Correctness of logic programs using proof schemes

Emmanouil Marakakis* and Nikos Papadakis

Sciences Department, Technological Educational Institute of Crete, Heraklion, Greece

Abstract. The correctness of logic programs which are constructed by a schema-based method is presented in this paper. This schema-based method constructs typed, moded logic programs by stepwise top-down design using five program schemata, data types and modes. Correctness proofs in this approach are guided by the constructed logic programs. A proof scheme is proposed for each program schema. It is claimed that the structure of a logic program constructed by this schema-based method is reflected in the structure of its correctness proof. The proof schemes which correspond to design schemata are followed in the correctness proofs of logic programs.

Keywords: Correctness of logic programs, specifications, transformations, proof schemes, program schemata

1. Introduction

This paper presents a method for proving correctness of logic programs which are constructed by a schema-based method [16]. This logic program construction method is also introduced.

This schema-based method constructs typed, moded logic programs by stepwise top-down design using five program schemata, data types (DTs) and modes [15, 17]. A program is constructed by this method by successively refining undefined predicates. The lowest refinement level involves refinement by DT operations. Modes in this method support the application of the refinement operations. The programs which are constructed by this method are polymorphic many-sorted programs. They satisfy the head condition and the transparency condition [10] which ensure that no runtime type checking is needed. Finally, the programs satisfy declared input-output modes when run using the standard left-right depth-first computation rule. The logic programs which are constructed by this schema-based method are called Schema-Instance Programs or SI-Programs.

There are two approaches to the problem of correctness of a program. One approach is to test the program by a finite set of tests. Testing can demonstrate that a program is free of bugs and behaves as expected for a given finite set of tests. For complex input/output it is hard to know whether it is correct or not. Testing cannot show complete absence of bugs. The other approach is to prove that a program is correct with respect to its specification by formal reasoning.

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 the construction of the program [5, 6, 9, 11, 12]. In this case, a specification is successively refined into a program. The construction process consists 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 [1,3,4,7].

A practical correctness method should offer guidance to the user. We follow the second approach because we would like our correctness method to offer guidance to the user. The knowledge for guidance can be gained mainly from the program itself and its construction method. Our correctness method benefits from the advantages of the construction method of logic programs in order to guide the user in the proof of correctness of logic programs. An interactive logic program verifier has been constructed based on this logic program cor-

*Corresponding author. E-mail: mmarak@cs.teicrete.gr.
rectness method [18]. In this approach, first a program is constructed by using this schema-based method 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 other words, there are proof schemes corresponding to design schemas which are followed in the correctness proofs. 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 schemas.
3. The structure of an SI-program is reflected in the structure of its correctness proof.
4. 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.

This paper is organized as follows. Related work is discussed in Section 2. Section 2 presents a comparative discussion of our method with other methods and discusses its advantages. The correctness of non-trivial size logic programs can be shown by our method. These logic programs are constructed by the development method in [15,17].

In Section 3, an overview of the underlying schema-based method is presented. In Section 4, the correctness method is presented. In Section 5, the proof schemes which correspond to each program schema are proposed. In Section 6, the application of proof schemes is discussed. In Section 7, an example of correctness proof is shown. Finally, conclusions are discussed in Section 8.

2. Related work

In the literature, there are several approaches which verify logic programs using first-order logic (FOL) as a specification language [1,3–5,7,11,12].

In these approaches, a specification consists of a set of FOL formulas each formula of which expresses the specification of one relation. Each relation may be specified in terms of other relations down to the level of primitive components. A formal specification in FOL is called in this paper logic specification.

A logic specification in our method essentially follows these approaches, but uses typed FOL with equality. Types are an important component of specifications. The nature of the knowledge involved in specifications is typed. Untyped logic specifications are not expressive enough for specifying large software systems. Popular specification languages whose aim is the specification of complex software use typed logic [14,19].

Specifications in [3,5,11,12] are expressed as formulas in FOL. Correctness of the constructed programs is ensured by construction. A method for formal development of logic programs from a FOL specification of a relation, a program schema and the inductive argument of the relation is presented in [11]. This development method produces programs which are partially correct with respect to a FOL specification. It is shown in [11] that the completed definition of the implemented relation is complete with respect to the implemented specification which is stronger than the actual specification. Program clauses in [11] are derived from the corresponding partially instantiated schema clauses by performing equivalence rewriting of formulas. The individual steps of the schemes of our approach are proved by equivalence preserving transformations as well. The approaches in [3,5,12] transform a specification into a program by correctness preserving transformations. Logic programs are constructed in [3] by deriving each Horn clause as a theorem from the specification theory of the relation it is supposed to compute. The constructed programs are partially correct by construction. A top-down method for synthesizing recursive logic programs by transforming FOL specifications is described in [12]. Partial correctness of the derived logic programs is guaranteed by construction. Logic programs are derived in [5] from a subset of the extended Horn clause subset of logic. The derived programs are guaranteed to be totally correct wrt the initial specification.

A specification theory in [1,4] consists from axioms defining the relations plus a theory of data types. In these approaches, first a program is constructed then its correctness is shown. Total correctness of definite programs is shown in [1]. Correctness theorems are shown by structural induction. A set of Horn clauses is derived in [4] from the axiomatic definition of the input-output relation of an algorithm. An induction schema is defined for each data structure which is used for proving correctness theorems for that data structure. The correctness of recursive predicates in our method is also proved by structural induction on a well-founded set.

The method in [8] constructs logic algorithms (LAs) which are specified by sets of examples and properties.
Verification is discussed with respect to these sets of examples and properties. A LA is complete with respect to a set of examples if the examples are covered by the relation defined by the LA. A similar criterion applies for the set of properties. Specifications which are based on examples have the advantage of simplicity but on the other hand the selection of an appropriate set of examples is not simple. Such specifications are not complete even if properties are used as additional specification aid.

The method in [6] constructs logic descriptions (LDs). The construction of LDs in [6] as the structured logic specifications of our approach are based on the structural cases of an argument whose type is defined inductively. It is shown in [6] that LDs are correct by construction and the construction process contains an implicit correctness proof.

A more recently work [7] suggests a declarative approach for proving correctness and completeness of normal logic programs. The semantics of normal programs is 3-valued completion semantics. In our approach we follow the standard [2,13] program completion semantics but we don’t add negative unit clauses for undefined predicates. The method in [7] refers to two specifications; one for correctness and one for completeness. Specifications in [7] are interpretations while in our approach specifications are FOL theories. The operational property of logic program termination is also discussed which does not influence correctness and completeness.

