# Run-time correctness checking is algorithmically undecidable for pointer data structures

Mikhail Auguston[1], Vladik Kreinovich[2], and Luc Longpré[2]

[1]Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
email mikau@cs.nmsu.edu

[2]Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
email {vladik,longpre}@cs.utep.edu

**Abstract**

Programs routinely use complicated pointer (linked list-type) data structures such as linked lists, doubly linked lists, different types of trees, etc. These data structures are usually defined *inductively*: e.g., a tree can be defined as a structure that results from an empty tree by an arbitrary sequence of adding and deleting elements.

When the program runs, these data structures take dynamically changing shapes. To test program correctness, it is important to check, at run-time, whether a current shape is a correct implementation of the corresponding structure. Algorithms are known for checking the "shape correctness" for *basic* pointer-based data structures such as linked list, binary tree, etc.

In this paper, we show that the *general* problem – verifying that a given shape is an instance of an inductively defined data structure – is algorithmically undecidable.

## 1 Introduction

**Pointer (linked list-type) data structures: brief reminder.** Programs routinely use complicated pointer data structures such as linked lists, doubly linked lists, different types of trees, etc. Unlike *static* data structures (such as static arrays or (non-variable) records) which have the same shape throughout the run-time, the "shape" of a pointer data structure is dynamically changing: new elements can be added or deleted, connections can be changed, etc.

A pointer data structures is, usually, a collection of records; each record contains:

- (one or several) *data fields* and
- (one or several) *pointer fields*, i.e., fields that contain *pointers* to other records (i.e., *addresses* of other records).

For example, a typical implementation of a linked list is:

$$\underset{r_0}{\overline{\underline{|r_1|}}}\quad \underset{r_1}{\overline{\underline{|*|r_2|}}}\quad \underset{r_2}{\overline{\underline{|*|\text{nil}|}}}$$

where $*$ indicates data, each record points to the next one, except for the last record that does not point to anything (this is expressed by giving a pointer a special non-address value of nil). Similarly, we can give examples of a double-linked list:

$$\underset{r_0}{\overline{\underline{|r_1|}}}\quad \underset{r_1}{\overline{\underline{|\text{nil}|*|r_2|}}}\quad \underset{r_2}{\overline{\underline{|r_1|*|r_3|}}}\quad \underset{r_3}{\overline{\underline{|r_2|*|\text{nil}|}}}$$

and of a binary tree:

$$
\begin{array}{cccc}
 & \multicolumn{2}{c}{\overline{\underline{|r|}}} & \\
 & \multicolumn{2}{c}{p} & \\
 & \multicolumn{2}{c}{\overline{\underline{|r_0| * |r_1|}}} & \\
 & \multicolumn{2}{c}{r} & \\
 & \overline{\underline{|r_{00}| * |r_{01}|}} & \overline{\underline{|\mathrm{nil}| * |r_{11}|}} & \\
 & r_0 & r_1 & \\
\overline{\underline{|\mathrm{nil}| * |\mathrm{nil}|}} & \overline{\underline{|\mathrm{nil}| * |\mathrm{nil}|}} & & \overline{\underline{|\mathrm{nil}| * |\mathrm{nil}|}} \\
r_{00} & r_{01} & & r_{11}
\end{array}
$$

These data structures are usually defined *inductively*: e.g., a tree can be defined as a structure that results from an empty tree by an arbitrary sequence of adding and deleting elements.

**It is important to check the shapes.** When the program runs, pointer data structures take dynamically changing shapes. To test program correctness, it is important to check, at run-time, whether a current shape is a correct implementation of the corresponding structure: e.g., that it is indeed a linked list, or a binary tree, etc.

If we only know the current shape, and the data structure is defined inductively, then it is not immediately clear how to check whether the current shape is indeed a (correct) implementation of the desired structure.

**What is known.** Algorithms are known that check shape correctness for *basic* pointer data structures such as linked lists, doubly linked lists, binary trees at run-time (see [1]).

**What we show.** In this paper, we show that the *general* problem – verifying that a given shape is an instance of an inductively defined data structure – is algorithmically undecidable.

