two or more subproblems. The solutions of the simpler problems imply the solution of the original problem.

The Case schema reduces a problem to two or more independent subproblems. Each subproblem corresponds to a different case of the original problem. A solution to the initial problem is implied by each solution of its subproblems.

The Search schema performs search in the space of states of a problem. It constructs the search tree in a stack called search stack. The search stack has an implicit representation of the state space which remains for searching. Backtracking is performed by using the search stack.

The construction process in our schema-based method involves the application of a sequence of refinements which are either schema refinements or refinements by DT operations [1], [2]. Let \( p/n \) be an undefined predicate with type \( \tau_1 \times \cdots \times \tau_n \) and mode \( m_1, \ldots, m_n \). Let \( p(x_1, \ldots, x_n) \) be a typed completely general atom (cga) where each \( x_i \) (\( 1 \leq i \leq n \)) has type \( \tau_i \) in \( p(x_1, \ldots, x_n) \). Initially, the programmer gives a typed cga of the predicate that he wants to define, i.e. \( p(x_1, \ldots, x_n) \), the type \( \tau_i \) of each \( x_i \) in \( p(x_1, \ldots, x_n) \) and the mode of \( p/n \). The initial refinement is applied to this typed cga. The next refinements are applied to typed cgas of undefined predicates which are created by the initial refinement, and so on. Eventually, the undefined predicates are expected to be refined by DT refinements. The construction process is a successive top-down, left-to-right application of refinements until the construction of the desired SI-program is complete. A program is considered to be complete when all
of its predicates are defined. During the program construction process, the signatures and the modes of the program predicates are also derived [1].

3 Correctness Method

For each relation \( p \) to be implemented, there are its specification \( p^S \) and its implementation \( p \).

**Definition 4.1.** Let \( p^S \) be a predicate. The logic specification \( (Spec^P) \) for \( p^S \) is defined to be a formula in polymorphic many-sorted FOL of the form

\[
Spec^P = \forall x \, \tau \, (p^S(x) \iff Def^P)
\]

where \( x \) is a tuple of distinct variables and \( \tau \) is a tuple of sorts corresponding to the variables in \( x \). \( Def^P \) is a formula in polymorphic many-sorted first-order logic which defines the relation \( p^S(x) \).

**Example 1.** The predicate \( \text{sum}^S(q,s) \) where \( Type(\text{sum}^S) = \text{seq}(Z) \times Z \) is true iff \( s \) is the sum of integers of sequence \( q \).

**Logic specification:**

\[
\forall q/\text{seq}(Z), s/Z (\text{sum}^S(q,s) \iff s = \sum_{i=1}^{\#q} q_i)\]

\( \#q \) stands for the length of the sequence \( q \) and \( \sum \) is the summation operator for all numeric data types.

**Example 2.** \( \text{incr}^S(q) \) where \( Type(\text{incr}^S) = \text{seq}(\alpha) \) is true iff the sequence \( q \) is in increasing order.

**Logic specification:**

\[
\forall q/\text{seq}(\alpha) (\text{incr}^S(q) \iff \forall i/N_1 (1 \leq i \leq (\#q - 1) \rightarrow q_i \leq q_{i+1}))
\]

The meaning of SI-programs is defined using program completion semantics.

**Definition 4.2.** Let \( Pr \) be a set of clauses. Let

\[
p(x_1, \ldots, x_n) \leftarrow L_1, \ldots, L_m
\]

be a clause where \( x_1, \ldots, x_n \) are distinct variables and \( L_1, \ldots, L_m \) are literals whose arguments are variables or constants. Let \( y_1, \ldots, y_{\#d} \) be the variables in the body of clause \( p \) which do not appear in its head. The clause \( p(x_1, \ldots, x_n) \leftarrow \exists y_1, \ldots, y_{\#d} (L_1 \land \ldots \land L_m) \) is equivalent to the previous one. Suppose that there are \( r \) such predicates for clause \( p/n \)

\[
p(x_1, \ldots, x_n) \leftarrow E_1 \ldots p(x_1, \ldots, x_n) \leftarrow E_r
\]

where each \( E_i \) stands for a corresponding formula of the form \( \exists y_1, \ldots, y_{\#d} (L_1 \land \ldots \land L_m) \). Then the completed definition of the predicate \( p/n \) is the formula

\[
\forall x_1 \ldots x_n \, (p(x_1, \ldots, x_n) \iff E_1 \lor \ldots \lor E_r)
\]