In summary, the novel features of our approach with respect to the approaches discussed in this section are the following. Our correctness method benefits from the design of logic programs. The user has guidance in order to show the correctness of a logic program. Our correctness method proposes a small set of proof schemes whose selection depends on design decisions taken during program development. These design decisions are depicted in the program structure.

3. Overview of our schema-based method

This section presents an overview of our schema-based method. Typed logic programs are constructed by this method [15,17]. The underlying language is many sorted logic with parametric polymorphism [10]. The alphabet of the language contains disjoint infinite sets $V, F, P, A, C$ representing respectively, variables, functions, predicate symbols, parameters and constructors. Functions of arity 0 are constants and constructors of arity 0 are called basic types.

3.1. Data types

A data type is a set of data objects or values. Each type is associated with a number of operations which are applied to the values of the type. A DT in this method is considered to include both the set of values and the associated set of operations, each with its mode and type signature. The operations are formally defined by FOL specifications.

In our language, types are denoted by type terms (or simply types). Type terms are defined similarly to first-order terms. That is, parameters correspond to individual variables, basic types correspond to constants and type constructors to functions.

Definition 1. Let $\alpha$ be a parameter. Let $B$ and $C$ stand for basic types and type constructors of arity $\geq 1$, respectively. A type is defined inductively as follows: A parameter $\alpha$ is a type. A basic type $B$ is a type. If $C$ is a type constructor of arity $n \geq 1$ and $\tau_1, \ldots, \tau_n$ are types, then $C(\tau_1, \ldots, \tau_n)$ is a type.

This method provides a set of DTs called built-in DTs. A programmer can define his own DTs called user-defined DTs. The available built-in DTs are sequences, sets, multisets (bags), tuples and the basic ones $Z$ (integers), $N$ (naturals), $Q$ (rationals) and strings. User-defined DTs are specified in terms of built-in DTs or previously defined user-defined DTs. The DT operations of user-defined DTs are implemented by using this method.

3.2. Program design schemata

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

Definition 2. An atom schema is a formula of the form $P(u_1, \ldots, u_n)$ where $P$ is a predicate variable of arity $n$ and $u_1, \ldots, u_n$ are distinct schema argument variables. A literal schema has the form $A$ or $\neg A$, where $A$ is an atom schema.

Definition 3. A clause schema has the form $A \leftarrow L_1 \land \ldots \land L_m$, where $A$ is an atom schema and $L_i$ (1 $\leq i \leq n$) are literal schemata.
In our language, the following notation denotes the type of a predicate \( p \) of arity \( n \). Type \((p)\): \( t_1 \times \ldots \times t_n \) where \( t_1, \ldots, t_n \) are the types of first, \ldots, \( n \)-th argument of \( p \) respectively. The same notation is used for predicate variables. Let \( P \) be a predicate variable of arity \( n \), its type is denoted as follows. Type \((P)\): \( \alpha_1 \times \ldots \times \alpha_n \) where \( \alpha_1, \ldots, \alpha_n \) are parameters which denote the types of first, \ldots, \( n \)-th argument of \( P \) respectively.

**Definition 4.** Let \( P \) be a predicate variable of arity \( n \). A type schema for \( P \), denoted by \( Type(P) \), is of the form \( Type(P) = \alpha_1 \times \ldots \times \alpha_n \) where \( \alpha_i \) (\( 1 \leq i \leq n \)) are parameter variables.

**Definition 5.** A typed clause schema is a clause schema \( C \) together with a type schema for each predicate variable occurring in \( C \), such that each schema argument variable in \( C \) has the same schema type wherever it occurs.

**Definition 6.** A typed procedure schema is a set of typed clause schemata with the same predicate variable in the head of each clause.

**Definition 7.** A typed program schema is a set of typed procedure schemata, with one distinguished predicate variable called the top procedure variable. The top predicate variable appears in the head of one procedure schema which is called the top procedure schema.

The modes which are used in this method are \( i \) and \( d \). They stand for “input” and “don’t know” respectively. \( i \) identifies the arguments which are expected to be ground. These terms are characterized as “input”. An argument with any instantiation can be assigned mode \( d \).

**Definition 8.** A mode schema for predicate variable \( P \) of arity \( n \), denoted by \( Mode(P) \), is \( Mode(P) = m_1, \ldots, m_n \) where each \( m_i \) (\( 1 \leq i \leq n \)) is either \( i \) or \( d \).

The mode schema for a predicate variable represents the expected use of the arguments. It is used during schema refinement as a heuristic to partition the arguments into two groups.

**Definition 9.** 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 designed 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. This feature makes this method practical and flexible. We find this small set of schemata surprisingly expressive for constructing logic programs. Mode schemata and modes are not relevant to this discussion because of that they are omitted.

The symbols in the schemata have the following meaning. 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 is parameter variable and it stands for parameters. The lower case Greek letter \( \tau \) possibly subscripted is type variable and it stands for arbitrary types.

### 3.2.1. Incremental program design schema

The Incremental schema assumes that the input data is an inductively defined type \( \tau \). The inductive definition of \( \tau \) 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.

**Type schema**

- \( Type(Incr) \): \( \alpha_1 \times \alpha_2 \)
- \( Type(Terminating) \): \( \alpha_3 \)
- \( Type(Initial_{result}) \): \( \alpha_1 \times \alpha_2 \)
- \( Type(Deconstruction) \): \( \alpha_1 \times \alpha_3 \times \alpha_1 \)
- \( Type(Non_{initial}_{result}) \): \( \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_2 \)

**Clause schemata**

- \( Incr(u_1, u_2) \) \iff \( Terminating(u_1) \land Initial_{result}(u_1, u_2) \)
- \( Incr(u_1, u_2) \) \iff \( \neg Terminating(u_1) \land Deconstruction(u_1, v_1, v_2) \land Incr(v_2, v_3) \land Non_{initial}_{result}(u_1, v_1, v_2, u_2) \)

### 3.2.2. Divide-and-conquer program design schema

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.
3.2.3. Subgoal program design schema

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.

