A Temporal Approach to Specification and Verification of Pointer Data-Structures*

Marcin Kubica

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland
fax: +48 22 5544400
kubica@mimuw.edu.pl

Abstract. We present a formalism for specification of pointer data-structures and programs operating on them, based on temporal specifications of dynamic algebras. It is an extension of first-order logic with temporal branching-time combinators. The use of this formalism is illustrated by examples. We also propose a Hoare-style calculus for verification of while-programs (operating on pointers) against specifications written in the proposed formalism, which is sound and complete in the sense of Cook.

1 Introduction

It is well known that pointer data-structures are widely used in implementation of efficient algorithms [4]. It is also well known that they are one of the main sources of programming errors. Therefore there is a need for formal verification of programs operating on pointers. On the other hand, pointers seem to be discriminated in the specification methods. There have been many attempts to incorporate pointers into the framework of Hoare logic (e.g. [2,3,11]). However, these cases are focused on verification of first-order specifications. They have limited applicability since first-order logic cannot express such basic properties as connectivity or existence of a cycle (cf. [17], Sect. 4). In case of [2], where the assertion language is a variant of second-order logic, specifications of common pointer data-structures are far from straightforward. We can also find specification methods based on monadic second-order logic [12,13] (granting decidability of many issues arising during verification), however their use concentrates on trees and lists.

We exploit the same analogy between pointer data-structures, graphs and transitive systems as in [10], where vertices are represented by sets of traces and assertions are second-order formulae on sets of traces. However, the formalism for specification of pointer data-structures proposed here is different. It is an adaptation of temporal specifications of dynamic algebras [5]—an extension of first-order logic with temporal branching-time combinators. In our approach we

* This research was supported by the Polish Scientific Research Committee (KBN) under grant 8T11C 01515.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 231–245, 2003.

interpret the notion of direct consequence in time as a pointer link. Hence, e.g., a possible history is an analog to a path of pointer links. We show on examples, that the presented formalism is concise and comprehensive. We also show a Hoare-style calculus for the verification of while-programs against specifications written in the presented formalism, which is sound and complete in the sense of Cook [14].

We assume that the reader is familiar with many-sorted algebras with partial functions (in short partial algebras) [7,18]. We allow overloading of function and variable names. Whenever it can cause ambiguities, the arity of a function symbol, or the sort of a variable is indicated. Whenever we use equality it is a strong equality, i.e. $t_1 = t_2$ is true iff both t_1 and t_2 are defined and their values are equal. The following notational convention is also used. Formula $t\downarrow$ is an abbreviation of $t = t$, and $t_1 \equiv t_2$ is an abbreviation of $t_1 = t_2 \vee (t_1 \neq t_1 \wedge t_2 \neq t_2)$. In other words, $t\downarrow$ means that the value of t is defined, and $\equiv$ is a weak equality, $t_1 \equiv t_2$ means that either both t_1 and t_2 are defined and their values are equal, or they are both undefined. If f is a (partial) function, then $f[a \rightarrow b]$ denotes such a (partial) function that $f[a \rightarrow b](a) = b$ and $\forall_{x \neq a} f(x) \equiv f[a \rightarrow b](x)$.

In Sect. 2 we model states and changes of pointer data-structures by partial many-sorted algebras. The formalism for specification of pointer data-structures and programs operating on them is described in Sect. 3. Section 4 contains definitions of syntax and semantics of while-programs. Hoare-style calculus for verification of while-programs operating on pointer data-structures is presented in Sect. 5. Section 6 contains final conclusions and remarks.

2 Modeling Data-Structures

In this section we describe how to model states and changes of states of pointer data-structures, using many-sorted partial algebras. We use Pascal, as an example programming language. Since we focus on common pointer data-structures, we consider only built-in types, records and pointer types. Our approach can be easily extended by arrays and enumeration types, however we omit them for simplicity of presentation.

The starting point of our considerations is the declaration of data-types. We do not allow complex declarations, i.e., data-type constructions cannot be nested. It allows us to identify all declared types by their names. We avoid type hiding. The interpretation of built-in types is not defined here, however we assume that it is known. We assume also that type `Boolean`, with its standard interpretation, is among built-in types. For the sake of simplicity we assume that the only operations allowed on built-in types are functions (without side-effects) and assignments. Let $\Sigma_B = \langle S_B, F_B \rangle$ be a fixed signature and B be a fixed partial Σ_B-algebra. We assume that sorts from B represent values of built-in types and functions from B model primitive operations on built-in types.

Definition 1. *Let D be a Pascal declaration of data-types of the form*

$$\texttt{Type } t_1 \texttt{ = } d_1\texttt{; } \ldots \; t_n \texttt{ = } d_n\texttt{;}$$

For $i = 1, \ldots, n$, if there exists such j that t_j is a type of pointers to t_i, then we denote t_j by $\uparrow t_i$ (if there are several such j, then we chose the smallest one). If D does not contain any type of pointers to t_i, then we treat $\uparrow t_i$ as a name of a new sort of pointers to t_i.

By $Sig(D) = \langle S, F \rangle$ we denote such a signature that:

$$S = S_B \cup \{t_1, \ldots, t_n, \uparrow t_1, \ldots, \uparrow t_n, \uparrow b_1, \ldots, \uparrow b_k\}, \quad F = F_B \cup \bigcup_{i=1,\ldots,n} f_i \ ,$$

where $S_B = \{b_1, \ldots, b_k\}$ and f_i is defined in the following way:

- *if t_i is a type of pointers to a non-record type u, then*

$$f_i = \{\_\uparrow : t_i \to u, \mathtt{Nil} :\to t_i\} \ ,$$

- *if t_i is a type of pointers to a record type u composed of the fields $p_1 : u_1, \ldots, p_l : u_l$, then*

$$f_i = \{\_\uparrow : t_i \to u, \mathtt{Nil} :\to t_i, \_|p_1 : t_i \to \uparrow u_1, \ldots, \_|p_l : t_i \to \uparrow u_l\} \ ,$$

- *if t_i is a record type composed of the fields $p_1 : u_1, \ldots, p_l : u_l$, then*

$$f_i = \left\{ \begin{array}{c} \_.p_1 : t_i \to u_1, \ldots, \_.p_l : t_i \to u_l, \\ (p_1 = \_, \ldots, p_l = \_) : u_1 \times \ldots \times u_l \to t_i \end{array} \right\} \ .$$

We use partial $Sig(D)$-algebras to model states of data-structures declared by D. Addresses of variables (of type t) are represented by values of an appropriate pointer type ($\uparrow t$). If D does not contain declaration of such a type, then a new sort named $\uparrow t$ is introduced. Function symbols in $Sig(D)$ have the following intuitive meaning:

- `Nil` is a constant representing a null pointer,
- $\_\uparrow$ returns value pointed to by a pointer; this function is defined for addresses of allocated variables,
- $\_.p$ returns the value of the field p of a record,
- $(p_1 = \_, \ldots, p_l = \_)$ constructs record values from values of record fields,
- $\_|p$ returns the address of the field p of a given record.