Let \( p_1, \ldots, p_k \) be all the predicate symbols which appear in the head atoms of program clauses in \( Pr \). The completion of program \( Pr \) denoted by \( \text{comp}(Pr) \) is the set of the completions of \( p_1, \ldots, p_k \).

Note that this definition of completion is slightly different from the standard one in [9]. We do not add in the \( \text{comp}(Pr) \) negative unit clauses for undefined predicates in \( Pr \) because these predicates are assumed to be implemented by DT operations. We also omit the equality theory, which is assumed to be part of the theory of DTs.

**Definition 4.3.** Let \( Pr \) be an SI-program, excluding the DT definitions. Let \( A \) be the theory of underlying DTs including the specifications of the DT operations. Then the meaning \( \text{Prog} \) of an SI-program is defined as follows:

\[
\text{Prog} = \text{comp}(Pr) \cup A
\]

**Definition 4.4.** The specification \( Spec \) of an SI-program with top level predicate \( p \) is a set of formulas including one of the form

\[
\forall x/\tau (p^S(x) \iff Def^P)
\]

**Definition 4.5.** Let \( Pr \) be an SI-program with top-level predicate \( p \) excluding DT definitions, and \( Spec \) its specification. \( p \) does not occur in \( Spec \) and \( p^S \) does not occur in \( \text{comp}(Pr) \). \( Pr \) is partially correct with respect to \( Spec \) if

\[
Spec \cup A \cup \text{comp}(Pr) \models \forall x/\tau (p(x) \iff p^S(x))
\]

\( Pr \) is complete with respect to \( Spec \) if

\[
Spec \cup A \cup \text{comp}(Pr) \models \forall x/\tau (p(x) \iff p^S(x))
\]

\( Pr \) is totally correct with respect to \( Spec \) if it is both partially correct and complete.

**Example 3.** The notation that is used is as follows. \( \rightarrow \) stands for the empty sequence. \( \cdots \) stands for the term constructor for sequences. \( x :: q \) is a sequence with head \( x \) and tail \( q \). Logic specification for relation \( \text{sum}^S(q,s) \) (Spec)

\[
\text{Spec} = \forall q/\text{seq}(Z), s/Z (\text{sum}^S(q,s) \iff s = \sum_{i=1}^{\#q} q_i)
\]

**Axioms - Lemmas - Specifications of DT operations \( A \):**

**Axioms**

**A1 Domain closure axiom for sequences**

\[
\forall s/\text{seq}(\alpha) (s = <> \iff \exists h/a, t/\text{seq}(\alpha) s = h :: t)
\]

**A2 Uniqueness axioms for sequences**

\[
\forall h/a, t/\text{seq}(\alpha) (\neg (h :: t = <>))
\]

**A3 Definition of summation operation over 0 entities**

\[
\forall s/\text{seq}(Z) (s = <> \iff \sum_{i=1}^{\#s} s_i = 0)
\]

**Lemmas**

**L1** \( \forall s/\text{seq}(\alpha) (s \neq <> \iff \exists h/a, t/\text{seq}(\alpha) s = h :: t) \)

**L2** \( \forall h/a, t/\text{seq}(\alpha) (s = h :: t \iff \forall i/N (2 \leq i \rightarrow s_i = t_{i-1}) \) \)

**L3** \( \forall s/\text{seq}(\alpha), h/\alpha (s = h :: t \iff s = <> \land s_i = t_{i+1}) \)

**L4** \( \forall h/\alpha, s/\text{seq}(\alpha) (s = h :: t \iff h = s_1) \)

**Logic specifications of DT operations**

\[
\forall q/\text{seq}(\alpha) (\text{empty}_q = <>)
\]

\[
\forall s/Z (\text{neutral_add}_s(s,t) \iff s = 0)
\]

\[
\forall s/\text{seq}(\alpha), h/\alpha, q (\text{head}(q,h) \iff q = <> \land \exists t/\text{seq}(\alpha) q = h :: t)
\]

\[
\forall s, h/s/Z (\text{plus}_s(s,t) \iff h = s + t)
\]