Type schemata

\[ Type(SubGoal) : \alpha_1 \times \alpha_2 \]
\[ Type(SubGoal1) : \alpha_1 \times \alpha_3 \times \alpha_2 \]
\[ Type(SubGoal2) : \alpha_1 \times \alpha_3 \times \alpha_2 \]
\[ \cdots \]
\[ Type(SubGoalN) : \alpha_1 \times \alpha_3 \times \alpha_2 \]

Clause schemata

\[ SubGoal(u_1, u_2) \leftarrow SubGoal1(u_1, v_1, v_2) \land \]
\[ SubGoal2(u_1, v_1, v_2) \land \]
\[ \cdots \]
\[ SubGoalN(u_1, v, u_2) \]

3.2.5. Search program design schema

A state of a problem is a particular configuration of the problem representation. The space of states of a problem are all possible configurations of a problem representation. Operations manipulate the problem representation. That is, they transform one state into another. The space of states is represented implicitly by a graph or tree. The Search schema performs search in the space of states of a problem. The search starts from the initial state, new states are produced by applying operations to it, next operations are applied again to these states producing new states and so on. As the search proceeds a tree is constructed which is called the search tree. If there is no operation to be applied or if it is understood that an incorrect operation has been performed then control can backtrack to a previous node of the search tree and try another operation. The Search schema 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.

Type schemata

\[ Type(Search) : \alpha_1 \times \alpha_2 \]
\[ Type(Initial\_state\_solution) : \alpha_1 \times \alpha_3 \times \alpha_2 \]
\[ Type(Search1) : \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_1 \]
\[ Type(Search\_termination) : \alpha_3 \times \alpha_2 \]
\[ Type(Solution\_assignment) : \alpha_3 \times \alpha_2 \]
\[ Type(Apply\_operation) : \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_3 \times \alpha_2 \]
\[ Type(Backtracking) : \alpha_3 \times \alpha_2 \times \alpha_3 \times \alpha_2 \]

Clause schemata

\[ Search(u_1, u_2) \leftarrow Initial\_state\_solution(u_1, v_1, v_2) \land \]
\[ Search1(u_1, v_1, v_2, u_2) \leftarrow \]
\[ Search\_termination(u_1, v_1, v_2) \land \]
\[ Solution\_assignment(v_1, v_2) \]
\[ Search1(u_1, v_1, v_2, u_2) \leftarrow \]
\[ \neg Search\_termination(u_1, v_1, v_2) \land \]
\[ Apply\_operation(u_1, v_1, v_2, v_3, v_4) \land \]
\[ Search1(u_1, v_3, v_4, u_2) \]
\[ Search1(u_1, v_1, v_2, u_2) \leftarrow \]
\[ \neg Search\_termination(u_1, v_1, v_2) \land \]
\[ \neg Apply\_operation(u_1, v_1, v_2, \ldots) \land \]
\[ Backtracking(v_1, v_2, v_3, v_4) \land \]
\[ Search1(u_1, v_5, v_6, u_2) \]

3.3. The construction process

Let \( \text{Prog} \) be a normal logic program. Let \( p \) be an n-ary predicate symbol in \( \text{Prog} \). The predicate \( p/n \) is
called an undefined predicate if there are no clauses whose heads contain the predicate symbol \( p \). The definition (or procedure) of the predicate \( p/n \) is the subset of clauses of \( \text{Prog} \) consisting of all clauses with the predicate symbol \( p \) in their heads.

**Definition 10.** A refinement is defined to be either of the following actions. The definition of an undefined predicate by instantiating a design schema. Such refinements are called schema refinements. The definition of an undefined predicate by a DT operation or equality or the negation of one of these. Such refinements are called DT refinements.

**Definition 11.** Let \( p/n \) be a predicate with type \( \text{Type}(p) = \tau_1 \times \ldots \times \tau_n \). Let \( p(x_1, \ldots, x_n) \) be an atom where \( x_1, \ldots, x_n \) are distinct variables. \( p(x_1, \ldots, x_n) \) is called typed completely general atom (cga) for predicate \( p/n \).

**Definition 12.** Let \( p(x_1, \ldots, x_n) \) be a typed cga of predicate \( p/n \). Let \( q(y_1, \ldots, y_k) \) be a typed cga of either a DT predicate or equality predicate. Basic clauses are clauses which have one of the next forms.

- \( p(x_1, \ldots, x_n) \leftarrow q(y_1, \ldots, y_k) \), or
- \( p(x_1, \ldots, x_n) \leftarrow \lnot q(y_1, \ldots, y_k) \)
  where \( k \leq n \), and \( \{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_n\} \).
- \( p(x_1, \ldots, x_n) \leftarrow eq(y, d) \), or
- \( p(x_1, \ldots, x_n) \leftarrow \lnot eq(y, d) \)
  where \( y \) is a variable, \( d \) is a constant and \( y \in \{x_1, \ldots, x_n\} \).

**Definition 13.** Let \( \text{Prog} \) be a normal logic program constructed by this method. Its set of clauses is \( S_1 \cup S_2 \cup S_3 \), where \( S_1 \) consists of instances of schema, \( S_2 \) consists of basic clauses and \( S_3 \) consists of definitions of DT operations and equality. We call \( \text{Prog} \) a Schema-Instance program or SI-program.

The construction process involves the application of a sequence of refinements. Let \( p/n \) be an undefined predicate with type \( \tau_1 \times \ldots \times \tau_n \) and mode \( m_1, \ldots, m_n \). Let \( p(x_1, \ldots, x_n) \) be a typed 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 cga(s) 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 [15].

### 3.4. An example

Let \( \text{sum}/2 \) be an undefined predicate with Type(sum) = \( \text{seq}(\alpha_1) \times \alpha_1 \) and expected mode \( \text{Mode}(\text{sum}) = (i, c) \). The predicate \( \text{sum}(q, s) \) is true iff \( s \) is the sum of integers in sequence \( q \). Let us assume that a programmer wants to construct a program for \( \text{sum}/2 \). Let \( \text{sum}(q, s) \) be a typed cga. The constructed SI-program is as follows.