Our intention is to model values of programming variables, by functions $\_\uparrow$. From here on, let D be the fixed type declarations. By *type* we mean any of the following sorts of $Sig(D)$: $b_1, \ldots, b_k, t_1, \ldots, t_n$.

Example 1. Let us consider the type declaration of singly-linked lists:

```
Type list = ↑elem;
     elem = record d: data; next: list end;
```

Let us denote these declarations by L. For the sake of simplicity we assume that `data` is a built-in type. $Sig(L)$ contains (among others):

- sorts: `list`, `elem`, `data`, ↑`list`, ↑`data`,

– function symbols:

$\_.\mathtt{d} : \mathtt{elem} \to \mathtt{data}$ $\quad \_\uparrow : \mathtt{list} \to \mathtt{elem}$ $\quad \mathtt{Nil} :\to \mathtt{list}$
$\_|\mathtt{d} : \mathtt{list} \to \uparrow\mathtt{data}$ $\quad \_\uparrow : \uparrow\mathtt{list} \to \mathtt{list}$ $\quad \mathtt{Nil} :\to \uparrow\mathtt{list}$
$\_.\mathtt{next} : \mathtt{elem} \to \mathtt{list}$ $\quad \_\uparrow : \uparrow\mathtt{data} \to \mathtt{data}$ $\quad \mathtt{Nil} :\to \uparrow\mathtt{data}$
$\_|\mathtt{next} : \mathtt{list} \to \uparrow\mathtt{list}$ $\quad (\mathtt{data} = \_, \mathtt{list} = \_) : \mathtt{data} \times \mathtt{list} \to \mathtt{elem}$

We are not interested in all partial $Sig(D)$-algebras as models of states of data-structures. We can restrict our considerations to partial algebras satisfying some natural conditions regarding pointers and records.

Definition 2. *By $DStr(D)$ we denote a class of all partial $Sig(D)$-algebras A satisfying the following conditions:*

– *for each record type t composed of fields $p_1 : u_1, \ldots, p_l : u_l$, we have:*
  - *functions returning values of fields ($\_.p_i$) are total,*
  - *function $(p_1 = \_, \ldots, p_l = \_)$ constructing record values is total,*
  - *a set of record values is isomorphic with the Cartesian product of sets of values of its fields,*
– *for each sort t of pointers to u the function $\_\uparrow : t \to u$ is undefined for* `Nil`*,*
– *for each sort t of pointers to a record type r composed of fields $p_1 : u_1, \ldots, p_l : u_l$ we have:*
  - *function $\_|p_j : t \to \uparrow u_j$ (for $j = 1, \ldots, l$) is defined for all pointers different from* `Nil`*,*
  - *if $x : t$ is an address of a record, then function $\_\uparrow : \uparrow u_j \to u_j$ (for $j = 1, \ldots, l$) is defined for $x|p_j$ iff x is the address of an allocated record, i.e. $\forall_{x:t}(x\uparrow)\downarrow \Leftrightarrow (x|p_j\uparrow)\downarrow$,*
  - *the value of a field of the record pointed to by a pointer is the same as the value pointed to by the pointer to the field, i.e. $\forall_{x:t}(x\uparrow)\downarrow \Rightarrow x\uparrow.p_j = x|p_j\uparrow$,*
  - *different fields of the same record, of the same type, have different addresses, i.e. for $1 \le j, k \le l$, $j \ne k$, $u_j = u_k$ we have $\forall_{x:t} x|.p_j \ne x|.p_k$,*
  - *fields (of the same type) of different records of the same type have different addresses, i.e. for $1 \le j, k \le l$, $u_j = u_k$ we have $\forall_{x \ne y:t} x \ne y \Rightarrow x|p_j \ne y|p_k$,*
– *fields (of the same type) of records, whose addresses belong to different sorts, have different addresses, i.e. for two different sorts t, u of pointers to record types, respectively, r composed of fields $p_1 : r_1, \ldots, p_l : r_l$, and s composed of fields $q_1 : s_1, \ldots, q_n : s_m$, and $1 \le j \le l$, $1 \le k \le m$, $r_j = s_k$ we have $\forall_{x:t \ne y:u} x|p_j \ne y|q_k$.*

Note that, for a given D, all conditions stated above can be expressed as a first-order formula.

Partial algebras from $DStr(D)$ can be used to model states of data-structures. One should remember, that one partial algebra models a single state of a data-structure. Our intention is to model such properties as data-structure invariants and preconditions of programs operating on pointer data-structures, with subclasses of $DStr(D)$. We should note, however, that the partial algebras from

$DStr(D)$ cannot model changes of data-structures, e.g. postconditions of programs. To do it, we need to express modification of $\_\uparrow$ functions. For this purpose, we extend $Sig(D)$ so that it contains two copies of $\_\uparrow$ functions—one related to the state of a data-structure before modification, and one after modification. We adopt here the notational convention of decorating symbols, similar to one used, for example, in the specification method Z [15]. Undecorated functions $\_\uparrow$ refer to the 'current' state of the data-structure, i.e. 'after' modification. Functions $\grave{\_}\uparrow$ refer to the 'previous' state of the data-structure, i.e. before modification. Later on, we use also other, auxiliary, decorations $\_\uparrow^{\circledast}$, $\_\uparrow^{\odot}$, etc. Let us denote by $\mathcal{D}$ a fixed infinite set of all decorations. For sake of simplicity we sometimes consider undecorated $\_\uparrow$ functions, as decorated with an empty word ε.

Definition 3. *Let Σ be a signature and $\circledast$ be a decoration. By $\Sigma^{\circledast}$ we denote a signature obtained from Σ by replacing all function symbols $\_\uparrow$ by $\_\uparrow^{\circledast}$. By convention $\Sigma^{[]} = \Sigma$.*

For a given $\Sigma_1 \supseteq \Sigma^{\circledast}$ we can treat $\circledast$ as a signature morphism $\circledast : \Sigma \to \Sigma_1$ renaming all $\_\uparrow$ functions to $\_\uparrow^{\circledast}$ and leaving all other function symbols and sort names unchanged. If φ is a formula and t is a term then $\varphi^{\circledast} = \circledast(\varphi)$ and $t^{\circledast} = \circledast(t)$.