670
4.2 Proof Scheme for Case Schema

If the Case schema has been applied to construct the predicate \( p \), then correctness proof is of the form

\[
\forall x (p(x) \rightarrow D_f(p))
\]

where \( \forall x (p(x) \rightarrow \exists \bar{y} (F_1 \land \ldots \land F_k)) \) and \( F_1, \ldots, F_k \) are first-order formulas. It is assumed that the arguments \( x \) are identical in each of the above formulas. In this case, perform the following steps, which are sufficient to establish the correctness result.

1. Try to reformulate \( D_f(p) \) as an equivalent formula of the form \( \exists \bar{z} (G_1 \lor \ldots \lor G_k) \) where \( \bar{z} \) is of the same type as \( \bar{y} \) in \( \exists \bar{y} (F_1 \lor \ldots \lor F_k) \).

2. Try to prove \( \forall x, \bar{y} (F_1 \rightarrow G_1[\bar{z}/\bar{y}]), \ldots, \forall x, \bar{y} (F_k \rightarrow G_k[\bar{z}/\bar{y}]) \).

4.3 Proof Scheme for Incremental and Divide-and-conquer Schemata

If the Incremental or Divide-and-conquer schema has been applied to construct \( p(x) \) then we apply structural induction on a well-founded set.

Let \( \tau \) be a type, and let \( \mu \) be a mapping from \( \tau \) to some well-founded set \( (S, \prec) \). \( \mu \) induces a well-founded ordering on \( \tau \), i.e., for all \( u_1, u_2 \) of type \( \tau \), \( u_1 \prec u_2 \) iff \( \mu(u_1) \prec \mu(u_2) \).

Let \( \forall \bar{u} (p(u, \bar{v}) \rightarrow D_f(p)) \) be the correctness theorem to be proved, abbreviated by \( \Phi(u) \). \( P(x) \) in the structural induction schema can now be instantiated by \( \Phi(u) \) where \( u \) is of type \( \tau \), and a well-founded ordering \( \prec \) has been defined on \( \tau \) by mapping \( \tau \) to some well-founded set.

The above induction scheme is used as the correctness proof method for the Incremental and Divide-and-conquer schemata. The following proof schemes for the base case and induction step of the above induction schema show how the proof can proceed.

In order to prove \( \forall u, \bar{v} (p(u, \bar{v}) \equiv D_f(p)) \) where \( p(u, \bar{v}) \) has been defined by an Incremental or Divide-and-conquer schema, perform the following steps. In the following, \( \bot \) represents the bottom element of the well-founded set \( S, \prec \).

Base case: \( \forall u, \bar{v} (u = \bot \rightarrow (p(u, \bar{v}) \equiv D_f(p))) \)

\( u \prec u \prec u \prec \ldots \prec u \)

Induction hypothesis: \( \forall u \ L_i(u) \equiv \forall u, u' \ L_i(u) \)

(\( u < u \rightarrow (p(u, u') \equiv D_f(p)) \))

Induction step: \( \forall u, \bar{v} (u \neq \bot \land L_i(u) \rightarrow (p(u, \bar{v}) \equiv D_f(p)) \)

4.4 Proof Scheme for Search Schema

If the Search schema has been applied to construct \( p(x) \) its completion has the following form.

\[
\forall x (p(x) \equiv \exists \bar{y}, \bar{w} (F_1 \land F_2[\bar{w}]))
\]
where $F_2(w)$ is an atom containing a recursive predicate and $w$ is the search stack.

This has the same form as the completed definition of an instance of Subgoal, and the same proof scheme can be applied. However, the proof of correctness of the definition of $F_2(w)$ has to be considered as well, since the Search schema provides a definition of $F_2(w)$.