**Signatures**

- \( \text{tail} : \text{seq}(\alpha_1) \times \text{seq}(\alpha_1) \)
- \( \text{neutral_add_subr} : \text{int} \times \text{int} \times \text{int} \)
- \( \text{head} : \text{seq}(\alpha_1) \times \alpha_1 \)
- \( \text{plus} : \text{int} \times \text{int} \times \text{int} \)
- \( \text{sum} : \text{seq}(\text{int}) \times \text{int} \)
- \( \text{p0} : \text{seq}(\text{int}) \)
- \( \text{p1} : \text{seq}(\text{int}) \times \text{int} \)
- \( \text{p2} : \text{seq}(\text{int}) \times \text{int} \times \text{int} \)
- \( \text{p3} : \text{seq}(\text{int}) \times \text{int} \times \text{int} \times \text{int} \)
- \( \text{p5} : \text{seq}(\text{int}) \times \text{int} \times \text{int} \)

**Constructed program clauses**

\[
\begin{align*}
\text{sum}(q, s) & \leftarrow \text{p1}(q) \land \text{p2}(q, s) \\
\text{sum}(q, s) & \leftarrow \lnot \text{p1}(q) \land \text{p3}(q, h, t) \land \text{sum}(t, s1) \land \text{p4}(q, h, s1, s) \\
\text{p1}(q) & \leftarrow \text{empty_seq}(q) \\
\text{p2}(q, s) & \leftarrow \text{neutral_add_subr}(s) \\
\text{p3}(q, h, t) & \leftarrow \text{p5}(q, h, t) \land \text{p6}(q, h, t) \\
\text{p5}(q, h, t) & \leftarrow \text{head}(q, h) \\
\text{p6}(q, h, t) & \leftarrow \text{tail}(q, t) \\
\text{p4}(q, h, s1, s) & \leftarrow \text{plus}(s1, h, s)
\end{align*}
\]

**Refinements**

1. Refinement 1: Incremental schema is applied to \( \text{sum}/2 \).
2. Refinement 2: The DT predicate \( \text{empty_seq} \) refines \( \text{p1}/1 \).
3. Refinement 3: The DT predicate \( \text{neutral_add_subr} \) refines \( \text{p2}/2 \).
4. Refinement 4: Subgoal schema is applied to \( \text{p3}/3 \).

**Definitions of DT Operations and Equality Predicates**

- empty_seq[](0)
- neutral_add_subtr_int(0)
- head([h | t], h)
- tail([h | t], t)
- plus_int(x1,x2,x3) ← x3 is x1 + x2

4. Correctness method

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

**Definition 14.** 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

\[
Spec_p = \forall \vec{x}/\vec{\tau} \ (p^S(\vec{x}) \iff \text{Def}_p)
\]

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

The following notation is used in the rest of the paper. \(<\to\) stands for the empty sequence. \( \cdot \) stands for the term constructor for sequences, i.e. \( x \cdot q \) is a sequence with head \( x \) and tail \( q \). \#q stands for the length of sequence \( q \). \( \sum \) is the summation operator for all numeric data types. \( x \preceq_y y \) stands for the less-than or equal comparison of unspecified type elements \( x \) and \( y \). \( x, \ldots, y/\tau \) stands for \( x/\tau, \ldots, y/\tau \).

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

**Logic specification:**

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

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

**Logic specification:**

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

where \( N_1 = N - \{0\} \).

The meaning of SI-programs is defined using program completion semantics. Our definition of completion is slightly different from the standard one in [2,13]. We do not add in the program completion negative unit clauses for undefined predicates in program \( Pr \) 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 15.** Let \( Pr \) be an SI-program, excluding the definitions of the DT operations. 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 which do not appear in its head. The clause \( p(x_1, \ldots, x_n) \leftarrow \exists y_1 \ldots \exists y_d (L_1 \land \ldots \land L_m) \) is equivalent to the previous one. Suppose that there are \( r \) such clauses for predicate \( p/n \)

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

where each \( E_i(1 \leq i \leq r) \) stands for a corresponding formula of the form \( \exists y_1 \ldots \exists y_d (L_1 \land \ldots \land L_m) \). Then the completed definition of the predicate \( p/n \) is the formula

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

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 \( p_1, \ldots, p_k \).

**Definition 16.** Let \( Pr \) be an SI-program, excluding the DT definitions. Let \( \mathcal{A} \) 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.

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

**Example 3.** Let \( Pr \) be the SI-program of the predicate \( \text{sum}/2 \) shown in Section 3 without the definitions of the DT operations. Its completion, \( \text{comp}(Pr) \), is as follows.

\[
\forall q/\text{seq}(Z), s/Z \ (\text{sum}(q,s) \iff (p1(q) \land p2(q,s)) \lor (\exists h/Z, t/\text{seq}(Z), s1/Z \ (\neg p1(q) \land p3(q,h,t) \land \text{sum}(t,s1) \land p4(q,h,s1,s)))))
\]

\[
\forall q/\text{seq}(Z) \ (p1(q) \iff \text{empty}(\text{seq}(q)))
\]

\[
\forall q/\text{seq}(Z), s/Z \ (p2(q,s) \iff \text{neutral_add_subtr_int}(s))
\]

\[
\forall q/\text{seq}(Z), h/Z, t/\text{seq}(Z) \ (p3(q,h,t) \iff p5(q,h,t) \land p6(q,h,t))
\]
∀q/seq(Z), h/Z, t/seq(Z) (p5(q, h, t) ↔ head(q, h))
∀q/seq(Z), h/Z, t/seq(Z) (p6(q, h, t) ↔ tail(q, t))
∀q/seq(Z), h, s1, s/Z (p4(q, h, s1, s) ↔ plus_int(s1, h, s))

The theory A of the DT operations including the specification of the DT operations is as follows.

Axioms
A1 Domain closure axiom for sequences
∀s/seq(α) (s =<> ∀∃h/α, t/seq(α) s = h :: t)
A2 Uniqueness axioms for sequences
i. ∀h/α, t/seq(α) (¬(h :: t =<>))
ii. ∀h1, h2/α, t1, t2/seq(α) (h1 :: t1 = h2 :: t2 → (h1 = h2 ∧ t1 = t2))
A3 Definition of summation operation over 0 entities
∀s/seq(Z) (s =<> → ∑#s i=1 s_i = 0)

Lemmas
L1 ∀s/seq(α) (s ≠ <> ↔ ∃h/α, t/seq(α) s = h :: t)
L2 ∀h/α, s, t/seq(α) (s = h :: t → ∀i/N (2 ≤ i ≤ #s → s_i = t_i-1))
L3 ∀s, t/seq(α), h/α (s = h :: t → #s = #t + 1)
L4 ∀h/α, s, t/seq(α) (s = h :: t → h = s1)

Logic specifications of DT operations
∀q/seq(α) (empty_seq(q) ↔ q =<>)
∀s/Z (neutral_add_subtr_int(s) ↔ s = 0)
∀q/seq(α), h/α (head(q, h) ↔ q ≠ <> ∧ ∃h/α seq(α) q = h :: t)
∀q, t/seq(α) (tail(q, t) ↔ q ≠ <> ∧ ∃h/α q = h :: t)
∀s1, h, s/Z (plus_int(s1, h, s) ↔ s = h + s1)