**This area of research is important.** Theorists should find this to be an important class of applied problems.

Our result is natural and expected. By proving the impossibility of a *general* algorithm, we wanted to emphasize the importance of *partial* algorithms, e.g., the ones developed in [1].

**Related results.** Several related problems are known to be undecidable, e.g., the problem of *static* shape analysis, i.e., the problem of checking, for a given program, whether it is possible, for some inputs, that at some moment of time, two pointer variables will point to the same place (i.e., be *aliases*) [3]. For a general overview of related undecidable and NP-hard problems, see [4].

## 2 Definitions and the main result

*Comment.* Our formal description of pointer data types is somewhat similar, e.g., to the formalism proposed in [2].

**Definition 1.** *By a record type $t$, we mean a finite sequence $t_1 \ldots t_F$ in which each $t_i$ is either a symbol $d$, or a symbol $p$. The total number $F$ of elements in this sequence is called a number of fields.*

- *If $t_i = d$, we say that $i$-th field is a data field.*
- *If $t_i = p$, we say that $i$-th field is a pointer field.*

*Example.* In the above linked list example:

- $r_0$ is a record of type $p$, with only one field $t_1 = p$ of pointer type;
- $r_1$ is a record of type $dp$, with two fields, of which:
  - the first field $t_1 = d$ is of data type, and
  - the second field $t_2 = p$ is of pointer type.

**Definition 2.** *By a shape, we mean a triple $S = \langle R, T, P\rangle$, where:*

- *$R$ is a finite set; its elements are called records;*
- *$T$ maps each record $r \in R$ into a record type $t = T(r)$; and*
- *$P$ maps, for each record $r \in R$, each pointer field of the corresponding record type $T(r)$ either into an element $r' \in R$, or into a special symbol* nil*; if the filed is mapped into a record $r' \in R$, we say that the corresponding pointer points to $r'$.*

*Example.* The shape of the above linked list is a triple $S_0 = \langle R, T, P\rangle$, where:

- the set $R$ consists of three records $R = \{r_0, r_1, r_2\}$;
- the mapping $T$ is defined as follows: $T(r_0) = p$ and $T(r_1) = T(r_2) = dp$;
- the mapping $P$ does the following:
  - for the record $r_0$, $P$ maps the only pointer field of the record type $T(r_0)$ into $r_1$;
  - for the record $r_1$, $P$ maps the only pointer field of the record type $T(r_1)$ into $r_2$;
  - for the record $r_2$, $P$ maps the only pointer field of the record type $T(r_2)$ into nil.

*Motivation for the next definition.* If we take only *some* records from a shape, then we get a *subshape*. The main difference between a shape and a subshape is as follows:

- In a *shape*, for each pointer, we have *two* options: the pointer is either nil, or points to some object from the same shape.
- In a *subshape*, we can have a *third* option, when a pointer is not nil, and it is not pointing to anything in this same subshape; instead, it points instead to an object *outside* this subshape.

Formally, a subshape ("pattern") can be defined as follows:

**Definition 3.** *A pattern is a quadruple $\mathcal{P} = \langle R_{\text{in}}, R_{\text{out}}, T, P\rangle$, where:*

- *$R_{\text{in}}$ is a finite set; its elements are called internal records of this pattern;*
- *$R_{\text{out}}$ is a finite set; its elements are called external records of this pattern;*
- *$T$ maps each record $r \in R_{\text{in}}$ into a record type $t = T(r)$; and*
- *$P$ maps, for each record $r \in R_{\text{in}}$, each pointer field of the corresponding record type $T(r)$ either into an element $r' \in R_{\text{in}} \cup R_{\text{out}}$, or into a special symbol* nil*, in such a way that for each record $r \in R_{\text{out}}$, there is exactly one field that is mapped into $r$.*

*Example.* We can view the beginning of the linked list

$$\begin{array}{cc} \overline{|u_1|} & \overline{|*|\tilde{u}_2|} \\ u_0 & u_1 \end{array}$$

as a pattern $\mathcal{P}_0 = \langle R_{\text{in}}, R_{\text{out}}, T, P\rangle$, in which:

- we have two internal records $R_{\text{in}} = \{u_0, u_1\}$;
- we have a single external record $R_{\text{out}} = \{\tilde{u}_2\}$;
- record types are $T(u_0) = p$ and $T(u_1) = dp$; and
- the mapping $P$ does the following:
  - for the record $u_0$, $P$ maps the only pointer field of the record type $T(u_0)$ into $u_1$;
  - for the record $u_1$, $P$ maps the only pointer field of the record type $T(u_1)$ into $\tilde{u}_2$.