Let $Y \subseteq \mathcal{D} \cup \{\varepsilon\}$ be a set of decorations. By Σ^Y we denote $\Sigma^Y = \bigcup_{[] \in Y} \Sigma^{[]}$. Let Σ_1 be a signature without $\grave{}$ decoration. By $\Delta\Sigma_1$ we denote the signature $\Sigma_1 \cup \grave{}\Sigma_1$. If $\circledast \in Y$ and A is a partial Σ^Y-algebra, then $A_{|\circledast}$ is a partial Σ-algebra with $\_\uparrow$ functions equal to their $\_\uparrow^{\circledast}$ equivalents from A.

Definition 4. *Let $Y \subseteq \mathcal{D} \cup \{\varepsilon\}$, $\varepsilon \in Y$. We extend the definition of DStr for $Sig(D)^Y$—by $DStr(Sig(D)^Y)$ we denote the class of such partial $Sig(D)^Y$-algebras A, that for all decorations $\circledast \in Y$ we have $A_{|\circledast} \in DStr(D)$.*

Partial algebras from $DStr(\Delta Sig(D))$ can be used to model changes of data-structures declared by D. One should keep in mind, however, that one partial algebra models one possible change of a data-structure. Our intention is to model properties of programs operating on pointer data-structures with subclasses of $DStr(\Delta Sig(D))$.

Definition 5. *Let $Y \subseteq \mathcal{D} \cup \{\varepsilon\}$, $\varepsilon \in Y$ and $A \in DStr(Sig(D)^Y)$. By $\overline{A}$ we denote the set of such partial algebras $A_1 \in DStr(Sig(D)^Y)$ that A_1 differs from A only in interpretation of function symbols of the form $\_\uparrow^{\circledast}$ (for $\circledast \in Y$). By $|\overline{A}|$ we denote $|A|$.*

Intuitively, $\overline{A}$ determines the sorts and interpretation of function symbols not representing states of data-structures.

Let $A \in DStr(\Delta Sig(D))$. We can model such properties as postconditions of programs by subsets of $\overline{A}$. Data-structure invariants and possible changes of a data-structure are modeled by subsets of $\overline{A_{|\grave{}}}$.

Dynamically allocated programming variables can be accessed only via pointers. Static and local programming variables can also be accessed by their names. We model allocation of these programming variables by variables valuations. For this purpose we adopt the following convention.

Definition 6. *Let* $Sig(D) = \langle S_{\mathbb{D}}, F_{\mathbb{D}} \rangle$, V *be a declaration of 'programming' variables of the following form* `Var` $x_1{:}t_1; \ldots x_n{:}t_n;$. *The set of variables induced by* V *is a many-sorted set of 'logical' variable names* $X = \langle X_s \rangle_{s \in S_{\mathbb{D}}}$, $X_s = \{\&x_i \mid \uparrow t_i = s\}$. *If the set of variables* X *is known from the context,* $\&x \in X_{\uparrow s}$, *then we write* x *as an abbreviation of* $\&x\uparrow$.

Intuitively, if x is a programming variable, then $\&x$ denotes the address of x. Note that the above convention allows us to describe the aliasing of programming variables, while we can write the name of a programming variable to reference its value. For the rest of this paper, let V be the fixed declaration of programming variables.

3 Temporal Specifications

Let us now consider the problem of specification of pointer data-structures and programs operating on them. It can be seen from Sect. 2, that this problem can be reduced (for given D, V, $\overline{A}$ and $v : X \to |\overline{A}|$) to specification of a subset of $\overline{A}$. Note that the first-order logic is too weak for this purpose, since it is known (cf. e.g. [17], Sect. 4) that we cannot express the notion of a path in it.

We often view pointer data-structures as graphs with labeled vertices and/or edges—records can be seen as vertices and pointers as edges. The analogy between such graphs, transitive systems and Kriepke models is obvious. Therefore it arises a natural question of usability of temporal logics for specification of pointer data-structures. The general idea of using temporal logics for the specification of abstract data-types is not new [5,8,16], however the idea of using it for the purpose of the specification of a pointer data-structure is novel.

One of the main notions appearing both in the specification of pointer data structures, and in transitive systems, is the notion of a path. In pointer data-structures a path is a sequence of records, which can be traversed by following pointer links. In transitive systems it represents a possible history of a computation. This analogy is also exploited in [10], where vertices are represented by sets of traces leading to them, and specifications are formulae on sets of traces.

In classical temporal logics, satisfaction of a temporal formula depends on the set of all possible paths (starting in a given state). In such a situation, it is not important, for example, whether two diverging paths cross again or not. On the contrary, in case of pointer data-structures, binary trees for example, it is crucial that for each record in a tree there is only one path leading to it from the root of the tree. Therefore classical temporal logics are of limited use for the specification of pointer data-structures. However, the formalism presented in [5] for the purpose of the specification of dynamic algebras does not have described disadvantages. Here we present its slightly modified version, adapted for our purposes.

Viewing a pointer data-structure as a graph, we often consider some of the pointers, abstracting from the rest of them. Usually we focus on pointers linking the records of a given type. In our approach, pointers are modeled by unary functions. We can describe edges representing selected pointers by a set of 'terms

with a hole'. Formally, the term with a hole is a term which can contain an additional, distinguished variable symbol '[]' called the hole. Such a term describes a set of edges in the following way: if we assign a source vertex to the hole and the value of the term is defined, then it is equal to the destination vertex. Since each record type is composed of a finite number of fields, there can be several, but limited, number of edges coming out of a vertex. Therefore, we can represent selected edges by a finite set of terms with a hole.

For the rest of this section, let $\Sigma = \langle S_{\square}, F_{\square} \rangle$ be a signature and $X = \langle X_s \rangle_{s \in S_{\square}}$ be a many-sorted set of variables. (We assume $[] \notin X$.)

Definition 7. *Let $s \in S_{\square}$. Any finite set e of terms with a hole $[] : s$, $e \subseteq T_{\square}(X \cup \{[] : s\})_s$ is called a description (over Σ) of edges between vertices of sort s. We denote the set of all such descriptions by $E_{\square}(X, s)$.*

Definition 8. *Let A be a partial Σ-algebra, $s \in S_{\square}$, $v : X \to |A|$ be a variables valuation, and $e \in E_{\square}(X, s)$. We denote by $Path(A, s, v, e)$ such a set of (finite or infinite) sequences $p = \langle p_0, p_1, \ldots \rangle \in |A|_s^+ \cup |A|_s^{\square}$ (called paths) that:*

- *for all $0 \leq k \leq length(p) - 2$ there exists such $t \in e$ that $(v[[] \to p_k])^A(t) = p_{k+1}$, and*
- *if $p = \langle p_0, \ldots, p_k \rangle$, then, for all $t \in e$, $(v[[] \to p_k])^A(t)$ is undefined.*

If $p \in Path(A, s, v, e)$, $p = \langle p_0, \ldots, p_k, \ldots \rangle$, then by $B(p)$ we denote the first element of p, $B(p) = p_0$, and by $p_{|k}$ we denote the result of removing the first k elements from p, $p_{|k} = \langle p_k, \ldots \rangle$.