Definition 17. The specification Spec of an SI-program with top level predicate p is a set of formulas including one of the form
∀¯x/¯y(p₅(¯x) ↔ Def_p)

Definition 18. 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 p̃ does not occur in comp(Pr). Pr is partially correct with respect to Spec if
Spec ∪ A ∪ comp(Pr) |=
∀¯x/¯y (p(¯x) → p₅(¯x))
Pr is complete with respect to Spec if
Spec ∪ A ∪ comp(Pr) |=
∀¯x/¯y (p(¯x) → p₅(¯x))

Example 4. The correctness theorem for predicate sum/2 and the theory which is used to prove it is as follows.
Spec ∪ A ∪ comp(Pr) |= ∀q/seq(Z), s/Z (sum(q, s) ↔ sum₅(q, s))

5. Schemata-based proof schemes
A proof scheme is a systematic plan of proof actions. Each design schema is associated with a correctness proof scheme. In the following, variables are not accompanied by types for simplicity of presentation.

5.1. Proof scheme for subgoal schema
If the Subgoal schema has been applied to construct the predicate p, then the correctness proof is of the form
∀¯x (p(¯x) ↔ Def_p)
where ∀¯x (p(¯x) ↔ ∃y (F₁ ∧ ... ∧ Fₖ)) and F₁, ..., Fₖ 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 Def_p as an equivalent formula of the form ∃z (G₁ ∧ ... ∧ Gₖ) where z is of the same type as y in formula ∃y (F₁ ∧ ... ∧ Fₖ).
2. Try to prove ∀¯x, ¯y (F₁ ↔ G₁[¯x/¯y], ... ∀¯x, ¯y (Fₖ ↔ Gₖ[¯x/¯y]).

5.2. Proof scheme for case schema
If the Case schema has been applied to construct the predicate p, then correctness proof is of the form
∀¯x (p(¯x) ↔ Def_p)
where \( \forall \bar{x} \ (p(\bar{x}) \leftrightarrow \exists \bar{y} \ (F_1 \lor \ldots \lor F_k)) \) and \( F_1, \ldots, F_k \) are first-order formulas. It is assumed that the arguments \( \bar{x} \) 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 \( \text{Def}_p \) as an equivalent formula of the form \( \exists x \ (G_1 \lor \ldots \lor G_k) \) where \( x \) is of the same type as \( y \) in formula \( \exists y \ (F_1 \lor \ldots \lor F_k) \).
2. Try to prove \( \forall \bar{x}, \bar{y} \ (F_1 \Rightarrow G_1[\bar{z}/\bar{y}]), \ldots, \forall \bar{x}, \bar{y} \ (F_k \Rightarrow G_k[\bar{z}/\bar{y}]) \).

5.3. Proof scheme for incremental and divide-and-conquer schemata

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

**Definition 19.** Let \( < \) be a binary relation. A partially-ordered set \( (S, <) \) in which every nonempty subset has a least element is called a well-founded set.

Let \( (S, <) \) be a well-founded set and \( P \) be a proposition over \( S \). If the following two conditions hold,

1. Base case: \( P(x) \) is true for each minimal element of \( S \).
2. Induction step: \( \forall x \in S \ (\forall y \in S \ (y < x \rightarrow P(y)) \rightarrow P(x)) \)

then infer that \( \forall z \in S \ P(z) \). This is the principle of structural induction.

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

Let \( \forall \bar{v} \ (p(\bar{u}, \bar{v}) \rightarrow \text{Def}_p) \) be the correctness theorem to be proved, abbreviated by \( \Phi(\bar{u}) \). \( P(x) \) in the structural induction schema can now be instantiated by \( \Phi(\bar{u}) \) where \( u \) is of type \( \tau \), and a well-founded ordering \( < \) 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 scheme show how the proof can proceed.

In order to prove \( \forall \bar{u}, \bar{v} \ (p(\bar{u}, \bar{v}) \rightarrow \text{Def}_p) \) where \( p(\bar{u}, \bar{v}) \) has been defined by an Incremental or Divide and conquer schema, perform the following steps. In the following, \( \perp \) represents the bottom element of the well-founded set \( (S, <) \), i.e. there is no \( \perp \in S \) such that \( \perp < \perp \).

**Base case:** \( \forall \bar{u}, \bar{v} \ (u = \perp \rightarrow (p(\bar{u}, \bar{v}) \rightarrow \text{Def}_p)) \)

**Induction hypothesis:** \( \forall u \ \text{Ind}(u) \equiv \forall u, u', \bar{v} \)

\( (u' < u \rightarrow (p(u', \bar{v}) \rightarrow \text{Def}_p)) \)

**Induction step:** \( \forall u, \bar{v} \ (u \not= \perp \land \text{Ind}(u) \rightarrow (p(u, \bar{v}) \rightarrow \text{Def}_p)) \)

5.4. Proof scheme for search schema

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

\( \forall \bar{x} \ (p(\bar{x}) \leftrightarrow \exists \bar{y}, w(F_1 \land F_2(w))) \)

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 \( \forall \bar{x} (p(\bar{x}) \leftrightarrow \text{Def}_p) \) where the variables \( \bar{x} \) are identical to \( \bar{x} \) in the formula defining \( p(\bar{x}) \) given above. Then try the following steps.

1. Try to transform \( \text{Def}_p \) into an equivalent formula \( \exists z, w' (G_1 \land G_2(w)) \) where \( w' \) is of the same type as \( w \).
2. Try to prove \( \forall \bar{x}, \bar{y} (F_1 \leftrightarrow G_1[\bar{z}/\bar{y}]) \)
3. Try to prove \( \forall \bar{x}, \bar{y}, \bar{w} (F_2(w) \leftrightarrow G_2(w)[\bar{z}/\bar{y}]) \)

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 search 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

\( \forall \bar{x}, \bar{y}, \bar{w} (F_2(w) \leftrightarrow G_2(w)[\bar{z}/\bar{y}]) \)
Following the induction proof scheme, with induction parameter \( w \), the steps of the proof are as follows.