Suppose the specification of $p$ is $\forall \bar{x} (p(\bar{x}) \rightarrow Defp)$ where the variables $\bar{x}$ are identical to $\bar{z}$ in the formula defining $p(\bar{z})$ given above. Then try the following steps.

1. Try to transform $Defp$ into an equivalent formula $\exists \bar{z}, \bar{w}' (G_1 \wedge G_2(\bar{w}'))$ where $\bar{w}'$ is of the same type as $\bar{w}$.

2. Try to prove $\forall \bar{z}, \bar{y}, \bar{w} (F_1(\bar{w}) \rightarrow G_1(\bar{z}/\bar{y}))$.

3. Try to prove $\forall \bar{z}, \bar{y}, \bar{w} (F_2(\bar{w}) \rightarrow G_2(\bar{w}[\bar{z}/\bar{y}]))$

These steps correspond to the Subgoal proof scheme. Now, to carry out step 3, a structural induction proof is appropriate. The induction argument is $w$, which will normally be a search stack.

The well-founded ordering on the stack will typically be a lexicographical ordering on the sequence of elements in the stack (starting from the bottom of the stack). The lexicographical ordering is induced by the well-founded ordering on the stack elements, which corresponds to the order in which the nodes of the search space will be visited, with the greatest nodes visited first. This is well-founded provided that the search space is finite.

Forward moves reduce the stack with respect to the lexicographical ordering since the top element is replaced by smaller elements. Backward moves reduce the stack with respect to the ordering since the top element is either replaced by a smaller one or removed. Hence $F_2(w)$ has an inductive structure based on the ordering on its stack argument $w$.

To return to step 3, the formula to be proved is

$$\forall \bar{z}, \bar{y}, \bar{w} (F_2(\bar{w}) \rightarrow G_2(\bar{w}[\bar{z}/\bar{y}]))$$

Following the induction proof scheme, with induction parameter $w$, the steps of the proof are as follows.

Base case: Show that $\forall (w = 1) \rightarrow (F_2(w) \rightarrow G_2(w)[\bar{z}/\bar{y}])$

Induction hypothesis: $\forall w Ind(w) \equiv \forall \bar{y}, \bar{w}, \bar{w}' (w < w \rightarrow (F_2(\bar{w}') \rightarrow G_2(\bar{w}')[\bar{z}/\bar{y}]))$

Induction step: Show that $\forall (w \neq 1) \rightarrow (Ind(w) \rightarrow (F_2(w) \rightarrow G_2(w)[\bar{z}/\bar{y}]))$

5 General Scheme to Prove Correctness

Suppose that we have formulated a logic specification for an SI-program. First, the specification is transformed into a structured form. Then, the proof scheme corresponding to the design schema of the top-level predicate is applied.

The structured form of a specification is built around the structural cases of a parameter as follows. Let $\tau$ be a type and let $x/\tau$. Suppose that there exists a finite set of constructors $f_1/n_{11} \ldots f_k/n_{1k}$ such that for all $x/\tau$, $x = f_1(y_1, \ldots, y_{n_{11}}) \lor \ldots \lor x = f_k(y_1, \ldots, y_{n_{1k}})$. The specification of a predicate $p$ consists of a disjunction of subformulas $F_1(x) \lor \ldots \lor F_2(x)$. Each subformula $F_i(x)$ $(1 \leq i \leq d)$ has the form $S_i \land G_i$, where $S_i$ is of the form $x = f_j(y_1, \ldots, y_{n_{j}})$ $(1 \leq j \leq k)$ and $G_i$ is either a conjunction of literals or a disjunction of conjunctions of literals. In addition, $x$ can be used in the correctness proofs as the induction parameter.

The structured forms of logic specifications in the next examples are derived from the corresponding initial logic specifications shown in Section 3.