We distinguish two kinds of formulae: dynamic formulae and path formulae. The latter ones, however, are used only as parts of dynamic formulae.

Definition 9. *Let $s \in S_{\square}$. The sets of dynamic formulae $F_{\square}(X)$ and path formulae $P_{\square}(X, s)$ are defined by the mutual induction:*

- *if $s_1 \in S_{\square}$, $t_1, t_2 \in T_{\square}(X)_{s_1}$, then $t_1 =_s t_2 \in F_{\square}(X)$,*
- *if $\varphi_1, \varphi_2 \in F_{\square}(X)$, then $\varphi_1 \Rightarrow \varphi_2, \neg\varphi_1 \in F_{\square}(X)$,*
- *if $s_1 \in S_{\square}$, x is an identifier and $\varphi \in F_{\square}(X \cup \{x : s_1\})$, then $\forall_{x:s_1}\varphi$, $\exists_{x:s_1}\varphi \in F_{\square}(X)$,*
- *if $t \in T_{\square}(X)_s$, $\{e_1, \ldots, e_k\} \in E_{\square}(X, s)$ and $\pi \in P_{\square}(X, s)$, then $\mathbf{A}_{t \langle e_1 \langle \ggg e_k} \pi$, $\mathbf{E}_{t \langle e_1 \langle \ggg e_k} \pi \in F_{\square}(X)$,*
- *if $\pi_1, \pi_2 \in P_{\square}(X, s)$, then $\pi_1 \Rightarrow \pi_2, \neg\pi_1 \in P_{\square}(X,)$,*
- *if $s_1 \in S_{\square}$, x is an identifier and $\pi \in P_{\square}(X \cup \{x : s_1\}, s)$, then $\forall_{x:s_1}\pi$, $\exists_{x:s_1}\pi \in P_{\square}(X, s)$,*
- *if $\varphi \in F_{\square}(X \cup \{x : s\})$, then $[\lambda x.\, \varphi] \in P_{\square}(X, s)$,*
- *if $\pi_1, \pi_2 \in P_{\square}(X, s)$, then $\pi_1 \mathcal{U} \pi_2 \in P_{\square}(X, s)$,*

The dynamic formulae are interpreted for a given variables valuation, and the path formulae are interpreted for a given variables valuation and a path. Both dynamic and path formulae can be build using first-order combinators (with standard interpretation). Dynamic formula $\mathbf{A}_{t \langle e_1 \langle \ggg e_k} \pi$ represents universal, and

$\mathbf{E}_{t\langle e_1\langle\!\langle\!\langle e_k}\pi$ represents existential quantification over paths starting in t and following edges from $e_1, \ldots, e_k$. In a path formula $[\lambda x.\, \varphi\,]$ variable x is assigned the first element of the path, over which it is interpreted. Operator $\mathcal{U}$ is a temporal operator 'until', interpreted along a given path.

Definition 10. *Let A be a partial Σ-algebra $v : X \to |A|$ be a variables valuation, $s \in S_{[]}$, $\{e_1, \ldots, e_k\} \in E_{[]}(X, s)$, $\sigma \in Path(A, s, v, \{e_1, \ldots, e_k\})$, $\varphi \in F_{[]}(X)$ and $\pi \in F_{[]}(X, s)$. The satisfaction of φ in A under v (written $A \models_v \varphi$), and the satisfaction of π in A under v and σ (written $A, \sigma \models_v \pi$) are defined by mutual induction:*

- $A \models_v t_1 = t_2$ *iff* $v^A(t_1)$ *and* $v^A(t_2)$ *are defined and* $v^A(t_1) = v^A(t_2)$,
- $A \models_v \varphi_1 \Rightarrow \varphi_2$ *iff* $A \models_v \varphi_1$ *implies* $A \models_v \varphi_2$,
- $A \models_v \neg\varphi$ *iff* $A \not\models_v \varphi$,
- $A \models_v \forall_{x:s_1}(\varphi)$ $(A \models_v \exists_{x:s_1}(\varphi))$ *iff* $A \models_{v[x\to a]} \varphi$ *for all (some)* $a \in |A|_{s_1}$,
- $A \models_v \mathbf{A}_{t\langle e_1\langle\!\langle\!\langle e_k}\pi$ $(A \models_v \mathbf{E}_{t\langle e_1\langle\!\langle\!\langle e_k}\pi)$ *iff* $v^A(t)$ *is defined, and for all (for some) paths* $\sigma \in Path(A, s, v, \{e_1, \ldots, e_k\})$ *(where s is a sort of t) such that* $B(\sigma) = v^A(t)$ *we have* $A, \sigma \models_v \pi$,
- $A, \sigma \models_v \pi_1 \Rightarrow \pi_2$ *iff* $A, \sigma \models_v \pi_1$ *implies* $A, \sigma \models_v \pi_2$,
- $A, \sigma \models_v \neg\pi$ *iff* $A, \sigma \not\models_v \pi$,
- $A, \sigma \models_v \forall_{x:s_1}\pi$ $(A, \sigma \models_v \exists_{x:s_1}\pi)$ *iff* $A, \sigma \models_{v[x\to a]} \pi$ *for all (some)* $a \in |A|_{s_1}$,
- $A, \sigma \models_v [\lambda x.\, \varphi\,]$ *iff* $A \models_{v[x\to B(\sigma)]} \varphi$,
- $A, \sigma \models_v \pi_1 \mathcal{U} \pi_2$ *iff there exists such* $j > 0$ *that* $\sigma_{|j}$ *is defined and* $A, \sigma_{|j} \models_v \pi_2$, *and for all* $0 < i < j$ *we have* $A, \sigma_{|i} \models_v \pi_1$.

We write $A \models \varphi$ iff $A \models_v \varphi$ for all $v : X \to |A|$.

We define boolean operators $\wedge$, $\vee$ and $\Leftrightarrow$ using $\Rightarrow$ and $\neg$ in a standard way. Similarly, we define temporal operators $\Box$ (everywhere on the path), $\Diamond$ (somewhere on the path) and $\mathcal{O}$ (next) using $\mathcal{U}$. We should stress out again that while using temporal analogies we do not consider possible histories, but we traverse a pointer data-structure along pointer links.

Let us consider the simple example of singly-linked lists.