**Base case:** Show that \( \forall (w = \bot \rightarrow (F_2(w) \leftrightarrow G_2(w) \bar{\gamma}/\bar{\delta})) \)

**Induction hypothesis:**
\[
\forall w \ (\text{Ind}(w) \equiv \forall w, w', \exists w' < w \rightarrow (F_2(w') \leftrightarrow G_2(w') \bar{\gamma}/\bar{\delta}))
\]

**Induction step:** Show that \( \forall w \ (w \neq \bot \rightarrow \text{Ind}(w) \rightarrow (F_2(w) \leftrightarrow G_2(w) \bar{\gamma}/\bar{\delta})) \)

6. Application of the proof schemes

Let us assume that \( Pr \) is an SI-program excluding the definitions of the DT operations. The correctness of the constructed SI-programs requires the establishment of the equivalence \( \forall x/\tau (p(x) \leftrightarrow \text{Def} f_y) \) in the theory comp(\( Pr \)) \( \cup \text{Spec} \cup A \). The correctness proof method for establishing this equivalence is guided by the design schemata that have been applied to construct the logic program \( Pr \) in a top-down fashion. It is assumed that the implementations of the DT operations satisfy their logic specifications. Otherwise the correctness of the DT operations with respect to their specifications would have to be proved as well.

6.1. Structured form of specifications

A logic specification for predicate \( p^S(x) \) can be expressed in several different ways. Which specification is appropriate for proving the correctness of the constructed SI-program? It is well-known that correctness proofs even for small programs are complex and long. A possible way to overcome this problem is by expressing logic specifications in a structured form suitable for applying the proof schemes. A form which often facilitates the correctness proof of an SI-program is the structured form of \( p^S(x) \).

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 \( f_1/n_1, \ldots, f_k/n_k \) such that for all \( x/\tau \), \( x = f_1(y_1, \ldots, y_{n_1}) \vee \ldots \vee x = f_k(z_1, \ldots, z_{n_k}) \). The specification of a predicate \( p \) consists of a disjunction of sub-formulas \( F_i(x) \vee \ldots \vee F_d(x) \). Each sub-formula \( F_i(x) (1 \leq i \leq d) \) 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 4.

**Example 5.** Logic specification in structured form for predicate \( \text{sum}^S(q, s) \):
\[
\forall q/\text{seq}(Z), s/Z (\text{sum}^S(q, s) \leftrightarrow (q =<> \land s = 0) \lor \exists s, h/Z, t/\text{seq}(Z) (q = h :: t \land s = h + s1 \land \text{sum}^S(t, s1)) \)
\]

**Example 6.** Logic specification in structured form for predicate \( \text{incrOrd}^S(q) \):
\[
\forall q/\text{seq}(\alpha) (\text{incrOrd}^S(q) \leftrightarrow \exists h, h1/\alpha, t, t1/\text{seq}(\alpha) (q =<> \lor (q = h :: t \land (t =<> \lor (t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t1)))) \)
\]

The first stage in a correctness proof is often to transform a specification into a structured form. The derivation of the structured logic specification for \( \text{sum}^S(q, s) \) from the initial one is shown in Section 7.

6.2. General scheme to prove correctness

Suppose that we have formulated a logic specification for an SI-program. How will we proceed to prove the correctness of the program with respect to its specification? The following steps have to be performed.

1. Try to manipulate the specification into a structured form.
2. Try to apply the proof scheme corresponding to the design schema applied to the top-level predicate.

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 SI-programs can be exploited when finding a proof.

**Example 7.** The predicate \( \text{isEven}(s) \) where \( \text{Type} (\text{isEven}) = \text{seq}(\alpha) \) is true iff the length of the sequence \( s \) is even.

**Logic specification Spec:**
\[
\forall s/\text{seq}(\alpha) (\text{isEven}^S(s) \leftrightarrow \exists n/N
(\#s = n \land \text{even}^S(n)))
\]
\[
\forall n/N (\text{even}^S(n) \leftrightarrow (n = 0 \lor \exists n1/N (n = \text{succ(\text{succ}(n1)))})
\]
The logic specification of the predicate isEven^5(s)/2 is already in structured form. The Subgoal proof scheme is immediately applicable to the correctness theorem, that is, \( \forall s/seq(\alpha) \ (isEven(s) \leftrightarrow isEven^5(s)) \). Its correctness theorem is transformed into the following form at some stage during the proof.

\[
\forall s/seq(\alpha) \ (\exists n/N \ (length \_ seq(s, n) \wedge p2(s, n)) \leftrightarrow \exists n/N (length \_ seq(s, n) \wedge even^5(n)))
\]

length_\_seq/2 is a DT operation for sequences. length_\_seq(s, n) is true if n is the length of sequence s.

In order to establish the above equivalence the following correctness theorem has to be proved, \( \forall s/seq(\alpha), n/N (p2(s, n) \leftrightarrow even^5(n)) \). This correctness theorem is used as lemma in the previous proof. The logic specification of the predicate even^5(s)/2 is already in structured form as well. The structural cases of \( n/N \) are \( n = 0, n = succ(0) \) (or \( \exists m/N (n = succ(m) \wedge m = 0) \)) which is omitted because it is false and \( \exists n1/N (n = succ(succ(n1))) \) (or \( \exists n1, m/N (n = succ(m) \wedge m = succ(n1)) \)). Incremental proof scheme is applied to this correctness theorem because the clauses of the predicate \( p2(s, n) \) with \( Type(p2) = seq(\alpha) \times N \) are an instance of Incremental schema. The well-ordering is based on \( n/N \).

Note that the argument s/seq(\alpha) of predicate p2/2 is unused. This schema-based method creates unused arguments. This does not cause any problem for correctness. It is better to perform correctness proof before removal of unused arguments because the structure of an SI-program reflects the structure of the applied schemata (design decisions). On the other hand, if removal of unused arguments is performed before correctness proof it will result in loss of useful structural information which is used by the correctness method.

The theories \( \text{comp}(Pr) \) and \( A \) which, in addition to Spec theory, are used to prove the above theorems are the following.