Example 1: Logic specification in structured form for predicate $\text{sum}(q, s)$ (Spec theory):

$$\forall q/\text{seq}(Z), s/Z (\text{sum}(q, s) \rightarrow (q = <> \land s = 0) \lor \exists t, h/\text{seq}(Z), t/\text{seq}(Z) (q = h :: t \land s = h + s1 + \text{sum}(t, s1)))$$

Example 2: Logic specification in structured form for predicate $\text{inrecOrd}(q)$:

$$\forall q/\text{seq}(t) (\text{inrecOrd}(q) \rightarrow \exists h, h1/\text{seq}(t), t/\text{seq}(q) (q = <> \lor (q = h :: t \land h \leq h1 :: t1 \land \text{inrecOrd}(t])))$$

A proof scheme is a systematic plan of proof actions. The proof scheme of the schema which has been applied for the instantiation of the top-level predicate is initially followed. The proof proceeds in a top-down fashion. If a program has nested instances of schemata the correctness proof may require the proof of some correctness theorems for predicates in lower levels. For such correctness theorems the proof scheme of the schema which has been applied for the instantiation of the predicate in the correctness theorem is followed. In this way, the structure of the SI-programs can be exploited when finding a proof. It is assumed that the implementations of the DT operations satisfy their logic specifications. Otherwise the correctness of the DT specifications would have to be proved as well.

The individual steps of the proof schemes are proved by performing equivalence preserving transformations. Let us assume that the correctness theorem has the form $\forall z/\tau (F_1 \rightarrow G_1)$ at some point during the transformation. Let us assume that equivalence preserving transformations have been performed on either or both of the formulas $F_1$ and $G_1$, i.e., $F_1 \rightarrow F_2$ and $G_1 \rightarrow G_2$. The correctness theorem is transformed by these transformations into an equivalent form. That is, $\forall z/\tau (F_1 \rightarrow G_1) \equiv \forall z/\tau (F_2 \rightarrow G_2)$. The process is continued until a formula $\forall z/\tau (F \rightarrow F)$ is reached.

The proof schemes given above generally provide sufficient conditions for proving correctness. They are not intended to be complete.
6 Examples

Two examples of correctness proofs are discussed in this section. Details about the proofs are not shown due to space limitations [1]. The logic specification $Spec$ of the first example is first transformed into its structured form $Spec$ then its correctness theorem is proved with respect to $Spec$'. The logic specification of the second example is already in structured form. The Subgoal proof scheme is immediately applicable to the correctness theorem.

Example 1: The logic specification $Spec$ for the predicate $sum^2(q,s)$ shown in Section 3 is transformed into its structured form $Spec$ shown in Section 5. Next, the $Spec$ theory is replaced by theory $Spec$'. The Incremental proof scheme is followed in order to prove its correctness theorem. That is, $\forall s/\exists q/\forall z/(sum(q,s) -> sum^2(q,s))$.

Example 2: The predicate isEven($s$) where $Type(isEven) = seq(a)$ is true iff the length of the sequence s is even.

$\forall s/\forall n/(length(seq(s,n) \land even^2(n)) \rightarrow 3n/N(length(seq(s,n) \land even^2(n)))$

In order to establish the above equivalence the next correctness theorem has to be proved. That is, $\forall s/\forall n/N(length(seq(s,n) \land even^2(n)) \rightarrow 3n/N(length(seq(s,n) \land even^2(n)))$

In addition, types enhance the expressive power of specifications and they facilitate the correctness proofs.

Correctness proofs are guided by the constructed SI-programs. That is, each schema is associated with a correctness proof scheme. The structure of an SI-program is reflected in the structure of its correctness proof. First, the correctness proof scheme that corresponds to the schema that has been applied for the construction of the top-level predicate is initially followed. Next, correctness proof schemes for schemata that have been applied for the construction of predicates in lower levels are followed. Correctness proofs for predicates in lower levels of an SI-program are used as lemmas for correctness proofs of predicates in higher levels of the program. Finally, correctness proofs are performed in a top-down manner, as the refinements in the program development process.

As further research we see the implementation of a semi-automatic verifier based on this correctness method. This verifier will enhance our development method [1], [2].

7 Conclusions

The main contribution of this paper on correctness is the guidance which is provided to the correctness proofs. In addition, types enhance the expressive power of specifications and they facilitate the correctness proofs.

References