Example 2. Let us recall type definitions from Example 1.

```
Type list = ↑elem;
     elem = record d:data; next:list end;
```

Heads of lists can be characterized by the following formula:

$$head(p) \Leftrightarrow (p\uparrow)\downarrow \wedge \forall_{q:\mathtt{list}}(q\uparrow.\mathtt{next} \neq p) \ .$$

The invariant of a list data-structure can be described as follows:

- each head of a list is pointed to by some external pointer:

$$\forall_{l:\mathtt{list}} head(l) \Rightarrow \exists_{p:\uparrow\mathtt{list}}(p\uparrow = l \wedge \forall_{q:\mathtt{list}} q|\mathtt{next} \neq p) \ ,$$

– each list terminates with a `Nil` pointer

$$\forall_{p:\mathtt{list}} head(p) \Rightarrow \mathbf{A}_{p\langle[]\Uparrow\mathtt{next}}\Diamond[\lambda x.\ x\ =\mathtt{Nil}])\ ,$$

– each allocated record appears on some list

$$\forall_{p:\mathtt{list}}(p\uparrow)\downarrow \Rightarrow \exists_{q:\mathtt{list}}(head(q) \wedge \mathbf{E}_{q\langle[]\Uparrow\mathtt{next}}\Diamond[\lambda x.\ x\ = p]))\ ,$$

– two separate lists do not intersect:

$$\forall_{l_1\langle l_2:\mathtt{list}}(l_1 \neq l_2 \wedge head(l_1) \wedge head(l_2) \Rightarrow \mathbf{A}_{l_1\langle[]\Uparrow\mathtt{next}}\Box[\lambda x.\ \mathbf{A}_{l_2\langle[]\Uparrow\mathtt{next}}\Box[\lambda y.\ x\ = y \Rightarrow x = \mathtt{Nil}]])\ .$$

Let us denote data-type invariant of lists, being the conjunction of the above conditions, by *list*. Let us consider a program replacing all values x appearing on a list p, with values y. We can specify this program by the following precondition:

$$list \wedge head(p) \wedge x\downarrow \wedge y\downarrow$$

and a postcondition:

$$\begin{aligned}\mathbf{E}_{\backprime p\langle[]\backprime\uparrow\Uparrow\mathtt{next}}(&\Box[\lambda q.\ q\backprime\uparrow.\mathtt{d} = \backprime x \Rightarrow q\uparrow.\mathtt{d} = \backprime y]\wedge\\ &\forall_{q:\mathtt{list}}((q\backprime\uparrow.\mathtt{d} \neq \backprime x \vee \Box[\lambda r.\ r\ \neq q]) \Rightarrow q\uparrow\equiv q\backprime\uparrow)\wedge\\ &\forall_{r:\uparrow\mathtt{data}}(\forall_{q:\mathtt{list}}\ q|\mathtt{d} \neq r) \Rightarrow r\uparrow\equiv r\backprime\uparrow) \wedge \psi\end{aligned}$$

where ψ is a formula stating that variables of any types other, then `elem` and `data` do not change.

Example 3. Doubly-linked cyclic lists can have the following type-declarations:

```
Type list = ↑elem;
     elem = record d: data; prev, next: list end;
```

The following data-type invariant describes possible shapes of lists, and expresses the fact, that each list is pointed to by an external pointer:

$$\begin{aligned}\forall_{p:\mathtt{list}}(p\uparrow)\downarrow \Rightarrow (&\mathbf{A}_{p\Uparrow\mathtt{next}\langle[]\Uparrow\mathtt{next}}\Diamond[\lambda x.\ x\ = p] \wedge \mathbf{A}_{p\Uparrow\mathtt{prev}\langle[]\Uparrow\mathtt{prev}}\Diamond[\lambda x.\ x\ = p] \wedge\\ &\exists_{q:\uparrow\mathtt{list}}\forall_{r:\mathtt{list}}(q \neq r|\mathtt{next}\ \wedge q \neq r|\mathtt{prev} \wedge\\ &\mathbf{E}_{q\uparrow\langle[]\Uparrow\mathtt{next}}\Diamond[\lambda x.\ x\ = p]) \wedge\\ &p\uparrow.\mathtt{next}\uparrow.\mathtt{prev} = p)\ .\end{aligned}$$

Example 4. Binary directed acyclic graphs (in short: binary DAGs) can have the following type declarations:

```
Type bdag = ↑ node;
     node = record x: data; l, r: bdag end;
```

All possible shapes of binary DAGs can be described by the following formula:

$$bDAG \Leftrightarrow \forall_{p:\mathtt{bdag}}((p\uparrow)\downarrow \Rightarrow \mathbf{A}_{p\langle[]\Uparrow\mathtt{l}\langle[]\Uparrow\mathtt{r}}\Diamond[\lambda x.\ x\ = \mathtt{Nil}])\ .$$

Binary trees can be seen as a sub-class of binary DAGs. Their possible shapes describes the following formula:

$$\begin{aligned}bDAG\ \wedge\ &\forall_{p\langle q\langle r:\mathtt{bdag}}(p\uparrow)\downarrow \wedge (q\uparrow.\mathtt{r} = p\ \vee\ q\uparrow.\mathtt{l} = p) \wedge\\ &(r\uparrow.\mathtt{r} = p\ \vee\ r\uparrow.\mathtt{l} = p) \Rightarrow q = r\ \wedge\\ &\forall_{p:\mathtt{bdag}}\ ((p\uparrow)\downarrow \wedge p\uparrow.\mathtt{l} = p\uparrow.\mathtt{p} \Rightarrow p\uparrow.\mathtt{l} = \mathtt{Nil})\end{aligned}\quad .$$

4 Programming Language

In this section we present a language of while-programs operating on pointer data-structures, and its semantics. For this section, let X be the set of variables induced by the declaration V.

Definition 11. *We denote by $Prog_D(V)$ the set of Pascal blocks of instructions (correct with respect to D and V) which can be derived from the following grammar:*

while-program → `begin` *instructions* `end`
instructions → *instruction* | *instruction* ; *instructions*
instruction → ε | `begin` *instructions* `end` | *designator* := *expression* | `New(` *designator* `)` | `while` *expression* `do` *instruction* | `if` *expression* `then` *instruction* `else` *instruction*
designator → *identifier* | *designator* . *identifier* | *expression* ↑ ,

where expression (after replacing each programming variable x by $\&x\uparrow$) is a term of an appropriate type.

Designators are also called l-values. They represent addresses of variables. We define a translation of designators to terms of appropriate pointer sorts. The translation gives us the semantics of designators.

Definition 12. *Let o be a designator of a variable of type s. By $\&(o)$ we denote a term, of a sort of pointers to type s, defined inductively in the following way:*

- *if o is a name of a programming variable, then $\&(o) = \&o$,*
- *if o has a form $o_1.p$, then $\&(o) = \&(o_1)|p$,*
- *if o has a form $o_1\uparrow$, then $\&(o) = o_1$.*

We can treat expressions appearing in a program as terms, and boolean expressions as logical formulae. However, we have to remember that if the value of a term is undefined, then the evaluation of an expression aborts the program execution. For this purpose we define a formula $OK(\alpha)$ which is true iff the evaluation of a boolean expression α is defined.