1. Axioms – Lemmas – Specifications of DT operations (A):

   **Axioms**

   **A1** Domain closure axiom for naturals
   \[
   \forall n/N \ (n = 0 \lor \exists m/N \ (n = succ(m)))
   \]

   **A2** Uniqueness axioms for naturals
   i. \( \forall n/N \ (\neg(succ(n) = 0)) \)
   ii. \( \forall n1, n2/N \ (succ(n1) = succ(n2) \rightarrow n1 = n2) \)

   **A3** \( \forall n/N \ (n > 0 \leftrightarrow \neg(n = 0)) \)

   **A4** \( \forall n/N \ (n = 0 + n) \)

   **A5** Transitivity of equality
   \[
   \forall n/N, m/N, k/N (n = m \land m = k \rightarrow n = k)
   \]

   **A6** Domain closure axiom for sequences
   \[
   \forall s/seq(\alpha), n/N (p1(s, n) \leftrightarrow p2(s, n))
   \]

   **A7** Uniqueness axioms for sequences
   i. \( \forall h/\alpha, t/seq(\alpha) \ (\neg(h : t = <>)) \)
   ii. \( \forall h1, h2/\alpha, t1, t2/seq(\alpha) (h1 :: t1 = h2 :: t2 \rightarrow (h1 = h2 \land t1 = t2)) \)

**Lemmas**

**L1** \( \forall n/N (n \neq 0 \rightarrow \exists m/N (n = succ(m)) \leftrightarrow m = pred(n)) \)

**L2** \( \forall n/N (n \neq 0 \rightarrow \exists m/N (m = pred(n)) \rightarrow n > m) \)

**L3** \( \forall n/N \ (\exists k, m/N (n = succ(k) \land k = succ(m)) \rightarrow n > 1) \)

**L4** \( \forall n, m, k/N (n > k \land m = n - k \rightarrow m > 0) \)

**L5** \( \forall n/N \ (n > 1 \rightarrow n > 0) \)

**L6** \( \forall n/N \ (n \neq 0 \rightarrow \exists m/N \ (n = succ(m))) \)

**L7** \( \forall s, t/seq(\alpha), h/\alpha \ (s = h :: t \rightarrow \#s = \#t + 1) \)

**L8** \( \forall s/seq(\alpha) (s = <> \rightarrow \#s = 0) \)

**L9** \( \forall m/N, n/M (m + 1 = n \rightarrow m = n - 1) \)

**Logic specifications of DT operations**

\[
\forall q/seq(\alpha), n/N (length \_ seq(q, n) \leftrightarrow ((q = <> \land n = 0) \lor \\
\exists h/\alpha, t/seq(\alpha) (q = h :: t \land \\
\#q = \#t + 1 \land \#t = n - 1)))
\]

\[
\forall n1, n2/N (succ \_ nat(n1, n2) \leftrightarrow \\
\neg n2 = n1 + 1)
\]

\[
\forall n, n1/N (pred \_ nat(n1, n) \rightarrow n > 0 \land \\
n1 = n - 1)
\]

\[
\forall n1, n2/N (gt \_ nat(n1, n2) \rightarrow n1 > n2)
\]

2. Completion of the logic program of isEven(s) (comp(Pr)):

\[
\forall s/seq(\alpha) (isEven(s) \leftrightarrow \\
\exists n/N (p1(s, n) \land p2(s, n)))
\]

\[
\forall s/seq(\alpha), n/N (p1(s, n) \leftrightarrow length \_ seq(s, n))
\]

\[
\forall s/seq(\alpha), n/N (p2(s, n) \leftrightarrow \exists m/N (\neg succ(n) \\
\land m = succ(n2) \rightarrow n1 = n2))
\]

\[
\forall s/seq(\alpha), n/N (p3(s, n) \leftrightarrow \\
p4(s, n) \lor \exists h/\alpha, t/seq(\alpha) (n = succ(\neg n1) \\
\land \neg t = n1 \land \neg \neg t = n1))
\]

\[
\forall s/seq(\alpha), n/N (p5(s, n) \rightarrow n = 0)
\]

\[
\forall s/seq(\alpha), n/N (p4(s, n) \leftrightarrow true)
\]

\[
\forall s/seq(\alpha), n/N, v/N, s_1/N, s_2/seq(\alpha), n1/N \ (p5(s, n, v, s_1, n1) \leftrightarrow \\
true)
\]
The correctness theorem is proved by performing equivalence preserving transformations. Let us assume that the correctness theorem has the form \( \forall x \exists y \) (Section 4). The correctness theorem is transformed by these transformations in an equivalent form. That is,

\[
\forall x \exists y (F_1 \iff G_1) \equiv \forall x \exists y (F_2 \iff G_2)
\]

The process is continued until a formula \( \forall x \exists y (F \iff G) \) is reached.

6.3. Equivalence preserving transformations

No suggestions were made in the proof schemes above for proving the individual steps of the schemes. The correctness theorem is proved by performing equivalence preserving transformations. Let us assume that the correctness theorem has the form \( \forall x \exists y (F_1 \iff G_1) \) at some point during the transformation. Let us assume that equivalence preserving transformations have been performed on either or on both of the formulas \( F_1 \) and \( G_1 \), i.e. \( F_1 \iff F_2 \) and \( G_1 \iff G_2 \). The correctness theorem is transformed by these transformations into an equivalent form. That is,

\[
\forall x \exists y (F_1 \iff G_1) \equiv \forall x \exists y (F_2 \iff G_2)
\]

The process is continued until a formula \( \forall x \exists y (F \iff G) \) is reached.

7. An example of correctness proof

The correctness proof of the predicate \( \text{sum}/2 \) is shown in this example. The correctness theorem and the theorem for \( \text{sum}/2 \) is presented in Section 4. The structured logic specification \( \text{Spec}' \) is shown in Section 6. Initially, the logic specification \( \text{Spec} \) of \( \text{sum}/2(q, s) \) is transformed into its structured form \( \text{Spec}' \). Next, the \( \text{Spec} \) is replaced by the theory \( \text{Spec}' \) and the Incremental proof scheme is followed in order to prove its correctness theorem.