**Definition 4.** *Let $S = \langle R, T, P\rangle$ be a shape, and let $R_0$ be a subset of the set $R$ of its records. We say that the pair $\langle R_0, S\rangle$ is* isomorphic *to a pattern $\mathcal{P} = \langle R_{\text{in}}, R_{\text{out}}, T, P\rangle$ if there exists a mapping $f : R_{\text{in}} \cup R_{\text{out}} \to R \cup \{nil\}$ for which the following four conditions are satisfied:*

- *The function $f$ is a one-to-one mapping between records from $R_{\text{in}}$ and records from $R_0$;*
- *For each record $r \in R_{\text{in}}$, the records $r$ and $f(r)$ have the same record type (i.e., $T(r) = T(f(r))$).*
- *The function $f$ maps each record from $R_{\text{out}}$ into some record $r' \in R - R_0$ or into* nil*; and*
- *For each record $r \in R_{\text{in}}$, and for each pointer field of this record, if in $R_{\text{in}}$, this field was pointing to $r'$, then in $R_0$, the corresponding field of the record $f(r)$ is pointing to $f(r')$.*

*The mapping $f$ that satisfies these four conditions is called an* isomorphism.

*Example.* For example, if $S_0 = \langle R, T, P\rangle$ is the above linked-list shape, and $R_0 = \{r_0, r_1\} \subset R = \{r_0, r_1, r_2\}$, then the isomorphism between the pair $\langle R_0, S_0\rangle$ and the pattern $\mathcal{P}_0$ (described after Definition 3) is established by the following mapping $f$: $f(u_0) = r_0$, $f(u_1) = r_1$, and $f(\tilde{u}_2) = r_2$.

**Definition 5.** *By an* elementary transformation *(or simply a* transformation*, for short), we mean a tuple $E = \langle \mathcal{P}_{\text{before}}, \mathcal{P}_{\text{after}}, ref\rangle$, where:*

- *$\mathcal{P}_{\text{before}}$ and $\mathcal{P}_{\text{after}}$ are patterns; and*
- ref *maps each record (internal or external) of the pattern $\mathcal{P}_{\text{before}}$ into either a record (internal or external) of the pattern $\mathcal{P}_{\text{after}}$, or into* nil.

*Example.* For example, adding an element at the beginning of a linked list means going from the initial pattern

$$\begin{array}{cc} \overline{\underline{|u_1|}} & \overline{\underline{|\ast|\tilde{u}_2|}} \\ u_0 & u_1 \end{array}$$

to a new pattern

$$\begin{array}{ccc} \overline{\underline{|u_{\text{new}}|}} & \overline{\underline{|\ast|u_1|}} & \overline{\underline{|\ast|\tilde{u}_2|}} \\ u_0 & u_{\text{new}} & u_1 \end{array}$$

Let us show that this transformation $E_0$ is indeed described by Definition 5. Indeed:

- The original pattern $\mathcal{P}_{\text{before}}$ is the same pattern $\mathcal{P}_0$ as described after Definition 3.
- Similarly, the new pattern, which after re-naming takes the form

  $$\begin{array}{ccc} \overline{\underline{|s_1|}} & \overline{\underline{|\ast|s_2|}} & \overline{\underline{|\ast|\tilde{s}_3|}} \\ s_0 & s_1 & s_2 \end{array},$$

  can be represented as a triple $\mathcal{P}_{\text{after}} = \langle R_{\text{in}}, R_{\text{out}}, T, P\rangle$, in which:
  - we have three internal records $R_{\text{in}} = \{s_0, s_1, s_2\}$;
  - we have a single external record $R_{\text{out}} = \{\tilde{s}_3\}$;
  - record types are $T(s_0) = p$ and $T(s_1) = T(s_2) = dp$; and
  - the mapping $P$ does the following:
    - for the record $s_0$, $P$ maps the only pointer field of the record type $T(s_0)$ into $s_1$;
    - for the record $s_1$, $P$ maps the only pointer field of the record type $T(s_1)$ into $s_2$.
    - for the record $s_2$, $P$ maps the only pointer field of the record type $T(s_2)$ into $\tilde{s}_3$.
- The mapping $ref$ maps $u_0$ into $s_0$, $u_1$ into $s_2$, and $\tilde{u}_2$ into $\tilde{s}_3$.

**Definition 6.** *We say that a transformation $E = \langle \mathcal{P}_{\text{before}}, \mathcal{P}_{\text{after}}, ref \rangle$ transforms a shape $S_{\text{before}} = \langle R_{\text{before}}, T_{\text{before}}, P_{\text{before}} \rangle$ into a shape $S_{\text{after}} = \langle R_{\text{after}}, T_{\text{after}}, P_{\text{after}} \rangle$ if there exists a set $R_0 \subseteq R_{\text{before}}$ such that:*

- *the pair $\langle R_0, S_{\text{before}} \rangle$ is isomorphic to the pattern $\mathcal{P}_{\text{before}}$, under some isomorphism $f$;*
- *the new shape $S_{\text{after}}$ is obtained from $S_{\text{before}}$ as follows:*
  - *the set $R_0$ is replaced by an (equivalent) set $\mathcal{P}_{\text{after}}$;*
  - *every pointer from each pointer field of each record $r \in R_{\text{before}} - R_0$ that points to a record $r' \in R_0$ is replaced by a pointer to $ref(f^{-1}(r'))$;*
  - *every pointer from each pointer field of each record $s \in ref(f^{-1}(R_0))$, that points to a record $s' \in ref(f^{-1}(R_{\text{before}} - R_0))$ is replaced by a pointer to $r' = f(ref^{-1}(s'))$.*

*Example.* Let us apply the above transformation $E_0$ to the following shape $S_{\text{before}} = \langle R_{\text{before}}, T_{\text{before}}, P_{\text{before}} \rangle$:

$$\begin{array}{ccc} \overline{\underline{|q_1|}} & \overline{\underline{|*|q_2|}} & \overline{\underline{|*|\text{nil}|}} \\ q_0 & q_1 & q_2 \end{array}$$

For this shape, $R_{\text{before}} = \{q_0, q_1, q_2\}$. If we take a set $R_0 = \{q_0, q_1\} \subseteq R_{\text{before}}$, then the pair $\langle R_0, S_{\text{before}} \rangle$ is isomorphic to the initial pattern $\mathcal{P}_{\text{before}}$ of the transformation $E_0$, i.e., to the pattern

$$\begin{array}{cc} \overline{\underline{|u_1|}} & \overline{\underline{|*|\tilde{u}_2|}} \\ u_0 & u_1 \end{array},$$

under the isomorphism $f$ for which $f(u_0) = q_0$, $f(u_1) = q_1$, and $f(\tilde{u}_2) = q_2$.

So, we can the above transformation $E_0$ to the original shape $S_{\text{before}}$. To describe the result of this transformation, let us follow the three steps listed in Definition 6:

- First, we replace the records from the set $R_0$ by the records from the set $\mathcal{P}_{\text{after}}$. As a result, we get the following picture:

$$\begin{array}{cccc} \overline{\underline{|s_1|}} & \overline{\underline{|*|s_2|}} & \overline{\underline{|*|\tilde{s}_3|}} & \overline{\underline{|*|\text{nil}|}} \\ s_0 & s_1 & s_2 & q_2 \end{array}$$