Definition 13. *Let α be a boolean expression. We denote by $OK(\alpha)$ a formula defined inductively in the following way:*

- *if $\alpha =$ `true` or $\alpha =$ `false`, then $OK(\alpha) = true$*
- *if $\alpha = (\alpha_1$ `and` $\alpha_2)$, then $OK(\alpha) = OK(\alpha_1) \wedge (\neg\alpha_1 \vee OK(\alpha_2))$—analogously in case of other boolean operators,*
- *if $\alpha = (t_1 = t_2)$, then $OK(\alpha) = t_1\downarrow \wedge t_2\downarrow$,*
- *otherwise $OK(\alpha) = \alpha\downarrow$.*

We define an auxiliary function returning a state of the data-structure obtained by storing, in a given state, a given value under a given address.

Definition 14. *Let $A \in DStr(D)$, t_1 be a sort of pointers to t_2, $a \in |A|_{t_1}$, $b \in |A|_{t_2}$. We denote by $Store(A, a, b)$ a partial algebra $A_1 \in DStr(D)$ obtained from A by storing value b under address a.*

Note that $\_\uparrow^{A_1}_{t_1 \to t_2} = \_\uparrow^{A}_{t_1 \to t_2} [a \to b]$. Moreover, if t is a record type containing values of type t_2 and u is a sort of pointers to t, then $\uparrow\colon u \to t$ is modified accordingly. Also, if t_2 is a record type containing values of type t, then $\uparrow\colon\uparrow t \to t$ is modified accordingly, etc.

We define semantics of while-programs operationally, as a function returning, for a given starting state of a data-structure and variables valuation, a set of possible final states of the data-structure.

Definition 15. *Let $P \in Prog_D(V)$, $A \in DStr(D)$, $v : X \to |A|$ be a variables valuation. We denote semantics of P by $[\![P]\!]_v(A)$ and define it by induction on the structure of P:*

- $[\![\texttt{begin } P \texttt{ end}]\!]_v(A) = [\![P]\!]_v(A)$,
- $[\![Q; R]\!]_v(A) = \overrightarrow{[\![R]\!]_v}([\![Q]\!]_v(A))$,
- $[\![\varepsilon]\!]_v(A) = \{A\}$,
- $[\![o \texttt{ := } e]\!]_v(A) = \begin{cases} \{Store(A, v^A(\&(o)), v^A(e))\} & \text{if } A \models_v o\downarrow \wedge\, e\downarrow \\ \emptyset & \text{otherwise} \end{cases}$,
- $[\![\texttt{New}(o)]\!]_v(A) = \left\{ A_1 \;\middle|\; \begin{array}{l} A \models_v o\downarrow \wedge \exists_{x \in |A|_{t_2}} \exists_{y \in |A|_{t_3}} (\neg(x\uparrow^A)\downarrow \wedge \\ A_1 = Store(Store(A, v^A(\&(o)), x), x, y)) \end{array} \right\}$ *where t_1 is a sort of $\&(o)$, t_2 is a sort of o and t_2 is a sort of pointers to t_3,*
- $[\![\texttt{if } e \texttt{ then } Q \texttt{ else } R]\!]_v(A) = \begin{cases} [\![Q]\!]_v(A) & \text{if } A \models_v OK(e) \wedge e \\ [\![R]\!]_v(A) & \text{if } A \models_v OK(e) \wedge \neg e \\ \emptyset & \text{otherwise} \end{cases}$,
- $[\![\texttt{while } e \texttt{ do } P]\!]_v(A) = \left\{ B \;\middle|\; \begin{array}{l} \exists_{\langle A_0 \ldots A_n \rangle \in \overline{A}^+} A_0 = A \;\wedge\; A_n = B \wedge \\ \bigwedge_{i=0 \ldots n-1} (A_i \models_v (OK(e) \wedge e) \wedge \\ A_{i+1} \in [\![P]\!]_v(A_i)) \wedge A_n \models_v OK(e) \wedge \neg e \end{array} \right\}$.

Note, that $[\![P]\!]_v(A) \subseteq \overline{A}$, i.e. the execution of a program can change only the contents of memory (represented by $\_\uparrow$ functions). All other functions and sorts remain unchanged.

5 Hoare Logic

In this section we deal with the problem of verification of while-programs operating on pointer data-structures against their specifications. A while-program specification can be expressed in the form of a precondition and a postcondition. We assume that these conditions are expressed in the formalism presented in Sect. 3. We present a version of Hoare logic [1,6,9] of while-programs, with dynamic formulae as assertions.

Usually, a postcondition is not only a condition on final states of the data-structure, but a relation between starting and final states. In the classical Hoare logic, this effect is achieved by the introduction of auxiliary (non-programming) variables. The precondition can state that a given programming variable is equal to an auxiliary one. Then the postcondition can refer to the previous value of the programming variable through the auxiliary one. However, in case of pointer data-structures, such auxiliary variables should store values of all dynamically

allocated variables. That would mean that assertions are expressed in second-order logic. Such an approach is presented in [2]. Note that these auxiliary variables are not quantified. Therefore they do not have to be variables. Instead of auxiliary variables, we allow the extension of the signature by introduction of function symbols $\_\uparrow$ with various decorations. We adopt the following notational convention:

- undecorated function symbols $\_\uparrow$ refer to the 'current' state of the data-structure,
- function symbols $\_\uparrow^{\backprime}$ can appear in postconditions and refer to the 'previous' state of the data-structure,
- function symbols $\_\uparrow$ decorated with other symbols are auxiliary.

For this section, let $Y \subseteq \mathcal{D} \setminus \{\wr\} \cup \{\varepsilon\}$, $\varepsilon \in Y$, $\Sigma = Sig(D)^Y$ and X be a set of variables induced by V.

Definition 16. *Each such triple $\{\varphi\}P\{\psi\}$ that $\varphi \in F_{\square}(X)$, $\psi \in F_{\square\,\square}(X)$, $P \in Prog_D(V)$ is called a dynamic Hoare formulae (over Σ and V). We denote the set of all such formulae (for a given D, Σ and V) by $DH_{\square}(V)$.*

In classical Hoare logic, satisfaction of a formula does not depend on (a single) valuation of variables. In our approach, logical variables represent addresses of programming variables and values of programming variables are represented by (undecorated and decorated) $\_\uparrow$ functions. Therefore we define satisfaction of dynamic Hoare formulae for the sets of partial algebras differing in interpretation of (undecorated and decorated) $\_\uparrow$ functions, and variables valuations.