1. Transform logic specification into structured form.

\[
\forall q/\text{seq}(Z), s/Z (\text{sum}^S(q, s))
\]

\[
\begin{align*}
\forall q/\text{seq}(Z), s/Z (\text{sum}^S(q, s)) & \iff s = \sum_{i=1}^{#q} q_i \\
& \quad \bullet \text{by axiom A1 and by FOL laws} \\
& \quad (\text{if } q = \langle \rangle \text{ then } P \land Q \iff P) \\
& \quad (q = \langle \rangle \lor \exists h/Z, t/\text{seq}(Z)) \\
& \quad q = \langle \rangle \land s = \sum_{i=1}^{#q} q_i \\
& \quad \bullet \text{by FOL law } (\lor \text{ distribution}) \\
& \quad (q = \langle \rangle \lor \exists s = \sum_{i=1}^{#q} q_i) \\
& \quad \bullet \text{by axiom A3} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad t/\text{seq}(Z) \lor \exists h/Z, t/\text{seq}(Z) \\
& \quad q = \langle \rangle \land s = \sum_{i=1}^{#q} q_i \\
& \quad \bullet \text{by definition of } \sum \text{ operator} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad t/\text{seq}(Z) \lor \exists h/Z, t/\text{seq}(Z) \\
& \quad q = \langle \rangle \land s = q_1 + \sum_{i=2}^{#q} q_i \\
& \quad \bullet \text{by introduction of new variable } s_1/ \\
& \quad \text{and by FOL laws} \\
& \quad (\forall x (P(x) \iff q_y(x = y \land P(y)))) \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad \exists s_1, h/Z, t/\text{seq}(Z) (q = \langle \rangle \land s = q_1 + s_1) \\
& \quad \bullet \text{by lemmas L2 and L3} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad \exists s_1, h/Z, t/\text{seq}(Z) (q = \langle \rangle \land s = q_1 + s_1) \\
& \quad \bullet \text{by lemma L4} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad \exists s_1, h/Z, t/\text{seq}(Z) (q = \langle \rangle \land s = h + s_1 + \text{sum}^S(t, s_1))
\end{align*}
\]

2. Proof: Incremental proof scheme

Structural induction is applied to the correctness theorem. \( \forall q_1, q_2/\text{seq}(q_1) q_1 < q_2 \text{ if } \#q_1 < \#q_2 \). The set \( (\text{seq}(q), \langle \rangle) \) is well-founded.

Induction base: \( q = \langle \rangle \)

\[
\forall q/\text{seq}(Z) (q = \langle \rangle \iff \forall s/Z (\text{sum}(q, s))
\]

\[
\begin{align*}
& \forall s/\text{seq}(Z) (q = \langle \rangle \iff \forall s/Z (\text{sum}(q, s)) \\
& \quad \bullet \text{by FOL law } (\lor \text{ distribution}) \\
& \quad (q = \langle \rangle \lor \exists s = \sum_{i=1}^{#q} q_i) \\
& \quad \bullet \text{by axiom A3} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad t/\text{seq}(Z) \lor \exists h/Z, t/\text{seq}(Z) \\
& \quad q = \langle \rangle \land s = \sum_{i=1}^{#q} q_i \\
& \quad \bullet \text{by definition of } \sum \text{ operator} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad t/\text{seq}(Z) \lor \exists h/Z, t/\text{seq}(Z) \\
& \quad q = \langle \rangle \land s = q_1 + \sum_{i=2}^{#q} q_i \\
& \quad \bullet \text{by introduction of new variable } s_1/ \\
& \quad \text{and by FOL laws} \\
& \quad (\forall x (P(x) \iff q_y(x = y \land P(y)))) \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad \exists s_1, h/Z, t/\text{seq}(Z) (q = \langle \rangle \land s = q_1 + s_1) \\
& \quad \bullet \text{by lemmas L2 and L3} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad \exists s_1, h/Z, t/\text{seq}(Z) (q = \langle \rangle \land s = q_1 + s_1) \\
& \quad \bullet \text{by lemma L4} \\
& \quad (q = \langle \rangle \lor s = 0) \\
& \quad \exists s_1, h/Z, t/\text{seq}(Z) (q = \langle \rangle \land s = h + s_1 + \text{sum}^S(t, s_1))
\end{align*}
\]
\[ \text{seq}(Z), s / Z (\text{sum}(q,s)) \]
\[ \forall q / \text{seq}(Z), s / Z (\text{sum}(q,s)) \]
\[ \left\{ \begin{array}{l}
q = <\rightarrow \land s = 0 \lor \\
\exists s_1, h / Z, t / \text{seq}(Z) (q = h :: t \land s = h + s_1 \land \text{sum}^3(t,s_1))
\end{array} \right. \]
\[
\circ \text{because of the induction base} \\
(q = <\rightarrow) \text{ and by axiom A2.1}
\]
\[ \left\{ \begin{array}{l}
q = <\rightarrow \land s = 0 \lor \\
\exists s_1, h / Z, t / \text{seq}(Z) (\text{false} \land s = h + s_1 \land \text{sum}^3(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land \text{false} \Rightarrow \text{false}) \\
\circ \text{by FOL law} (P \lor \text{false} \Rightarrow P) \\
q = <\rightarrow \land s = 0)
\]
\[
\circ \text{because of the logic specifications of DT operations empty_seq1 and neutral_add/subtr_int1/1}
\]
\[ \left\{ \begin{array}{l}
\text{empty_seq}(q) \land \\
\text{neutral_add/subtr_int}(s)
\end{array} \right. \]
\[
\circ \text{because of the completion of p1/1 and p2/2} \\
\left\{ \begin{array}{l}
q1(q) \land p2(q,s)
\end{array} \right. \]
\[
\circ \text{because of the completion of sum}/2 \\
\circ \left\{ \begin{array}{l}
\text{sum}(q,s)
\end{array} \right. \\
\circ \text{by lemma L1 and by FOL law} \\
(\text{if } P \Rightarrow Q \text{ then } P \land Q \Rightarrow P)
\]
\[ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land s = h + s_1 \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL laws} (P \land P \Rightarrow P, \land \text{commutativity}) \\
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
q = <\rightarrow \land q \neq <\rightarrow \land q = h :: t \land s = h + s_1 \land \text{sum}(t,s_1))
\]
\[
\circ \text{because of the logic specification of the DT operation head/2} \\
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{head}(q,h) \land s = h + s_1 \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by the logic specifications of the DT operation tail/2} \\
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{head}(q,h) \land s = h + s_1 \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{tail}(q,t) \land s = h + s_1 \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{plus}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{plus}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
\[
\circ \text{by FOL law} (P \land P \Rightarrow P) \\
\circ \left\{ \begin{array}{l}
\exists s_1, h / Z, t / \text{seq}(Z) (q \neq <\rightarrow \land q = h :: t \land \text{sum}(t,s_1) \land \text{sum}(t,s_1))
\end{array} \right. \]
8. Conclusions

The main contribution of this paper on correctness is the guidance which is provided to the correctness proofs. Correctness proofs are guided by the constructed SI-programs. That is,

1. Each schema is associated with a correctness proof scheme.
2. The structure of an SI-program is reflected in the structure of its correctness proof.

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.

References