- Second, we must replace all pointers from all pointer field of all the records $r \in R_{\text{before}} - R_0$ that point to records $r' \in R_0$. In our example, the difference set $R_{\text{before}} - R_0$ contains only one record $r_3$, and the only pointer field of this record does not point to a record from $R_0$. Thus, in our example, we do not make any replacements on the second step.
- Finally, we must replace, for each record $s \in ref(f^{-1}(R_0))$, each pointer which points to a record $s' \in ref(f^{-1}(R_{\text{before}} - R_0))$ by a pointer to $r' = f(ref^{-1}(s'))$. In our example:
  - We have $R_0 = \{q_0, q_1\}$, so $f^{-1}(R_0) = \{u_0, u_1\}$, and $ref(f^{-1}(R_0)) = ref(\{u_0, u_1\} = \{s_0, s_2\}$.
  - Similarly, $R_{\text{before}} - R_0 = \{q_2\}$, so $f^{-1}(\{q_2\}) = \{\tilde{u}_2\}$, and $ref(f^{-1}(R_{\text{before}} - R_0)) = \text{ref}(\{\tilde{u}_2\} = \{\tilde{s}_3\}$.

  Thus, we are looking for pointers that go from a record $s \in \{s_0, s_2\}$ to a record $s' \in \{\tilde{s}_3\}$. One can easily see that there is exactly one such pointer: from $s_2$ to $\tilde{s}_3$. According to our definition, we replace it with a pointer to $f(\text{ref}^{-1}(\tilde{s}_3)) = f(\tilde{u}_2) = q_2$. As a result, we get the following shape:

$$\begin{array}{cccc} \overline{\underline{|s_1|}} & \overline{\underline{|*|s_2|}} & \overline{\underline{|*|q_2|}} & \overline{\underline{|*|\text{nil}|}} \\ s_0 & s_1 & s_2 & q_2 \end{array}$$

In this example, we started with a linked list with *two* data-containing records, and ended up with a linked list with *three* such records.

**Definition 7.**

- *By a pointer data structure $D$, we mean a set consisting of finitely many shapes (called basic shapes) and finitely many elementary transformations.*

- *We say that a shape $S$ is a (correct) implementation of the pointer data structure $D$ if $S$ can be obtained from one of the basic shapes from $D$ by using a finite sequence of elementary transformations from the list $D$.*

*Example.* For example, a linked list can be described as a set $D$ that consists of the two basic shapes

$$\underset{r_0}{\underline{\overline{|\text{nil}|}}}$$

and

$$\underset{r_0}{\underline{\overline{|r_1|}}}\quad \underset{r_1}{\underline{\overline{|*|\text{nil}|}}}$$

and of a single transformation (the above-described one). One can easily see that if we apply this transformation to a linked list with $k$ data-containing records, then we get a new linked list with $k+1$ data-containing records. Thus:

- each shape that can be obtained from the basic shapes from $D$ by applying transformations from $D$ is a linked list, and

- every linked list shape is either in $D$ already (if it has 0 or 1 data-containing records), or it can be obtained from the single-data-record linked list by applying an appropriate number of transformations from $D$.

Thus, we get exactly shapes that correspond to linked list.

Similarly, we can describe doubly linked list, binary tree, and other pointer data structures.

**PROPOSITION.** *The problem of checking whether a given shape $S$ is a correct implementation of a given pointer data structure $D$ is algorithmically undecidable.*

# 3 Proof

To prove the Proposition, we will show that the halting problem for Turing machines (that is known to be algorithmically undecidable, see, e.g., [5]) can be reduced to the problem of checking whether a given shape is indeed a correct implementation of given data structure.

For this reduction, we will represent a state of a Turing machine by an appropriate shape. This shape will include the following:

- records that correspond to possible states $s_1, \ldots, s_m$ of the head; each of these records consists of a single field: a pointer filed, with a pointer pointing to nil;

- records that correspond to possible symbols $a_1, \ldots, a_p$ of the tape (empty cell $\Lambda$ corresponds to nil); each of these records also consists of only one field: a pointer filed, with a pointer pointing to nil;

- a doubly linked list that describes the contents of the tape; each element of this linked list describes a cell on the tape, and the middle field of this element points to the symbol written in this cell; and, finally,

- a special record $s_0$ with two pointers that indicate:
  - the current *state* of the head, and
  - the current *location* of the head.