Definition 17. *Let $A \in DStr(\boldsymbol{\Delta}\boldsymbol{\Sigma})$, $v : X \to |A|$ be a variables valuation, $\varphi \in F_{\square}(X)$, $\psi \in F_{\square\,\square}(X)$, $P \in Prog_D(V)$. We denote by $Post(\overline{A}, v, \varphi\, P)$ and $Pre(\overline{A}, v, \psi\, P)$ such sets of partial $\boldsymbol{\Delta}\boldsymbol{\Sigma}$ -algebras that:*

$$Post(\overline{A}, v, \varphi\, P) = \{A_1 \in \overline{A} \mid A_1 \models_v {\backprime}\varphi\, A_{1\,|\square} \in [\![P]\!]_v(A_{1\,|\backprime})\}\ ,$$

$$Pre(\overline{A}, v, \psi\, P) = \{A_1 \in \overline{A} \mid \forall_{A_2 \in \overline{A}}(A_2 \textit{ differs from } A_1 \textit{ only in interpretation of function symbols } \_\uparrow, A_{2\,|\square} \in [\![P]\!]_v(A_{2\,|\backprime})) \Rightarrow A_2 \models_v \psi\}\ .$$

Intuitively, *Post* defines a set of partial $\boldsymbol{\Delta}\boldsymbol{\Sigma}$ -algebras representing possible (terminating) executions of P starting in states satisfying φ. *Pre* defines a set of partial $\boldsymbol{\Delta}\boldsymbol{\Sigma}$ -algebras having such 'starting' states, that if P terminates, then ψ is true.

Definition 18. *We say that a dynamic Hoare formula $\{\varphi\}P\{\psi\} \in DH_{\square}(V)$ is satisfied for $\overline{A}$ ($A \in DStr(\boldsymbol{\Delta}\boldsymbol{\Sigma})$) and variables valuation $v : X \to |\overline{A}|$ (written $\overline{A} \models_v \{\varphi\}P\{\psi\}$), when for all $A_1 \in Post(\overline{A}, v, \varphi\, P)$, we have $A_1 \models_v \psi$.*

Proposition 1. *Let us assume that $\overline{A} \models_v \{\varphi\}P\{\psi\}$, $A_1 \in \overline{A}$, then:*

- *if $A_1 \models_v {\backprime}\varphi$, then $A_1 \in Pre(\overline{A}, v, \psi\, P)$,*
- *if there exists such $\pi \in F_{\square}(X)$ that $Pre(\overline{A}, v, \psi\, P) = \{A_1 \in \overline{A} \mid A_1 \models_v {\backprime}\pi\}$, then for all $A_1 \in \overline{A}$ we have $A_1 \models_v \varphi \Rightarrow \pi$.* □

Proposition 2. *If* $\pi \in F_{\square}(X)$ *and* $\psi_1, \psi_2 \in F_{\square\,\square}(X)$ *such that* $Pre(\overline{A}, v, \psi_1, P) = \{A_1 \in \overline{A} \mid A_1 \models_v \backprime\pi\}$ *and* $Post(\overline{A}, v, \pi, P) = \{A_1 \in \overline{A} \mid A_1 \models_v \psi_2\}$, *then* $\forall_{A_1 \in \overline{A}} A_1 \models_v \psi_2 \Rightarrow \psi_1$. □

Let $\circledast$ be a decoration. By $NChng(\circledast)$ we will denote a first-order formula stating that functions $\_\uparrow$ and $\_\uparrow^{\circledast}$ are identical (for all pointer sorts).

Let $p_1, \ldots, p_k, s_1, \ldots, s_k \in S_{\square}$, p_i be a sort of pointers to s_i, $o_1 \in T_{\square}(X)_{p_1}, \ldots, o_k \in T_{\square}(X)_{p_k}$, $t_1 \in T_{\square}(X)_{s_1}, \ldots, t_k \in T_{\square}(X)_{s_k}$ be terms without decorations. By $Stored(o_1, t_1, \ldots, o_k, t_k)$ we will denote a first-order formula describing the results of storing the value of t_1 at o_1, t_2 at o_2, $\ldots$, t_k at o_k, i.e.: for all $A \in DStr(\Delta\Sigma)$, $v : X \to |A|$ we have $A \models_v Stored(o_1, t_1, \ldots, o_k, t_k)$ iff

$$A_{|\square} = Store(\ldots(Store(A_{|\backprime}, v^A(o_1), v^A(t_1)), \ldots), v^A(o_k), v^A(t_k)) \ .$$

Obviously, such a formula always exists, but its form depends on declarations of record types. Note that if, for example, s is a record type, then $Stored(o, t)$ has to express appropriate changes of record fields (and if some of these fields are also records, then changes of their fields, etc.). Therefore we do not give here a general form of *Stored*.

Now we present a deduction system for a dynamic Hoare logic.

Definition 19. *Let* $\Lambda \subseteq F_{\square\,\square}(X)$ *be such a set of formulae (called assumptions), that for each signature morphism* $\sigma : \Delta\Sigma \to \Delta\Sigma$ *permuting decorations, if* $\alpha \in \Lambda$, *then* $\sigma(\alpha) \in \Lambda$. *Let* $\{\varphi\}P\{\psi\} \in DH_{\square}(V)$. *We denote by* $\Lambda \vdash \{\varphi\}P\{\psi\}$ *the fact, that basing on assumptions from* Λ *and using the proof system described below, we can prove* $\{\varphi\}P\{\psi\}$. *We will write* $\vdash \{\varphi\}P\{\psi\}$ *instead of* $\emptyset \vdash \{\varphi\}P\{\psi\}$.

– *Axioms:*

$$\vdash \{\varphi\}\varepsilon\{\backprime\varphi \wedge NChng(\backprime)\} \ , \tag{1}$$

$$\vdash \{\varphi\}o\mathtt{:=}e\left\{\backprime\varphi \wedge (\backprime o)\downarrow \wedge (\backprime e)\downarrow \wedge Stored(\backprime\&(o), \backprime e)\right\} \ , \tag{2}$$

$$\vdash \{\varphi\}\mathtt{New}(o)\left\{\backprime\varphi \wedge (\backprime o)\downarrow \wedge \exists_{x:t_1 \backprime y:t_2}(\neg(x\backprime\uparrow)\downarrow \wedge Stored(\&(o), x, x, y))\right\} \ , \tag{3}$$

where t_1 *is a sort of* o *and it is a sort of pointers to* t_2,

– *Rules:*

$$\frac{\varphi \Rightarrow \varphi_1 \in \Lambda, \Lambda_1 \vdash \{\varphi_1\}P\{\psi_1\}, (\psi_1 \wedge \backprime\varphi) \Rightarrow \psi \in \Lambda}{\Lambda \vdash \{\varphi\}P\{\psi\}} \ , \tag{4}$$

where $\Lambda_1 \subseteq \Lambda$,

$$\frac{\Lambda \vdash \{\varphi \wedge NChng(\circledast)\}P\{\psi\}}{\Lambda \vdash \{\varphi\}P\{\psi\}} \ , \tag{5}$$

where $\circledast$ *is a decoration different from* $\backprime$, *and not appearing in* φ *or* ψ,

$$\frac{\Lambda \vdash \{\varphi\}P\{\psi\}}{\Lambda \vdash \{\varphi\}\mathtt{begin}\ P\ \mathtt{end}\{\psi\}} \ , \tag{6}$$

$$\frac{\Lambda \vdash \{\varphi\}P\{\xi\}, \Lambda \vdash \{\sigma(\xi)\}Q\{\sigma(\psi)\}}{\Lambda \vdash \{\varphi\}P; Q\{\psi\}} \ , \tag{7}$$

where $\sigma : \mathbf{\Sigma} \to \mathbf{\Sigma}$ is a signature morphism changing $\setminus$ decorations to $\circledast$, and $\circledast$ is a decoration different from $\setminus$ and not appearing in φ, ξ or ψ,

$$\frac{\Lambda \vdash \{\varphi \wedge OK(\alpha) \wedge \alpha\} P \{\psi\},\ \Lambda \vdash \{\varphi \wedge OK(\alpha) \wedge \neg\alpha\} Q \{\psi\}}{\Lambda \vdash \{\varphi\}\texttt{if } \alpha \texttt{ then } P \texttt{ else } Q\{\psi\}}\ , \tag{8}$$

$$\frac{\Lambda \vdash \{\varphi \wedge OK(\alpha) \wedge \alpha\} P \{\varphi\}}{\Lambda \vdash \{\varphi\}\texttt{while } \alpha \texttt{ do } P\{\varphi \wedge OK(\alpha) \wedge \neg\alpha\}}\ . \tag{9}$$

We denote the above proof system by DH.

Intuitively, Λ contains all known facts concerning data-structures, which does not depend on the actual state. For example, we can instantiate Λ with the set of dynamic formulae satisfied by all partial algebras from a given $\overline{A}$ (for a given variables valuation v), i.e. $\Lambda = Th_{\square\,\square}(\overline{A}, v) = \{\varphi \in F_{\square\,\square}(X) \mid \forall_{A_1 \in \overline{A}} A_1 \models_v \varphi\}$.

Theorem 1. *The proof system DH is sound, i.e. if $\Lambda \vdash \{\varphi\} P \{\psi\}$, $A \in DStr(\Sigma)$ and $v : X \to |A|$ are such, that for all $\alpha \in \Lambda$ and $A_1 \in \overline{A}$ we have $A_1 \models_v \alpha$, then $\overline{A} \models_v \{\varphi\} P \{\psi\}$*

Proof of this theorem can be found in [14].

The proof system *DH* is also complete in the sense of Cook.

Definition 20. *Let $A \in DStr(\mathbf{\Sigma})$ and $v : X \to |A|$ be variables valuation. We say that the set of dynamic formulae $F_{\square\,\square}(X)$ is expressive, with respect to $\overline{A}$, v and $Prog_D(V)$, if for all $\psi \in F_{\square\,\square}(X)$ and $P \in Prog_D(V)$ there exists such $\pi \in F_{\square}(X)$ that $Pre(\overline{A}, v, \psi P) = \{A_1 \in \overline{A} \mid A_1 \models_v \setminus\pi\}$.*

Theorem 2. *The proof system DH is complete in the sense of Cook, i.e. for all $A \in DStr(\mathbf{\Sigma})$ and $v : X \to |A|$, if $F_{\square\,\square}(X)$ is expressive, with respect to $\overline{A}$, v and $Prog_D(V)$, and $\overline{A} \models_v \{\varphi\} P \{\psi\}$, then, for ι being natural insertion of $\mathbf{\Sigma}$ into $Sig(D)^{\mathcal{D}}$ and such $A_1 \in DStr(Sig(D)^{\mathcal{D}})$ that $A_{1|\square} = A$, we have $Th_{Sig(D)^{\mathcal{D}}}(\overline{A_1}, v) \vdash \{\varphi\} P \{\psi\}$.*

The proof of this theorem can be found in [14]. It follows the classical schema presented e.g. in [1], with all the necessary adaptations.

6 Conclusions

This work is a step forward in merging specification methods with efficient algorithms and data-structures. We have adopted temporal specifications of dynamic algebras [5] for the purpose of specification of pointer data-structures and programs operating on them. In the proposed formalism temporal combinators have been used to describe structure of pointer links. As it can be seen from examples, this idea allows compact and relatively readable specification of pointer data-structures. Its practical usefulness should be verified by thorough case-studies.

We have embedded the specification formalism within the framework of Hoare logic. Presented proof system satisfies standard properties of soundness and completeness in the sense of Cook. This gives us an elementary tool for verification of programs operating on pointer data-structures.

References

[1] K.R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.

[2] H. Bickel and W. Struckmann. The Hoare logic of data types. Technical Report 95-04, Technische Universität Braunschweig, Deutschland, 1995.

[3] R. Cartwright and D. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.

[4] T.H. Cormen, C.E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] G. Costa and G. Reggio. Specification of abstract dynamic-data types: A temporal approach. *Theoretical Comput. Sci.*, 173(2):513–554, 1997.

[6] O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[7] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specificaton 1*. Number 6 in EATCS MTCS. Springer-Verlag, 1985.

[8] Y. Feng and J. Liu. A temporal approach to algebraic specifications. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90: Theories of Concurrency: Unification and Extension*, number 458 in LNCS, pages 216–229. Springer-Verlag, 1990.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–583, 1969.

[10] C.A.R. Hoare and H. Jifeng. A trace model for pointers and objects. In *ECOOP'99*, number 1628 in LNCS, pages 1–18. Springer-Verlag, 1999.

[11] T.M.V. Janssen and P. van Emde Boas. On the proper treatment of referencing, dereferencing and assignment. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming*, number 52 in LNCS, pages 282–300. Springer-Verlag, 1977.

[12] N. Klarlund and M.I. Schwartzbach. Graph types. In *Proceedings of the 20th Symposium on Principles of Programming Languages*, pages 196–205. ACM, 1993.

[13] N. Klarlund and M.I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In S. Tison, editor, *Trees in Algebra and Programming — CAAP'94. Proceedings*, number 787 in LNCS, pages 187–201. Springer-Verlag, 1994.

[14] M. Kubica. Temporal-style specifications of pointer data-structures. Technical Report TR 02–03 (268), Institute of Informatics, Warsaw University, 2002.

[15] J.M. Spivey. *The Z notation, A Reference Manual*. Second edition. Prentice-Hall, 1992.

[16] A. Szałas. Towards the temporal approach to abstract data types. *Fundamenta Informaticae*, XI:49–63, 1988.

[17] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, volume 3, chapter 7. Springer-Verlag, 1997.

[18] M. Wirsing. Algebraic specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 676–788. Elsevier, 1990.