For example, consider a Turing machine with:

- states $\{s_1, s_2, s_3\}$;
- an alphabet $\{a_1, a_2, a_2, a_4, a_5\}$
- current state $s_3$;
- its tape consists of:
  - an empty 0-th cell ($c_0 = \Lambda$),
  - cell 1 containing $a_1$,
  - cell 2 containing $a_5$, and
  - cell 3 with $a_2$,

  (i.e., the tape contains a sequence $\overline{\underline{|\Lambda|a_1|a_5|a_2|}}$), and
- the head pointing to the second cell.

Then, this state of this Turing machine is represented by the following shape:

$$\begin{array}{ccc} \overline{\underline{|\text{nil}|}} & \overline{\underline{|\text{nil}|}} & \overline{\underline{|\text{nil}|}} \\ s_1 & s_2 & s_3 \end{array}$$

$$\begin{array}{ccccc} \overline{\underline{|\text{nil}|}} & \overline{\underline{|\text{nil}|}} & \overline{\underline{|\text{nil}|}} & \overline{\underline{|\text{nil}|}} & \overline{\underline{|\text{nil}|}} \\ a_1 & a_2 & a_3 & a_4 & a_5 \end{array}$$

$$\begin{array}{cccc} \overline{\underline{|\text{nil}|\text{nil}|c_1|}} & \overline{\underline{|c_0|a_1|c_2|}} & \overline{\underline{|c_1|a_5|c_3|}} & \overline{\underline{|c_2|a_2|\text{nil}|}} \\ c_0 & c_1 & c_2 & c_3 \end{array}$$

$$\begin{array}{c} \overline{\underline{|s_3|c_2|}} \\ s_0 \end{array}$$

The initial state of a Turing machine is thus naturally represented as a shape. Each step of a Turing machine is *local* and therefore, can also be naturally represented as an elementary transformation of shapes.

To complete the proof, we must add transformations that delete (record by record) everything except the halting state cell if the machine has reached the halting state. After this translation, checking whether the Turing machine will halt is equivalent to checking whether a one-record state is a correct implementation of the corresponding data structure. Since halting problem is undecidable, checking whether a given shape is a correct implementation of a given data structure is, therefore, also undecidable.

# 4 Future plans

For inductively defined data shapes, it is necessary to check whether a given shape is indeed a (correct) implementation of the desired structure.

- Algorithms are known that check it for *basic* pointer-based data structures.
- In this paper, we have shown that, *in general* (for arbitrary pointer data structures), this problem is algorithmically undecidable.

Our main future plan is to find reasonable classes of data structures for which such shape-checking algorithm is still possible.

**Acknowledgments.** This work was supported in part by NSF under grants No. EEC-9322370 and DUE-9750858, by NASA under cooperative agreement NCCW-0089, and by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-95-1-0518. The authors are thankful to Michael Gelfond for valuable discussions.

## References


[1] M. Auguston and Miu Har Hon, "Assertions for Dynamic Shape Analysis of List Data Structures", *Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging, AADEBUG'97*, Linkoping, Sweden, May 26–27, 1997, pp. 37–42.

[2] J. L. Jensen, M. E. Jorgensen, N. Klarlund, and M. I. Schwartzbach, "Automatic verification of pointer programs using monadic second-order logic", *ACM SIGPLAN Notices*, 1997, Vol. 32, No. 5, pp. 226–234.

[3] W. A. Landi, "Undecidability of static analysis", *ACM Letters on Programming Languages and Systems*, 1992, Vol. 1, pp. 323–337.

[4] H. D. Pande, W. A. Landi, and B. G. Ryder, "Interprocedural def-use associations for C systems with single level pointers", *IEEE Transactions on Software Engineering*, 1994, Vol. 20, No. 5, pp. 385–403.

[5] C. H. Papadimitriou, *Computational Complexity*, Addison Wesley, San Diego, 1994.